

AFRL-RI-RS-TR-2009-158
Final Technical Report
June 2009



PROGRAMMING METHODOLOGY FOR HIGH PERFORMANCE APPLICATIONS ON TILED ARCHITECTURES

Georgia Institute of Technology

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. V301/01

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (<http://www.dtic.mil>).

AFRL-RI-RS-TR-2009-158 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
CHRISTOPHER J. FLYNN
Work Unit Manager

/s/
EDWARD J. JONES, Acting Chief
Advanced Computing Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE**Form Approved**
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**1. REPORT DATE (DD-MM-YYYY)**
JUN 09**2. REPORT TYPE**
Final**3. DATES COVERED (From - To)**
Jan 06 – Dec 08**4. TITLE AND SUBTITLE**PROGRAMMING METHODOLOGY FOR HIGH PERFORMANCE
APPLICATIONS ON TILED ARCHITECTURES**5a. CONTRACT NUMBER**

N/A

5b. GRANT NUMBER

FA8750-06-1-0012

5c. PROGRAM ELEMENT NUMBER

62303E

6. AUTHOR(S)

Mark A. Richards and Daniel P. Campbell

5d. PROJECT NUMBER

V301

5e. TASK NUMBER

XM

5f. WORK UNIT NUMBER

OR

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)Georgia Institute of Technology
505 10th Street NW
Atlanta, GA 30332-0001**8. PERFORMING ORGANIZATION
REPORT NUMBER**

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)DARPA
3701 N. Fairfax Drive
Arlington, VA 22203-1714AFRL/RITB
525 Brooks Road
Rome NY 13441-4505**10. SPONSOR/MONITOR'S ACRONYM(S)**
N/A**11. SPONSORING/MONITORING
AGENCY REPORT NUMBER**
AFRL-RI-RS-TR-2009-158**12. DISTRIBUTION AVAILABILITY STATEMENT**

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW-2009-2559, Date Cleared: 15-Jun-2009

13. SUPPLEMENTARY NOTES**14. ABSTRACT**

The core activity of the original Polymorphous Computing Architectures (PCA) program was to advance the design and implementation of several emerging academic multiprocessor-on-a-chip architecture projects (chip multiprocessors, or CMPs). The Georgia Tech team's primary role in the PCA program was to facilitate definitions and implementation of the Morphware Stable Interface (MSI), application software development architecture intended to be portable across PCA architectures, offer high development productivity, and maintain high performance in deployed software. The GT team organized and led the Morphware Forum, a PCA program-wide activity meeting quarterly to propose, debate, and develop MSI concepts and elements. In addition, Georgia Tech participated in related efforts to extend aspects of the MSI approach to additional hardware targets, such as graphical processing units, cluster computers, and cognitive computers, and to incorporate work on advanced parallel libraries taking place in other programs. This report summarizes the work accomplished in support of these endeavors.

15. SUBJECT TERMS

Polymorphous Computing Architectures, morphware, chip multiprocessors

16. SECURITY CLASSIFICATION OF:**a. REPORT**
U**b. ABSTRACT**
U**c. THIS PAGE**
U**17. LIMITATION OF
ABSTRACT**

UU

**18. NUMBER
OF PAGES**

431

19a. NAME OF RESPONSIBLE PERSON

Christopher J. Flynn

19b. TELEPHONE NUMBER (Include area code)

N/A

TABLE OF CONTENTS

<u>Paragraph</u>	<u>Page</u>
SECTION 1 INTRODUCTION	1
SECTION 2 THE MORPHWARE FORUM AND THE MORPHWARE STABLE INTERFACE ...	3
2.1 The Morphware Stable Interface.....	3
2.2 The Morphware Forum.....	3
2.2.1 Background	3
2.2.2 xPCA Morphware Forum Activities.....	4
2.2.3 Final Morphware Forum Products	4
2.3 Web site and e-mail	7
SECTION 3 RESEARCH INTO PROGRAMMING METHODS FOR HIGH PERFORMANCE TILED ARCHITECTURES	11
3.1 Signal Processing Libraries for Tiled Architectures	11
3.2 Alternative High Level Compiler	11
3.3 Graphics Processing Units for High Performance Embedded Computing	14
3.3.1 Performance Improvement Experiments	14
3.3.2 GPU Middleware.....	20
SECTION 4 INTERNET TOOLS SUPPORT FOR THE HIGH PRODUCTIVITY COMPUTING SYSTEMS PROGRAM	26
4.1 Background.....	26
4.2 HPCS Web Site	26
REFERENCES	28
List of Acronyms.....	30
APPENDIX A INTRODUCTION TO MORPHWARE: SOFTWARE ARCHITECTURE FOR POLYMORPHOUS COMPUTING ARCHITECTURES	A-1
APPENDIX B THE PCA METADATA SYSTEM.....	B-1
APPENDIX C PCA MACHINE MODEL.....	C-1
APPENDIX D STREAMING VIRTUAL MACHINE SPECIFICATION.....	D-1
APPENDIX E THREADED VIRTUAL MACHINE - HARDWARE ABSTRACTION LAYER (TVM-HAL) SPECIFICATION	E-1
APPENDIX F USER-LEVEL TVM (UVM) INTERFACE SPECIFICATION	F-1
APPENDIX G SVM ISSUES SUMMARY	G-1
APPENDIX H MACHINE MODEL (MM) ISSUES SUMMARY	H-1
APPENDIX I RUN TIME (RT) ISSUES SUMMARY	I-1
APPENDIX J SIGNAL PROCESSING LIBRARIES ON TILED ARCHITECTURES	J-1

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
Figure 1. The current Morphware Stable Interface (MSI) architecture.	3
Figure 2. Screen shot of the public home page of the Morphware Forum web site at www.morphware.org as of April 2009.....	8
Figure 3. Screen shot of the Morphware Forum web site internal “home page” for project participants.	9
Figure 4. Example Streamit code for a notional FM radio and corresponding implied graph....	12
Figure 5. Example FIR filterbank benchmark application.	13
Figure 6. Growth of GPU single precision floating point performance.	14
Figure 7. Wrapped (a) and unwrapped (b) two-dimensional phase functions.....	15
Figure 8. Runtime for the 2D phase unwrapping algorithm on a CPU, a GPU, and in MATLAB on a CPU.	16
Figure 9. Point updates per second for the 2D phase unwrapping algorithm on a CPU, a GPU, and in MATLAB on a CPU.	17
Figure 10. Illustration of RaiderTracer RCS prediction for a simple modeled airplane.	18
Figure 11. Speedup of GPU VSIPL++ vs. CPU VSIPL++ for the FIR operation.	22
Figure 12. Speedup of GPU VSIPL vs. TASP Reference VSIPL for various binary vector operations.....	24
Figure 13. Speedup of GPU VSIPL vs. TASP Reference VSIPL for 1D FFTs.	24
Figure 14. Output of GPU VSIPL range-Doppler map application showing three target with sidelobes in both range (vertical) and Doppler (horizontal) dimensions.	25
Figure 14. Screen shot of the home page of the High Productivity Computing Systems web site at www.highproductivity.org as of April 2009.	27

LIST OF TABLES

<u>Table</u>	<u>Page</u>
Table 1. Final comparative performance of native and SVM implementations of the benchmark of Figure 5 with four processors and 500,000 input data samples.....	13
Table 2. HPEC challenge Performance Results in msec.	20

PREFACE

This document is the Final Technical Report under Georgia Institute of Technology (Georgia Tech, GT) Project 21066Q6, “Programming Methodology for High Performance Applications on Tiled Architectures.” This project was sponsored by the Information Processing Technology Office (IPTO) of the US Defense Advanced Research Projects Agency (DARPA) and was conducted under the administration of the U.S. Air Force Research Laboratory (AFRL) under contract FA8750-06-1-0012. It was one component of DARPA’s Polymorphous Computing Architectures (PCA) program.

The authors would like to thank Dr. William Harrod of DARPA/IPTO, the PCA program manager, and Mr. Christopher Flynn of AFRL for their support of this effort. The authors also thank the principal investigators and other contributors to the other PCA program projects for their contributions to the Morphware Forum activity.

SECTION 1

INTRODUCTION

“Polymorphous Computing Architectures” (PCA) was a U.S. Defense Advanced Research Projects Agency (DARPA) program [1] that sought to substantially advance not only the performance but also the reconfigurability of Department of Defense (DoD) embedded computers. Phase I of the PCA program ran from June 2001 to June 2003. Phase II ran through December 2005. Some PCA participants, including the Georgia Institute of Technology (GT, Georgia Tech), continued in an “extended PCA program” (xPCA) from January 2006 to various termination dates. This report describes the research conducted by Georgia Tech as part of the xPCA program.

The core activity of the original PCA program was to advance the design and implementation of several emerging academic multiprocessor-on-a-chip architecture projects (chip multiprocessors, or CMPs). All of these projects seek to develop means to flexibly tie explicitly separate sub-processors into larger processors. The “polymorphous” aspect comes from the fact that the resulting larger processors can be tailored to fit different classes of applications, *e.g.*, conventional or digital signal processing (DSP), and even re-optimized (“morphed”) on the fly in response to application or environmental demands.

Georgia Tech’s activities within the original PCA program are fully detailed in the final technical report of that project [2]. The GT team’s primary role in the PCA program was to facilitate definition and implementation of the Morphware Stable Interface (MSI), an application software development architecture intended to be portable across PCA architectures, offer high development productivity, and maintain high performance in deployed software. To carry out that function, the GT team organized and led the Morphware Forum, a PCA program-wide activity meeting quarterly to propose, debate, and develop MSI concepts and elements. As Phase II of the PCA program ends, much work remains to be done on the MSI. The Morphware Forum will be continued, possibly in modified form, in DARPA’s “extended PCA” (xPCA) program. In addition, Georgia Tech is participating in related efforts to extend aspects of the MSI approach to additional hardware targets, such as graphical processing units, cluster computers, and cognitive computers, and to incorporate work on advanced parallel libraries taking place in other programs.

Under the xPCA extension, GT’s primary role was to continue development of the MSI. A successful MSI will provide an application development approach that serves the interest of DoD and other end users of CMPs. Because of the rising importance of CMP devices, their proliferation, and their differing native application development approaches, there remains a clear need to continue development of a device-portable environment for building high performance applications on CMPs and other parallel architectures. A single common application development approach enables end users such as DoD system integrators to build large-scale applications that can target various parallel implementation architectures. The MSI will provide an important step toward the development of an effective application development environment for a wide spectrum of embedded parallel machines.

Georgia Tech's specific responsibilities under this effort included

- Scheduling, organization, and leading quarterly Morphware Forum meetings at rotating locations;
- Contributing to the technical development of the MSI architecture;
- Contributing to the development of the MSI specifications, as well as editing and serving as the repository and archive for those specifications;
- Enhancement and maintenance of the Morphware Forum web site and e-mail reflector; and
- Beginning in April 2007, maintenance of the High Productivity Computing systems (HPCS) program web site and e-mail reflector.

As various participants concluded their PCA and xPCA projects, the Morphware Forum also concluded its active development of the MSI. The last Morphware Forum meeting was held in December 2006. Georgia Tech continued to refine and update the documents defining the MSI through February 2007. A complete set of MSI documents is included as appendices to this report.

In addition to leading the Morphware Forum and the MSI effort, GT also conducted independent research into programming methods for high performance tiled architectures. The following specific efforts were conducted:

- An analysis of issues in implementing signal processing libraries on tiled architectures, conducted by consultant Randall Judd;
- Modification of the Massachusetts Institute of Technology's (MIT) Streamit streaming language compiler to target the stream virtual machine (SVM) component of the MSI as its backend. GT developed a symmetric multiprocessor (SMP) implementation of the SVM. The modified Streamit/SMP SVM combination achieved nearly the same performance as a "native" Streamit/SMP backend configuration (*i.e.*, not using the SVM as a portability layer);
- Investigation of the applicability of commercial graphics processing units (GPUs) for select signal processing and electromagnetic (EM) codes and benchmarks; and
- Initial development of an implementation of the Vector, Signal, Image Processing Library (VSIPL) for GPUs. Continued development under a subsequent contract has resulted in the GPU VSIPL library, available at gpu-vsimpl.gtri.gatech.edu, which at this writing implements most of the VSIPL Core profile, with some additional extensions.

These independent research activities continued through September 2007. Beginning in October 2007 and continuing to the end of the contract, only activities in support of the Morphware Forum and HPCS web sites and e-mail lists were funded.

The following sections of this report provide additional details on each of these activities.

SECTION 2

THE MORPHWARE FORUM AND THE MORPHWARE STABLE INTERFACE

2.1 The Morphware Stable Interface

At the end of the original PCA program in December 2005, the participants in the Morphware Forum had developed the structure shown in Figure 1, known as the Morphware Stable Interface (MSI), as a model of a software stack for reconfigurable tiled architectures. The MSI utilizes a metadata description of a target platform (the “machine model”) and a two-level compilation process to create high performance yet portable application codes for PCA machines.

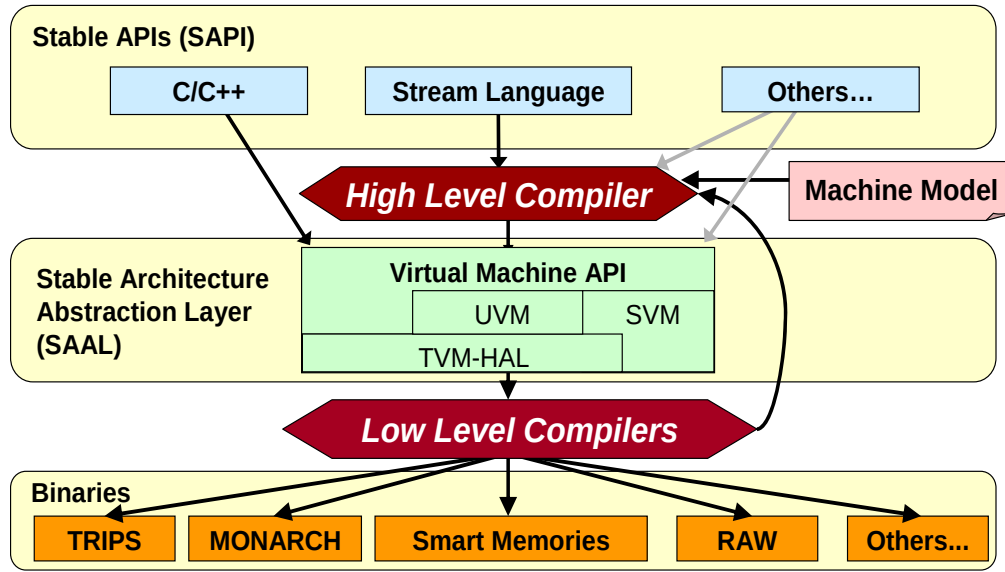


Figure 1. The current Morphware Stable Interface (MSI) architecture.

A full discussion of the elements of the MSI and the reasoning that led to this architecture is given in [2]. Specification documents for each of the major elements are included as appendices to this report.

2.2 The Morphware Forum

2.2.1 Background

The Morphware Forum (MF) was a joint activity of the direct contractual participants in the PCA program, as well as other invited developers and users of embedded computing hardware, software, and application technology. The Forum activity was initiated under the GT’s predecessor PCA contract. The first eighteen meetings were conducted under that contract, beginning in June 2001.

The purpose of the Forum was to define an open, portable software environment for the development of high performance applications on PCA platforms. That design became the MSI

discussed above. The Forum developed a program-wide consensus on the architecture of the MSI, followed by development of detailed, documented, and widely available standards for the application programming interfaces (APIs) and other elements that define the MSI. The primary method for developing the MSI architecture was a series of quarterly Forum meetings at rotating locations.

2.2.2 xPCA Morphware Forum Activities

Three additional meetings of the Forum were conducted under this project, in March, July, and December 2006. Full minutes and presentations from all 21 meetings are archived on the Morphware Forum web site at www.morphware.org.

The 19th meeting (April 2006) concentrated on the following topics:

- A major update of the Threaded Virtual Machine – Hardware Architecture Layer (TVM-HAL, or just HAL);
- Review of the OS and runtime systems of the PCA teams;
- Continued work on the SVM and the Machine Model (MM) specifications; and
- Results regarding performance of the MSI compile chain.

The 20th meeting (July 2006) concentrated on these topics:

- A detailed review of the recent updates to the SVM and MM documents, with the intent of finalizing and releasing version 1.2 of each; and
- A discussion of lessons learned and “what’s next?” in the context of the Morphware effort and the recent workshops on FPGA tools, “EDGE” computing devices, and the DARPA Heterogeneous Embedded Computing Systems (HECS) workshop that immediately preceded this Forum meeting.

Finally, the 21st and final meeting (December 2006) focused on these topics:

- What are the best results to date on the various PCA architectures using the high-level compiler/low-level compiler (HLC./LLC) structure, and what are the one or two most important obstacles to remove to allow better results?
- Continued discussion of lessons learned and how the PCA community could go about defining an “MSI 2” strawman design.

2.2.3 Final Morphware Forum Products

When disbanded at the end of 2006, the Morphware Forum had produced a comprehensive set of background and specification documents describing the elements of the MSI, and a variety of software elements implementing aspects of the MSI. The major documents were as follows:

- PCA Systems and Software Overview
 - o Introduction to Morphware: Software Architecture for PCAs (GT)(2/23/04)

- PCA Specifications
 - o SVM Specification v1.2 (draft 5) (January 2007)
 - o Machine Model v1.2 (draft 4) (January 2007)
 - o TVM-HAL Specification version 1.0 (draft) (Stanford) (May 30, 2006)
 - o PCA User-Level VM (UVM) Interface Specification 0.9a (UT Austin) (3/16/2005)
 - o PCA Metadata System (3/2/04)
- PCA Specification Supporting Documents
 - o SVM Issues Document (February 2007)
 - o Run-time Issues Document (May 2006)
 - o Machine Model Issues Document (February 2007)
 - o SVM Design for Cell (Reservoir) (November 2005)
 - o Machine Model Naming and SVM Initialization and Configuration (Reservoir Labs) (March 24, 2005)
- Morph Scenario Documents
 - o Morphing Scenarios for the PCA Integrated Radar Tracker (IRT) (MIT/LL) (6/6/05)
 - o Morph Taxonomy (Georgia Tech / SPAWAR) (2/3/04)
 - o Morphing Scenarios for the GMTI Portion of the PCA Integrated Radar Tracker (MIT/LL) (2/20/04)
- Verification & Validation
 - o Run-time Environment and Design Application for Polymorphous Technology Verification and Validation (Lockheed Martin) (11/19/04)

The major software elements were as follows:

- Machine Models
 - o Raw (MIT, USC/ISI)
 - o TRIPS (UT Austin)
 - o Smart Memories (Stanford)
 - o MONARCH (USC/ISI)

- Tools
 - o MSI Implementations
 - Stanford TVM-HAL Implementation (Feb. 11, 2006)
 - USC/ISI SVM Library for Raw (4/6/05)
 - o Metadata-Related Tools
 - Lockheed-Martin Visual Editor (June 25, 2003)
 - Metadata Authoring Toolkit for Machine Models (Vanderbilt University) (September, 2004)
 - o Stream Language Tools
 - R-Stream 2.1 High Level Compiler (Reservoir, Inc.) (March 2006)
 - Streamit 2.0 Compiler (MIT) (December 12, 2003)
- Benchmark Software and Data
 - o Kernel Benchmark Performance on Raw (MIT/LL) (June 2006)
 - o C-Language Implementation of PCA Integrated Radar-Tracker Application (provided by Lockheed-Martin ATL) (December 2005)
 - o MATLAB Specification of PCA Integrated Radar-Tracker Application (provided by MIT/LL) (11 Feb 2005)
 - Includes the following Design Review Documents:
 - PCA Integrated Radar-Tracker Application (IRT) (6 Feb 2004)
 - GMTI Processing for the PCA IRT (6 Feb 2004)
 - GMTI Narrowband for the Basic PCA IRT (6 Feb 2004)
 - Kinematic Tracking for the PCA IRT (6 Feb 2004)
 - Database files for the feature-aided tracker (30 MB each)
 - MATLAB code
 - o PCA Kernel Benchmarks (provided by MIT/LL)
 - “Generic C” version of the kernel benchmarks (updated 11/15/2005).
 - Report describing the kernel benchmarks (23 Jan 2004)
 - PowerPC G4 kernel benchmark measurements (23 Jan 2004)
 - Archive of kernel benchmark code

- o Java Version of CFAR Kernel Benchmark (provided by Lockheed-Martin ATL) (24 June 2003)
- o Fully Parallelized MATLAB version of CFAR Kernel Benchmark (provided by Lockheed-Martin ATL) (2 July 2003)
- o Mercury SCE Image Scaling Example (21 Oct 2002)
- o PCA Application Examples (provided by MIT/LL) (31 Jul 2001)

All of these documents and software elements, as well as others not included in this list for brevity, are available to government-approved users at www.morphware.org.

2.3 Web site and e-mail

From the beginning of the PCA program and continuing through the xPCA program, GT maintained a web site to support Forum activities and the MSI development at www.morphware.org.¹ GT owns the rights to this domain name until May 2010 and will maintain the site until at least that time. Figure 2 shows the public home page as of April 2009.

The public home page currently provides

- A link to the so-called “PCA 101” report [3], an introduction to the MSI and its major concepts;
- A page of publicly-released MSI standards documents, currently including the 1.0 versions of the Metadata System, Machine Model, and Stream Virtual Machine specifications, along with a web page for providing comments back to the Forum;
- A listing of all of the PCA program projects and, where available, a link to their individual home pages;
- Links to a number of publicly-released briefings and papers describing elements of the PCA program technologies; and
- Links to related standards activities and DARPA programs.

¹ In order to reduce the chance of conflicting uses by other parties, the names <morphware.com>, <morphwareforum.org>, and <morphwareforum.com> were purchased as well, but are not being actively used. The names <{msi-forum,pca-msi}.{org,com}> were controlled by GT at one time but were allowed to expire.



Figure 2. Screen shot of the public home page of the Morphware Forum web site at www.morphware.org as of April 2009

Figure 3 shows a screen shot of the “internal home page,” the non-public main navigation page for approved, registered users seen after successfully logging in. From this point, the site provides the following information:

- A list of points of contact for each PCA project, both for the project as a whole and for the Morphware Forum;
- Meeting information, including minutes of past Morphware Forum and PCA PI meetings as well as travel and agenda information for upcoming Morphware Forum meetings;

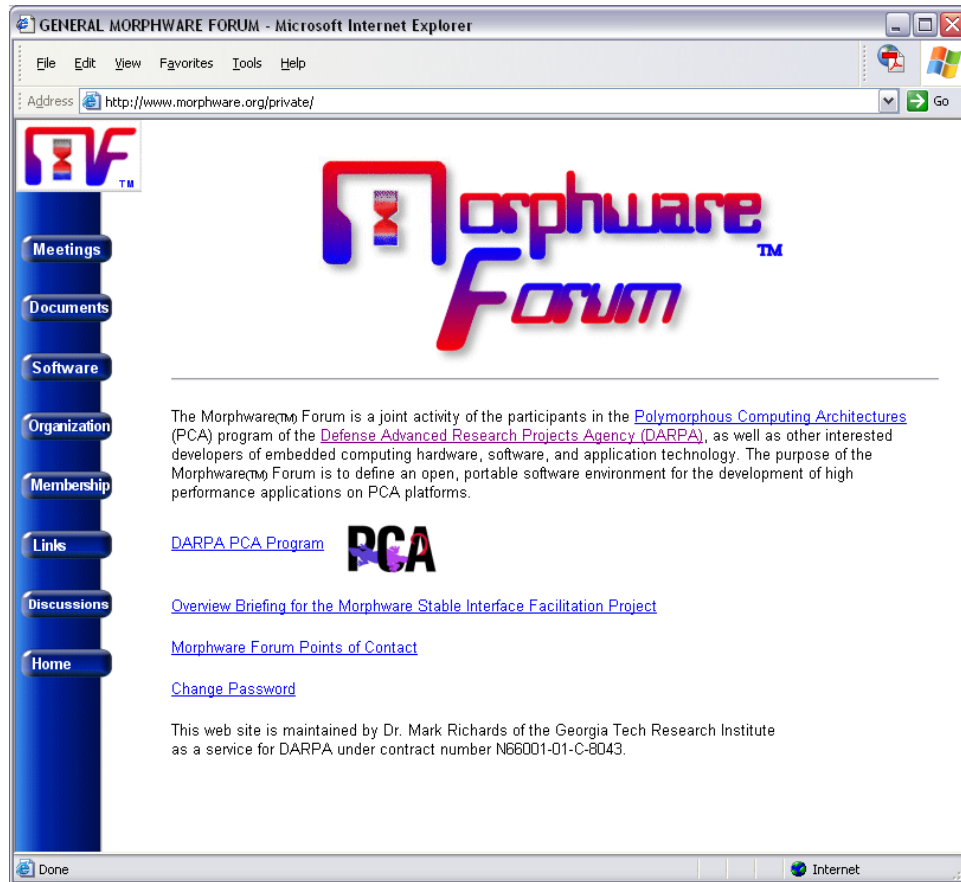


Figure 3. Screen shot of the Morphware Forum web site internal "home page" for project participants.

- Documents, such as "homework assignments" and project responses, benchmark descriptions, selected PCA participant technical reports, and draft Forum products such as the various MSI specifications;
- Application benchmark codes from the Massachusetts Institute of Technology Lincoln Laboratory (MIT/LL) and other PCA participants;
- Software components such as library implementations of the SVM from PCA participants;
- Software tools such as a "high-level compiler", metadata authoring and design exploration tools, and architecture representation generators, again from PCA participants;
- A list of Forum members, both voting and non-voting;
- A page of links, including links to all available PCA home pages, the home pages of other DARPA programs of interest such as Data Intensive Systems (DIS) and High Productivity Computing Systems (HPCS), and the home pages of other embedded software programs of interest such as the Vector, Signal, and Image

Processing Library (VSIPL) and the High Performance Embedded Computing Software Initiative (HPEC-SI); and

- Organizational information on the Morphware Forum, such as membership requirements and voting procedures.

The web site was the primary collaborative tool for the project. However, GT also maintains a Morphware Forum e-mail reflector at <morphware@lists.gatech.edu>. Its primary use was for document dissemination, meeting announcements, and similar planning information.

SECTION 3

RESEARCH INTO PROGRAMMING METHODS FOR HIGH PERFORMANCE TILED ARCHITECTURES

3.1 Signal Processing Libraries for Tiled Architectures

GT retained Mr. Randall Judd as a consultant during the period June 20, 2006 to June 30, 2007. Mr. Judd was one of the original developers of the VSIPL API and participated in the development of the MSI throughout the original PCA program. Mr. Judd was tasked to investigate the implications of tiled PCA architectures for the implementation of signal processing libraries such as VSIPL, and to identify methods for improving the performance of such libraries on tiled architectures. The “Tera-op Reliable and Intelligently adaptable Processing System” (TRIPS) device developed by the University of Texas – Austin was used as a specific example of a tiled chip, principally because a TRIPS compile tool chain was available for experimentation.

The study concluded that work was needed in the following areas to enable development of good signal processing libraries for tiled architectures:

- High level language features to support explicit expression of concurrency;
- Low level compilers corresponding to these language features; and
- Development of algorithms tuned to tiled and multicore architectures, for instance low latency algorithms.

The full analyses and conclusions of this study were reported in a white paper titled “Signal Processing Libraries on Tiled Architectures” and dated June 24, 2007, which is included in this report as Appendix J. The reader is referred to that Appendix for details.

3.2 Alternative High Level Compiler

As noted earlier, the MSI uses a two-level compile structure, comprising a high level compiler (HLC) and a low level compiler (LLC). Most PCA and xPCA work used Reservoir Labs’ R-Stream [17], developed under the PCA program, as the HLC. R-Stream accepted C with certain extensions as the input language, and targeted the PCA stream virtual machine (SVM).

Georgia Tech conducted an experiment to retarget MIT’s Streamit compiler to generate SVM code. The purpose of the experiment was to test the MSI infrastructure portability enabled by the MSI, leverage existing stream language research, examine the performance penalties of the MSI in a stream context, and further vet the SVM. Full results of this experiment are given in the minutes of Morphware Forum meeting #21, December 2006, available at the Morphware Forum web site www.morphware.org

Streamit is a novel language for streaming developed at MIT. It exposes parallelism and communication, is architecture independent, and is modular and composable. Streamit builds up complex functionality by composing elemental filters in a graph. A finite impulse response (FIR) filter is an example of a Streamit filter. Figure 4 is an example of a Streamit graph, in this

case for a notional FM radio, and the corresponding Streamit code. Each parallel channel containing lowpass and highpass filters (LPF and HPF) implements a different radio channel.

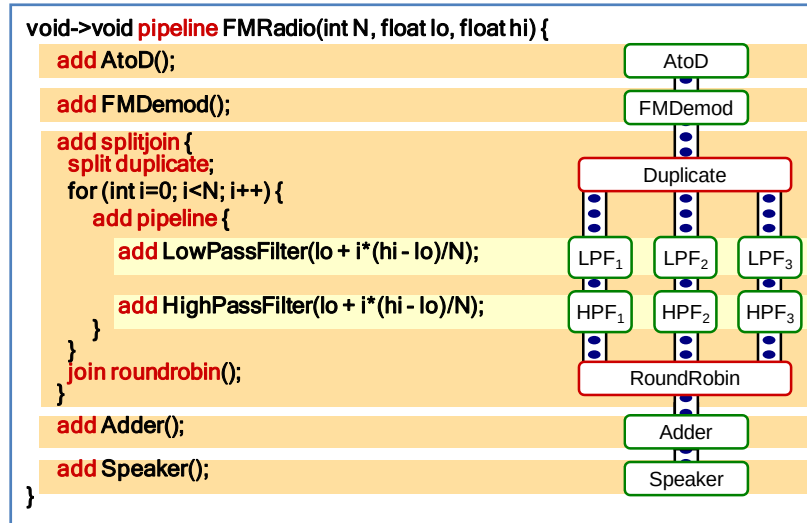


Figure 4. Example Streamit code for a notional FM radio and corresponding implied graph. (Courtesy of William Thies, MIT.)

GT modified the Streamit compiler's code generator to produce SMV-compliant code. The major changes required were

- Creation of a user-defined kernel for each filter
 - Input, output streams
 - Init function
- Implementation of SVM calls for hardware configuration
 - Allocate memory for kernels and streams to execute
 - Configure processors requested at compile-time
- Translation of stream read/write operations, generation of temporaries
- Generation of `svm_kernelRun()`, `svm_kernelWait()` calls in `main()`
- Various miscellaneous translations

The modified system targeted an x86 architecture SMP system with a choice of one, two, or four processors. This work was accomplished by one graduate student in a few weeks, demonstrating that the SVM can be successfully targeted with modest effort.

Reservoir's `svmref` library, which is an SMP SVM system simulator, was modified to control on which processor a Streamit filter runs. This has the disadvantage of interfering with code

portability, but was done as a convenience for this experiment. Some issues were noted with the handling of streams, since R-Stream primarily uses blocks.

Several benchmark applications of increasing complexity were implemented and compared to a “native” implementation, meaning an implementation using the same source code but MIT’s Streamit compiler to directly target the SMP rather than pass through the intermediate SVM layer. Figure 5 is an example of a 4-way parallel FIR filterbank used for many of the experiments. Experimentation with several variants of this benchmark revealed a need to make sure that Streamit optimizations were turned on to avoid stalls, which could occur due to having more kernels than processors, a lack of interleaving, and a lack of filter fusion.

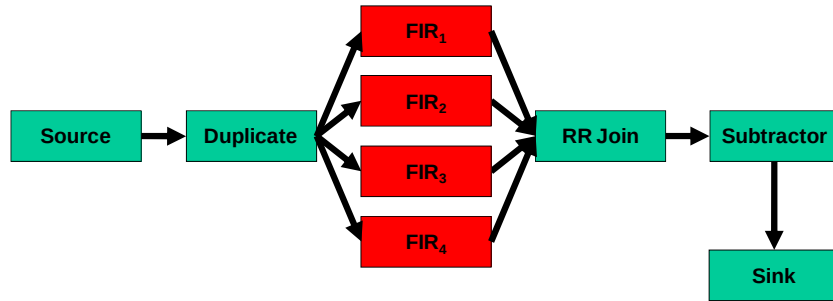


Figure 5. Example FIR filterbank benchmark application.

It was also discovered that a modulo operation and depth checking in an inner loop in the SVM streampeek function were causing significant performance losses. A non-depth checking variant of streampeek was defined and implemented in both the native and SVM versions of the benchmark code. With this change, the performance obtained in Table 1 was obtained. As the number of taps grew large, the performance penalty for using the SVM dropped from about 330% down to about 10%. It is expected that further refinement would reduce the penalties for all filter sizes. The table also shows relatively linear scaling of SVM runtimes with the filter size, for filter sizes of 2048 and larger. The native implementation maintains relatively linear scaling down to the 256-tap filter case.

Table 1. Final comparative performance of native and SVM implementations of the benchmark of Figure 5 with four processors and 500,000 input data samples.

Native:				
filter taps:	256	2048	4096	8192
runtime (secs):	0.3	2.2	4.2	8.5
SVM:				
filter taps:	256	2048	4096	8192
runtime (secs):	1.0	2.8	4.9	9.3
increase relative to native:	3.3x	1.3x	1.2x	1.1x

3.3 Graphics Processing Units for High Performance Embedded Computing

During the course of the xPCA project, the use of graphical processing units (GPUs) for “general purpose” scientific computing, including signal processing, grew rapidly in interest and acceptance in the high performance computing community. This more general application of GPUs beyond their original purpose is referred to as “general purpose GPU” (GPGPU) computing. During this time, the two key GPU vendors produced significant new software development tool chains, the Compute Unified Device Architecture (CUDA, released in Feb. 2007) for Nvidia, and ATI’s Close-to-the-Metal (CTM), subsequently renamed the Stream Software Development Kit (SDK) and released in Dec. 2007. Figure 6 illustrates the growth in GPU single precision floating point performance since 2001. Note that the fastest GPUs now offer one teraflops performance in a single chip!

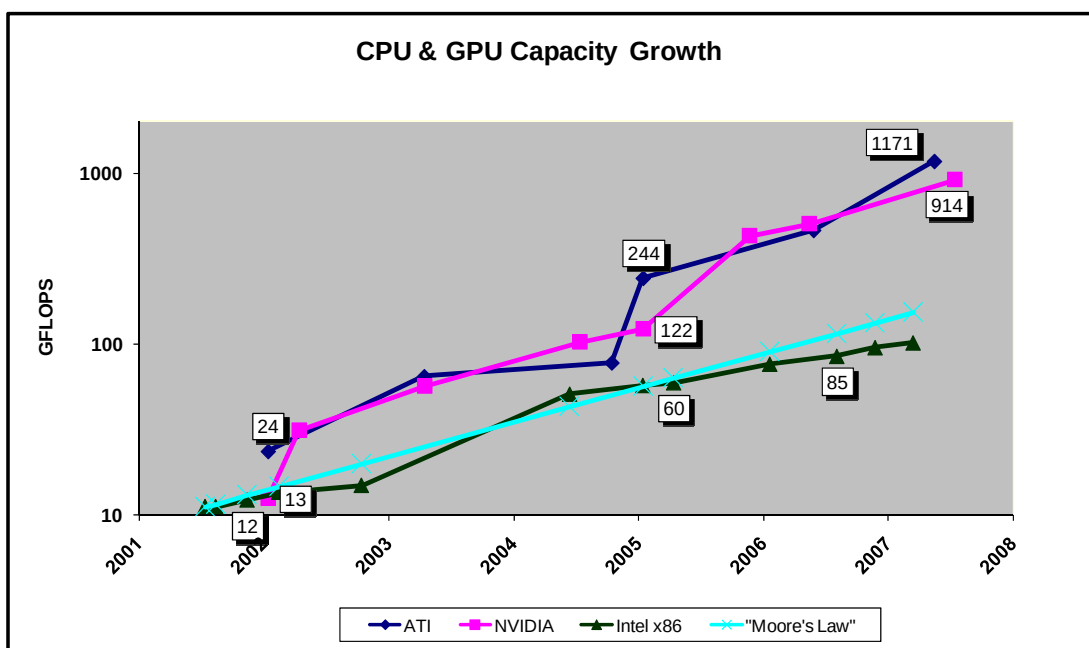


Figure 6. Growth of GPU single precision floating point performance.

GPUs are many-core devices. This fact, coupled with the rapid performance increases, relatively low cost, and improving toolsets led Georgia Tech to investigate their applicability to defense-relevant high performance embedded computing in the context of this project.

Georgia Tech’s work with GPUs under this project has fallen into two general categories: characterization of the performance improvements obtainable with GPUs relative to conventional CPUs, and development of GPU middleware based on VSIPL

3.3.1 Performance Improvement Experiments

Georgia Tech conducted three separate demonstrations of the performance improvements obtainable with GPUs. These were:

- Speedup of a two-dimensional phase unwrapping algorithm for 3D radar imaging;
- Speedup of a radar cross section (RCS) prediction code;
- Implementation of the HPEC Challenge benchmarks.

Each of these is now described in turn.

3.3.1.1 Two-Dimensional Phase Unwrapping

In 2007, GT investigated the application of GPUs to several problems in electromagnetics and sensor signal processing. The majority of the effort was devoted to an implementation of two-dimensional phase unwrapping for 3D interferometric radar imaging (interferometric synthetic aperture radar, or IFSAR). The need for phase unwrapping arises because the third (height) dimension in IFSAR is obtained by scaling the phase difference between corresponding pixels in two complex SAR images of a scene. However, the phase difference measured by a radar is highly ambiguous (“wrapped”); that is, relatively small changes in height produce changes in the pixel-to-pixel phase difference that exceed 2π radians. Accurate height determination requires disambiguation (“unwrapping”) of the wrapped phase. 2D phase unwrapping is a critical and time-consuming step in IFSAR. There are several classes of algorithms typically used, including path-following, least-squares, and linear programming methods [4]. Figure 7 illustrates an arbitrary, artificially generated 2D phase function on the left that varies over a total range of 63 radians, and its wrapped equivalent, confined to the range of $[-\pi, \pi]$, on the right.

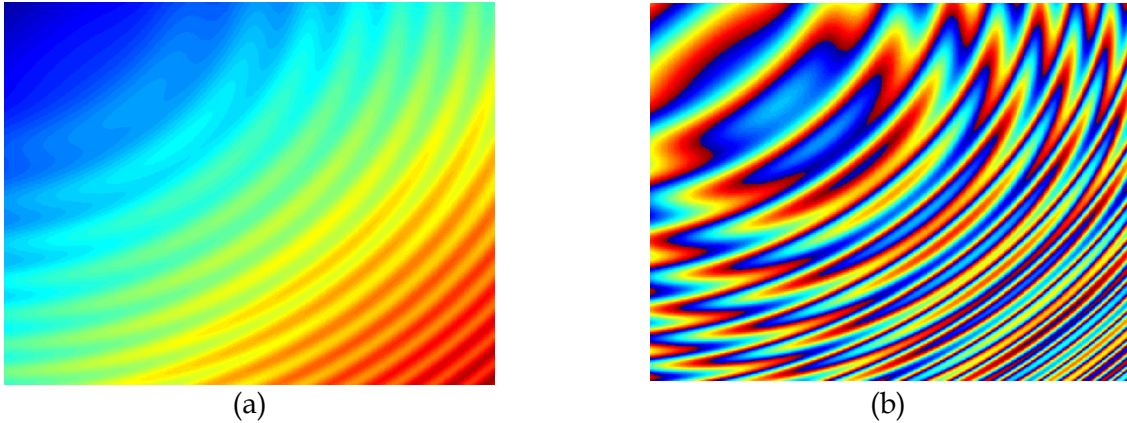


Figure 7. Wrapped (a) and unwrapped (b) two-dimensional phase functions.

One of the key issues in effective use of GPUs is using algorithms that are well-matched to the GPUs architectural constraints. This work was done with Nvidia GPUs but pre-dated the release of the CUDA development environment; instead, Nvidia’s Cg (“C for Graphics”) language was used. Because of the structure of the GPU pipeline, its use for efficient general purpose computation is confined to algorithms that perform a very similar operation on a number of data points in an array (*texture*). The process of applying the mathematical operations to each data point (pixel in the texture) is called a *rendering* pass. Other output patterns are possible but require additional time or rendering passes, reducing the overall performance improvements gained by using the GPU.

For this research, a weighted least square method was adopted. While normally implemented using fast transforms such as the discrete cosine transform, that implementation is not well-suited to the GPU. Thus, a classical Gauss-Seidel or Jacobi solver was used, which iterates the equation of the form

$$\phi_{m,n} = \frac{U_{m,n}\phi_{m+1,n} + U_{m-1,n}\phi_{m-1,n} + V_{m,n}\phi_{m,n+1} + V_{m,n-1}\phi_{m,n-1} + \rho_{m,n}}{U_{m,n} + U_{m-1,n} + V_{m,n} + V_{m,n-1}} \quad (1)$$

where $\phi_{m,n}$ is the phase function, $U_{m,n}$ and $V_{m,n}$ are weight functions typically related to local signal-to-noise ratios, and $\rho_{m,n}$ is a “driving” function derived from the input wrapped phase values

The iteration is considered to have converged when the change in the mean-square difference in unwrapped phase between iterations falls below a pre-set threshold. This was combined with a multigridging technique to further accelerate the algorithm. Full details of the implementation are given in [5]. Figure 8 and Figure 9 show two metrics of the speedup obtained. The computer used was a dedicated AMD Athlon64 PC using Windows XP with an NVIDIA GeForce 8800GTX video card having 768 MB video RAM and 1 GB of system RAM. Figure 8 shows the wall clock time for various phase image sizes in MATLAB on the host, in compiled C on the host (“CPU”), and on the GPU. The GPU obtained speedups relative to the CPU of 5.3x on the 256x256 images, rising to 34.5x on the 2048x2048 images.

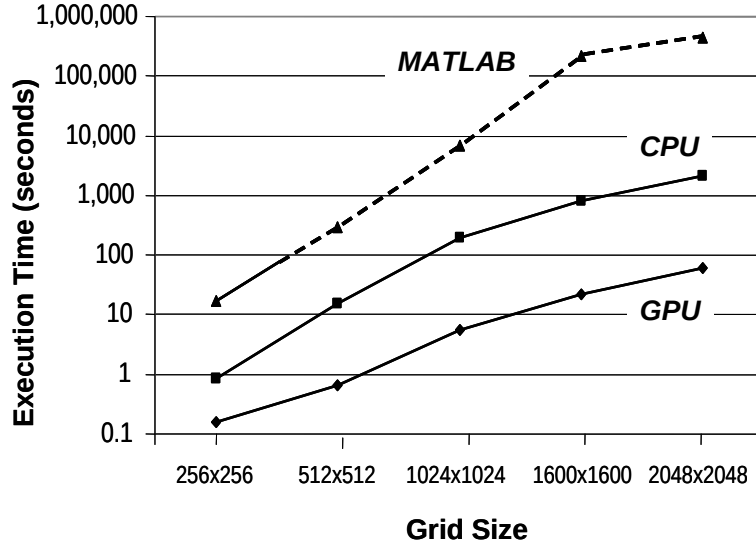


Figure 8. Runtime for the 2D phase unwrapping algorithm on a CPU, a GPU, and in MATLAB on a CPU.

Figure 9 compares the implementations in terms of the number of “point updates” per second. A point update performs the operation in Eq. (1) at one grid point during one iteration; this

metric takes into account the varying grid sizes during the multigrid algorithm's runtime and becomes nearly constant once the grid is large enough to saturate all caches on a given architecture (this is not the case for a software-based environment like MATLAB which degrades further with grid size). This metric gives a more general sense of the relative computational advantage of the GPU as it might be applied to other radar signal processing algorithms. These per-point speedups are the fundamental reason for the advantage of the GPU implementation of the phase unwrapping algorithm.

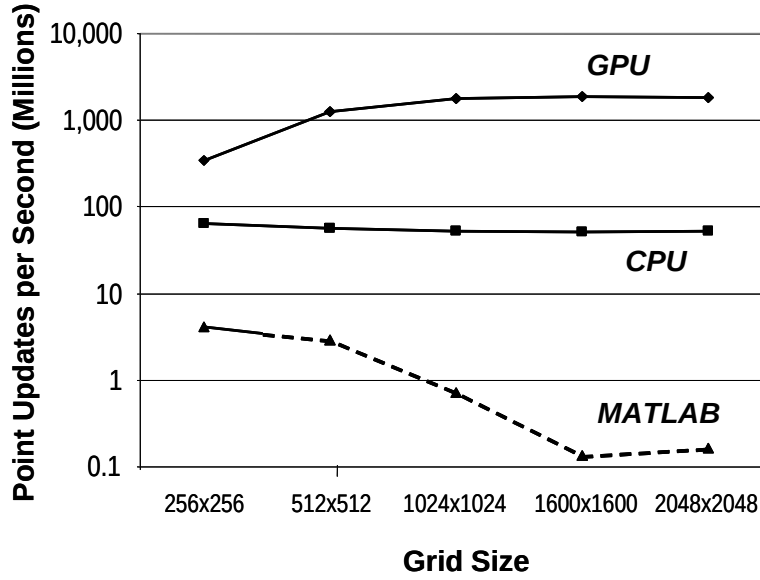


Figure 9. Point updates per second for the 2D phase unwrapping algorithm on a CPU, a GPU, and in MATLAB on a CPU.

3.3.1.2 RCS Prediction

A second experiment considered porting of an open-source ray-tracing radar cross-section prediction (RCS) tool to the GPU. The prototype code being used is RaiderTracer, developed by Prof. Brian Rigling of Wright State University [6]. Figure 10 illustrates two different RCS predictions for a simple modeled airplane shape at X band (10 GHz). Prof. Rigling provided a MATLAB source code to serve as a reference.

GT analyzed the algorithm structure and developed two alternate architectures believed to be well-suited to GPU implementation. GT then developed a C version of the MATLAB code for one architecture, including restructuring to enable more efficient mapping to a GPU, to serve as the reference for a GPU-accelerated version.

The newly-released Nvidia CUDA toolset was used to begin development of a GPU version of RaiderTracer. This work was not completed because the student developing the program left the project. However, at the time of cessation of work, speedups ranging from 6-12x over the MATLAB version of the RaiderTracer code had been achieved, depending on the machine, number of frequency samples, and number of facets. This preliminary result suggests that the GPU may be useful in EM prediction codes, which is useful in and of itself, but also could be helpful in certain automatic target recognition (ATR) applications.

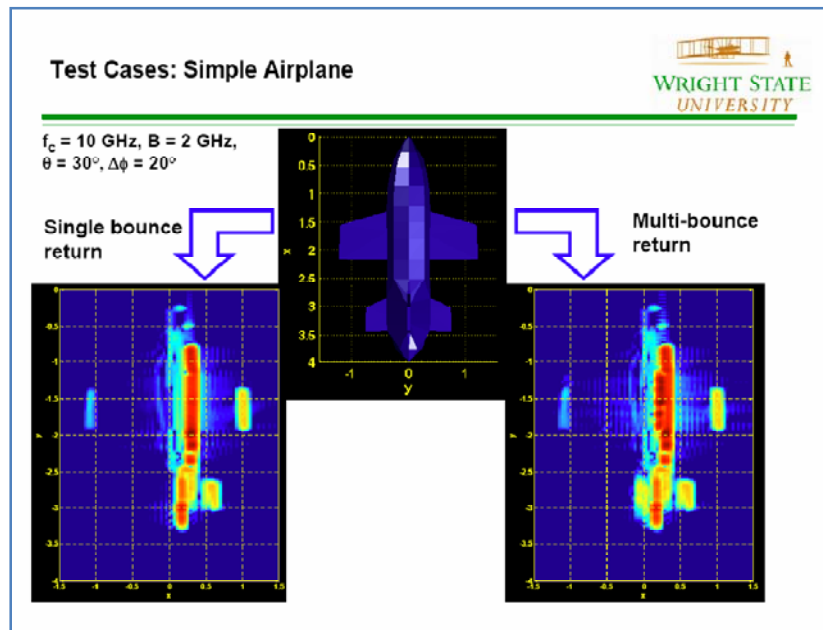


Figure 10. Illustration of RaiderTracer RCS prediction for a simple modeled airplane.
(Image courtesy of Brian Rigling, Wright State University.)

3.3.1.3 HPEC Challenge Implementation

The HPEC Challenge benchmark suite [7] consists of a set of single-processor kernel benchmarks and a multiprocessor application benchmark. The kernel benchmarks address important operations across a broad range of DoD signal and image processing applications. The application benchmark implements a scalable synthetic aperture radar (SAR) application that is representative of one of the most common functions in DoD surveillance systems. The nine kernel benchmarks are

1. Time-Domain Finite Impulse Response Filter Bank - TDFIR
2. Frequency-Domain Finite Impulse Response Filter Bank - FDFIR
3. QR Factorization - QR
4. Singular Value Decomposition - SVD
5. Constant False-Alarm Rate Detection - CFAR
6. Pattern Matching - PM
7. Graph Optimization via Genetic Algorithm - GA
8. Database Operations - DB
9. Corner Turn Benchmark - CT

GT implemented seven of these kernel benchmarks in the Nvidia CUDA 1.1 environment on a GeForce 8800 GTX GPU board with a Intel Core2 Q6600 2.4 GHz host. The seven kernels implemented were the time-domain FIR filter bank, frequency-domain FIR filter bank, QR factorization, constant false-alarm rate detection, pattern matching, graph optimization via genetic algorithms, and corner turn. Details are given in [8].

The HPEC Challenge specifies several performance metrics. Latency, $L_1(k; d_i)$, is the total time required to perform one kernel k for data set size d_i . The HPEC Challenge specifies that input data sets should reside initially in system memory, and that latency measurements should include time required to transfer data to coprocessor memory spaces such as the GPU's global memory. Because GPUs have enough on-board memory to allow many real-world applications to perform several operations consecutively before transmitting final results to system memory, GT also provided a strict kernel latency measurement denoted $L'_1(k; d_i)$ that excludes time spent transferring data between GPU and system memory. GT then defines speedup in terms of the strict kernel latency $L'_1(k; d_i)$.

Table 1 shows the results obtained for the seven kernel benchmarks attempted. Speedups range from a very good 151x for the time domain FIR on data set 1, to 46.8x for data set 3 in the CFAR algorithm, to a rather poor 1.5x for data set 2 in the QR factorization. The better results occur with benchmarks having straightforward parallelization schemes that result in high speedup without significant optimization effort. For example, the time-domain FIR filtering benchmark assigns one block per filter bank and performs convolution with an unrolled looping structure and data pipeline through shared memory. CFAR is embarrassingly parallel and specifies a large data cube. Each Doppler bin may be processed independently of the others, and overall runtime is bounded by global memory bandwidth.

The QR factorization benchmark specifies a specific algorithm, the Fast Givens procedure [9]. While Fast Givens is well-suited to CPU architectures, it is not necessarily the best algorithm for performing QR factorization on GPUs. Fast Givens is designed to reduce the number of square roots performed. Because square root is a computationally intensive operation, this method is preferable to the Givens method of QR factorization for CPU implementations if accuracy requirements are relaxed. GPUs, however, are capable of performing many square root calculations in parallel with relatively low latency. QR factorization with Givens rotations is likely to be faster than the Fast Givens algorithm on GPUs. Moreover, the Givens rotation method is typically more accurate than Fast Givens implemented for the same architecture and should be selected for future GPU implementations if the implementer is free to choose the underlying algorithm.

Table 2. HPEC challenge Performance Results in msec.

Benchmark	Data set	$L_1(k, d_i)$	$L'_1(k, d_i)$	Speedup
Corner Turn	Set 1	2.49	0.30	8.32
	Set 2	28.2	4.60	11.4
TD FIR	Set 1	3.92	2.54	151
	Set 2	0.76	0.09	22.2
FD FIR	Set 1	9.02	3.25	19.7
	Set 2	2.0	0.26	11.5
Pattern Matching	Set 1	2.00	0.24	12.7
	Set 2	3.99	1.65	23.1
CFAR	Set 1	2.43	0.29	2.3
	Set 2	56.6	3.5	166
	Set 3	19.1	3.4	46.8
	Set 4	10.5	2.7	25.6
Genetic Algorithms	Set 1	2.12	0.5	15.6
	Set 2	13.6	11.7	33.3
	Set 3	2.65	1.0	21.9
	Set 4	6.4	4.1	23.7
QR	Set 1	23.5	20.3	4.6
	Set 2	6.54	4.5	1.5
	Set 3	3.91	1.8	5.6

3.3.2 GPU Middleware

Georgia Tech first investigated applications of GPUs to tiled architectures in 2005 when, in collaboration with Reservoir Labs, an implementation of the SVM for GPUs was prototyped [10]. This work was performed under the predecessor PCA program contract and is not further described here. Under this project, GT concentrated on topics related to the use of the Vector, Signal, Image Processing Library (VSIPL) with GPUs.

3.3.2.1 VSIPL and VSIPL++

VSIPL is a portable API for implementing high-performance signal processing applications while retaining platform independence. The API specification document, currently at version 1.3, is available at the VSIPL web site [11]. VSIPL supports memory abstractions for utilizing coprocessors with disjoint memory spaces. A signal processing application may structure input data in a *block*, *admit* it once to VSIPL's memory management, perform computations on that data, and *release* only the block containing the final result. Intermediate results are not transferred between system and GPU memory, avoiding unnecessary latencies and communications overhead. This capability distinguishes VSIPL from other signal processing libraries that permit random access to data.

The initial cost to an implementer of developing a complete VSIPL library can be substantial. Consequently, VSIPL defines two *profiles*, VSIPL Core and VSIPL Core Lite. Each defines a reduced, but very useful subset of the full VSIPL functionality that can be implemented at lower

cost. Documents defining the exact contents of each profile are available at [11]. Generally, the Core Lite profile supports single-precision real and complex-valued vectors, and provides the following functionality:

- Support functions for creating, modifying, and destroying blocks and managing associated buffers;
- Element-wise mathematical operations;
- Inner products and scalar operations;
- Extrema searching;
- FIR filtering;
- Out-of-place Fast Fourier Transform (FFT); and
- Histogram and portable random number generator.

The Core profile adds support for single-precision real and complex-valued matrices, and provides the following additional functionality:

- Matrix element-wise operations;
- Window creation;
- Convolution and correlation;
- Elementary linear algebraic operations (transpose, matrix-vector products and sum variations, *etc.*); and
- Linear equation solvers.

In recent years, active development of the VSIPL specification has focused on defining a C++ binding known as VSIPL++. This binding provides parallel and object-oriented extensions to VSIPL with the goal of unifying computation and communication in a single high performance, productive, and portable API. Most of the work on extending VSIPL to VSIPL++ has been done by the VSIPL Forum operating in conjunction with the High Performance Embedded Computing Software Initiative (HPEC-SI) [12]. An API specification has been developed for VSIPL++ version 1.02, with a version 1.03 draft pending [12].

3.3.2.2 VSIPL++ for GPUs

VSIPL++ is well suited to exploit the capabilities of GPUs. It is designed for signal processing, image processing, and linear algebra tasks that typically have similar levels of inherent parallelism to graphics rendering. It includes explicit mechanisms for managing separate memory spaces, and presents a data and storage abstraction that maps well to both its target application space and the available data storage mechanisms on GPUs. The elements of VSIPL++ that are specific to the C++ expansion and distinct from the original VSIPL are also particularly well suited to GPUs. For example, GPUs impose relatively high per-loop and per-

data-access latency costs compared to general purpose processors. VSIPL++ includes provisions for expression-level specialization and loop fusion, which significantly mitigate these costs.

Georgia Tech implemented a portion of VSIPL++ for the GPU in a pre-CUDA development environment [13]. GPU VSIPL++ was implemented in a layered approach, using a modified version of the reference implementation of VSIPL++ created by CodeSourcery, LLC, and a GPU-based implementation of portions of VSIPL (GPU VSIPL). GPU VSIPL blocks were implemented as OpenGL textures. Details of the implementation are given in [13].

Several simple GPU VSIPL++ functions were benchmarked and compared to CPU-only execution of the reference VSIPL++ implementation, using the Tactical Advanced Signal Processing (TASP) VSIPL Core Plus reference implementation as a backend [11]. The host computer was an Athlon64 2.4GHz 1GB RAM desktop PC, while the GPUS utilized were an ATI Radeon 1900XT and an Nvidia 7800GTX single/SLI.

For small vector sizes, the per-operation cost dominates, and the CPU-only VSIPL++ outperforms the GPU VSIPL++. However, for larger vector sizes, GPU VSIPL++ allows significant speedups over the CPU-only implementation. Figure 11 shows the speedup obtained by each GPU, relative to the CPU-based VSIPL++, on the FIR operation. Asymptotic speedups of 15-20x were delivered for simple vector operations, and as high as 79x for FIR operations on larger vectors.

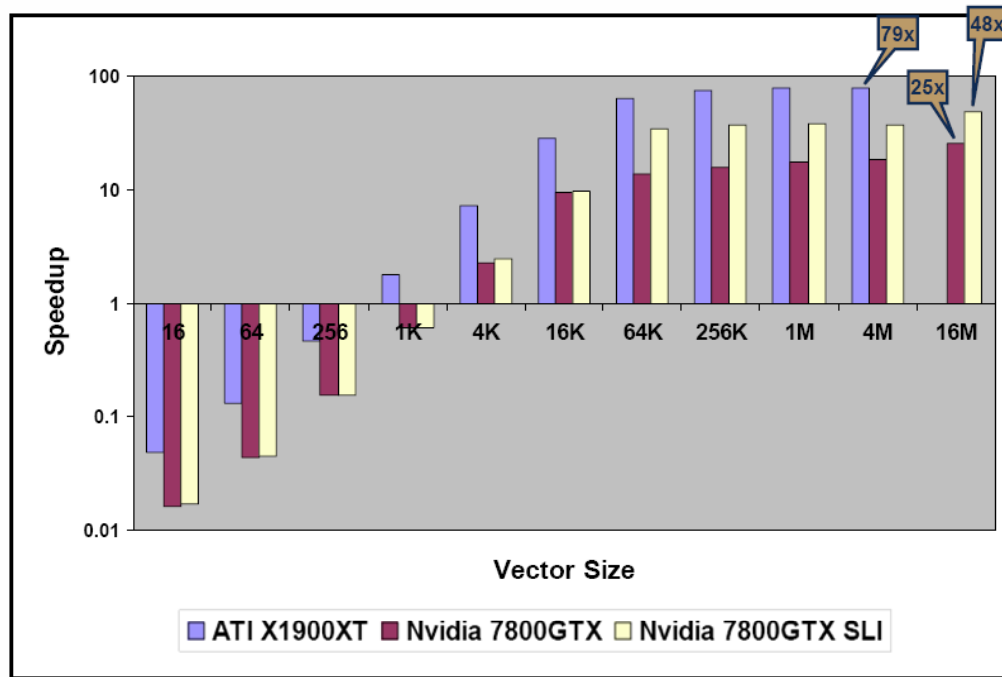


Figure 11. Speedup of GPU VSIPL++ vs. CPU VSIPL++ for the FIR operation.

3.3.2.3 GPU VSIPL Library

With Nvidia's release of the CUDA environment in 2007, Georgia Tech began development of a more complete and high performance implementation of C VSIPL for Nvidia GPUs. That effort was begun under this contract, and was continued under a subsequent contract.² This work has culminated in the GPU VSIPL Library [14].

GPU VSIPL is an implementation of VSIPL that, at this writing, now includes most of the VSIPL Core functionality (not just Core Lite as in [14]), with the exception of some of the linear equation solvers and random number functions, but with the addition of a large number of matrix arithmetic operations not included in VSIPL Core profile. It passes all compliance tests of the VSIPL Test Suite [11]. GPU VSIPL was implemented with NVIDIA's CUDA 2.0 programming language and C++ compiled with Visual Studio 2005, and will run on all CUDA 2.0-compatible GPUs. Details of the implementation techniques for the library are given in [14]

GPU VSIPL is freely distributed as a static binary library with C linkage at the GPU VSIPL web site [15]. Georgia Tech is investigating licensing options for the GPU VSIPL source code.

To establish the performance of GPU VSIPL, an application was written to determine the average runtime of VSIPL functions. By linking the application with first the TASP VSIPL Core Plus library [11] and then the GPU VSIPL library, performance results from each were obtained. The test platform was an Intel Core2 Q6600 at 2.4 GHz running Windows XP Professional with 2 GB of system memory. The GPU was an NVIDIA GeForce 8800 GTX with 768 MB of video memory. GPU VSIPL demonstrated speedups of 20x - 40x for a variety of binary vector operations compared to the reference implementation, as seen in Figure 12. Figure 13 illustrates GPU VSIPL's 1D FFT performance as the input length is varied from 64 to 1024K elements. The speedup is greater than one for inputs of 1K or longer, and rises steadily with increasing input vector length.

The GPU VSIPL web site provides a sample radar range-Doppler map calculation application, intended as both a demonstration and a tutorial application for new users of VSIPL and GPU VSIPL. The basics of range-Doppler processing are described in [16]. In brief, range-Doppler mapping is a radar signal processing technique, common in both ground and especially airborne radars, that separates the echo energy of multiple radar targets from one another based on the Doppler shift of the targets, which is related to their velocity, and their time delay, which is related to the distance (range) to the target. It also amplifies the target echoes relative to the system noise, making target detection easier. Figure 14 shows the output of this application.

² "Exascale Computing Study", U.S. Air Force Contract FA8650-07-C-7724.

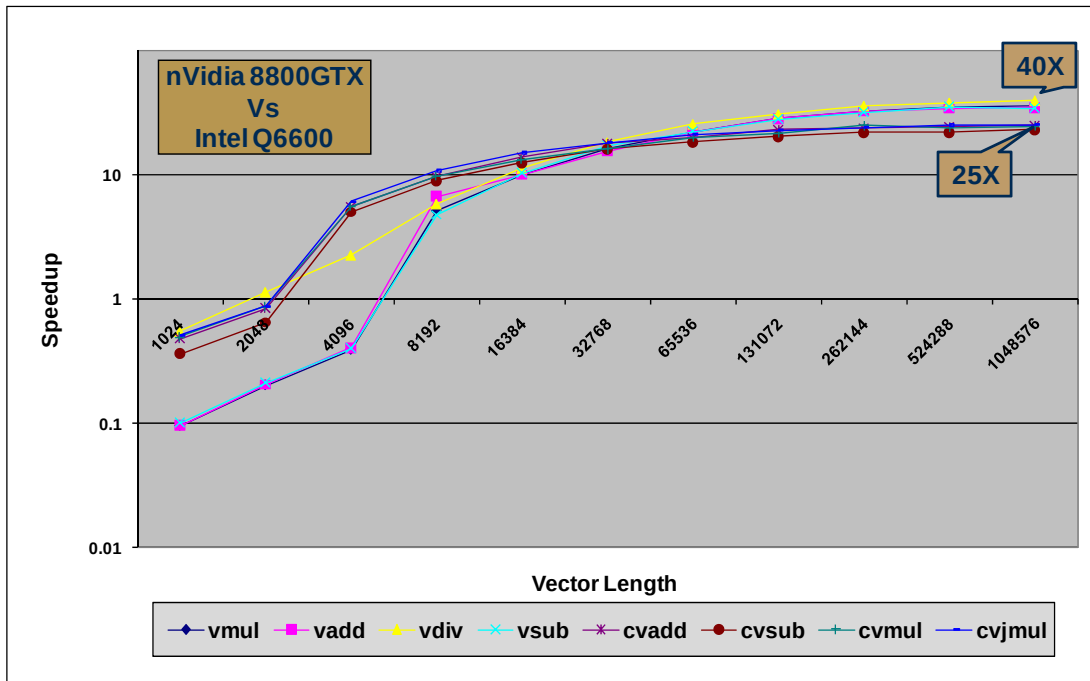


Figure 12. Speedup of GPU VSIPL vs. TASP Reference VSIPL for various binary vector operations.

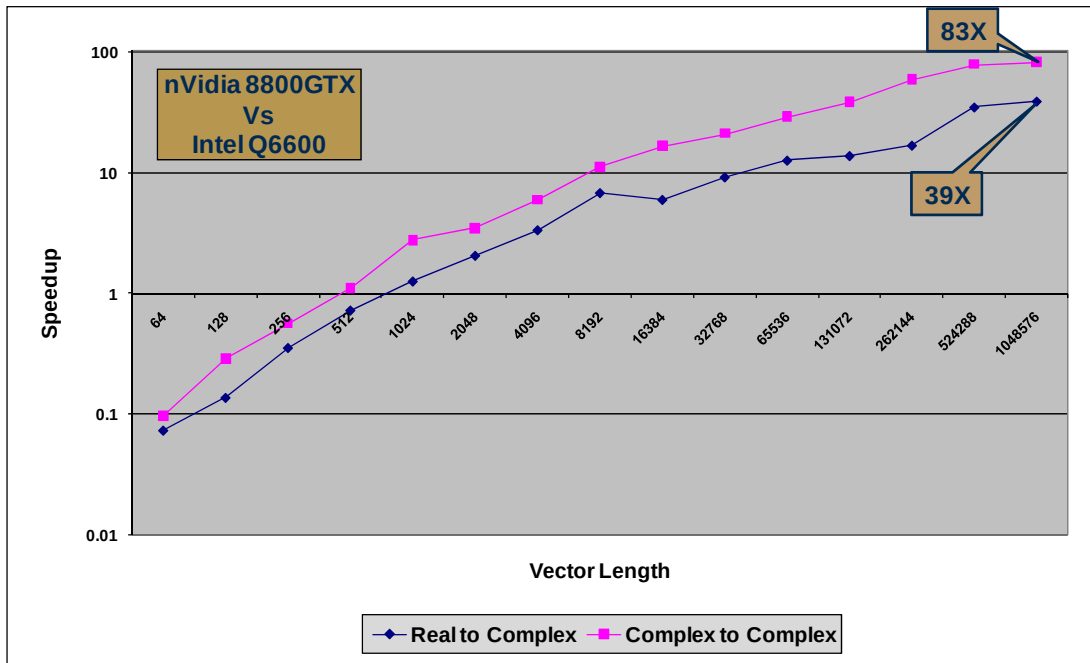


Figure 13. Speedup of GPU VSIPL vs. TASP Reference VSIPL for 1D FFTs.

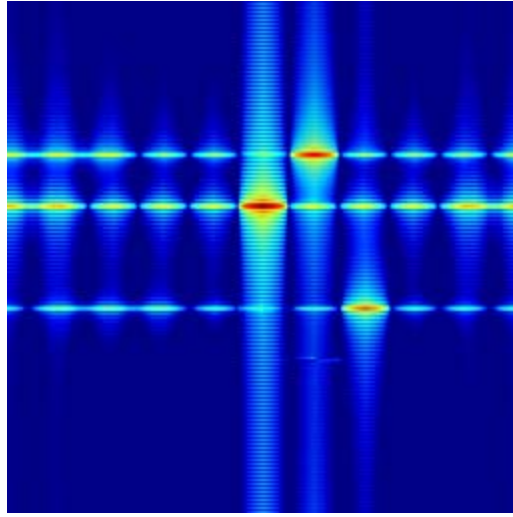


Figure 14. Output of GPU VSIPL range-Doppler map application showing three target with sidelobes in both range (vertical) and Doppler (horizontal) dimensions.

Computationally, a 2D matrix of complex-valued floating point data is the input to the process. Independent 1D FIR filtering operations with complex coefficients are performed on each column of the data. Independent 1D FFTs are then performed on each of the rows. Finally, the magnitude of the result is calculated, usually on a decibel scale, and displayed. Variations involve the use of time- or frequency-domain convolution for the FIR operations, and the use (or not) of windows for sidelobe control in both dimensions. Using a GeForce GTX 280 and GPU VSIPL, a speedup of 75x was obtained compared to an implementation using the TASP VSIPL Core Plus library on a 2.83 GHz Intel Core 2 processor.

SECTION 4

INTERNET TOOLS SUPPORT FOR THE HIGH PRODUCTIVITY COMPUTING SYSTEMS PROGRAM

4.1 Background

Beginning in April 2007, GT was tasked to provide “internet tools” support for DARPA’s High Productivity Computing Systems program as part of this project. This support consisted of hosting and maintaining the HPCS website and e-mail reflectors as required. This support continued through December 2008; in fact, after October 2007, the primary research on this project was concluded, and the HPCS support was the only funded activity through the project completion at the end of December, 2008.

4.2 HPCS Web Site

GT obtained ownership of the HPCS web site domain name, www.highproductivity.org. That ownership expires in July 2009. GT then designed and constructed an HPCS web site based on previous experience with the Morphware Forum, VISPL, and other sites, and sponsor and user guidance.

Figure 15 shows a screen shot of the HPCS site as of April 2009. The publicly available portion of the site provides background information and references on the HPCS program and goals, information on past and planned meetings, a list of participants, and links to various pertinent benchmark sites. Information on program-specific (as opposed to public) meetings and documents is password-protected. Analysis of early 2009 server logs indicate that the site is currently receiving approximately 10,000 hits per month.

In anticipation of completion of this project, GT prepared a template of the HPCS web site during the fourth quarter of 2008 and provided it to Dr. Jeffrey Vetter of Oak Ridge National Laboratory. That template, not yet implemented as of this writing, looks substantially similar to the existing site depicted in Figure 15. At this writing, it appears likely that GT will continue to host and maintain the site under new funding and under Dr. Vetter’s direction.



Figure 15. Screen shot of the home page of the High Productivity Computing Systems web site at www.highproductivity.org as of April 2009.

REFERENCES

Note: Several of the references below are denoted as being available at the Morphware Forum web site, www.morphware.org. All such documents are available to members of the Morphware Forum, but may not be available in the public area of the web site. Standards documents are released to the public area when they become sufficiently stable.

- [1] Defense Advanced Research Projects Agency Polymorphous Computing Architectures program web site, www.darpa.mil/ipto/research/pca/index.html.
- [2] "Facilitating Middleware for Polymorphous Computing Architectures", Final Technical Report #3, Contract N66001-01-C-8043, GTRI Project A6594, Georgia Tech Research Institute, December 2005.
- [3] The Morphware Forum, "Introduction to Morphware: Software Architecture for Polymorphous Computing Architectures," version 1.0, February 23, 2004. Available at www.morphware.org.
- [4] M. A. Richards, "A Beginner's Guide to Interferometric SAR Concepts and Signal Processing," IEEE Aerospace and Electronics Systems Magazine, Tutorial Issue IV, vol. 22, no. 9, pt. 2, pp. 5-29, September 2007.
- [5] P. A. Karasev, D.P. Campbell, and M. A. Richards, "Obtaining a 35x Speedup in 2D Phase Unwrapping Using Commodity Graphics Processors", *Proceedings IEEE Radar Conference*, 2007, Boston, MA, pp. 574-578.
- [6] B. Rigling, *RaiderTracer* web site, Wright State University, www.cs.wright.edu/~brigling/RaiderTracer/RaiderTracer.htm.
- [7] The HPEC Challenge Benchmark Suite, www.ll.mit.edu/HPECchallenge.
- [8] A. R. Kerr, D. P. Campbell, and M. A. Richards, "GPU Performance Assessment with the HPEC Challenge", *Proceedings 2008 High Performance Embedded Computing Workshop*, MIT Lincoln Laboratory, September 23-25, 2008. Available at <http://www.ll.mit.edu/HPEC/agendas/proc08/agenda.html>.
- [9] G. H. Golub and C. F. Van Loan. *Matrix Computations*. (Johns Hopkins University Press, 3rd edition, 1996.)
- [10] K. Mackenzie, D. P. Campbell, and P. Szilagyi, "A Streaming Virtual Machine for GPUs", *Proceedings 2005 High Performance Embedded Computing Workshop*, MIT Lincoln Laboratory, September 20-22, 2005. Available at <http://www.ll.mit.edu/HPEC/agendas/proc05/agenda.html>.
- [11] Vector, Signal, Image Processing Library (VSIPL) web site, www.vsipl.org.
- [12] High Performance Embedded Computing Software Initiative (HPEC-SI) web site, www.hpec-si.org/.
- [13] D. P. Campbell, "VSIPL++ Acceleration Using Commodity Graphics Processors", *Proceedings 2006 High Performance Embedded Computing Workshop*, MIT Lincoln Laboratory, September 19-21, 2006. Available at <http://www.ll.mit.edu/HPEC/agendas/proc06/agenda.html>.

- [14] A. R. Kerr, D. P. Campbell, and M. A. Richards, "GPU VSIPL: High-Performance VSIPL Implementation for GPUs", *Proceedings 2008 High Performance Embedded Computing Workshop*, MIT Lincoln Laboratory, September 23-25, 2008. Available at <http://www.ll.mit.edu/HPEC/agendas/proc08/agenda.html>.
- [15] GPU VSIPL web site, gpu-vsimpl.gtri.gatech.edu/.
- [16] M. A. Richards, *Fundamentals of Radar Signal Processing*. (McGraw-Hill, New York, 2005.)
- [17] R. Lethin, A. Leung, B. Meister, P. Szilagyi, N. Vasilache and D. Wohlford, "R-STREAM 3.0 COMPILER", AFRL-RI-RS-TR-2008-160, Final Technical Report, June 2008.

LIST OF ACRONYMS

API	Application Programming Interface
ATR	Automatic Target Recognition
CFAR	Constant False Alarm Rate
CMP	Chip Multiprocessor
CTM	Close-to-the-Metal
CUDA	Compute Unified Device Architecture
DARPA	Defense Advanced Research Projects Agency
DoD	Department of Defense
DSP	Digital Signal Processing
ECE	Electrical and Computer Engineering
EM	Electromagnetic
FIR	Finite Impulse Response
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
FM	Frequency Modulation
GB	Gigabyte
GMTI	Ground Moving Target Indication
GPGPU	General Purpose GPU
GPU	Graphical Processing Unit
GT	Georgia Tech
GTRI	Georgia Tech Research Institute
HAL	Hardware Architecture Layer
HLC	High Level Compiler
HPCS	High Productivity Computing Systems
HPEC	High Performance Embedded Computing
HPEC-SI	High Performance Embedded Computing Software Initiative
HPF	High Pass Filter
IFSAR	Interferometric Synthetic Aperture Radar
IPTO	Information Processing Technology Office
IRT	Integrated Radar-Tracker
LLC	Low Level Compiler

LIST OF ACRONYMS

LPF	Low Pass Filter
MF	Morphware Forum
MIT	Massachusetts Institute of Technology
MIT/LL	Massachusetts Institute of Technology Lincoln Laboratory
MONARCH	Morphable Networked Microarchitecture
MSI	Morphware Stable Interface
PC	Personal Computer
PCA	Polymorphous Computing Architectures
Raw	Raw Architecture Workstation
RCS	Radar Cross Section
SAAL	Stable Architecture Abstraction Layer
SAPI	Stable Application Programming Interface
SAR	Synthetic Aperture Radar
SDK	Software Development Kit
SMP	Symmetric MultiProcessor
SVM	Stream Virtual Machine
TRIPS	Tera-op Reliable and Intelligently adaptable Processing System
TVM	Threaded Virtual Machine
TVM-HAL	Threaded Virtual Machine Hardware Architecture Layer
UVM	User-level Virtual Machine
VM	Virtual Machine
VSIPL	Vector, Signal, Image Processing Library
XML	eXtensible Markup Language
xPCA	Extended PCA program

APPENDIX A

Last Available Version of the Morphware Stable Interface Document

INTRODUCTION TO MORPHWARE: SOFTWARE ARCHITECTURE FOR POLYMORPHOUS COMPUTING ARCHITECTURES

Version 1.0

February 23, 2004

Introduction to Morphware

Software Architecture for Polymorphous Computing Architectures

Georgia Institute of Technology
and
Space and Naval Warfare Systems Center San Diego

Version 1.0
February 23, 2004



©2004 Georgia Tech Research Corporation, all rights reserved.

This material is based in part upon work supported by the U.S. Defense Advanced Research Projects Agency (DARPA) and other agencies of the U.S. Department of Defense (DoD). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or the DoD. THE INFORMATION IN THIS DOCUMENT SHOULD NOT BE CONSTRUED AS A COMMITMENT OF DEVELOPMENT BY ANY OF THE ABOVE PARTIES.

The US Government has a license under these copyrights, and this material may be reproduced by or for the U.S. Government.



ACKNOWLEDGEMENTS

Authors

The primary authors of this document were:

Daniel P. Campbell	Georgia Tech Research Institute
Dennis M. Cottel	Space and Naval Warfare Systems Center San Diego
Randall R. Judd	Space and Naval Warfare Systems Center San Diego
Mark A. Richards	Georgia Institute of Technology

The authors wish to thank the members of the Morphware Forum who reviewed and contributed to this document. The authors also thank the Defense Advanced Research Projects Agency (DARPA) and the US Navy's Space and Naval Warfare Systems Center San Diego for their support of this work.

The Morphware Forum

The Morphware Forum is a joint activity of the participants in DARPA's Polymorphous Computing Architectures (PCA) program, as well as other interested developers of embedded computing hardware, software, and application technology. The purpose of the Morphware Forum is to define an open, portable software environment for the development of high performance applications on PCA platforms. Morphware Forum products and information are available at www.morphware.org.

The following organizations are voting members of the Morphware Forum at this writing:

- o Defense Advanced Research Projects Agency
- o Georgia Institute of Technology
- o Lockheed Martin Advanced Technology Laboratory
- o Mercury Computing
- o Massachusetts Institute of Technology
- o MIT Lincoln Laboratory
- o MPI Software Technology, Inc.
- o Protean Devices, Inc.
- o Reservoir Labs, Inc.
- o Stanford University
- o University of Texas, Austin
- o University of Southern California Information Sciences Institute

Additional contributing organizations include:

- o Air Force Research Laboratory
- o Applied Photonics
- o BAE
- o Brigham Young University
- o California Institute of Technology
- o George Mason University
- o IBM Austin Research Laboratory
- o IBM T. J. Watson Research Center
- o Lockheed Martin Aerospace
- o Lockheed Martin NE&SS
- o Los Alamos National Laboratory
- o Mississippi State University
- o North Carolina State University
- o Northrop Grumman
- o Raytheon
- o Sandia National Laboratory
- o South West Research Institute
- o Space and Naval Warfare Systems Center San Diego
- o University of California, Berkeley
- o University of California, Irvine
- o University of Illinois at Urbana-Champaign
- o University of Maryland
- o University of Pennsylvania
- o Vanderbilt University
- o Vermont University
- o VLSI Photonics

DOCUMENT CHANGE HISTORY

This is the initial public release.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	i
Authors	i
The Morphware Forum	i
DOCUMENT CHANGE HISTORY	iii
TABLE OF CONTENTS	iv
TABLE OF ACRONYMS	v
INTRODUCTION	1
Who Should Read this Document?	1
Content of this Document	1
THE PCA PROGRAM	2
PCA SYSTEMS AND APPLICATIONS	3
Introduction to PCA Systems	3
Morphing	5
Example of a PCA Application	5
PCA LANGUAGES AND COMPILATION	9
Streaming Algorithms and Languages	9
Two Stage Compilation	10
ELEMENTS OF THE MSI	12
The Stable Architecture Abstraction Layer	13
Stream Virtual Machine Overview	13
Thread Virtual Machine Overview	14
Metadata	15
Machine Model Metadata Context	15
CREATING AN APPLICATION	17
FUTURE DEVELOPMENTS	19
REFERENCES	20
APPENDIX: STREAM LANGUAGE EXAMPLES	22
Brook Example	22
StreamIt Example	24

TABLE OF ACRONYMS

API	Application Programming Interface
ALU	Arithmetic-Logic Unit
ASIC	Application-Specific Integrated Circuit
CAT	Classification-Aided Tracking
DARPA	Defense Advanced Research Projects Agency
DSP	Digital Signal Processor
FAT	Feature-Aided Tracking
FIFO	First In, First Out
FPU	Floating Point Unit
GMTI	Ground Moving Target Indication
HAL	Hardware Abstraction Layer
HLC	High-Level Compiler
I/O	Input/Output
IRT	Integrated Radar-Tracker
ISR	Intelligence, Surveillance, and Reconnaissance
LLC	Low-Level Compiler
MIT	Massachusetts Institute of Technology
MIT/LL	Massachusetts Institute of Technology Lincoln Laboratory
MSI	Morphware Stable Interface
OS	Operating System
PCA	Polymorphous Computing Architecture
RAM	Random Access Memory
Raw	Raw Architecture Workstation
SAAL	Stable Architecture Abstraction Layer
SAPI	Stable Application Programming Interface
SAT	Signature-Aided Tracking
SVM	Stream Virtual Machine
SWEPT	Size, Weight, Energy, Performance, and Time
TFLOPS	TeraFLOPS (Tera-Floating Point Operations per Second)
TRIPS	Tera-op Reliable and Intelligently Adaptive Processing System
TVM	Thread Virtual Machine
UVM	User Virtual Machine
V&V	Validation and Verification

INTRODUCTION

Who Should Read this Document?

This document is a product of the Polymorphous Computing Architectures (PCA) program of the Defense Advanced Research Projects Agency (DARPA). It is intended for application programmers, system engineers, and managers as a high-level introduction to PCA [1] systems in general, and software development for PCA systems in particular¹. PCA platforms offer a significant increase in capability and flexibility over traditional computing platforms. However, the systems are also significantly more complex, thus presenting new problems requiring new solutions for application development.

The PCA program established the Morphware Forum [2] to develop and establish a portable application development methodology for PCA systems. The Morphware Forum is a joint activity of the participants in the DARPA PCA program, as well as other interested developers of embedded computing hardware, software, and application technology. A methodology that includes new source languages, a new application development process, and a framework for expressing system and application metadata has been developed to allow application developers, hardware developers, and build-tool developers to deploy high performance, flexible PCA-based computing systems. Those elements of this methodology defined by the Morphware Forum are termed the Morphware Stable Interface (MSI) or simply *morphware*. The defined software, metadata, and programming standards of the MSI allow the developer to abstract diverse PCA hardware targets from the application software requirements.

Content of this Document

This document gives an overview of the elements and structure of the MSI. After reading this document an application programmer will understand the basic concepts used in developing PCA applications. Also, as an example of the type of defense applications that will benefit from a PCA system, a benchmark developed under the PCA program, the Integrated Radar Tracker, is described.

Detailed descriptions of the concepts introduced in this document are found in other MSI documents. At the time of this writing, the MSI is still under development, so the framework described here is subject to change.

¹ The acronym “PCA” is used in this document to refer variously to PCA microprocessor devices, computing systems based on these devices, and the DARPA research program. The specific meaning of each usage should be clear from the context.

THE PCA PROGRAM

DARPA's Polymorphous Computing Architectures program [1] is developing a revolutionary approach to implementing embedded computing systems that support reactive, multi-mission, multi-sensor, and in-flight retargetable missions, and that reduce the time needed for payload adaptation, optimization, and validation from years to days to minutes. The PCA program breaks the current development approach of "hardware first and software last" point solutions by moving beyond conventional computer hardware and software to flexible, "polymorphous" computing systems. A polymorphous computing system (chips, processing architecture, memory, networks, and software) will "morph" (take on or pass through varying forms or implementations) to best fit changing mission requirements, sensor configurations, and operational constraints during a mission, for changing operational scenarios, or over the lifetime of a deployed platform.

In current practice, a processor is typically selected to perform a particular class of processing, such as real-time data signal processing or, at the other extreme of the processing spectrum, cognitive reasoning. A corresponding spectrum of domain-specific processors, such as specialized data-intensive signal processors, general digital signal processors (DSP), general-purpose microprocessors, and the server-class devices, is then required to perform the complete range of mission processing. Through their ability to reconfigure resources and architectural elements to implement a broad range of architectural implementations, PCA systems can span this processing continuum with a single class of device. This is accomplished by dynamically reorganizing the PCA device's processing elements or micro-architectural components to provide an optimized architectural implementation for each specific set of mission or system requirements.

To achieve this capability, the PCA program is implementing a family of novel malleable micro-architecture processing elements to include compute cores, caches, memory structures, data paths, network interfaces, network fabrics with incremental instructions, OS, and network protocols. To support the use of these polymorphous computing systems, the program is creating a model-based software framework for reactive monitoring, optimization, modeling, resource negotiation and allocation, regeneration, and verification. A set of measurement metrics are being developed to support processing system design and optimization; these include size, weight, energy, performance, and time (SWEPT). Finally, the PCA program is establishing benchmark and standards groups that are creating community standards to enable broad application and commercial support of PCA program developments. The Morphware Stable Interface described in this document is a product of this latter activity.

PCA SYSTEMS AND APPLICATIONS

Introduction to PCA Systems

PCA devices are programmable computing devices that possess a significantly greater degree of reconfigurability than traditional general purpose computing devices. PCA devices are designed to deliver performance approaching that of Application Specific Integrated Circuits (ASICs) on a wide variety of tasks, and to rapidly adjust to meet changing, task-specific needs. This ability will increase the total efficiency of a computing platform by allowing high utilization of a large portion of the computing resources during all phases of a mission or application. For instance, in signal processing applications the resources may be configured to efficiently support data parallel operations, and then reconfigured (*morphed*) to support analysis and knowledge processing when data is available.

DARPA is supporting PCA architecture and chip development for four systems: the Raw architecture at the Massachusetts Institute of Technology [3], the Stanford University Smart Memories project [4], the MONARCH project at the University of Southern California Information Sciences Institute [5], and the TRIPS system from the University of Texas at Austin [6].

PCA devices have a pool of configurable computing resources on a single integrated circuit, connected by a configurable communication network. Computing resources consist of such elements as arithmetic logic units (ALUs), floating point units (FPUs), and various types of memory. Each resource typically can operate in one of several modes, and has some configurability within a mode. Some PCA devices support aggregation of several of the computing resources to behave as a single more complex and powerful type of resource. Computing resources are typically placed into a repeating tiled arrangement, with each tile consisting of one or more instruction processors and associated local memory. Communication networks typically consist of a fixed data path from each computing tile to external I/O and memory, as well as a configurable local path from each tile to one or more of its neighboring tiles. The notional, generic depiction of a PCA device shown in Figure 1 suggests some of these key features. Specific PCA devices, however, vary in whether they have one or multiple procesors per tile, the number and type of interconnect networks, and so forth. Although the four PCA systems supported by DARPA all feature a tiled architecture, they vary widely in their details.

Tiled computing resources with dedicated local memory increase the efficiency of processors by reducing the average distance between a processing element and the memory used by that element. With clock rates on general purpose processors constantly increasing, this distance becomes important when it exceeds the distance that memory information can travel during a clock cycle. Each of the processing cores in a tiled configuration is typically smaller and less capable than the processing core on a traditional CPU, but is able to achieve higher utilization. The presence of several such tiles on a single IC allows very high bandwidth and low latency communication between processing cores, which in turn allows applications to be parallelized more effectively

than on platforms with less efficient inter-process communication networks (e.g., symmetric multiprocessors or cluster computers).

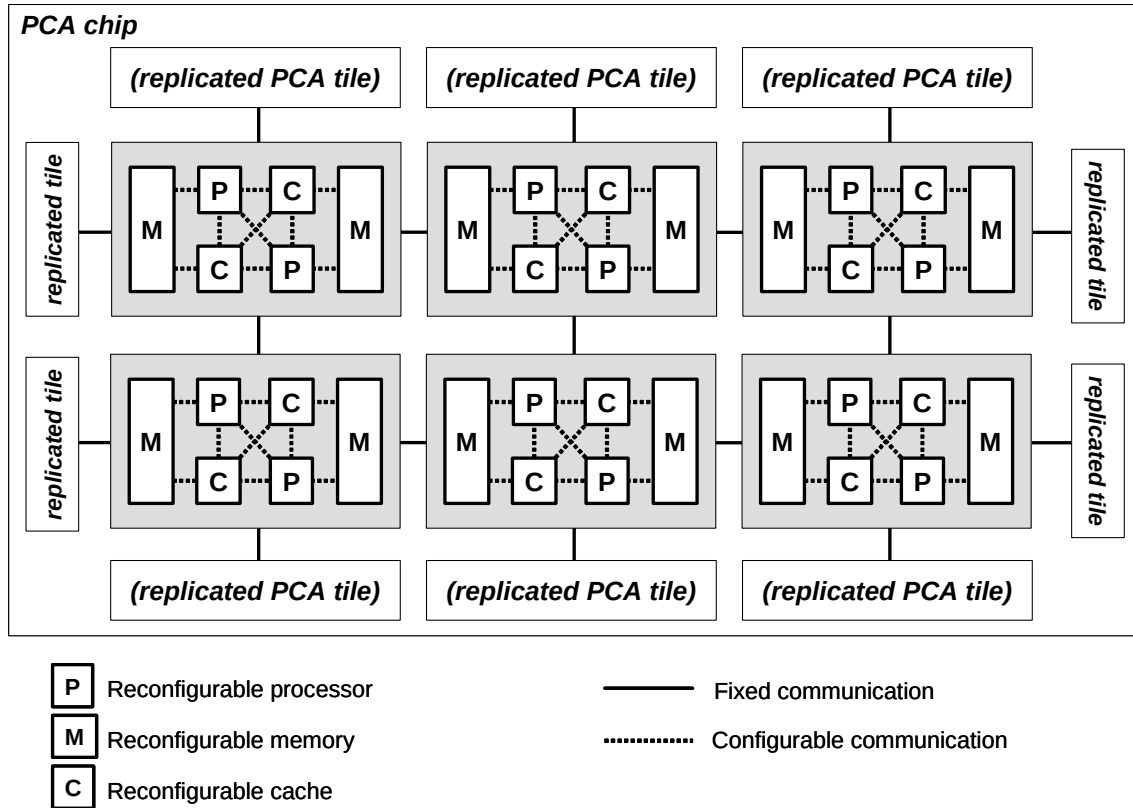


Figure 1. Generic PCA device architecture.

Two other aspects of PCA architectures contribute to their high performance. One is the high ratio of device area dedicated to computation resources such as ALUs and memory to the area devoted to control overhead. The second is the unusually high degree of control over the configuration and allocation of those resources available to the programmer and compiler.

These features allow PCA architectures to achieve a significantly higher utilization than is typical for modern traditional CPUs, for a wide variety of applications. To achieve these levels of performance, however, the application development tools must be very effective at utilizing the disparate and configurable resources present in a PCA system. In particular, the tools must be capable of identifying the parallelism in an application and partitioning that application to take advantage of the specific resources of a particular PCA chip. New languages and a new framework have been developed that allow application developers to expose data dependencies and opportunities for parallelism to the build chain, and allow the configuration space of PCA platforms to be expressed in a structured and analyzable way.

The MSI has two main goals: to reduce the effort required by tool developers, and to allow productive development of high performance portable applications. The level of effort required for tool development is reduced by standardizing and abstracting multiple

portability layers within the MSI. Creating a portable virtual machine abstraction of PCA hardware, as well as portable application level APIs, reduces development effort by presenting new tools and architectures with a common abstraction target.

Morphing

There are many possible scenarios and situations in which a change, or *morph*, in the configuration of a PCA system may be desired. The Morphware Forum has identified and categorized the types of these situations to aid in identifying the hardware and software services required by applications, operating systems, and run-time resource managers. This categorization encodes three orthogonal aspects of the attributes of a morph [7]. These are:

- whether the morph is initiated directly by an API call within the application code, or is initiated by the run-time system or compiler invisibly to the application programmer;
- whether the physical resources allocated to the application must change or stay the same; and
- whether the components of the application (or the entire application) continue to execute or are reloaded or replaced.

The set of morph types resulting from these attributes is summarized in Figure 2. At this time morph type 4a, where the platform configuration is determined by the build tools at compile time and set by the run-time environment at load time, is the only morphing type supported by the MSI. Support for additional morph types will be developed in the future.

Example of a PCA Application

Many high performance applications process data from a sensor network to solve a physical problem. Examples range from radar and sonar surveillance systems to audio and video multimedia devices. These applications can often be intuitively described by a directed graph of well-defined, computation-intensive tasks (“kernels”). Each kernel in the graph receives data from the system input or from other kernels, processes it, and passes the modified data to still other kernels or to the ultimate system output. The concept is shown in Figure 3. A common constraint of such applications is that data must pass through the graph fast enough to keep up with the real-time sensor input stream. The need for high performance, flexible implementations of applications of this type is one of the major motivators for DARPA’s development of PCA technology.

	Run-time System		Application Programmer		Compiling System	
	Components continue	Components change	Components continue	Components change	Components continue	Components change
Resource allocation doesn't change	Type 0a	Type 1a	Type 2a	Type 3a	Type 4a	Type 5a
	Run-time environment changes transparently to the running application.	Run-time system changes components to reconfigured but equivalent set of resources.	Application makes API call to make suggestions.	Application makes API call to change processing mode but does so within existing resource set.	Compiler instructions reconfigure allocated resources.	Compiler switches to a different library able to use the same resources.
Resource allocation changes	Type 0b	Type 1b	Type 2b	Type 3b	Type 4b	Type 5b
	Run-time system changes resource allocation of a running application transparently to the application.	Run-time system configures resources and loads components at application startup.	Application makes API call to give up or gain some resources.	Application makes API call to add or replace one or more components using different resources.	Compiler requests different resources to meet change in performance specified by metadata.	Compiler switches to a different library that uses different resources.

Figure 2. Taxonomy of Morph Types.

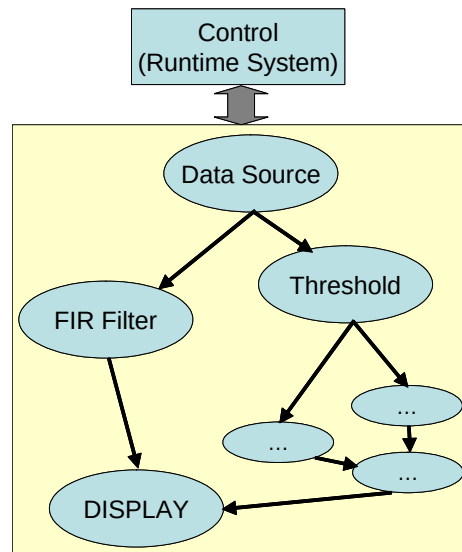


Figure 3. Illustration of the streaming computation concept. (Figure courtesy of MIT.)

To get a sense of the type and scale of a particular example of a streaming sensor application, consider the Integrated Radar-Tracker (IRT) benchmark application [8]. The IRT is an end-to-end specification of a modern intelligence, surveillance, and reconnaissance (ISR) radar system. Motivated by a space-based radar application, it embodies all of the major attributes required in a defense-oriented PCA application test: both streaming and data-dependent threaded computation with multiple sub-types of each (*e.g.*, fast transforms vs. vector-matrix arithmetic in the streaming elements); heavy computational loads; and multiple application-level parallelization and morphing opportunities. Developed by MIT Lincoln Laboratory (MIT/LL), the benchmark consists of a MATLAB simulation that serves as an executable specification, sample data sets, spreadsheets for estimating the computational loading of the application, and instructions for installation and operation. The benchmark is being developed in stages; as of this writing, the initial version of the IRT, which does not yet include all of the functionality described below, is available to the PCA community at the Morphware Forum web site.

Figure 4 shows a very high level view that divides the IRT application into two major blocks, ground moving target indication (GMTI) and feature-aided tracking (FAT), based on computational load and processing type. The FAT block has two options, signature- or classification-aided tracking (SAT or CAT). The total system load is a strong function of the processing parameters and the number of targets to be tracked; one set of default parameters provided by MIT/LL gives an estimated load in excess of six TeraFLOPS (TFLOPS).

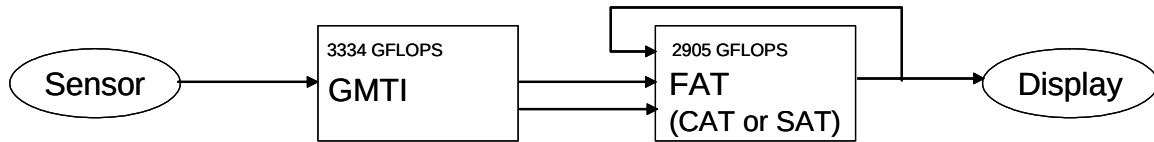


Figure 4. High-level decomposition of the IRT benchmark.

The IRT embodies both major computing types important in PCA systems, *streaming* and *threaded*. These classes of computing are described further in the next section. The large majority of the processing in the GMTI block consists of a variety of signal processing operations such as polyphase FIR filtering (for the subband analysis and synthesis), matrix-vector operations (adaptive beamforming and space-time adaptive processing), fast Fourier transforms (Doppler filtering), and correlation or convolution (pulse compression). All of these are examples of streaming operations. They apply non-data dependent, fixed (except for parameter choices) kernels to incoming data samples on a continuous or block basis, producing modified data streams or blocks that are input to the next kernel in the functional flow graph. Operations are sequenced in a dataflow manner, with each kernel able to “fire” when all of its inputs are available. Because of the fixed kernels and lack of data-dependent control flow and data production, the system can be deterministically scheduled.

In contrast, the set of operations that implement the tracking processing represented by the FAT block are highly data-dependent, with the computational load depending not only on the number of targets to be tracked, but the actual physical behavior (kinematics) of those targets. As a result, even though most of the load comes from arithmetic

calculations, the number and sequencing of those calculations varies with the sensor scenario. In addition, the tracker uses stored databases (reference signatures for a mean-squared error calculation, and a dynamic database of track histories for the kinematic tracker) as well as incoming sensor data.

The IRT can be parallelized in a number of ways, including both data-parallel and pipeline-parallel approaches, with varying levels of granularity for each. The IRT also supports application-level morphing of various granularities in both functional complexity and time scale. For instance, reconfiguring the same PCA resources from GMTI to FAT processing would require a major functional reorganization, probably with very low latency, while operator-selected parameter changes would require only a relatively minor scaling of the existing functional flow and be infrequent and tolerant of greater latency. These parallelization and morphing options provide ample opportunities to exercise PCA and MSI capabilities.

PCA LANGUAGES AND COMPILATION

The Morphware Stable Interface is composed of several novel elements. These include the use of streaming languages, a two-stage compilation with a virtual machine middle layer, and metadata to describe the target architecture and guide high-level compiler optimization. The following subsections provide more detail on each of these elements.

Streaming Algorithms and Languages

As discussed in the last section, high performance streaming sensor applications are of significant interest to the PCA program. Applications in this form may be efficiently mapped to physical resources in several ways. For instance, kernels may be multiplexed in space or time. Multiplexing in space statically assigns each kernel in the graph to disjoint physical resources. In this scheme, each kernel must be assigned to enough resources to ensure that its throughput equals to the input rate. In time multiplexing, kernels take turns using all of the system resources to process a block of input data, so long as the system is able to rotate through all the kernels quickly enough to keep up with the input data. Hybrid combinations are also possible.

Many opportunities for partitioning a streaming computing problem are masked from compilers and linkers by traditional general-purpose languages. Streaming languages are efficient on PCA hardware because they expose data parallelism and the kernel structure directly in the applications' representation in the programming language. Compilers can then determine which data may be kept local to the compute resources and streamed directly between logic units via internal buffers, greatly improving utilization efficiency, power, and throughput. Streaming languages and the associated methodology are an integral part of the MSI.

Because of the potential efficiency of streaming on PCA systems, several of the PCA hardware teams are developing new stream-based languages to aid in exploiting the capabilities of their hardware. Two of these, Brook and StreamIt, are currently supported by the MSI. While both StreamIt and Brook are similar in their use of streams and kernels to process data, they also have several significant differences. StreamIt represents a program as a single stream graph that operates on a conceptually infinite stream, while Brook supports multiple stream graphs that operate on finite streams and are controlled from a pointer-less subset of C. Kernels in StreamIt must have static input and output rates, while Brook kernels can be dynamic. StreamIt kernels can have internal state that is preserved between invocations, while Brook kernels must be stateless; also, StreamIt kernels can peek at input items without popping them from the stream, while Brook kernels must pop any items that they inspect. Finally, the stream graphs in StreamIt are composed of hierarchical units, each of which has a single input stream and single output stream, while Brook supports a flat graph of kernels, each of which can have multiple input streams and multiple output streams. StreamIt and Brook are fully detailed in their respective specifications [9,10]. Rudimentary examples of each are given in the Appendix of this document. More substantial examples of Brook code are included in Reservoir's R-Stream 1.0 high-level compiler release [11].

Two Stage Compilation

The implied abstract machine model on which traditional programming languages such as C are built has become an increasingly poor match to actual modern processor architectures. Greater use of multiple processors, multiple independent process flows within a processor, a complex memory hierarchy, and the common use of many isolated computing systems on a single application (clusters and distributed computing) have all created important deviations from the underlying computational engine for which these languages were designed. Many approaches have been taken in an attempt to maintain the usability of traditional languages, while exploiting the emergent capabilities of modern computers, including domain-specific middleware, explicit communication libraries, special codes inserted into source code to serve as compiler hints, and a litany of general guidelines and approaches for structuring programs to best suit particular platforms [12]. Each of these approaches has had some success in exploiting new technology, but has resulted in software that is tightly coupled to a particular deployment platform, and often requires a significant level of human effort to maximize performance for that configuration.

With a finite, but increasing set of source languages, APIs, and PCA target platforms, each new API or source language introduced requires an application development toolset for each possible target PCA platform, and each new PCA platform requires a development toolset for every source language and API supported. The resulting proliferation of toolsets is depicted in Figure 5.

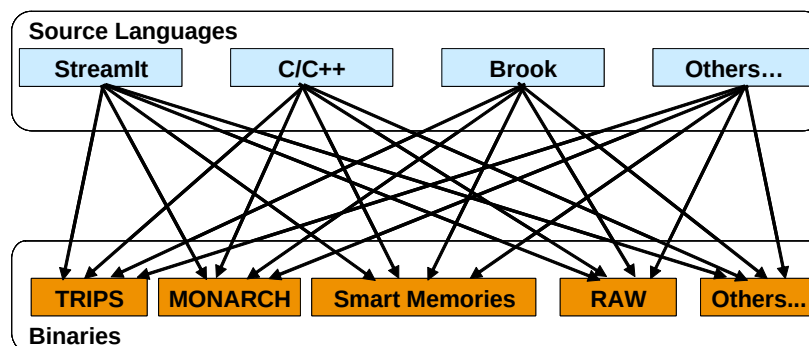


Figure 5. Multiple source languages and multiple hardware targets require development of many different software tool sets.

The approach of building multiple independent compilers and build tools is not efficient for PCA platforms, which can change and reconfigure frequently. The build-chain tools must be portable and capable of deploying an application onto a wide variety of platform configurations and in addition must be capable of selecting the optimal hardware/software configurations for a particular task and set of constraints. In order to support increased portability and to speed the deployment of new PCA devices and PCA programming languages, the Morphware Forum recognized the need to establish an explicit expression of the common abstractions of the PCA processors in the MSI. In particular, since the source languages and Application Programming Interfaces (APIs) (e.g., C, C++, StreamIt, Brook) developed for PCA architectures are targeted to similar implicit abstract machines, and since PCA platforms share common abstract

characteristics, it is natural to introduce a portability layer between these APIs and the PCA hardware that encapsulates the common abstractions.

In the MSI, this portability layer is the Stable Architecture Abstraction Layer (SAAL). The SAAL is a set of portable APIs that encapsulate abstractions of the computing resources present in PCA devices, as well as the operations on those resources used by the MSI source languages and APIs. This portability layer abstracts and simplifies PCA hardware for the source languages, and provides a consistent abstract set of resource types and functional support requirements for PCA hardware developers. The SAAL portability layer also simplifies the deployment of new PCA platforms and new MSI source languages and APIs by providing a single common target for each. New languages and APIs must only provide a mapping to the SAAL portability layer, instead of build tools targeting every possible target platform. New PCA platforms need only supply a compiler for the SAAL to work with all of the existing source languages and APIs. This simplified toolset architecture is depicted in Figure 6.

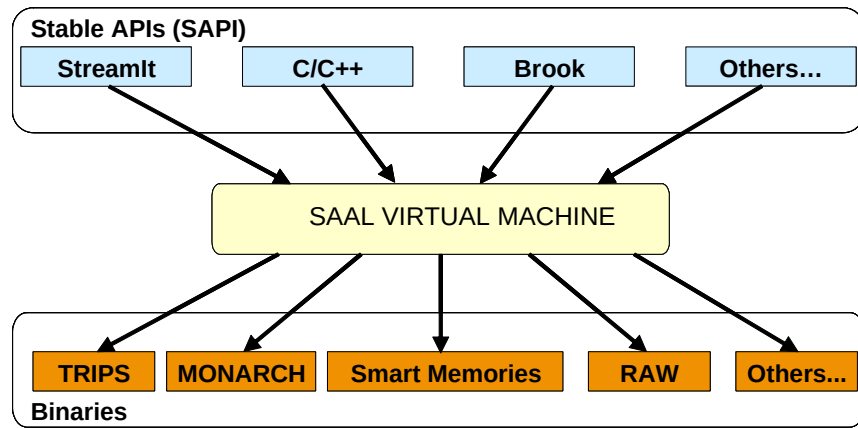


Figure 6. Introduction of a virtual machine layer reduces the number of software tools required.

The MSI framework thus uses two separate compile steps. The source languages and APIs collectively are referred to as the Stable Application Programming Interface (SAPI) layer. The top-level input at the SAPI level is processed by the high-level compiler (HLC) appropriate to the source language(s) used. The high-level compiler outputs SAPI code, which is the input to the low-level compiler (LLC) for the target PCA platform of choice. At this writing, the first alpha version of a PCA high-level compiler, Reservoir Inc.'s R-Stream, has been released [11]

ELEMENTS OF THE MSI

Figure 7 illustrates the major elements of the MSI. Applications are written in one or more of the SAPI source languages. The target platform and the set of possible configurations of the target platform are described using the PCA machine model metadata context, described below. The Morphware Forum has begun defining a metadata context for application information including such elements as performance requirements (*e.g.*, computational throughput, latency and power), input rates, and module interconnections. This metadata context is currently undefined, and no existing high-level compiler uses this context. Currently all high-level compilers optimize for execution time, and do not support specific performance requirements.

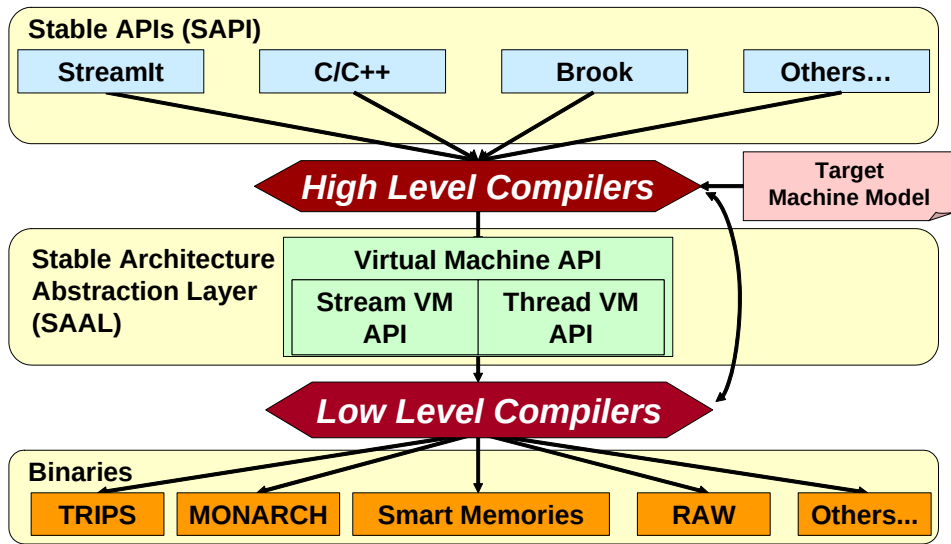


Figure 7. Elements of the MSI.

The high-level compiler is responsible for selecting the desired platform configuration, and for converting the SAPI source code to SAAL code. The high-level compiler performs coarse-grained parallelization of the input application, based on the granularity appropriate to the target platform. In order to accomplish this effectively, the high-level compiler must have a description of the target platform. This description is provided in the PCA machine model metadata context [14], where the capabilities and resources of PCA platforms are described in terms of a common, high-level model that allows the high-level compiler to perform accurate performance estimates. The output from a high-level compiler is SAAL code and a desired initial platform configuration. Depending on the application source language, the SAAL intermediate code may be Streaming Virtual Machine (SVM) code, User Virtual Machine (UVM) code, or Threaded Virtual Machine-Hardware Abstraction Layer (TVM-HAL) code, described below.

The low-level compiler is an architecture-specific build tool that accepts the SAAL code and desired platform configuration metadata output by the high-level compiler and produces native machine code suitable for execution on the target PCA platform. If a

performance constraint, resource requirement, or desired configuration is unobtainable by a low-level compiler, that low-level compiler fails and reports the error.

The Stable Architecture Abstraction Layer

The SAAL is composed of a number of parts, each addressing particular aspects and requirements of application programs and the high-level compilers. Streaming code is primarily translated into an intermediate language described by the Streaming Virtual Machine [15]. The SVM supports the mapping of kernels and streams onto virtualized processing and memory resources. Conventional code and the control code for the streaming languages are translated into an intermediate language described by the Threaded Virtual Machine. The TVM includes support for numerous non-streaming activities. A portion of the TVM, called the Hardware Abstraction Layer (TVM-HAL) [16], defines a thin abstraction layer for low-level hardware services such as memory mapping and exception handling. Many TVM capabilities, such as access to privileged instructions, are used only by operating system functions. Therefore, a User Virtual Machine (UVM) interface has been defined that contains somewhat restricted functionality that is intended to be used directly by application code and libraries [17].

Application programmers will not normally interact with any of the interfaces at the SAAL level. This code is either produced by the high-level compilers, or is contained in special-purpose libraries or operating system code written by expert library programmers. Figure 8 depicts the relationships between the various interfaces.

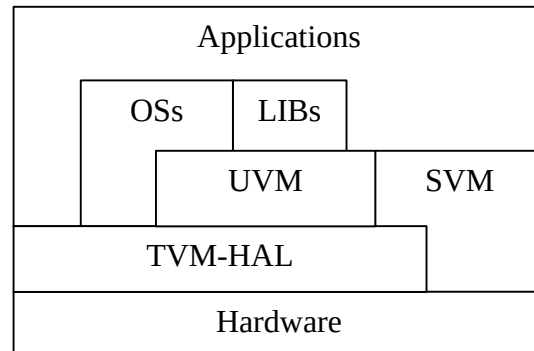


Figure 8. The relationships between applications, SAAL layer interfaces, and the underlying PCA hardware architecture.

Stream Virtual Machine Overview

The streaming virtual machine is the portion of the SAAL API that implements streaming functionality on virtual resources. The SVM allows streaming applications to be expressed in terms of operations on the set of streaming virtual resources described below. These resources are mapped directly to physical resources in a PCA system by the low-level compiler.

The virtual resources within the SVM API are:

- *Streams* are sequences of data elements of a specified type. Streams are used for sequential production and consumption of data elements.

- *Kernels* are computational engines that consume zero or more input streams, and produce zero or more output streams. Kernels are described using a subset of C, and also maintain execution state information.
- *Blocks* provide random, indexed access to a fixed set of data elements.
- *Controls* are independent computational engines that can initiate, monitor and control the state of, and terminate kernels. Controls are described with a subset of C.
- *Data elements* are primitives supported by a particular platform, arrays of fixed length, or fixed-sized `structs` containing primitives and arrays.

The SVM API is a C API that can be implemented as a library, if desired, for testing and debugging purposes. The SVM API is described in detail in the document titled “PCA Morphware Virtual Machine Specification” [15].

Thread Virtual Machine Overview

The thread virtual machine is comprised of two major elements, the TVM Hardware Abstraction Layer, and the User Virtual Machine. The TVM-HAL provides a SAAL level virtual machine interface for threaded portions of PCA applications and libraries. The TVM-HAL is a privileged interface that provides a thin layer of abstraction on the available PCA architecture and presumes total control over the available resources. The UVM is also a low-level interface, but does not presume total resource control over the platform. This allows a more direct mapping from application code, but also requires a richer interface to allow user-level response to changes in resource availability, or failure to obtain requested resources. The UVM and TVM-HAL together address the major areas of processor control, exception handling, and memory management, as well as several other low-level requirements to support general purpose threaded software.

Processor control and exception handling in the UVM and TVM-HAL include support for exception vectoring of up to 255 exceptions, with special state maintenance code for exception handlers; instructions to convert a processor from a threading mode to streaming mode; and a scheduler activation interface. The scheduler activations interface [18] provides a communication mechanism between a kernel-level scheduler and application code. This interface provides for a set of callbacks to user code when a processor has been assigned to or taken away from a process, or when a thread has become blocked or unblocked (for example, while waiting for I/O).

PCA architectures differ from conventional architectures in that they have many distinct blocks of memory distributed across the system. This requires special handling for memory addressing in the UVM and TVM-HAL interfaces. Memory management in the UVM and TVM-HAL includes support for segmented memory as well as paged memory. Paged memory control is available in the TVM-HAL but not in the UVM. The segmented memory model allows for virtual addressing and configuration of individual segments, including whether a segment is visible to a particular processor, initial virtual and physical addresses, and caching control.

The UVM and TVM-HAL support several other control constructs including active DMA, multiprocessor synchronization, mutex locks, cache memory control, performance counters, IEEE 754 floating point control, and condition variables.

Metadata

PCA systems are designed to meet a variety of goals and constraints, and to function on a wide variety of platform configurations. PCA applications require a large amount of information in addition to the procedural definitions of programs in source languages. Examples of this information are the computing resources available on a particular host platform, the set of possible configurations of these resources, desired optimization goals for a particular piece of software, and computing resources required by a particular piece of compiled software. This set of descriptive and extra-functional information is known collectively as *metadata*.

Each of the specific uses for metadata within the MSI is known as a metadata *context*. At this writing, the PCA machine model is the only defined metadata context. Others are under consideration by the Morphware Forum, for example an application information metadata context and a context describing high-level compiler output assumptions and decisions.

In order to ease the development of build-chain tools and speed the development of PCA applications, the MSI includes a standard method for describing metadata contexts, as well as a standard method for expressing and storing metadata. For example, metadata is stored in XML text files. The PCA metadata system is fully described in the document titled “PCA Metadata System” [13].

Machine Model Metadata Context

A PCA machine model is a generic model used to describe specific PCA platforms. The primary purpose of the machine model is to describe PCA platforms in a common manner to the various tools in the MSI framework. The consumers of this description are high-level compilers, which are responsible for dividing applications into appropriately sized portions for a target platform. The parameters of these models are communicated to the compilers and other tools as metadata in the machine model metadata context, which is formalized in the document titled “PCA Machine Model” [14].

A PCA machine model contains three types of elements: *resources*, *ingredients*, and *morphs*. A resource is a description of a discrete device with well-defined functionality. There are three kinds of resources in a machine model. These are:

- *Processors*: anything that consumes, produces, or moves data. Examples of processors are functional units, DMA engines, and I/O devices.
- *Memories*: Any resource that stores data. Examples are RAM, FIFOs, and cache memory.
- *Network Links*: Any communication pathway between resources. Network links may have an arbitrary number of senders and receivers.

Resources have a variety of parameters to allow detailed descriptions of their capabilities.

An ingredient is a description of an arbitrary, configurable division of underlying hardware. An ingredient can be configured, possibly in concert with other ingredients, into one or more resources. Ingredients provide a mechanism for identifying resources that cannot coexist on the physical platform.

Finally, a morph in the machine model is a description of a state into which a PCA platform may be configured. The metadata describing the morph defines how the machine model ingredients can be configured into resources by the compiler. More than one morph may be active at once, but no two morphs that share an ingredient may be active at once.

CREATING AN APPLICATION

The general flow for creating a PCA application using the Morphware Stable Interface is illustrated in Figure 9. Application programmers using the MSI first partition their application according to the high-level language that will be used to express their algorithms. Parts that fit the streaming paradigm are written in Stream or Brook, while non-streaming parts may be written in C or C++. The high-level language code, augmented with application performance metadata and a machine model metadata description of the target architecture, is compiled by a high-level compiler to SAAL code, which is based on the SVM and UVM APIs, respectively. The HLC may be provided by a PCA architecture vendor or by a third party. Reservoir Labs, Inc.'s R-Stream is an example of a third-party HLC [11]. Application performance metadata must be provided by the application programmer, while the machine model is supplied by the architecture vendor. The application source code may also call functions in libraries that were written by experts using the UVM or TVM-HAL APIs directly.

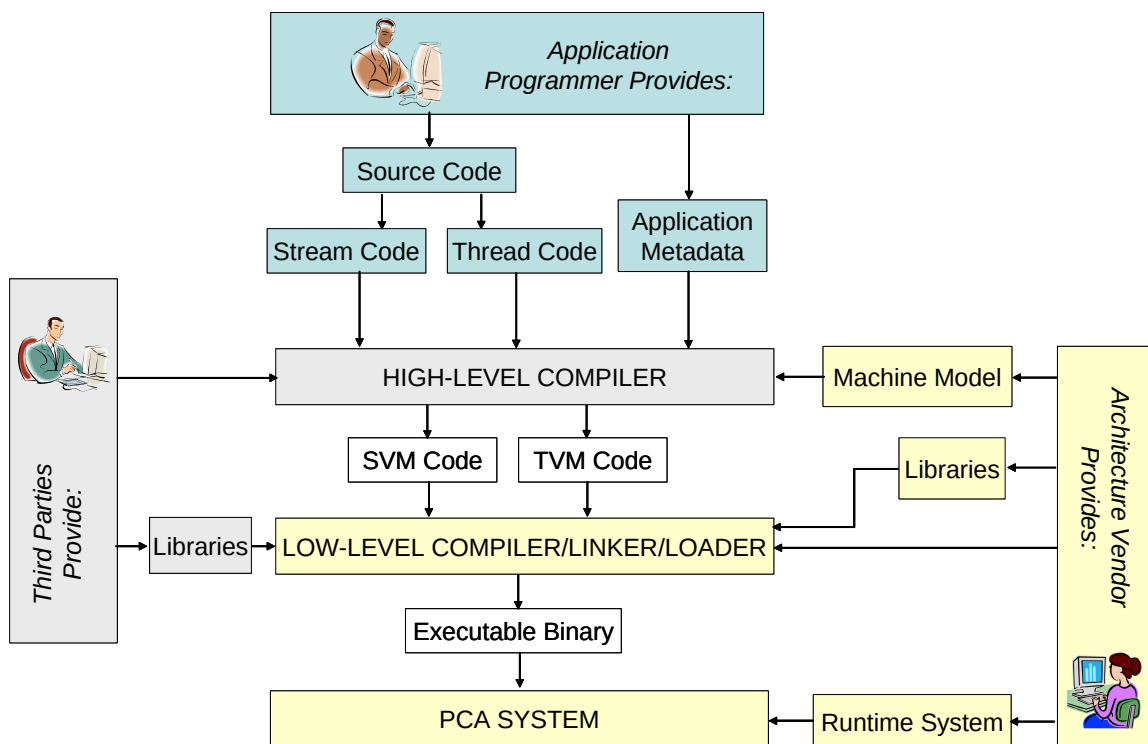


Figure 9. Methodology for creating a PCA application using the MSI.

Each architecture vendor provides one or more low-level compilers specific to their PCA system or device architecture. The various files in the SAAL intermediate formats are compiled for the target architecture, producing architecture-specific binary files. These are linked as appropriate to create one or a set of binaries that are ready to be executed. These binaries are annotated with a set of hardware resource requirements needed for the program to successfully execute. When the program is executed, the PCA run-time system on the target verifies that the required resources are available, allocates the

resources and assigns them to the executing program, and configures the PCA hardware into the initial configuration before starting and running the program. During execution, the hardware may be reconfigured by the application, or by the system responding to changes in the run-time environment.

More detailed and specific information on the process of creating and executing an application using PCA hardware and the MSI is available in the references mentioned elsewhere in this document as well as at the web sites of the various PCA architecture providers.

FUTURE DEVELOPMENTS

To date, the Morphware Forum has focused on development of the two-level compile structure and machine model, thus creating a portable development process for compiling basic program units and supporting libraries. However, these elements of the MSI are not sufficient to meet the full set of goals of the PCA program. The Forum is working actively to identify and prioritize the additional services, APIs, and metadata contexts needed for a fully functional PCA system, and to extend the MSI to include them. Extensions that have been discussed include:

- New extensions to program loaders and linkers to support selection and composition of program components from multiple options;
- New operating system (OS) services and calls to support the unique requirements of application morphing at the various levels described in Figure 2;
- A portable resource management system to manage the real-time selection and control of alternative program instantiations;
- A component-based application software architecture; and
- A composable development system that can provide a stable, portable interface to the application programmer while still being customized to the unique lower-level development environment for each PCA architecture.

Once implemented, these MSI services and architectures will support the advanced capabilities made possible by PCA devices. Most fundamental will be the capability to implement morphing across the full spectrum of circumstances detailed in Figure 2. This in turn is an essential capability to support important end-user system attributes such as validation and verification (V&V) of PCA applications, and real-time fault tolerance in mission-critical applications. For instance, V&V is reported to account for as much as 40 to 70% of system costs [20]. The PCA program is seeking to implement a significant portion of the V&V capability in the application build cycle by extending standard techniques with run-time monitoring and checking. Such an approach would likely be significantly integrated with the run-time resource management portion of the MSI. Similarly, the morphing capability of PCA technology offers significant new flexibility in the system response to run-time faults. Utilizing this capability effectively will take advantage of the component-based approach to PCA software and will again require tight integration with the resource management functions.

REFERENCES

1. The Polymorphous Computing Architectures (PCA) Program, Information Processing Technology Office (IPTO), Defense Advanced Research Projects Agency (DARPA), www.darpa.mil/ipto/Programs/pca/index.htm.
2. The PCA Morphware Forum, www.morphware.org.
3. M. B. Taylor *et al*, “The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs,” *IEEE Micro*, March-April 2002. See also cag.lcs.mit.edu/raw/.
4. K. Mai *et al*, “Smart Memories: A Modular Reconfigurable Architecture,” *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000. See also www-vlsi.stanford.edu/smart_memories.
5. J. Granacki and M. Vahey, “MONARCH: A Morphable Networked micro-ARCHitecture,” presentation to High Performance Embedded Computing Workshop, October 2002. See also www.isi.edu/asd/monarch/.
6. S. W. Keckler *et al*, “A Wire-Delay Scalable Microprocessor Architecture for High Performance Systems,” *International Solid-State Circuits Conference (ISSCC)*, pp. 1068-1069, February 2003. See also www.cs.utexas.edu/users/cart/trips/.
7. “Morph Taxonomy,” Georgia Institute of Technology and SPAWAR Systems Center San Diego, white paper dated Feb. 3, 2004. Available at www.morphware.org.
8. B. Coate, J. Lebak, J. McMahon, and A. Reuther, “Basic Deliverable of the Integrated Radar-Tracker Application,” MIT Lincoln Laboratory, February 21, 2003. This document in turn references more detailed design documents for each major IRT component. Available at www.morphware.org.
9. “StreamIt Language Specification”, Version 2.0, MIT Computer Science and Artificial Intelligence Laboratory, October 17, 2003. Available at cag.lcs.mit.edu/streamit/papers/streamit-lang-spec.pdf.
10. P. Mattson, E. Schweitz, M. Engle, V. Litvinov, and K. Mackenzie, “R-Stream 1.0 Brook Clarification”, Reservoir Labs, November 5, 2003.
11. R-Stream 1.0 release package, Reservoir, Inc., November 2003. Package may be requested via the Morphware Forum web site, www.morphware.org.
12. Stanford University “Streaming Languages” web page, graphics.stanford.edu/streamlang/.
13. “The PCA Metadata System”, v. 0.9, Georgia Institute of Technology and SPAWAR Systems Center San Diego, Jan. 28, 2004. To be available at www.morphware.org.
14. “PCA Machine Model,” September 17, 2003. Available at www.morphware.org.

15. “PCA Morphware Virtual Machine Specification,” November 4, 2003. Available at www.morphware.org.
16. Lance Hammond, “TVM-HAL Specification”, Smart Memories Group, Stanford University, June 2003. Available at www.morphware.org.
17. “User-level TVM (UVM) Interface Specification,” September 3, 2003 revision, TRIPS Project, Department of Computer Sciences, The University of Texas at Austin. Available at www.morphware.org.
18. T. Anderson, B. Bershad, E. Lazowska, and H. Levy. “Scheduler activations: Effective kernel support for the user-level management of parallelism”, *ACM Transactions on Computer Systems*, vol. 10(1), pp. 53-79, February 1992.
19. S. Amarasinghe and W. Thies, “Stream Languages and Programming Models,” presentation at PACT 2003, Sept. 27, 2003. Available at www.cag.lcs.mit.edu/streamit/talks/pact03tutorial/pact03languages.pdf.
20. M. Amduka, D. Kreckler, and O. Sokolsky, “Run-Time Environment and Design Application for Polymorphous Technology Verification and Validation (READAPT V&V)”, presented at Morphware Forum meeting, December 10, 2003. Available at www.morphware.org.

APPENDIX: STREAM LANGUAGE EXAMPLES

Brook Example

The Reservoir, Inc. R-Stream 1.0 release contains several sample source codes, including a simple Brook program that sums all of the elements of two streams [11]. Figure A-1 is a block diagram of that program; the program code follows.

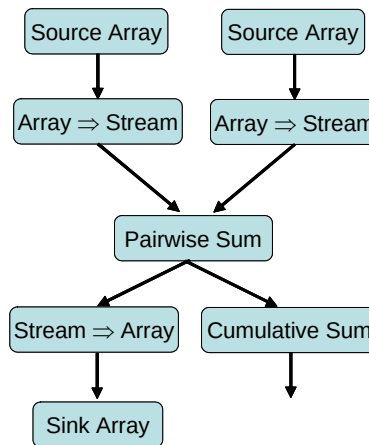


Figure A-1. Block diagram of example Brook program for pairwise sum of two streams.

```

/****--**--*
*
* Copyright (C) 2003 Reservoir Labs. All rights reserved.
*
*--**--***/

/*-----*
* Stream Sum                                     *
*                                             *
*   Get the sum of all the elements in a stream *
*-----*/

extern int printf(char [], ...);

/*-----*
* Simple kernel which computes the pair-wise sum of two streams
* of elements. For example, the 5th element of the output stream
* c is the sum of the 5th elements of the input streams a and b.
*-----*/
kernel void pairwiseSum(int a<>, int b<>, out int c<>) {
    c = a + b;
}

```

```

/*-----
Simple kernel which computes the total of a stream of elements.
Specifically, the reduction value total is set equal to the sum
of all elements in the input stream a.
-----*/
kernel void computeTotal(int a<>, reduce int total) {
    total += a;
}

/*-----
Brook code which reads the contents from arrays a_array and
b_array into streams a and b using the streamReadAll stream
operator, applies pairwiseSum kernel to those streams, and
writes the output stream c back to the array c_array using the
streamWriteAll stream operator.

Also uses the computeTotal kernel to compute the total of all
elements in c and prints the result.
-----*/
int main(void) {
    int i;
    int total = 0;
    int a_array[100];
    int b_array[100];
    int c_array[100];
    int sum = 0;
    int a<>, b<>, c<>;

    for(i = 0; i < 100; i++) {
        a_array[i] = i;
        b_array[i] = 100 + i;
        sum += i + 100 + i;
    }

    streamReadAll(a, a_array);
    streamReadAll(b, b_array);

    pairwiseSum(a, b, c);

    streamWriteAll(c, c_array);

    computeTotal(c, total);

    printf("\n\nReal sum = %d\n", sum);
    printf("Calculated sum = %d\n", total);

    if(sum == total) {
        printf("PASSED!\n\n");
    } else {
        printf("FAILED!\n\n");
    }

    return (total != sum);
}

```

StreamIt Example

The following example of a frequency band detector, consisting of an A/D converter followed by a bandpass filter and four detectors, is taken from [19]. Figure A-2 shows a block diagram of the system; StreamIt code to implement the system follows.

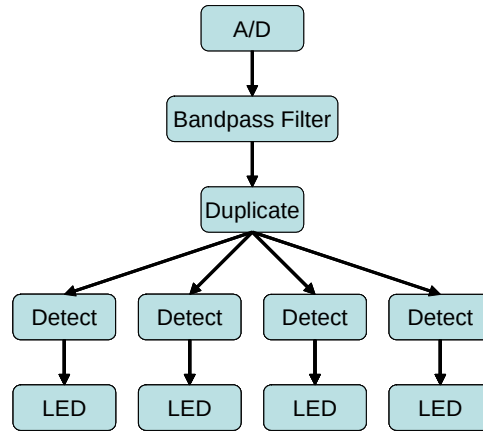


Figure A-2. Block diagram of frequency band detector. (After [19].)

```

void -> void pipeline FrequencyBand {
    float sFreq = 4000;
    float cFreq = 500/(sFreq*2*pi);
    float wFreq = 100/(sFreq*2*pi);

    add BandPassFilter(1, cFreq-wFreq, cFreq+wFreq, 100);

    add splitjoin {
        split duplicate;
        for (int i=0; i<4; i++) {
            add pipeline {
                add Detector(i/4);
                add LEDOutput(i);
            }
        }
        join roundrobin(0);
    }
}
  
```

As an example of a kernel, the following code implements a low pass filter:

```

float -> float filter LowPassFilter (int N, float freq) {
    float[N] weights;

    init {
        weights = calcWeights(N, freq);
    }

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i]*peek(i);
        }
    }
}
  
```

```
    }  
    push(result);  
    pop();  
  }  
}
```

APPENDIX B

Last Available Version of the Morphware Stable Interface Document

THE PCA METADATA SYSTEM

Version 1.0

March 2, 2004

The PCA Metadata System

Georgia Institute of Technology
and
Space and Naval Warfare Systems Center San Diego

Version 1.0
March 2, 2004



©2004 Georgia Tech Research Corporation, all rights reserved.

A non-exclusive, non-royalty bearing license is hereby granted to all persons to copy, modify, distribute and produce derivative works for any purpose, provided that this copyright notice and following disclaimer appear on all copies: THIS LICENSE INCLUDES NO WARRANTIES, EXPRESSED OR IMPLIED, WHETHER ORAL OR WRITTEN, WITH RESPECT TO THE SOFTWARE OR OTHER MATERIAL INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF PERFORMANCE OR DEALING, OR FROM USAGE OR TRADE, OR OF NON-INFRINGEMENT OF ANY PATENTS OF THIRD PARTIES. THE INFORMATION IN THIS DOCUMENT SHOULD NOT BE CONSTRUED AS A COMMITMENT OF DEVELOPMENT BY ANY OF THE ABOVE PARTIES.

This material is based in part upon work supported by the U.S. Defense Advanced Research Projects Agency (DARPA) and other agencies of the U.S. Department of Defense (DoD). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or the DoD.

The US Government has a license under these copyrights, and this material may be reproduced by or for the U.S. Government.



Acknowledgements

Authors

The primary authors of this document were:

Daniel P. Campbell	Georgia Tech Research Institute
Dennis M. Cattel	Space and Naval Warfare Systems Center San Diego
Randall R. Judd	Space and Naval Warfare Systems Center San Diego
Mark A. Richards	Georgia Institute of Technology

The authors wish to thank the members of the Morphware Forum who reviewed and contributed to this document. The authors also thank the Defense Advanced Research Projects Agency (DARPA) and the US Navy's Space and Naval Warfare Systems Center San Diego for their support of this work.

Document Change History

This is the initial public release.

Table Of Contents

Acknowledgements	i
Authors	i
Document Change History	ii
Table Of Contents	iii
1. Introduction	1
1.1. Background	1
1.2. Approach	2
2. Requirements	3
3. The PCA Metadata System	5
3.1. Metadata Content Description	5
3.2. Metadata XML Representation	7
4. An XML Example	9
5. References	11
6. Appendix: Additional Metadata System Issues	12

1. Introduction

The Polymorphous Computing Architectures (PCA) program [1] is a U.S. Defense Advanced Research Projects Agency (DARPA) effort to develop integrated computing systems with architectures that can be changed, or *morphed*, to closely match the resource requirements of Department of Defense (DoD) application programs. Those resources can include computation, memory, and communication capabilities.

The PCA program also explicitly addresses the issue of software commonality between the various PCA hardware systems so that DoD applications can be moved from one system to another with minimal software modifications and minimal impact on performance. To carry out this function, the PCA program has created the Morphware Forum [2], a program-wide activity meeting quarterly to debate and develop software concepts and standards.

This document describes the standard approach to handling PCA-related metadata as defined by the Morphware Forum. Specific metadata content is defined in other documents establishing the Morphware Stable Interface (MSI) and standardized by the Forum.

1.1. Background

Within the PCA program, the term *metadata* refers to any information needed to create, build, and run a PCA application, that is not contained in the source code or the executable binary. This broad definition of metadata includes, for instance, such information as file dependencies and compiler options normally kept in UNIX Makefiles. Some metadata, such as that in the Makefiles, continues to be recorded in special formats as determined by other development tools. However, in those cases where the metadata is unique to the PCA program, the metadata system described in this document was designed to record the metadata and to transfer metadata information from one program or tool to another.

Various kinds of PCA information are included in the broad term metadata. They include simple data, such as program parameters, as well as complex data defining the arbitrary graphs that are used to describe PCA architectures and interconnections among programs in a large application. The information in the metadata system can also be dynamic: programs can get run-time configuration information, modify metadata at run time, and access real-time status variables such as the current operating temperature.

PCA metadata may be accessed or modified throughout the development and execution of a PCA application. For example, PCA compilers manipulate metadata at compile time, the PCA run-time system uses and creates metadata at application load time, and the application itself will access metadata at run time. In addition, various tools manipulate PCA metadata throughout the process of development and deployment of an application.

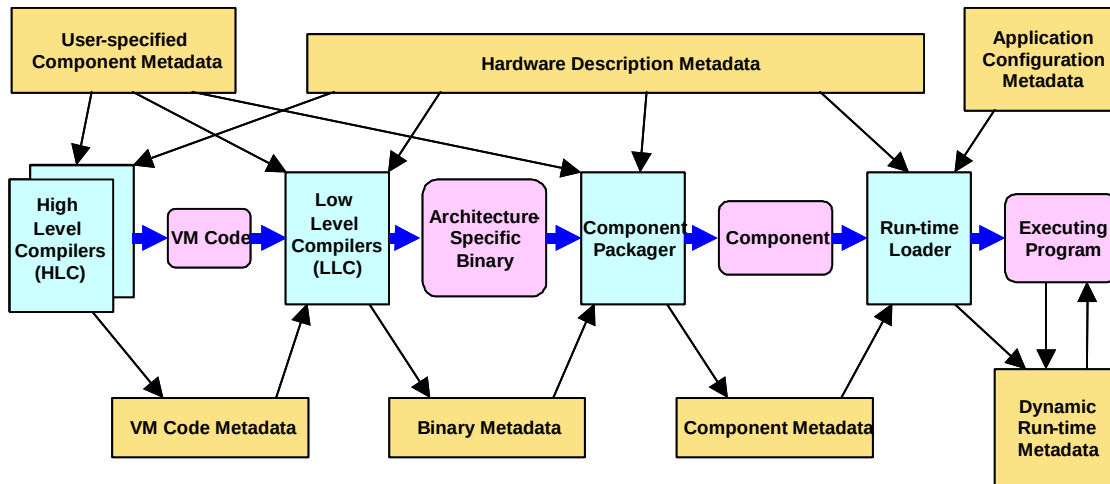


Figure 1. A possible PCA development environment illustrating that there are a number of different metadata groupings (contexts), and that each program concerns itself with more than one context.

The illustration in Figure 1 shows a typical example of how the various aspects of a PCA development system might work together. There are a number of different groupings of metadata used for different purposes. Each of the programs that use metadata works with more than one of these groupings. It is clear that throughout the PCA program there must be a single metadata system for manipulating metadata so that users, application writers, and tool developers do not have to work with multiple metadata formats.

1.2. Approach

Each of the metadata groupings described in the paragraph above and in Figure 1 is called a *metadata context*. The purpose of the metadata system is to provide a defined way to document and handle metadata for each of the contexts used in the PCA program.

The PCA approach to handling metadata information is to separate the problem into two orthogonal tasks:

- Design a simple, flexible, extensible, and usable common metadata system.
- For each metadata context, define what metadata is required and map it onto the common metadata system.

This document describes the single standard metadata system used by the PCA program. Other documents standardized by the Morphware Forum describe the actual metadata content for specific contexts.

2. Requirements

The metadata system must meet the following general requirements:

- R1. The system must allow metadata to be accessed or modified during compile and link time, at application load time, during execution of the application, and at any other time during program development.
- R2. The system must be extensible so that new kinds of metadata can be added without breaking existing programs and applications.
- R3. The system must allow programs to create and modify metadata as well as to access it.
- R4. The system must be independent of programming language, hardware architecture, and internal implementation.

The following additional requirements must be met when the system is used by an executing PCA application:

- R5. Programs must access *current* metadata values, for instance, after a morph or when the metadata value represents current performance metrics or environment measurements.
- R6. Programs must be able to request notification when a metadata value changes.
- R7. The system must provide acceptable performance in terms of speed of operation and memory resources used.

During discussions among PCA participants, a number of other capabilities were brought up that were not included in the final requirements. Among them:

- The system does not provide a capability for configurability of the metadata. Such a feature could allow expressions of metadata values (*e.g.*, “\$some_value = 2 * \$other_value”) and conditional statements (*e.g.*, “if \$display then load GUI”). These capabilities are handled with other tools.
- Access controls for reading and writing the metadata are not included in the system. The design and implementation of such a facility would require resources beyond that available to the PCA program, and would add considerable complexity to the system.
- The system does not attempt to specify a general approach to querying the contents of the metadata “database.” For specific cases, features such as this can be implemented using the capabilities provided by the system.
- Large binary objects were considered for inclusion directly as metadata. In the final system, large objects are stored separately, and normal metadata text values provide the locating information (*e.g.*, file pathnames).
- An inherent capability for describing ranges of values or constraints was suggested, but in the final system, this information is provided with additional metadata items when needed. For instance, two metadata entries could be used to contain the minimum and maximum values.

REQUIREMENTS

- The system does not validate the metadata content. When wanted, this capability is provided with other tools.

3. The PCA Metadata System

The PCA metadata system provides a standard way to document and store the various metadata contexts that will be used throughout the PCA program. This system consists of two parts. The first part is documentation that describes all metadata content in terms of *objects*, *parameters* of those objects, and *references* to other objects. The second part is a standard mapping for these metadata constructs into Extensible Markup Language (XML) [3]. XML is a flexible text format for data and document interchange standardized by the World Wide Web Consortium (W3C) [4].

The PCA metadata system does not specify the Application Program Interface (API) used by programs to manipulate metadata. Rather, the format of how metadata objects are stored in XML is standardized. This allows developers and users to take advantage of the many free and commercial tools for XML *data binding* [5] to build APIs that are specific to their requirements and implementation language. These tools, given a description of the context, can automatically generate a language-specific API to access and modify the XML content. For instance, a developer writing a compiler in Java could choose to use Java Architecture for XML Binding (JAXB) [6] tools to automatically generate a set of Java classes that directly map to the compiler's metadata. The developer of each tool needing to read or write metadata in XML format will need to acquire a free or commercial tool to generate an API for the language they are using, or they could choose to implement a custom API themselves.

Ultimately, the complete and formal specification of the allowed content for a particular metadata context will be given by an XML *schema*. XML schemas provide a means for defining the structure and content of XML documents. The XML Schema language is a W3C standard [7].

3.1. Metadata Content Description

All metadata content is described in terms of objects and parameters of those objects. Parameters can be various scalar types, other objects, references to other objects, and ordered lists of parameters. This system is capable of describing metadata of arbitrary complexity including graphs of objects. For instance, object references may be used to describe a parent-child hierarchy among the objects in the metadata. Also note that an object may be entirely contained in another object allowing hierarchical design decisions to be reflected in the metadata content if desired.

For each metadata context, the allowed metadata content is documented using whatever format is appropriate for the data, usually a combination of diagrams, tables, and text.

For example, suppose the metadata in an example metadata context must describe an architecture family that has one or more processors and one or more memories connected to a single bus as illustrated in Figure 2.

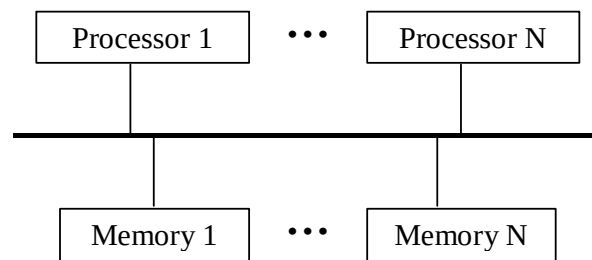


Figure 2. A family of computer architectures with one or more processors and one or more memories connected to a single bus.

One possible way to document the metadata in this context is shown in Figure 3. Parameters have been added to each object for illustration purposes.

Note: The context documentation example in Figure 3 uses a Unified Modeling Language (UML) diagram [8]. UML is a graphical modeling language standardized by the Object Management Group (OMG) [9]. The use of UML diagrams for metadata content documentation is not a requirement of the metadata system. In Figure 3, metadata objects are shown as classes with attributes but no methods. Object parameters are shown using class attributes. Parameters that are references to other objects are represented with associations between classes. Parameter names for object references are given using the UML role name for the association. Lists of object references are used when the association's multiplicity designates the possibility of more than one reference, and all such lists are ordered. An object contained in another object is shown with a composition relation.

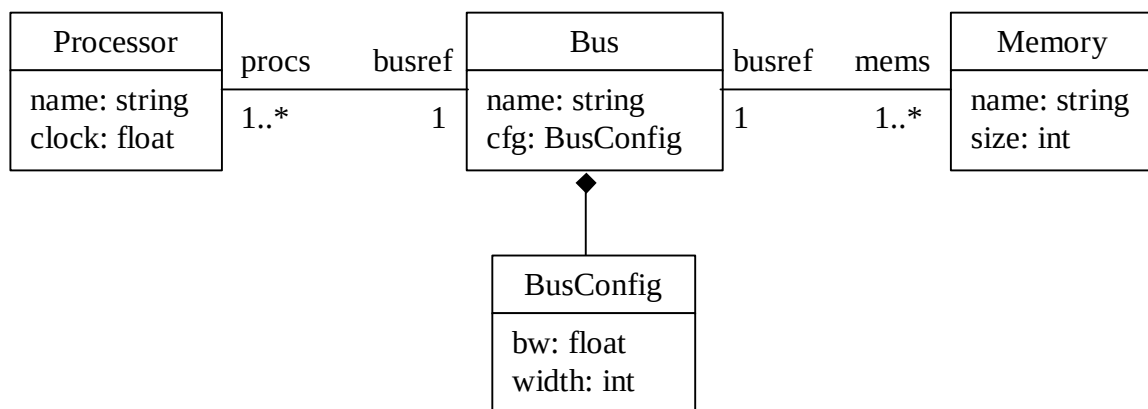


Figure 3. A diagram documenting the metadata context for describing the family of systems shown in Figure 2.

Figure 3 shows that metadata in this context is allowed to consist of objects of four types. Processor objects have three parameters: a string `name`; `clock`, of type `float`; and `busref`, a reference to a `Bus` object. Memory objects have three parameters: a string

`name`; `size`, of type `int`; and `busref`, a reference to a Bus object. Bus objects have four parameters: a string `name`; `procs`, a list of one or more references to Processor objects; `mems`, a list of one or more references to Memory objects; and `cfg`, of type `BusConfig` where a `BusConfig` is an object with two parameters of its own: `bw`, of type `float`, and `width`, of type `int`.

Objects do not necessarily have references to other objects. For instance, an object may always be contained in other objects, as is the `BusConfig` object in Figure 3. Or, one or more objects might be used solely to contain parameters describing global metadata values.

3.2. Metadata XML Representation

As pointed out above, the PCA program requires the transfer of metadata from one application to another. To facilitate this, as well as to minimize development time and the use of program resources, the transfer format must be standardized. There are a number of reasons why this format should be text; among them, it allows the metadata to be created and modified with standard text manipulation tools, and it provides a human-readable document for inspection and problem solving. Fortunately, there exists a widely used, standard text format, XML, standardized by the W3C.

PCA metadata will be represented in an XML format that maps PCA context documentation to XML in a standard way. This allows a user to easily predict the XML form from the documentation, or to derive the metadata context from the XML. The choice of XML also allows the use of many available tools to manipulate PCA metadata.

XML requires that there be exactly one top-level element, defined here as `<pca_metadata>`. An additional sub-element will be used to identify PCA metadata contexts (for example, `<example_context>`). Top-level metadata objects appear as elements within this context element where the object element start tag corresponds to the object type name in the context documentation. Each object element has an attribute called `ident` with a string value that uniquely identifies the object and that may be used to reference this object. This attribute is associated with the `ID/IDREF` feature of the XML Document Type Description (DTD) format and the XML Schema specification.

Within an object element, each object parameter is described using an element with a start tag that is the same as the parameter name. Scalar parameter values are given as text in the content of the element, and may be strings, integers, and floating point numbers. Object references are empty elements that have an attribute `objref` with a string value the same as that of the `ident` attribute of the object being referenced. A parameter that is another object (not an object reference) is represented as an element nested within its containing object element. A parameter may appear more than once within an object to designate a variable list of values. The order of elements in a list is maintained when reading and writing the metadata.

Figure 4 summarizes at a high level how metadata information corresponds to the elements of an XML document.

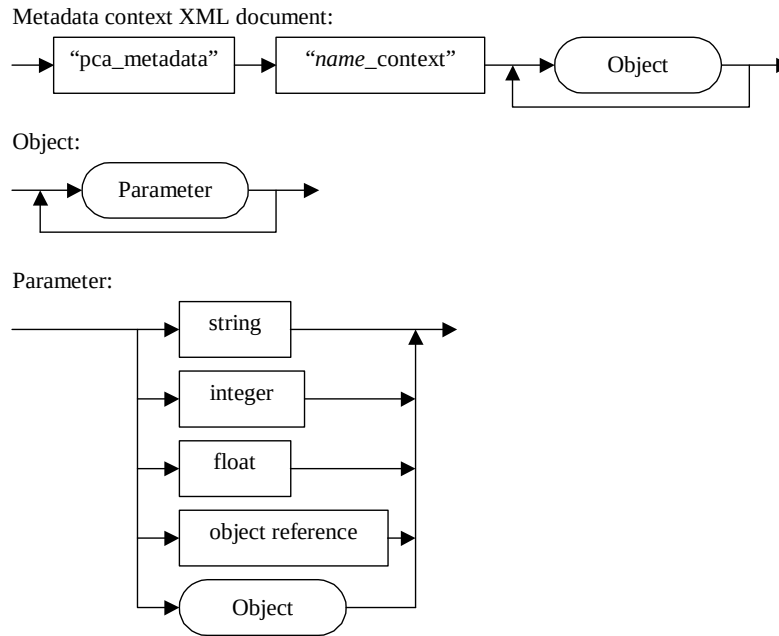


Figure 4. A high-level depiction of how an XML document is used to represent a metadata context. Each box is an XML element. Rectangular boxes are terminals (they have no further description) while rounded boxes have an additional sub-diagram describing their contents.

4. An XML Example

Suppose it is necessary to describe a specific system, from the family of systems shown in Figure 1 and Figure 3, that has exactly two processors and two memories. The XML file containing the metadata for this system is shown in Figure 5.

```
<?xml version="1.0"?>
<pca_metadata>
  <example_context>
    <Processor ident="proc001" >
      <name> proc1 </name>
      <busref objref="bus001" />
      <clock> 1.5e9 </clock>
    </Processor>
    <Processor ident="proc002" >
      <name> proc2 </name>
      <busref objref="bus001" />
      <clock> 2.0e9 </clock>
    </Processor>
    <Bus ident="bus001" >
      <name> bus </name>
      <procs objref="proc001" />
      <procs objref="proc002" />
      <mems objref="mem001" />
      <mems objref="mem002" />
      <cfg>
        <BusConfig ident="busconfig001" >
          <bw> 1.0e9 </bw>
          <width> 128 </width>
        </BusConfig>
      </cfg>
    </Bus>
    <Memory ident="mem001" >
      <name> mem1 </name>
      <busref objref="bus001" />
      <size> 512 </size>
    </Memory>
    <Memory ident="mem002" >
      <name> mem2 </name>
      <busref objref="bus001" />
      <size> 512 </size>
    </Memory>
  </example_context>
</pca_metadata>
```

Figure 5. XML description of the metadata for a two-processor, two-memory system. A diagram describing the metadata context for this family of systems is shown in Figure 3.

If it is ever necessary to commingle PCA metadata files with XML files from other sources, an XML *namespace* may be used to ensure that there are no name conflicts. This is done by adding a “pca” prefix and colon to all PCA elements and attributes, and then identifying the namespace prefix by adding an attribute to the `pca_metadata` element. The beginning of the XML metadata file in Figure 5 would then be:

AN XML EXAMPLE

```
<pca:pca_metadata xmlns:pca="http://www.darpa.mil/ipto/pca/">
  <pca:example_context>
    <pca:Processor pca:ident="proc001">
      ...
```

5. References

1. The Polymorphous Computing Architectures (PCA) Program, Information Processing Technology Office (IPTO), Defense Advanced Research Projects Agency (DARPA), www.darpa.mil/ipto/Programs/pca/index.htm.
2. The PCA Morphware Forum, www.morphware.org.
3. The Extensible Markup Language (XML), World Wide Web Consortium, www.w3.org/XML.
4. The World Wide Web Consortium (W3C), www.w3.org.
5. Ronald Bourret, "XML Data Binding Resources," October 6, 2003. Available at www.rpbouret.com/xml/XMLDataBinding.htm.
6. Java Architecture for XML Binding (JAXB), java.sun.com/xml/jaxb.
7. XML Schema, World Wide Web Consortium, www.w3.org/XML/Schema.
8. The Unified Modeling Language (UML), Object Management Group, www.omg.org/technology/documents/formal/uml.htm.
9. The Object Management Group (OMG), www.omg.org.

6. Appendix: Additional Metadata System Issues

This section discusses additional issues related to the metadata system. These are outside the scope of this metadata system document but are recorded here for reference.

Procedural interface. Some PCA participants expressed concern that the metadata system should be procedural rather than declarative. The thought was that this would give the compiler direct access to metadata values allowing for the possibility of some sophisticated optimizations. It appears, however, that this capability can be layered above the current declarative approach. For instance, a compiler could recognize the API calls used by its source language to access metadata, and then read the metadata values directly. Further, this capability would be implementation specific and so it is not suitable for standardization by the PCA program. Consequently, this standard metadata system does not further address the procedural concepts.

Change notification (R6). The requirement that a running PCA application be able to request notification of a change in the value of a metadata item is not directly addressed by this standard as this document does not standardize the run-time API. Nevertheless, it is expected that this capability will be provided by the various run-time interfaces provided by PCA vendors. The notification requirement may mean that tool developers will have to extend an automatically generated API, as some XML data-binding tools may not include this feature.

Version compatibility (R2). In practical use, programs that have previously been developed for an earlier definition of a metadata context will have to access metadata from a newer, updated context. If the metadata format were too rigid, this scenario would cause the program to fail, an unacceptable result. The PCA XML format described here is general enough to allow new types of metadata to be added without causing the XML parsers to fail. Many of the data binding tools will validate that the XML conforms to the schema of a particular context, but they also allow this validation to be skipped in cases where it is undesirable.

Specification time. Some metadata values may be specified early in development, and others may not be known until, say, application execution time. The runtime system must provide a non-fatal way for a program to determine whether a metadata value exists. If additional information is available to the program designer that tells when in the process a metadata value will be specified, that information is assigned to additional metadata items on a case-by-case basis.

Persistence of changes. There is no underlying database that preserves metadata values across program invocations. If a program changes a metadata value, the change does not persist after that program terminates unless the metadata context is explicitly written to a file.

Scope. A program may consist of multiple threads, possibly running on multiple processors. It is possible that a metadata item could have a different value when read by different threads (*e.g.*, processor clock speed, thread role). A change to a metadata value

made by one thread need not be propagated to other threads and processes in the application.

Read-only values. Certain values in a metadata context may be defined by the context documentation to be read only. However, there need not be any runtime support for this concept. It is up to the application writer to manipulate metadata within the semantics specified by the context documentation. The result of attempting to write a read-only value is implementation dependent.

APPENDIX C

Last Available Version of the Morphware Stable Interface Document

PCA MACHINE MODEL

Version 1.2

Draft 4

January 2007

PCA Machine Model

Version 1.2
Draft 4

January 2007



©2007 Georgia Tech Research Corporation, all rights reserved.

A non-exclusive, non-royalty bearing license is hereby granted to all persons to copy, modify, distribute and produce derivative works for any purpose, provided that this copyright notice and following disclaimer appear on all copies: THIS LICENSE INCLUDES NO WARRANTIES, EXPRESSED OR IMPLIED, WHETHER ORAL OR WRITTEN, WITH RESPECT TO THE SOFTWARE OR OTHER MATERIAL INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF PERFORMANCE OR DEALING, OR FROM USAGE OR TRADE, OR OF NON-INFRINGEMENT OF ANY PATENTS OF THIRD PARTIES. THE INFORMATION IN THIS DOCUMENT SHOULD NOT BE CONSTRUED AS A COMMITMENT OF DEVELOPMENT BY ANY OF THE ABOVE PARTIES.

This material is based in part upon work supported by the U.S. Defense Advanced Research Projects Agency (DARPA) and other agencies of the U.S. Department of Defense (DoD). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or the DoD.

The US Government has a license under these copyrights, and this material may be reproduced by or for the U.S. Government.



Acknowledgements

Authors

The primary author of this document was:

Reservoir Labs, Inc.

The authors wish to thank the members of the Morphware Forum who reviewed and contributed to this document. The authors also thank the Defense Advanced Research Projects Agency (DARPA) for their support of this work.

The Morphware Forum

The Morphware Forum is a joint activity of the participants in DARPA's Polymorphous Computing Architectures (PCA) program, as well as other interested developers of embedded computing hardware, software, and application technology. The purpose of the Morphware Forum is to define an open, portable software environment for the development of high performance applications on PCA platforms. Morphware Forum products and information are available at www.morphware.org.

The following organizations are active members of the Morphware Forum at this writing:

- o Defense Advanced Research Projects Agency
- o Exogi, LLC
- o Georgia Institute of Technology
- o Lockheed Martin Advanced Technology Laboratory (ATL)
- o Lockheed Martin Maritime Systems and Sensors (MS2)
- o Mercury Computer Systems, Inc.
- o Massachusetts Institute of Technology
- o MIT Lincoln Laboratory
- o MITRE
- o Raytheon Corporation
- o Reservoir Labs, Inc.
- o Stanford University
- o University of Southern California Information Sciences Institute
- o University of Texas, Austin
- o U. S. Air Force Research Laboratory, Eglin AFB Site
- o U. S. Air Force Research Laboratory, Rome NY Site
- o U. S. Air Force Research Laboratory, Wright-Patterson AFB Site
- o U.S. Army Tank Automotive Research, Development, and Engineering Center (RDECOM)
- o U.S. Navy SPAWAR Systems Center

Additional contributing organizations include:

- o BAE Systems
- o Black River Systems Company, Inc.
- o Honeywell Space Systems
- o IBM Austin Research Laboratory
- o Lockheed Martin Aerospace
- o Nallatech Ltd.
- o National Reconnaissance Office
- o North Carolina State University
- o Northeastern University
- o Pentum Group, Inc.
- o Protean Devices, Inc.
- o Rose-Hulman Institute of Technology
- o Science & Technology Associates, Inc.
- o University of Maryland
- o University of Pennsylvania
- o Vanderbilt University

Document Change History

Version 1.0 is the first approved version of the document.

Version 2.0 draft 1 is a proposed revision and simplification that reflects the machine model as used by the Reservoir Labs, Inc. *R-Stream* compiler, prototype 2.0.

Version 1.2 makes the following changes:

- Changes the numbering scheme for consistency with the Streaming Virtual Machine (SVM) document. This version 1.2 is the immediate successor to Version 2.0 draft 4.
- Changes the name of the `InitialProcessor` root parameter to `InitialControlProcessor`.
- Defines the `InitialGlobalMemory` root parameter.
- Adds `SupportsCoherency` and `CacheLineSize` fields to Cache memory parameters
- Added the `RAMInLaneBandwidth` parameter.
- Changed the type of the root parameter `IgnoreUnnamedBitfieldAlignment` from `int32` to `boolean`.
- Defined the terms “lane”, “inlane”, and “crosslane” and clarified their use in the memory bandwidth parameters.
- Corrected the definition of crosslinks and uplinks.
- Rewrote the introduction to the network modeling parameters.
- Replaced the single-latency model for crosslinks with a base latency and per-hop latency.
- Updates the XML schema for consistency with above changes
- Numerous formatting and minor editorial changes.

Table Of Contents

Acknowledgements.....	i
Authors.....	i
The Morphware Forum.....	i
Document Change History.....	iii
Table Of Contents.....	iv
List of Acronyms	vi
1. Overview.....	1
1.1. Elements of the PCA Machine Model	1
1.2. Levels.....	2
1.3. Relation to PCA Metadata System	2
2. Objects	3
2.1. Root Object	3
2.2. Level Object.....	3
2.3. Primary Objects	4
2.4. Secondary Objects	4
2.5. Reference Object.....	4
3. Machine Model Parameters	5
3.1. Primary Object Parameters	5
3.2. Root Parameters.....	5
3.3. Level Parameters.....	6
3.4. Processor Parameters	7
3.4.1. User-code processor parameters	8
3.5. Memory Parameters	9
3.6. Network Parameters.....	10
3.7. Link Parameters	11
3.8. Ingredient Parameters	11
3.9. Morph Parameters.....	11
3.9.1. IngredientUse.....	12
3.9.2. LatencyFromMorph	12
3.10. Reference Parameters.....	12

4. Machine Model Naming Convention.....	13
5. References.....	14
6. Index of Object and Parameter Names	15
Appendix: Machine Model XML Schema.....	19

List of Acronyms

ALU	Arithmetic-Logic Unit
API	Application Programming Interface
DARPA	Defense Advanced Research Project Agency
DMA	Direct Memory Access
FIFO	First In, First Out
I/O	Input/Output
MHz	Megahertz
MSI	Morphware Stable Interface
PCA	Polymorphous Computing Architecture
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
SIMD	Single Instruction Multiple Data
SVM	Streaming Virtual Machine
TRIPS	Tera-op, Reliable, Intelligently adaptive Processing System
XML	eXtensible Markup Language

1. Overview

The PCA Machine Model is a generic conceptual model used to describe a specific PCA architecture. The phrase “the machine model” refers to the model. Phrases like “a machine model” or “the TRIPS machine model” refer to a description of a specific architecture using the machine model. The machine model is an element of the Morphware Stable Interface (MSI) [1].

The primary role of the machine model in the MSI is to describe the architecture to software tools common to all PCA architectures, such as a component manager or high-level compiler. Tools common to all PCA architectures are relatively high-level; consequently, the machine model provides a relatively abstract model of the architecture, rather than a finely detailed description.

The goal of the machine model is to describe the architecture in a simple, uniform manner adequate to clearly define functionality and enable accurate performance estimates. Toward this end, it describes capabilities rather than implementation and makes quantitative rather than qualitative distinctions whenever possible.

1.1. Elements of the PCA Machine Model

A machine model contains three kinds of elements:

1. An *ingredient* is an arbitrary, configurable division of underlying hardware. An ingredient can be configured, possibly in concert with other ingredients, into one or more resources.
2. A *resource* is a discrete generic computing element with well-defined functionality. Four classes of resources are defined in the machine model: processors, memories, networks, and links.
3. A *morph* defines a specific set of ingredients that can be configured into a specific set of resources.

For example, suppose a hypothetical “PCAPower” machine model divides the underlying hardware into three ingredients:

- “Foo” ingredient
- “Bar” ingredient
- “Baz” ingredient

The architecture described by the machine model contains two resources that are always present:

- “MX” memory
- “NX” network

The machine model also defines three morphs:

- The “IrregularComp” morph configures the “Foo” ingredient into a “P1000” processor resource.

- The “RegularComp” morph configures the “Bar” and “Baz” ingredients into a “P2000” processor resource.
- The “MaximumPower” morph configures the “Foo”, “Bar”, and “Baz” ingredients into the “P3000” processor resource.

It may not be possible to configure the system to make all of the possible resources available at the same time. The available ingredients and morphs define the possible combinations of resources. For example, in the “PCAPower” machine model, the “RegularComp” and “IrregularComp” morphs can both be configured at the same time since they require disjoint ingredients. However, neither can be configured in conjunction with the “MaximumPower” morph since that morph demands all three ingredients.

1.2. Levels

The machine model also introduces *levels* to organize the resources, ingredients, and morphs to reflect hierarchy and symmetry of the system. A level may contain some number of resources, ingredients, morphs, and possibly nested levels. If a level contains another level, it is said to be the parent of that level, and that level is said to be its child.

Parent-child relationships define a tree of levels. Each machine model has one root level (level with no parent), branch levels (levels with both a parent and one or more children), and leaf levels (levels with no children). For instance, the example machine model could be expanded by grouping the memory “MX” and the network “NX” in a root level “LA”, and grouping the ingredients and morphs in a leaf level “LB”. “LA” is then the parent of “LB.”

A level defines an array of instances. Each level specifies a list of dimensions, and has cumulative dimensions given by those and the concatenated dimensions of all its ancestor levels. All resources, ingredients, and morphs contained in a level have the same dimensions as the level. For example, “LA” might have dimensions [1] and “LB” might have dimensions [4][4]. Level “LB” would have cumulative dimensions [1][4][4]. All the ingredients, morphs, and processors in level “LB” would also have those dimensions.

1.3. Relation to PCA Metadata System

The machine model is an element of the PCA metadata system [2]. The machine model is a conceptual model. The machine model is documented as an XML schema [3] and is included in the appendix of this document.

2. Objects

The machine model is composed of a hierarchy of *objects*. An object is a structure that describes a part of the machine model with a collection of *parameters*. Each parameter may be a simple value (numeric, enumerated, or string), an object, or a list of values or objects. Use of some parameters is conditional on the values of other parameters. For example, the cache parameters for a Memory will not be used unless the **Subtype** parameter for the memory specifies that it is a cache. Some parameters have default values and may be omitted.

The hierarchy of objects consists of one root object and multiple primary, secondary, and reference objects. The *root object* represents the entire machine model. *Primary objects* represent the key elements identified in the introduction: ingredients, resources, and morphs. Some parameters of a primary object may be *secondary objects*, such as the ingredient usage information of a morph. Some parameters of an object may be *reference objects* that refer to objects elsewhere in the hierarchy.

2.1. Root Object

Object	Description
Root	Single root object that represents the entire machine model.

2.2. Level Object

Object	Description
mm_Level	A level object contains groups of related objects. A level may contain another level. Nested levels are used to reflect hierarchy. A level may specify dimensions, indicating an array of instances of the level. Levels with dimensions are used to reflect symmetry.

2.3. Primary Objects

Object	Description
mm_Processor	A processor resource. A processor reads and/or writes data.
mm_Memory	A memory resource. A memory stores data.
mm_Network	A network resource. A network transmits data.
mm_Link	A link resource. A link connects two resources.
mm_Ingredient	An architecture-specific hardware ingredient.
mm_Morph	Defines a set of hardware ingredients that can be configured to produce a set of resources.

2.4. Secondary Objects

Object	Description
mm_IngredientUse	Specifies an ingredient required to configure a morph.
mm_LatencyFromMorph	Specifies minimum latency to configure a morph if one or more ingredients were previously part of another specific morph.

2.5. Reference Object

Object	Description
mm_Ref	Reference to another object.

3. Machine Model Parameters

3.1. Primary Object Parameters

Certain parameters are common to all primary and level objects:

Parameter	Type – Description
Name	string – Globally unique identifier of an object, composed only of alphanumeric characters and underscores.

3.2. Root Parameters

A single root object specifies parameters that apply to the system as a whole, such as the data layout, and the root level of the level hierarchy.

Parameter	Type – Description
AddressSize	int32 – Size of address in bits.
AddressAlignment	int32 – Alignment of address in bits.
WordSize	int32 – Size of a word in bits.
BigEndian	boolean – True if data is Big-Endian.
CharSigned	boolean – True if char type is signed.
SizeChar, SizeShort, SizeInt, SizeLong, SizeLongLong, SizeFloat, SizeDouble, SizeLongDouble	int32 – Size of type in bits.
AlignmentChar, AlignmentShort, AlignmentInt, AlignmentLong, AlignmentLongLong, AlignmentFloat, AlignmentDouble, AlignmentLongDouble	int32 – Alignment of type in bits.
MinStructSize	int32 – Size of minimum size struct in bits.
MinUnionSize	int32 – Size of minimum size union in bits.

Parameter	Type – Description
MinStructAlignment	int32 – Alignment of minimum size struct in bits.
MinUnionAlignment	int32 – Alignment of minimum size union in bits.
OnlyPackBitFields	boolean – True if only bitfields are packed in structures and unions.
IgnoreUnnamedBitfieldAlignment	boolean – Ignore unnamed bitfields when packing
DefaultMorph	List of String – The names of the morph instances to configure at boot.
InitialControlProcessor	String – The name of the processor instance on which to execute <code>main()</code> . The processor must be configured by the default morph.
InitialGlobalMemory	String – The name of the memory instance on which to locate the control processor's stack and heap. The memory must be configured by the default morph and accessible by the initial control processor.
RootLevel	mm_Level – Root level of the hierarchy.

3.3. Level Parameters

Each level specifies one or more dimensions and a set of resources, nested levels, ingredients, and morphs.

Parameter	Type – Description
Dimension	List of int32 – Dimension of the level.
Processor	List of mm_Processor – A processor contained in the level.
Memory	List of mm_Memory – A memory contained in the level.
Network	List of mm_Network – A network contained in the level.
Link	List of mm_Link – A link contained in the level.
Level	List of mm_Level – A nested level.
Ingredient	List of mm_Ingredient – An ingredient contained in the level.
Morph	List of mm_Morph – A morph contained in the level.

3.4. Processor Parameters

A processor represents anything that consumes, produces, or moves data. For instance, a processor object can represent a RISC core, a DMA engine, or an I/O device.

A Single Instruction, Multiple Data (SIMD) processing unit performs some number N of operations in parallel. A *lane* corresponds to an individual operation of the group of N within the SIMD unit or, in other words, one of the N parallel paths through the SIMD unit. Typically, the path between memory and the SIMD unit transmits data at a higher rate when the addresses of the data values being read or written are *aligned* with the SIMD unit. An address is said to be aligned with the SIMD unit when that address is an integer multiple of the quantity $N \cdot W$, where W is the data width in bytes and N is the number of lanes N in the SIMD unit. To model the memory bandwidth difference between aligned and unaligned accesses, *inlane* accesses refer to aligned memory references, while *crosslane* accesses refer to unaligned memory accesses.

Parameter	Type – Description
RequiredMaster	mm_Ref to mm_Processor [default: none] – If specified, processor can only execute kernels requested by the specified master processor.
SupportsUserCode	boolean [default: false] – True if the processor can execute arbitrary C code subject to the limits of its supported types and memory hierarchy.
SupportsBlockDMA	boolean [default: false] – True if the processor can execute block DMA built-in kernels per the SVM API specification. See reference [4].
SupportsStreamDMA	boolean [default: false] – True if the processor can execute streaming DMA built-in kernels per the SVM API specification. See reference [4]
KernelOverhead	int32 – Number of processor cycles required to start a new kernel.

3.4.1. User-code processor parameters

The following parameters are used only for processors running user code.

Parameter	Type – Description
SupportsChar, SupportsShort, SupportsInt, SupportsLong, SupportsLongLong, SupportsFloat, SupportsDouble, SupportsLongDouble	boolean [default: true] – True if the processor can execute C code that uses the specified type.
Freq	float – Frequency of processor in cycles/nanosecond. For processors that support a configurable processor frequency, this parameter provides a list of valid processor frequency settings.
NumALUs	int32 – Number of ALUs in the processor.
ALUOccupancy	int32 – Average ALU occupancy achieved.
NumSIMDLanes	int32 – Number of parallel SIMD execution engines. Also implicitly a factor in the alignment at which memory access is inlane per RAMInLaneBandwidth memory parameter defined in Section 0.
SIMDSubword	boolean – True if processor can execute a number of operations equal to (largest type width / actual type width) when utilizing smaller data widths.
NumRegs	int32 – Number of registers in the processor.
SupportsSpilling	boolean – If True, processor can spill values to any RAM to which it is connected. If False, all values must fit in registers.
ISize	int32 – Size of an instruction (one operation for one ALU) in bits.
IStrict	boolean – True if all instructions for a kernel must fit in the instruction cache.

3.5. Memory Parameters

A memory represents anything that stores data.

Parameter	Type – Units Interpretation
Subtype	{FIFO, RAM, CACHE} – Mode of operation supported by this memory resource.
Size	int64 – Size of the memory (number of bytes available to store instructions and/or data).
FIFO parameters	
FIFO Peek	int32 – The number of bytes in FIFO that can be peek'ed.
RAM parameters	
RAMLinearBandwidth	float – Bandwidth for linear accesses into memory (normal stream pop/push) in bytes/nanosecond.
RAMRandomBandwidth	float [default: RAMLinearBandwidth] – Bandwidth for random access into memory (e.g., unpredictable peek, block read/write, etc.) in bytes/nanosecond.
RAMInlaneBandwidth	float [default: RAMLinearBandwidth] – Bandwidth for aligned (within lanes) access into memory for SIMD architectures in bytes/nanosecond.
RAMCrosslaneBandwidth	float [default: RAMLinearBandwidth] – Bandwidth for random access into memory (unaligned, across lanes) for SIMD architectures in bytes/nanosecond.
Cache parameters	
CacheContent	{DATA, INSTRUCTIONS, UNIFIED} – Specifies allowable contents of cache.
SupportsCoherency	boolean – If True, hardware maintains cache coherence with all other caches that support coherency.
CacheLineSize	int32 – The size of a cache line (bytes).
CacheMissRate	float – The cache miss rate.
CacheMissSize	int32 – The average amount of data transmitted per cache miss (bytes).

3.6. Network Parameters

A network transmits data between resources.

A network that transmits data between two resources is termed a *crosslink* if

- The resources are contained in levels that have a common ancestor, and
- That ancestor level is a parent of the level that contains the network.

Otherwise, the network is termed an *uplink*.

Parameters are provided to characterize a very coarse model of the network that transmits messages of a relatively uniform size. *Traffic*, expressed in bytes, refers to a message of one or more bytes transmitted on a network. The *maximum traffic* (capacity) of a network, expressed in terms of bytes per nanosecond, is consumed by the *base traffic* imposed by the application plus a *per-byte traffic* overhead on that traffic. The per-byte overhead may differ between uplinks and crosslinks. The network topology for crosslinks is assumed to be a two-dimensional mesh, and the number of hops between two processors in the network equals the Manhattan distance between them. A *base latency* for messaging is augmented by a *per-hop latency*.

Parameter	Type – Description
MaximumTraffic	float – Maximum traffic per nanosecond allowed by network, in bytes.
UplinkBaseLatency	float – Base latency to transmit data, in nanoseconds, on an uplink network.
UplinkHopLatency	float – Per-hop latency to transmit data, in nanoseconds, on an uplink network.
UplinkTrafficPerByte	float – Overhead traffic to transmit a byte, per byte, in bytes, on an uplink network..
CrosslinkBaseLatency	float – Base latency to transmit data, in nanoseconds, on a crosslink network.
CrosslinkHopLatency	float – Per-hop latency to transmit data, in nanoseconds, on a crosslink network.
CrosslinkTrafficPerByte	float – Overhead traffic to transmit a byte, per byte, in bytes, on a crosslink network.

3.7. Link Parameters

A link connects one resource to another.

Parameter	Type – Description
Sender	mm_Ref to resource – Resource that can send data to this link.
Receiver	mm_Ref to resource – Resource that can receive data from this link.
Bidirectional	Boolean – True if the link is bidirectional.

3.8. Ingredient Parameters

Ingredients are arbitrary portions of hardware that can be configured into resources by morphs.

Parameter	Type – Description
Count	List of int32 – Number of identical, equivalent instances of the ingredient available.

3.9. Morph Parameters

A morph expresses a set of ingredients that can be configured into a set of resources.

Switching between morphs that use one or more of the same ingredients requires a certain latency, which depends on the morphs that previously used the ingredient(s). During this time, the resources contained in either morph must be unused. The state of resources that are not contained in both morphs are lost.

This definition creates a distinction between morphs and *settings*: changing morphs (*i.e.*, from a threaded to a streaming processor) destroys state, while changing settings (*i.e.*, changing a processor's frequency from 600 MHz to 700 MHz) does not.

Parameter	Type – Description
IngredientUse	List of mm_IngredientUse – Defines ingredients used by the morph.
LatencyFromMorph	List of mm_LatencyFromMorph – Defines latency of morph based on the morphs that the ingredients previously belonged to.
Processor	List of mm_Processor – A processor produced by the morph.
Memory	List of mm_Memory – A memory produced by the morph.
Network	List of mm_Network – A network produced by the morph.
Link	List of mm_Link – A link produced by the morph.
Level	List of mm_Level – A level produced by the morph.

3.9.1. *IngredientUse*

Parameter	Type – Description
Ingredient	mm_Ref to mm_Ingedient – Ingredient used.
Count	int32 – Number of ingredients used.

3.9.2. *LatencyFromMorph*

Parameter	Type – Description
FromMorph	mm_Ref to mm_Morph – Morph that previously contained the ingredients used by this morph.
Latency	int32 – Minimum latency required to configure this morph if any of the ingredients were previously used by FromMorph (nanoseconds).

3.10. *Reference Parameters*

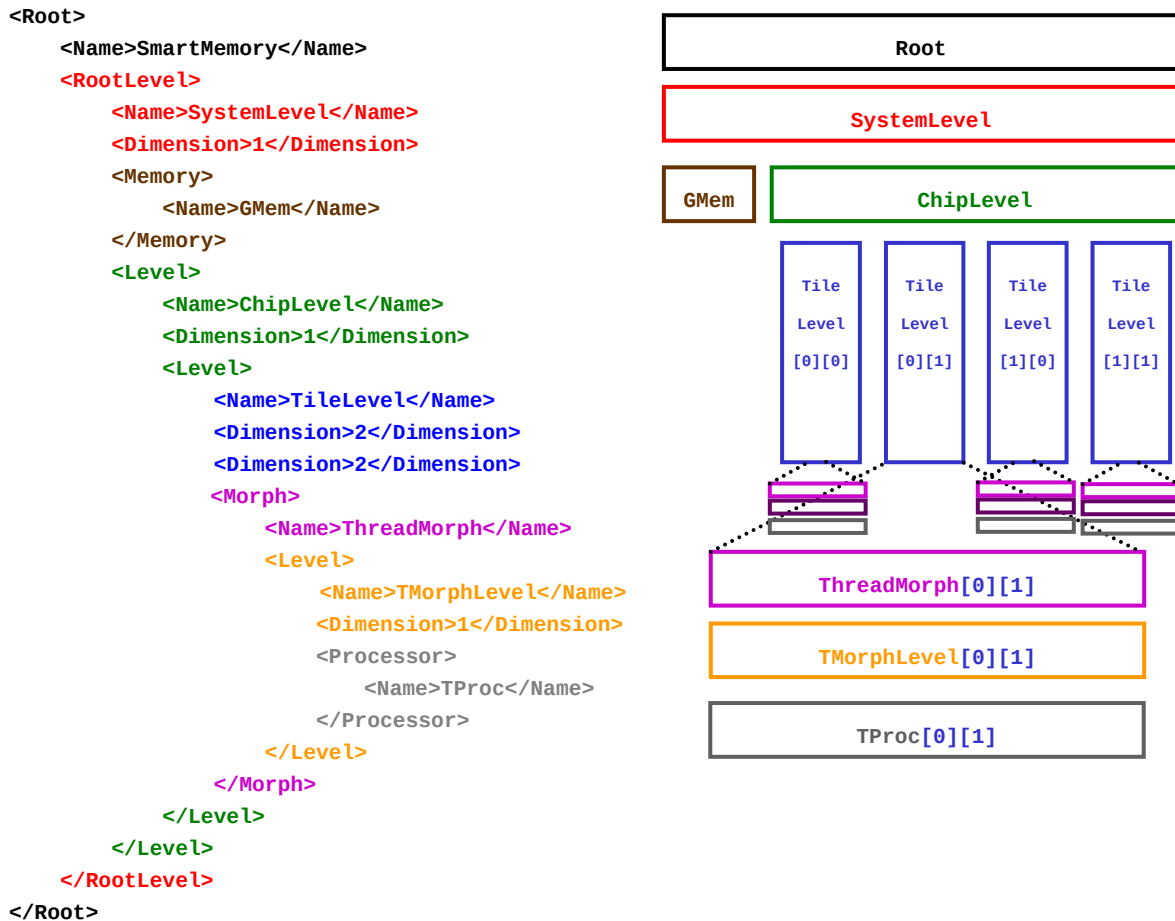
Every object in the machine model has a unique name. A reference object refers to another object by this name.

Parameter	Type – Units Interpretation
Name	string – Name of object that is referenced.

4. Machine Model Naming Convention

The machine model defines a unique name for each *instance* of a machine model object. This enables users of the machine model to refer to specific instances. A machine model consists of a hierarchy of level objects. Each level represents an array of instances of that level, with some specified dimensions. When one level nests inside another, it has dimensions given by the concatenation of its parent level's dimensions and its own dimensions. Each ingredient, morph, and resource that is contained in a level represents an array of instances with the same dimensions as the parent level. Each instance of an object has a name given by the concatenation of the object name and instance indices, each enclosed in square brackets. For example, an instance of “StreamProc” might be named `StreamProc [2][15]`. For simplicity, indices into any unit size dimensions are omitted.

The following figure shows a more complicated example with an abbreviated form of the XML Machine Model on the left and a diagram showing all of the object instances with their names on the right. Color coding indicates which objects in the XML define which instances in the diagram, and which levels define which indices.



5. References

1. “Introduction to Morphware: Software Architecture for Polymorphous Computing Architectures.” Version 1.0, February 23, 2004. Available at www.morphware.org.
2. “The PCA Metadata System”, Georgia Institute of Technology and SPAWAR Systems Center San Diego, March. 2, 2004. Available at www.morphware.org.
3. XML Schema, World Wide Web Consortium, www.w3.org/XML/Schema.
4. “Streaming Virtual Machine Specification”, Version 1.2 draft 5, January 2007. Available at www.morphware.org.

6. Index of Object and Parameter Names

AddressAlignment	5
AddressSize.....	5
AlignmentChar.....	5
AlignmentDouble.....	5
AlignmentFloat	5
AlignmentInt	5
AlignmentLong	5
AlignmentLongDouble	5
AlignmentLongLong.....	5
AlignmentShort.....	5
ALUOccupancy	8
Bidirectional.....	11
BigEndian	5
CacheContent.....	9
CacheLineSize	9
CacheMissRate	9
CacheMissSize	9
CharSigned.....	5
Count.....	11, 12
CrosslinkBaseLatency.....	10
CrosslinkHopLatency	10
CrosslinkTrafficPerByte	10
DefaultMorph.....	6
Dimension	6
FIFOPeek	9
Freq	8
FromMorph	12
IgnoreUnnamedBitfieldAlignment	6
Ingredient	6, 12
IngredientUse	12

InitialControlProcessor	6
InitialGlobalMemory	6
ISize	8
IStruct.....	8
KernelOverhead	7
Latency.....	12
LatencyFromMorph	12
Level	6, 12
Link.....	6, 12
MaximumTraffic.....	10
Memory	6, 12
MinStructAlignment	6
MinStructSize	5
MinUnionAlignment.....	6
MinUnionSize	5
mm_Ingredient.....	4
mm_IngredientUse.....	4
mm_LatencyFromMorph.....	4
mm_Level	3
mm_Link.....	4
mm_Memory.....	4
mm_Morph	4
mm_Network	4
mm_Processor.....	4
mm_Ref.....	4
Morph.....	6, 12
Name	5, 12
Network.....	6, 12
NumALUs.....	8
NumRegs.....	8
NumSIMDLanes	8
OnlyPackBitFields	6

Processor	6, 12
RAMCrosslaneBandwidth	9
RAMInlaneBandwidth	9
RAMLinearBandwidth	9
RAMRandomBandwidth	9
Receiver	11
RequiredMaster	7
Root	3
RootLevel	6
Sender	11
SIMDSubword	8
Size	9
SizeChar	5
SizeDouble	5
SizeFloat	5
SizeInt	5
SizeLong	5
SizeLongDouble	5
SizeLongLong	5
SizeShort	5
Subtype	9
SupportsBlockDMA	7
SupportsChar	8
SupportsCoherency	9
SupportsDouble	8
SupportsFloat	8
SupportsInt	8
SupportsLong	8
SupportsLongDouble	8
SupportsLongLong	8
SupportsShort	8
SupportsSpilling	8

SupportsStreamDMA..... 7

SupportsUserCode 7

UplinkBaseLatency 10

UplinkHopLatency..... 10

UplinkTrafficPerByte 10

WordSize..... 5

Appendix: Machine Model XML Schema

Following is the complete XML Schema for the PCA Machine Model.

```
<?xml version="1.0"?>
<!-- Copyright (c) 2004 Reservoir Labs, Inc. -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Root">
    <xs:complexType>
      <xs:all>
        <xs:element name="Name" type="xs:string" />
        <xs:element name="AddressSize" type="xs:int" />
        <xs:element name="AddressAlignment" type="xs:int" />
        <xs:element name="WordSize" type="xs:int" />
        <xs:element name="BigEndian" type="xs:boolean" />
        <xs:element name="CharSigned" type="xs:boolean" />
        <xs:element name="AlignmentChar" type="xs:int" />
        <xs:element name="AlignmentShort" type="xs:int" />
        <xs:element name="AlignmentInt" type="xs:int" />
        <xs:element name="AlignmentLong" type="xs:int" />
        <xs:element name="AlignmentLongLong" type="xs:int" />
        <xs:element name="AlignmentFloat" type="xs:int" />
        <xs:element name="AlignmentDouble" type="xs:int" />
        <xs:element name="AlignmentLongDouble" type="xs:int" />
        <xs:element name="SizeChar" type="xs:int" />
        <xs:element name="SizeShort" type="xs:int" />
        <xs:element name="SizeInt" type="xs:int" />
        <xs:element name="SizeLong" type="xs:int" />
        <xs:element name="SizeLongLong" type="xs:int" />
        <xs:element name="SizeFloat" type="xs:int" />
        <xs:element name="SizeDouble" type="xs:int" />
        <xs:element name="SizeLongDouble" type="xs:int" />
        <xs:element name="MinStructSize" type="xs:int" />
        <xs:element name="MinUnionSize" type="xs:int" />
        <xs:element name="MinStructAlignment" type="xs:int" />
        <xs:element name="MinUnionAlignment" type="xs:int" />
        <xs:element name="OnlyPackBitFields" type="xs:boolean" />
        <xs:element name="IgnoreUnnamedBitfieldAlignment" type="xs:boolean" />
        <!-- <xs:element name="AlignmentBitfield" type="xs:int" /> -->
        <!-- <xs:element name="BitfieldCrossWord" type="xs:boolean" /> -->
        <xs:element name="RootLevel" type="mm_Level"/>
      </xs:all>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="mm_Level">
    <xs:sequence>
      <xs:element name="Name" type="xs:string" />
      <xs:element name="Dimension" type="xs:int" minOccurs="1"
maxOccurs="unbounded" />
      <xs:element name="Processor" type="mm_Processor" minOccurs="0"
maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

```

        <xs:element name="Memory" type="mm_Memory" minOccurs="0"
maxOccurs="unbounded" />
        <xs:element name="Network" type="mm_Network" minOccurs="0"
maxOccurs="unbounded" />
        <xs:element name="Link" type="mm_Link" minOccurs="0" maxOccurs="unbounded"
/>

        <xs:element name="Level" type="mm_Level" minOccurs="0"
maxOccurs="unbounded" />
        <xs:element name="Ingredient" type="mm_Ingredient" minOccurs="0"
maxOccurs="unbounded" />
        <xs:element name="Morph" type="mm_Morph" minOccurs="0"
maxOccurs="unbounded" />
    </xs:sequence>
</xs:complexType>
<xs:complexType name="mm_Processor">
    <xs:sequence>
        <xs:element name="Name" type="xs:string" />
        <xs:element name="RequiredMaster" type="mm_Ref" minOccurs="0" />
        <xs:element name="SupportsUserCode" type="xs:boolean" minOccurs="0" />
        <xs:element name="SupportsBlockDMA" type="xs:boolean" minOccurs="0" />
        <xs:element name="SupportsStreamDMA" type="xs:boolean" minOccurs="0" />
        <xs:element name="KernelOverhead" type="xs:int" />
        <xs:element name="SupportsChar" type="xs:boolean" minOccurs="0" />
        <xs:element name="SupportsShort" type="xs:boolean" minOccurs="0" />
        <xs:element name="SupportsInt" type="xs:boolean" minOccurs="0" />
        <xs:element name="SupportsLong" type="xs:boolean" minOccurs="0" />
        <xs:element name="SupportsLongLong" type="xs:boolean" minOccurs="0" />
        <xs:element name="SupportsFloat" type="xs:boolean" minOccurs="0" />
        <xs:element name="SupportsDouble" type="xs:boolean" minOccurs="0" />
        <xs:element name="SupportsLongDouble" type="xs:boolean" minOccurs="0" />
        <xs:element name="Freq" type="xs:float" minOccurs="0" />
        <xs:element name="NumALUs" type="xs:int" minOccurs="0" />
        <xs:element name="ALUOccupancy" type="xs:int" minOccurs="0" />
        <xs:element name="NumSIMDLanes" type="xs:int" minOccurs="0" />
        <xs:element name="SIMDSubword" type="xs:boolean" minOccurs="0" />
        <xs:element name="NumRegs" type="xs:int" minOccurs="0" />
        <xs:element name="SupportsSpilling" type="xs:boolean" />
        <xs:element name="ISize" type="xs:int" minOccurs="0" />
        <xs:element name="IStrict" type="xs:boolean" minOccurs="0" />
    </xs:sequence>
</xs:complexType>
<xs:complexType name="mm_Memory">
    <xs:sequence>
        <xs:element name="Name" type="xs:string" />
        <xs:element name="Subtype">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="FIFO" />
                    <xs:enumeration value="RAM" />
                    <xs:enumeration value="CACHE" />
                </xs:restriction>
            </xs:simpleType>
        </xs:element>
        <xs:element name="Size" type="xs:long" />
    </xs:sequence>
</xs:complexType>

```

```

<xs:element name="FIFOpeek" type="xs:int" minOccurs="0" />
<xs:element name="RAMLinearBandwidth" type="xs:float" minOccurs="0" />
<xs:element name="RAMRandomBandwidth" type="xs:float" minOccurs="0" />
<xs:element name="RAMInlineBandwidth" type="xs:float" minOccurs="0" />
<xs:element name="RAMCrosslaneBandwidth" type="xs:float" minOccurs="0" />
<xs:element name="CacheContent" minOccurs="0">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="DATA" />
      <xs:enumeration value="INSTRUCTIONS" />
      <xs:enumeration value="UNIFIED" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="SupportsCoherence" type="xs:boolean" minOccurs="0" />
<xs:element name="CacheLineSize" type="xs:int" minOccurs="0" />
<xs:element name="CacheMissRate" type="xs:float" minOccurs="0" />
<xs:element name="CacheMissData" type="xs:int" minOccurs="0" />
</xs:sequence>
</xs:complexType>
<xs:complexType name="mm_Network">
  <xs:sequence>
    <xs:element name="Name" type="xs:string" />
    <xs:element name="MaximumTraffic" type="xs:float" />
    <xs:element name="UplinkBaseLatency" type="xs:float" />
    <xs:element name="UplinkHopLatency" type="xs:float" />
    <xs:element name="UplinkTrafficPerByte" type="xs:float" />
    <xs:element name="CrosslinkBaseLatency" type="xs:float" />
    <xs:element name="CrosslinkHopLatency" type="xs:float" />
    <xs:element name="CrosslinkTrafficPerByte" type="xs:float" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="mm_Link">
  <xs:sequence>
    <xs:element name="Name" type="xs:string" />
    <xs:element name="Sender" type="mm_Ref" />
    <xs:element name="Receiver" type="mm_Ref" />
    <xs:element name="Bidirectional" type="xs:boolean" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="mm_Ingredient">
  <xs:sequence>
    <xs:element name="Name" type="xs:string" />
    <xs:element name="Count" type="xs:int" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="mm_Morph">
  <xs:sequence>
    <xs:element name="Name" type="xs:string" />
    <xs:element name="IngredientUse" type="mm_IngredientUse" minOccurs="0"
maxOccurs="unbounded" />
    <xs:element name="LatencyFromMorph" type="mm_LatencyFromMorph"
minOccurs="0" maxOccurs="unbounded" />

```



```
        <xs:element name="Processor" type="mm_Processor" minOccurs="0"
maxOccurs="unbounded" />
        <xs:element name="Memory" type="mm_Memory" minOccurs="0"
maxOccurs="unbounded" />
        <xs:element name="Network" type="mm_Network" minOccurs="0"
maxOccurs="unbounded" />
        <xs:element name="Link" type="mm_Link" minOccurs="0" maxOccurs="unbounded"
/>
        <xs:element name="Level" type="mm_Level" minOccurs="0"
maxOccurs="unbounded" />
    </xs:sequence>
</xs:complexType>
<xs:complexType name="mm_IngredientUse">
    <xs:sequence>
        <xs:element name="Ingredient" type="mm_Ref" />
        <xs:element name="Count" type="xs:int" />
    </xs:sequence>
</xs:complexType>
<xs:complexType name="mm_LatencyFromMorph">
    <xs:sequence>
        <xs:element name="FromMorph" type="mm_Ref" />
        <xs:element name="Latency" type="xs:float" />
    </xs:sequence>
</xs:complexType>
<xs:complexType name="mm_Ref">
    <xs:sequence>
        <xs:element name="Name" type="xs:string" />
    </xs:sequence>
</xs:complexType>
</xs:schema>
```

This Page Deliberately Left Blank



©2007 Georgia Tech Research Corporation, all rights reserved.



APPENDIX D

Last Available Version of the Morphware Stable Interface Document

STREAMING VIRTUAL MACHINE SPECIFICATION

Version 1.2

Draft 5

January 2007

Streaming Virtual Machine Specification

Version 1.2
Draft 5

January 2007



©2007 Georgia Tech Research Corporation, all rights reserved.

A non-exclusive, non-royalty bearing license is hereby granted to all persons to copy, modify, distribute and produce derivative works for any purpose, provided that this copyright notice and following disclaimer appear on all copies: THIS LICENSE INCLUDES NO WARRANTIES, EXPRESSED OR IMPLIED, WHETHER ORAL OR WRITTEN, WITH RESPECT TO THE SOFTWARE OR OTHER MATERIAL INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF PERFORMANCE OR DEALING, OR FROM USAGE OR TRADE, OR OF NON-INFRINGEMENT OF ANY PATENTS OF THIRD PARTIES. THE INFORMATION IN THIS DOCUMENT SHOULD NOT BE CONSTRUED AS A COMMITMENT OF DEVELOPMENT BY ANY OF THE ABOVE PARTIES.

This material is based in part upon work supported by the U.S. Defense Advanced Research Projects Agency (DARPA) and other agencies of the U.S. Department of Defense (DoD). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or the DoD.

The US Government has a license under these copyrights, and this material may be reproduced by or for the U.S. Government.



Acknowledgements

Authors

The primary authors of this document were:

Peter Mattson	Reservoir Labs, Inc.
Bill Thies	Massachusetts Institute of Technology

Additional major authors were:

Lance Hammond	Stanford University
Michael Vahey	Raytheon

Additional substantial contributions were made by the following organizations:

University of Southern California Information Sciences Institute
University of Texas at Austin
IBM Austin Research Laboratory
Georgia Institute of Technology

The authors wish to thank the members of the Morphware Forum who reviewed and contributed to this document. The authors also thank the Defense Advanced Research Projects Agency (DARPA) for their support of this work.

The Morphware Forum

The Morphware Forum is a joint activity of the participants in DARPA's Polymorphous Computing Architectures (PCA) program, as well as other interested developers of embedded computing hardware, software, and application technology. The purpose of the Morphware Forum is to define an open, portable software environment for the development of high performance applications on PCA platforms. Morphware Forum products and information are available at www.morphware.org.

The following organizations are active members of the Morphware Forum at this writing:

- o Defense Advanced Research Projects Agency
- o Exogi, LLC
- o Georgia Institute of Technology
- o Lockheed Martin Advanced Technology Laboratory (ATL)
- o Lockheed Martin Maritime Systems and Sensors (MS2)
- o Mercury Computer Systems, Inc.
- o Massachusetts Institute of Technology
- o MIT Lincoln Laboratory

- o MITRE
- o Raytheon Corporation
- o Reservoir Labs, Inc.
- o Stanford University
- o University of Southern California Information Sciences Institute
- o University of Texas, Austin
- o U. S. Air Force Research Laboratory, Eglin AFB Site
- o U. S. Air Force Research Laboratory, Rome NY Site
- o U. S. Air Force Research Laboratory, Wright-Patterson AFB Site
- o U.S. Army Tank Automotive Research, Development, and Engineering Center (RDECOM)
- o U.S. Navy SPAWAR Systems Center

Additional contributing organizations include:

- o BAE Systems
- o Black River Systems Company, Inc.
- o Honeywell Space Systems
- o IBM Austin Research Laboratory
- o Lockheed Martin Aerospace
- o Nallatech Ltd.
- o National Reconnaissance Office
- o North Carolina State University
- o Northeastern University
- o Pentum Group, Inc.
- o Protean Devices, Inc.
- o Rose-Hulman Institute of Technology
- o Science & Technology Associates, Inc.
- o University of Maryland
- o University of Pennsylvania
- o Vanderbilt University

Document Change History

Version 1.0 was the first approved version of the document.

Version 1.0.1 made the following changes:

- removed `streamInitWithDataFIFO`
- added `streamClearFIFO`; removed old FIFO-draining protocol from appendix
- clarified `kernelPause` for predefined kernels
- required kernels to finish, FIFO streams to be empty before going out of scope

Version 1.1 made the following changes:

- removed mention of `maxBuffering` hint
- clarified that dynamically-sized blocks must fit within corresponding memory resource (this property does not need to be checked by the Low-Level Compiler)
- clarified `kernelWaitMultiple` usage scenario
- removed reference code for predefined kernels from Appendix
- clarified that `streamPeek` and `streamPeekEOS` on FIFO streams require a peek support in corresponding hardware FIFO
- clarified that blocks, streams, and kernels must be local variables (not global variables) to simplify analysis

Version 1.2 made the following changes

- specified that SVM is based on C89 standard
- made blocks pass-by-value
- made `BLOCKREAD/WRITE` macros
- removed the `maxBuffering` performance hint from the stream initialization functions
- specified that stream EOS information is allocated by the LLC or the run-time system
- clarified the role of the kernel argument `struct`
- added flags controlling when the kernel argument `struct` is copied
- specified that the kernel argument `struct` is allocated by the LLC or the run-time system
- clarified the kernel dependence examples
- specified that C code in kernels is only restricted by the machine model

- revised the definition of kernels
- specified that C code in control is unrestricted
- made hardware identifiers implementation-specific types
- introduced Machine Model identifier queries and configuration calls
- specified code restrictions based on the machine model
- eliminated Section 5.7 on black box kernels, and most other references to black box kernels
- numerous editorial and formatting changes

Table of Contents

Acknowledgements.....	i
Authors.....	i
The Morphware Forum	i
Document Change History.....	iii
Table of Contents.....	v
List of Acronyms	vii
1. Introduction.....	1
1.1. Two-Level Compilation.....	1
1.2. The Streaming Virtual Machine API	2
2. Terminology.....	4
2.1. Logical Entity Types.....	4
2.2. Resource Entity Types	5
2.3. Processor Properties.....	5
2.4. Logical to Resource Binding	5
3. Streams.....	6
3.1. Stream Flags.....	7
3.2. Stream Initialization.....	8
4. Blocks	14
4.1. Block Initialization.....	14
4.2. Block I/O.....	15
5. Kernels	16
5.1. Kernel Base Data Type	16
5.2. Kernel Execution Model	17
5.3. Kernel Initialization	20
5.4. Kernel Control	21
5.5. User-Defined Kernels	26
5.6. Pre-Defined Kernels.....	27
6. Control and Kernel Threads.....	39
6.1. Transferring Data Between Kernels.....	40
6.2. Transferring Data Between Control Code and Kernels	41

6.3. FIFO Cleanup.....	42
7. Machine Model Interaction.....	43
7.1. Machine Model Restrictions	46
8. Error Handling	47
References.....	48
Index of SVM Calls and Data Types	49

List of Acronyms

ALU	Arithmetic-Logic Unit
API	Application Programming Interface
DARPA	Defense Advanced Research Project Agency
DMA	Direct Memory Access
DSP	Digital Signal Processor, Digital Signal Processing
EOS	End of Stream
FIFO	First In, First Out
HLC	High Level Compiler
I/O	Input/Output
LLC	Low Level Compiler
MM	Machine Model
MSI	Morphware Stable Interface
PCA	Polymorphous Computing Architecture
RAM	Random Access Memory
SVM	Streaming Virtual Machine

1. Introduction

Recent trends in both applications and architectures point to the streaming model of computation as a cornerstone for the next generation of high-performance computing. On the applications side, there is a growing class of programs that falls within the streaming domain. Good candidate applications are distinguished by a large degree of task and data parallelism and regular patterns of communication between concurrently executing modules. Such applications appear in the areas of signal processing, graphics, networks, and scientific computing [1], [2]. On the architecture side, there is an emerging class of communication-exposed machines that contains distributed memories, replicated processing units, and sometimes high-performance “stream co-processors.” Because resources in these machines are controlled by a software layer, a compiler can leverage widespread parallelism and regular communication patterns to obtain good performance.

A number of communication-exposed architectures are being developed as part of the Polymorphous Computing Architectures (PCA) initiative [3]. These machines offer alternate modes of operation, called *morphs*, which can be selected to meet the demands of the application and the constraints of the environment. Mapping a stream application to these architectures is a challenging compiler problem. To obtain good performance, the compiler needs to detect data, task, and pipeline parallelism in the source program and calculate a load-balanced mapping onto the processor resources, all while taking into account the large array of possible streaming morphs.

1.1. Two-Level Compilation

To address this challenge, the Morphware Forum [4] is defining elements of a portable application development methodology for architectures of the PCA program that includes new source languages, a new application development process, and a framework for expressing system and application metadata. One element of the methodology, referred to as the Morphware Stable Interface (MSI) [5], is a strategy as shown in Figure 1 that partitions the compilation process into two stages: a High-Level Compiler (HLC) and a Low-Level Compiler (LLC).

The HLC accepts source code written in a stream language (*e.g.*, Brook or StreamIt [6], [7]) and the description of an architecture via a PCA Machine Model (MM), and it produces a mapping of logical stream constructs to the physical resources available in the given architecture. The mapping is expressed using the Streaming Virtual Machine Application Programming Interface (SVM API), which is built upon the C programming language as defined by the C89 standard. The SVM API abstracts and simplifies PCA hardware, providing a consistent abstract set of resource types and functional support requirements that must be targeted by HLC developers and supported by the LLCs of PCA hardware developers.

The LLC accepts mappings expressed using the SVM API and produces architecture-specific assembly language. Because the SVM API uses the C programming language, many of its forms are common C constructs that the LLC maps to hardware using traditional compilation techniques. The feature that distinguishes an LLC from a standard C compiler is its ability to recognize and compile SVM API idioms that control stream hardware such as kernel processors,

hardware FIFOs, and DMA engines. An LLC can be implemented as a dedicated compiler for the SVM API, or as a combination of a conventional C Compiler with an SVM library.

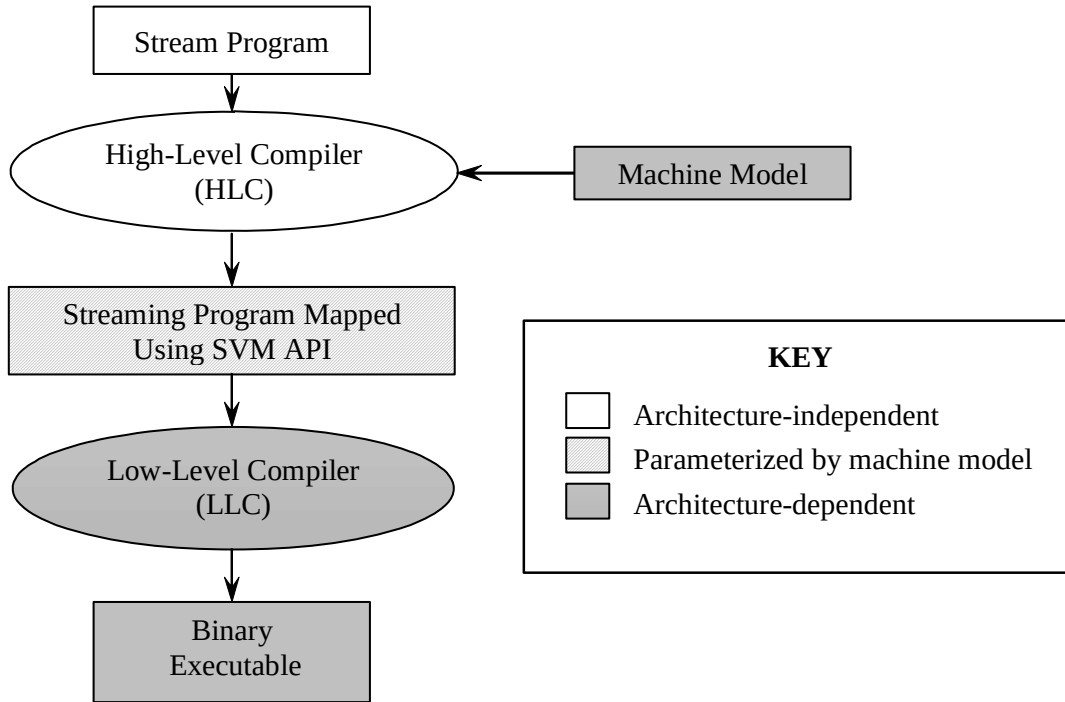


Figure 1. Streaming components of the Morphware Stable Interface.

This two-stage compilation process offers at least two benefits. First is a degree of application portability, achieved through partial infrastructure reuse. An application can be re-targeted to a new architecture by recompiling it through the HLC using an appropriate new machine model to generate new SVM code. It is expected that multiple architectures and their accompanying LLCs will be able to reuse an existing HLC and attain high performance without significantly modifying it. Consequently, it is also expected that many fewer HLCs than LLCs will be needed to implement the MSI model on a wide range of architectures. The PCA program is supporting the development of one HLC, *R-Stream* by Reservoir Labs, Inc. [8].

A second benefit is a degree of support for *morphing*, which is a change of the operational mode of the application or of the architecture resources available, possibly in real time. There are many types of morphing; one taxonomy is described in [5]. As an example, a change of target resources might be implemented in the MSI by replacing one executable with another compiled to a different machine model.

1.2. The Streaming Virtual Machine API

The SVM API is the interface between the HLC and the LLC. Its goal is to provide a means for the HLC to express a mapping from the streaming application to the physical resources of a given architecture. Towards this end, the SVM API embodies the following design features:

- **Separation of control and data-intensive code.** The SVM API provides the *kernel* abstraction as a unit of computation. Kernels are threads with swapping restrictions

(explained in Section 6) that generally encapsulate clean, data-parallel code. Kernels can be asynchronously monitored and controlled from their spawning thread.

- **Explicit communication via streams.** The SVM API exposes the streaming nature of the computation with a stream abstraction. A *stream* is defined as an ordered collection of values of homogenous type through which processors can communicate. Streams are mapped either to RAMs or to hardware FIFOs. Hardware FIFOs provide primitive support for stream operations and can be used for direct processor-to-processor communication.
- **Explicit memory management for streaming data.** The SVM API expresses the size and placement of memory-mapped streams using orchestrated data movement with explicit DMA operations, instead of relying on hardware caching. This functionality is essential for good performance on data-intensive code that operates out of small local memories (*e.g.*, a Stream Register File). With explicit memory management, stream data may be fetched ahead of time to keep the execution units busy. The SVM API includes streaming DMA, which is continuous data movement in contrast with discontinuous transfers of quanta of data.

The design features above, and many of the details of the API that follow, reflect the principal design goal of simplifying the translation process for the LLC. While the resulting API is comprehensible by a human, this was a secondary consideration; the SVM API is not intended to be a programming mechanism for humans but rather as a vehicle for expressing compiler-generated high-level mappings. Because the API is built upon C, it can be compiled by a standard C compiler with a simple runtime library for testing and debugging purposes.

The remainder of this document gives the specification for the SVM API. Section 2 reviews terminology. Sections 3 – 6 discuss the parts of the API having to do with streams, blocks, kernels, and controls, respectively. Section 7 discusses interaction between the SVM and the PCA Machine Model [9]. Section 8 contains discussion of error handling.

2. Terminology

The SVM API is a vehicle for defining logical computing entities and expressing their mapping to physical computing entities.

Logical entities are used to encapsulate the streaming application as transformed by the HLC from the original source input; they represent the program's calculations, state, control, and data flow. The particular logical entity types defined in Section 2.1 encourage a program structure with partitioning, scheduling, and communication that correspond to the physical entity types typical of PCA architectures.

Physical entities correspond to hardware structures in architectures of the PCA program. The particular physical entity types referenced by the SVM API reflect an architecture philosophy in which control processors initiate large quanta of memory accesses, communications, and computations. The physical entities described in Section 2.2 are defined in the PCA Machine Model specification document [9]. For each source application and configuration of the target architecture, a PCA Machine Model is constructed that describes the hardware resources available to the HLC for mapping logical entities.

Like physical entities, processor properties are defined in the PCA Machine Model specification document and referenced by the SVM API. Processor properties are parameters that describe the functionality of processors and their relationship with other processors. Section 2.3 defines the processor properties referenced by the SVM API.

In Section 2.4, the legal bindings of logical to physical entities are defined. The SVM API embodies these bindings, allowing for the construction of legal mappings of stream programs to PCA architectures.

2.1. Logical Entity Types

The SVM API contains the following types of logical entities:

- **Streams** – Entities that contain an ordered collection of data elements of a given type. From the perspective of the controlling program, streams are sequential, but at the low level stream processors may access streams in any order.
- **Blocks** – Provide indexed (random) access to a fixed-sized set of data elements of a given type.
- **Kernels** – Entities that contain a locus of code. They have an execution state, are declared using standard C constructs, and are controlled with SVM API functions. Two types of kernels exist: *user-defined* and *pre-defined*. User-defined kernels perform computation specific to a given PCA application. Pre-defined kernels move data between kernels and streams. Pre-defined kernels subsume the concept of DMA.
- **Controls** – Entities that embody a locus of control code. Control code can initiate, monitor, and terminate the execution of kernels. Controls are described using standard C constructs in conjunction with SVM API functions and data structures.
- **Data Elements** – Data elements are entities that are the units of state stored in blocks and streams. They must have a fixed size. Program semantics requiring variable-sized data elements must synthesize this capability.

2.2. Resource Entity Types

The following resource entity types are contained within the PCA Machine Model and are referenced by the SVM API; see [9] for details.

- **Memory** – A memory stores data. Two types of memory are currently referenced in this document:
 - **FIFO** – A memory in which data is accessed in first-in first-out fashion.
 - **RAM** – Hardware that provides random access storage with indexed data.
- **Processor** – Logic with the ability to execute instructions and to read and write data. Processors have internal state and can be started and stopped. They may support the execution of kernels or controls. DMA engines, RISC cores, and I/O deices are all types of processors.

Other types of resource entities are defined in the PCA Machine Model but are not currently referenced in this document. These include *networks*, *links*, and *cache* memories; see [9] for details. PCA architectures may also include resource types not included in either the PCA Machine Model or this Stream Virtual Machine. Registers are one example of such a resource.

2.3. Processor Properties

The following processor properties in the PCA Machine Model are relevant to the operation of the SVM:

- **mm.Proc.RequiredMaster** – This property, if set, indicates that the processor can only execute kernels invoked by a specific processor.
- **mm.Proc.SupportsBlockDMA** – This property indicates that a processor can execute all pre-defined kernels with block inputs and outputs.
- **mm.Proc.SupportsStreamDMA** – This property indicates that a processor can execute all pre-defined kernels with stream inputs and outputs.
- **mm.Proc.SupportsUserCode** – This property indicates whether or not user code, including user-defined kernels, is supported on a given slave processor.

2.4. Logical to Resource Binding

Following are the permitted mappings of logical to resource entities.

- A stream is mapped to a memory.
- A block is mapped to a RAM.
- A kernel is mapped to a slave processor that supports the given kernel.

A control is mapped to a master processor.

3. Streams

A *stream* is a computational entity type that contains an ordered collection of elements of a fixed size.¹ At any given time, each stream has a single producer and a single consumer. However, the data elements in memory can be aliased such that several different streams can read the same values and write to the same place in memory.

The stream type is declared as a black box:

```
typedef struct {
    // not exposed
} svm_Stream;
```

The SVM API includes support for end-of-stream (EOS) indicators. In some cases, the number of elements that are pushed onto a stream is data-dependent and not determined until runtime. Thus, an explicit EOS indicator needs to be included in order for a consumer kernel to know when its input stream has ended. While it is possible to encode EOS information into the data stream, doing so may ignore hardware EOS support and incur unnecessary overhead. Including EOS functionality in the SVM API standardizes the use of EOS indicators, allows the LLC to take advantage of hardware EOS support, and allows other API functions to take advantage of EOS. Using an EOS tag instead of a separate EOS element placed in the stream serves to simplify the stream semantics.

For these reasons, the API includes `svm_streamPushWithEOS`, a function that places an EOS tag onto the next element pushed onto a stream. The `svm_streamPeekEOS` function is provided for consumers to check whether or not the next element in the stream has an EOS tag. Note that there might be several EOS tags in the lifetime of a stream, or even at a given point in time; for example, EOS tags could be used as a boundary between successive lines of video data. However, the `svm_Stream_One_EOS` flag allows the HLC to indicate cases where a stream will contain at most one EOS tag at a given point in time. The LLC or run-time is responsible for allocating space to hold the EOS tag information.

The API also requires the LLC to implement RAM-based streams with a very specific data layout: they must be circular buffers. At the program level, data elements are placed sequentially starting at a given address and wrapping around once the capacity of the RAM has been reached. This is important for allowing reuse of memory space during the lifetime of a stream and for transferring data between control code and streaming code. However, in cases where the data layout is inconsequential, the HLC can use the `svm_Stream_Unaliased_RAM` flag to allow layout optimizations by the LLC.

In order to improve the readability of SVM API code, there are several aliases for the `svm_Stream` type:

```
typedef svm_Stream svm_IStream;
typedef svm_Stream svm_OStream;
typedef svm_Stream svm_IOStream;
```

¹ Variable sized elements can be supported at the application language level and compiled into the fixed size scheme using a data encoding.

The `svm_IStream`, `svm_OStream`, and `svm_IOStream` types may be used to indicate the relationship between kernels and streams. If a kernel takes a variable of type `svm_IStream`, then that stream should only be used for input; likewise, an `svm_OStream` should only be used for output, while an `svm_IOStream` may be used for both input and output.² Note that the use of these types is strictly optional; they are only provided as a means to improve the readability of SVM code.

The API for streams consists of a set of initialization and I/O functions. An initialization function sets the size and location of a stream, while the I/O functions transfer data into and out of streams. A stream must be initialized before being used. These functions are described in detail in the following sections.

3.1. Stream Flags

These flags may be used by the HLC to pass performance hints to the LLC. They appear as an argument to the stream initialization functions in Section 3.2.

```
typedef enum {
    svm_Stream_Unordered = 0x1,
```

This flag is true if FIFO semantics may be relaxed such that popping does not return elements in the same order that the elements were pushed. This might enable improved routing or instruction scheduling, for example in applications dealing with unordered network packets. Because the elements are unordered, `svm_streamPeek` and `svm_streamPeekEOS` cannot be used if this flag is set. Note that this implies that there is no EOS support for unordered streams.

```
    svm_Stream_Unaliased_RAM = 0x2,
```

This flag may be set if other streams and blocks are guaranteed *not* to be aliased with this stream. If set, the LLC can use an optimized, architecture-specific memory layout for this stream rather than guaranteeing that it is implemented as an in-order circular buffer.

```
    svm_Stream_Never_Wraps = 0x4,
```

This flag is true if the number of elements pushed onto a given stream will never exceed the stream's capacity. This allows the LLC to treat the stream as a single-assignment block rather than a circular buffer.

```
    svm_Stream_One_EOS = 0x8
```

This flag is true if at most one element in the stream will ever have an EOS tag at any given time. For RAM-based streams, this allows the LLC to implement EOS with a single pointer instead of tracking a set of EOS tokens.

```
} svm_Stream_Flags;
```

² Note that an `svm_IOStream` is intended to be used by a single kernel to both pop and push elements. The `svm_IOStream` should not be used between kernels because such use would be output for one kernel (`svm_OStream`) and input for the other (`svm_IStream`). The primary use of the `svm_IOStream` is to share a single computational resource for both an output and input stream. Data pushed onto the stream can be popped. When the resource is full, the push function blocks. When the resource is empty, pop blocks.

3.2. Stream Initialization

streamInitRAM

```
void svm_streamInitRAM(  
    svm_Stream* s,  
    svm_Memory ramLocation,  
    size_t address,  
    size_t capacity,  
    size_t elementSize,  
    svm_Stream_Flags flags)
```

Initializes an empty stream that resides in RAM with the specified location, size, and performance hints. The address and capacity might be dynamically calculated at runtime.

Unless the `svm_Stream_Unaliased_RAM` flag is set, the stream must be implemented as a circular buffer that stores items in consecutive locations starting at address and wrapping around when more than capacity elements have been pushed. That is, items 0, capacity, 2×capacity, etc., are all stored at location address.

Parameters:

s - stream to initialize.

ramLocation - RAM in which to hold the stream (from the PCA Machine Model).

address - address at which stream begins.

capacity - maximum number of elements in stream at any one time.

elementSize - number of bytes per element.

flags - performance hints to the LLC (see Section 3.1). These can be ignored or assumed to be zero without sacrificing correctness; they are only for the sake of optimization.

Constraints:

ramLocation, elementSize, and flags are literals and are constant across all calls to initialization functions for s.

s is not used for input or output in a kernel currently Waiting, Running, or Paused.

Called by:

Controls.

streamInitWithDataRAM

```
void svm_streamInitWithDataRAM(  
    svm_Stream* s,  
    svm_Memory ramLocation,  
    size_t address,  
    size_t capacity,  
    size_t elementSize,  
    size_t initLength,  
    int initEOS,  
    svm_Stream_Flags flags)
```

Initializes a stream in RAM with some elements already in place. If `initEOS` is true, then the last item is tagged with an EOS tag. The stream will have the specified location, size, and performance hints. The address and capacity might be dynamically calculated at runtime. The stream must be implemented as a circular buffer. Unlike `svm_streamInitRAM`, the `svm_Stream_Unaliased_RAM` flag may not be used. The LLC or run-time is responsible for allocating space to hold the EOS information.

Parameters:

`initLength` - The number of items already in place at address that should be pushed onto `s`.

`initEOS` - Indicates whether or not the item at position `initLength-1` should be tagged with an EOS tag. That is, it indicates the value of `svm_streamPeekEOS(s, initLength-1)` after initialization. 0 indicates false; otherwise true.

Other parameters are the same as for `svm_streamInitRam`.

Constraints:

`initLength > 0`.

`svm_Stream_Unaliased_RAM` flag is *not* set and items to be pushed onto `s` appear sequentially in the RAM. This is to ensure that the layout of data in the stream matches the layout of the data in the RAM.

`ramLocation`, `elementSize`, and `flags` are literals and are constant across all calls to initialization functions for `s`.

`s` is not used for input or output in a kernel currently `Waiting`, `Running`, or `Paused`.

Called by:

Controls.

streamInitFIFO

```
void svm_streamInitFIFO(  
    svm_Stream* s,  
    svm_Memory fifoLocation,  
    size_t elementSize,  
    svm_Stream_Flags flags)
```

Initializes an empty stream mapped to a FIFO resource with the given element size and performance hints. Only one stream may be allocated to a FIFO at a time and the size of this stream is determined by the size of the FIFO.

Parameters:

`s` - stream to initialize.

`fifoLocation` - FIFO resource in which to hold stream (from PCA Machine Model).

Other parameters are the same as for `svm_streamInitRam`.

Constraints:

`fifoLocation`, `elementSize`, and `flags` are literals and are constant across all calls to initialization functions for `s`.

`fifoLocation` represents an empty FIFO (one in which no other stream retains state).

`s` is not used for input or output in a kernel currently `Waiting`, `Running`, or `Paused`.

Called by:

Controls.

streamClearFIFO

```
void svm_streamClearFIFO(
    svm_Memory fifoLocation)
```

Clears all data items from a FIFO resource. This is a synchronous call from the control thread: it does not return until the items are removed.

Parameters:

`fifoLocation` - FIFO resource to clear (from PCA Machine Model).

Called by:

Controls.

streamPush

```
void svm_streamPush(
    svm_Stream* s,
    void* element)
```

Enqueues a data element onto the end of a stream. If the stream cannot hold any more elements, then `svm_streamPush` blocks until there is space available.

Parameters:

`s` - stream to push onto.

`element` - data element to push.

Constraints:

`s` has been initialized and has a positive capacity.

Called by:

Kernels, Controls.

streamPushMulticast

```
void svm_streamPushMulticast(
    size_t n,
    void* element,
    svm_Stream* s1,
    svm_Stream* s2, ...)
```

Pushes a single data element onto the end of multiple streams, in any order or simultaneously. Does not return until the element has been pushed onto each stream.

For example, if `x` is the element to push, `s1` is empty, and `s2` is full, then the following are all legal implementations of `svm_streamPushMulticast(2, &x, &s1, &s2)`:

- Push `x` onto `s1`, wait until `s2` is not full, then push `x` onto `s2`.
- Push `x` onto `s1` while waiting until `s2` is not full, then push `x` onto `s2`.
- Wait until `s2` is not full, push `x` onto `s2`, then push `x` onto `s1`.
- Wait until `s2` is not full, push `x` onto `s1`, then push `x` onto `s2`.

Aside from the blocking behavior, this method is equivalent to a series of calls to `svm_streamPush`. It is included in the API in order to give the LLC an opportunity to optimize broadcast messages.

Parameters:

`n` - number of streams.
`s1, s2, ..., sn` - streams to push onto.
`element` - element to push.

Constraints:

`s1, s2, ..., sn` have been initialized and each has a positive capacity.

Called by:

Kernels, Controls.

streamPop

```
void svm_streamPop(  
    svm_Stream* s,  
    void* element)
```

Dequeues an element from the front of the stream `s` and stores it into `element`. If the stream is empty, then `svm_streamPop` blocks until an element is present. Note that if the popped element has an EOS tag, the tag is also popped, and the EOS information is no longer accessible.

Parameters:

`s` - stream to pop from.
`element` - location in which to store the popped element.

Constraints:

`s` has been initialized and has a positive capacity.

Called by:

Kernels, Controls.

streamPeek

```
void svm_streamPeek(  
    svm_Stream* s,  
    size_t n,  
    void* element)
```

Retrieves the data element at position *n* from the front of the stream without changing any of the stream's state. The index *n* is zero-based, such that `svm_streamPeek(s, 0, &x)` and `svm_streamPop(s, &x)` both put the same value in *x*. If the stream contains less than *n*+1 elements, then `svm_streamPeek` blocks until *n*+1 elements are available.

Parameters:

s - stream to peek at.
n - index at which to peek (zero-based and starting from front of stream).
element - location in which to store the peeked element.

Constraints:

n < capacity of *s*.
s does not have the `svm_Stream_Unordered` flag set.
s has been initialized and has a positive capacity.
If *s* is allocated to a FIFO, then `FIFOPeek` > 0 for that FIFO in the machine model.

Called by:

Kernels, Controls.

streamPushWithEOS

```
void svm_streamPushWithEOS(  
    svm_Stream* s,  
    void* element)
```

Pushes a data element onto the end of the stream with an EOS tag attached to the element. If the stream is full, this blocks until there is space to push the element.

Parameters:

s - stream to push onto.
element - element to push.

Constraints:

s has been initialized and has a positive capacity.

Called by:

Kernels, Controls.

streamPushMulticastWithEOS

```
void svm_streamPushMulticastWithEOS(  
    size_t n,  
    void* element,  
    svm_Stream* s1,  
    svm_Stream* s2, ...)
```


Pushes a data element with EOS tag attached onto the given streams, in any order or simultaneously. Will not return until the element has been pushed onto all streams. See `svm_streamPushMulticast` for an example.

Parameters:

`n` - number of streams.
`s1, s2, ..., sn` - streams to push onto.
`element` - element to push.

Constraints:

`s1, s2, ..., sn` have been initialized and each has a positive capacity.

Called by:

Kernels, Controls.

streamPeekEOS

```
int svm_streamPeekEOS(  
    svm_Stream* s,  
    size_t n)
```

Returns true if there is an EOS tag on the `n`th element from the front of the stream, otherwise false; blocks if less than `n+1` elements are available.

Parameters:

`s` – stream to peek for EOS tag.
`n` – index at which to peek (zero-based and starting from front of stream).

Returns:

True (anything except 0) if there is an EOS tag at index `n`.

Constraints:

`n < capacity` of `s`.
`s` does not have the `svm_Stream_Unordered` flag set.
`s` has been initialized and has a positive capacity.
If `s` is allocated to a FIFO, then `FIFOpeek > 0` for that FIFO in the machine model.

Called by:

Kernels, Controls.

4. Blocks

To provide random access to a set of data elements, the SVM API provides the *block* abstraction. A block is a region of memory that can be read from and written to through indexed random access. As with streams, the block type is a black box with aliases to indicate its relationship to a given kernel:

```
typedef struct {  
    // not exposed  
} svm_Block;  
  
typedef svm_Block svm_IBlock;  
typedef svm_Block svm_OBlock;  
typedef svm_Block svm_IOBlock;
```

4.1. Block Initialization

blockInit

```
void svm_blockInit(  
    svm_Block* b,  
    svm_Memory ramLocation,  
    size_t address,  
    size_t capacity,  
    size_t elementSize)
```

Initializes a block at the given location with the given size. The address and capacity might be dynamically calculated at runtime.

Parameters:

b - block to initialize.

ramLocation - memory resource in which to hold block (from the PCA Machine Model).

address - memory address at which block begins.

capacity - maximum number of elements in the block.

elementSize - number of bytes per element.

Constraints:

ramLocation and elementSize are literals and are constant across all calls to svm_blockInit on b.

b is not used for input or output in a kernel that is currently Waiting, Running, or Paused.

address, capacity, and elementSize do not cause the block to exceed the size of the ramLocation resource as defined in the machine model.

Called by:

Controls.

4.2. Block I/O

BLOCK_WRITE

```
SVM_BLOCK_WRITE(  
    /* svm_Block */ b,  
    /* size_t */ index,  
    /* void* */ element,  
    /* TYPE */ type)
```

A macro that stores a data element into a memory block.

Parameters:

b - block to write into.
index - offset within b to write to.
element - element to write.
type – type of the element to write.

Constraints:

b has been initialized and has a positive capacity.
index < capacity of b.

Called by:

Kernels, Controls.

BLOCK_READ

```
SVM_BLOCK_READ(  
    /* svm_Block */ b,  
    /* size_t */ index,  
    /* void* */ element,  
    /* TYPE */ type)
```

A macro that reads a data element from a memory block b and writes it into element.

Parameters:

b - block to read from.
index - offset within b to read from.
element - location to load element into.
type – type of the element to read.

Constraints:

b has been initialized and has a positive capacity.
index < capacity of b.

Called by:

Kernels, Controls.

5. Kernels

Kernels consist of computation expressed with a restricted subset of the C language; operate on a set of input and output streams or blocks; may contain local state; and generally encapsulate data-parallel code. Kernels are assigned to specific processors and do not migrate. They run to completion before a new kernel is initiated. Kernels can be asynchronously monitored and controlled from a control thread running on a different processor.

The SVM API has support for two types of kernels. Section 5.5 describes general *user-defined kernels* whose behavior is specific to a given application. Section 5.6 describes *pre-defined kernels* that move data; these kernels are required elements of the API and must be supported by every LLC. A third type of kernel, *black box kernels*, may be added in the future to provide support for external library routines.

Both types of kernels share a common execution model and a base data type that specifies the location for kernel execution. The base data type is passed into functions that control a kernel's execution. The kernel execution model is described in section 5.2.

Kernels have the following components:

1. An initialization function that receives the following:
 - The processor resource where the kernel will execute.
 - (Optional) A block of memory for spilling local variables.
 - The input and output streams and/or blocks for the kernel.
 - Any kernel-specific initialization data.
2. A work function that defines the computation of the kernel.
3. (Optional) A `struct` that contains fields used to pass arguments to, and return results from, the kernel, henceforth called the “*kernel argument struct*”. Space for this structure must be allocated by the LLC or run-time system.
4. A status that reflects the current state of the kernel.
5. Control functions used to execute the kernel, defined in section 1.1.

5.1. Kernel Base Data Type

All kernels share a base data type that is extended to create specific kernels. The kernel type and enumerations are declared as follows:

```
typedef struct {  
    // not exposed  
} svm_Kernel;
```

This is a function pointer to the work function of a kernel:

```
typedef void (*svm_ExtKernelWork) (void* extKernelData);
```

The work function represents the entire execution of the kernel; it is called only once by the control thread and should contain internal loops if some computation is to be executed repeatedly.

In our terminology, status codes refer to the state of a kernel. An uninitialized kernel does not have a well-defined state. Valid status codes are defined by the following enumeration:

```
typedef enum {
    svm_Kernel_Null,
    svm_Kernel_Unstarted,
    svm_Kernel_Waiting,
    svm_Kernel_Running,
    svm_Kernel_Paused,
    svm_Kernel_Finished
} svm_Kernel_Status;
```

For the remainder of the document, these status codes are referred to as `Null`, `Unstarted`, `Waiting`, `Running`, `Paused`, and `Finished`. Section 5.2 defines the execution model for kernels and the valid transitions between kernel states.

5.2. Kernel Execution Model

Kernels have a simple execution model. A transition diagram for the legal states of a kernel appears in Figure 2, while a timeline for a sample interaction with a kernel appears in Figure 3. Note that these are not required transitions. For instance, a kernel might never enter the `Paused` or `Finished` states.

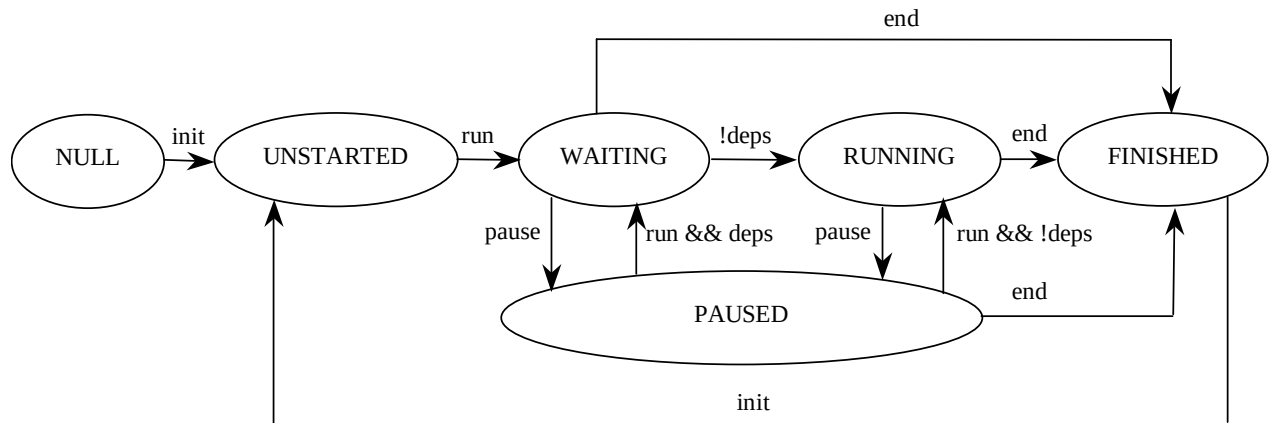


Figure 2. Legal transitions of a kernel's status.

Kernel state transitions are triggered by the kernel SVM API initialization and control functions detailed in Section 5.3 and Section 1.1 and referenced extensively in this section.

First, the initialization function `svm_kernelInitNull` sets the kernel's status to `Null`. This ensures that no code will interact with a kernel whose status is not well defined. Then, `svm_kernelInit` is used to initialize the kernel and place it in the `Unstarted` state. After a call to `svm_kernelRun` by the control thread, the kernel's status changes to `Waiting`. When all kernels that it depends on (as indicated through the use of the `svm_kernelAddDependence` function) have entered the `Finished` state, the kernel's status becomes `Running`. The kernel then executes its work function. If either the control thread or the work function call `svm_kernelPause`, the kernel pauses in executing the work function and its status becomes

Paused. It remains Paused until the control thread calls `svm_kernelRun` again. If either the control thread or the work function calls `svm_kernelEnd` at any time when the kernel's status is not `Finished`, the kernel stops executing the work function and its status becomes `Finished`. The kernel implicitly calls `svm_kernelEnd` when the work function returns. If control flow causes a kernel to be initialized again, its status is reset to `Unstarted`. Calling `svm_kernelInit` is the only way to reset a kernel so that it may be run again.

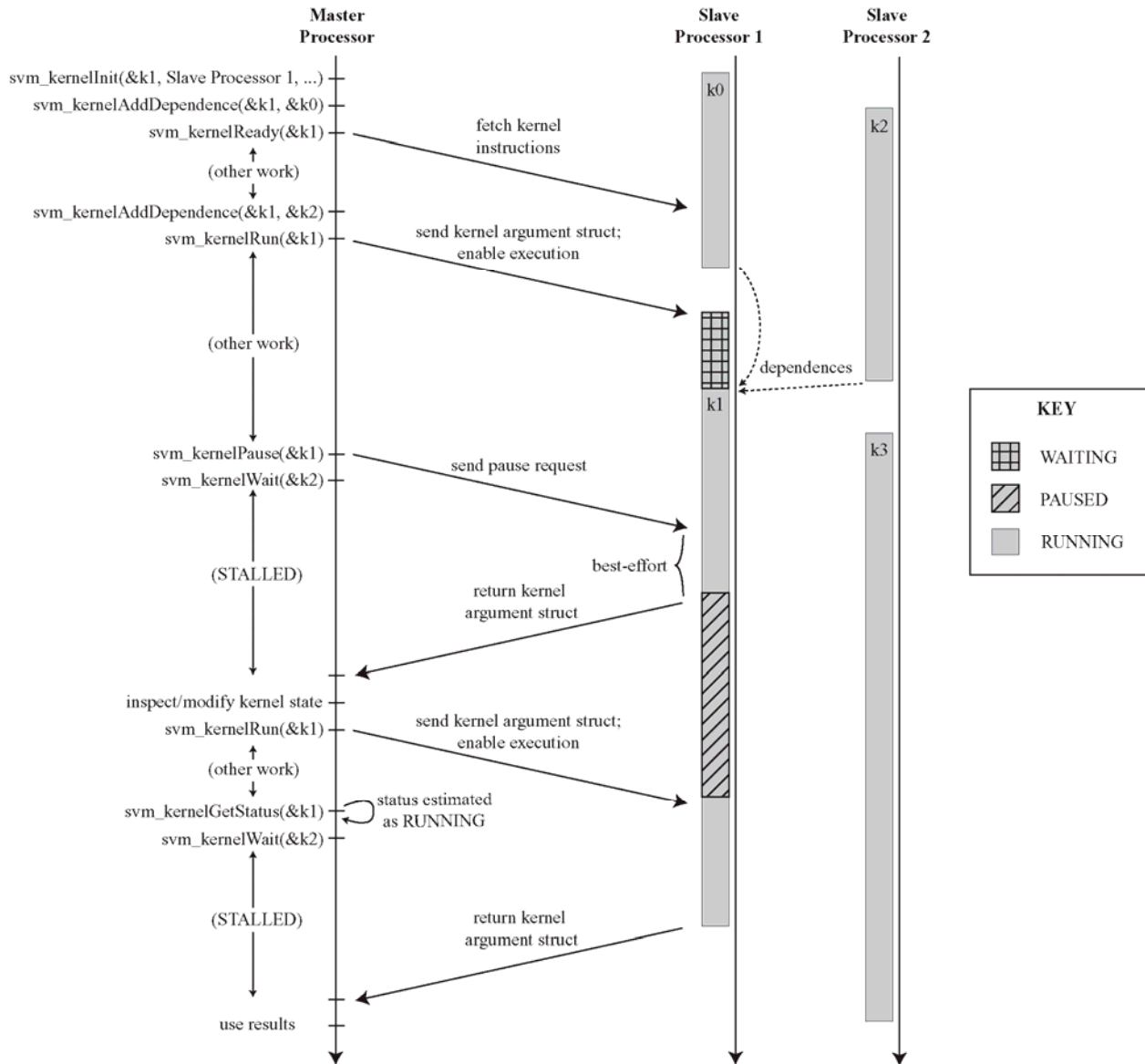


Figure 3. Sample timeline for an interaction between the control thread and a kernel.

Section 6 gives a detailed explanation of how kernels should be controlled. However, we give some brief examples here to clarify the model of computation for kernels. The most common control sequence for initializing and running a kernel is as follows:

```
MyKernel k;  
svm_kernelInitNull(&k.kernel);  
myKernelInit(&k, ...);  
svm_kernelRun(&k.kernel);  
svm_kernelWait(&k.kernel);
```

In addition, multiple kernels can be run in parallel, with items being transferred from one to another through a stream. For example:

```
svm_Stream s;  
  
// Declare 2 kernels and set their status to Null  
MyKernel1 k1; // k1 will write to s  
MyKernel2 k2; // k2 will read from s  
svm_kernelInitNull (&k1.kernel);  
svm_kernelInitNull (&k2.kernel);  
  
svm_streamInitRAM (&s, ...);  
  
myKernelInit(&k1, &s, ...);  
svm_kernelRun (&k1.kernel);  
  
myKernelInit(&k2, &s, ...);  
svm_kernelRun (&k2.kernel);
```

In this case, the data dependency is explicitly communicated through the use of the stream, and the `svm_kernelAddDependence` function does *not* need to be used to synchronize the transfer of data elements. The following is an example in which `svm_kernelAddDependence` is required:

```
svm_Block b;  
  
MyKernel1 k1; // k1 will write to b  
MyKernel2 k2; // k2 will read from b  
svm_kernelInitNull (&k1.kernel);  
svm_kernelInitNull (&k2.kernel);  
  
svm_blockInit (&b, ...);  
  
myKernel1Init(&k1, b, ...);  
svm_kernelRun (&k1.kernel);  
  
myKernel2Init(&k2, b, ...);  
svm_kernelAddDependence (&k2.kernel, &k1.kernel);  
svm_kernelRun (&k2.kernel);
```

As shown, `svm_kernelAddDependence` is intended for cases where there is a memory dependence that is not captured by the stream semantics. This can occur either when a kernel writes to a block from which a subsequent kernel reads or when two kernels are reading/writing to streams or blocks that point to overlapping regions in memory. It would be possible to use `svm_kernelWait` to enforce these dependencies from the control thread, but `svm_kernelAddDependence` is added for performance reasons; with stream processing, a round-trip communication to the control thread is often expensive.

Additionally, the API supports the use of dependent kernels in a loop. The following code illustrates this concept. Two kernels called k1 and k2, contained within a common loop, update some of the contents of block b. Hence, k1 depends on k2 and vice versa.

```
MyKernel1 k1;
MyKernel2 k2;
svm_kernelInitNull (&k1.kernel)
svm_kernelInitNull (&k2.kernel)

while (...) {
    svm_kernelWait (&k1.kernel);
    myKernel1Init(&k1, b, ...); // b used for both input and output
    svm_kernelAddDependence (&k1.kernel, &k2.kernel);
    svm_kernelRun (&k1.kernel);

    svm_kernelWait(&k2.kernel);
    myKernel2Init(&k2, b, ...); // b used for both input and output
    svm_kernelAddDependence(&k2.kernel, &k1.kernel);
    svm_kernelRun(&k2.kernel);
}
```

In this scenario, each kernel is run multiple times, and each call to `svm_kernelAddDependence(k2, k1)` makes k2 dependent on the last execution of k1. If k1 is Unstarted, the `svm_kernelAddDependence` call is ignored.

5.3. Kernel Initialization

Like blocks and streams, kernels must be initialized before use. The `svm_kernelInitNull` function is used to set the kernel's status to Null, and `svm_kernelInit` function is used to initialize kernels for execution on a given processor with an optional scratch space for spilling local variables.

kernelInitNull

```
void svm_kernelInitNull(
    svm_Kernel* k)
```

Sets the status of a kernel to Null. `svm_kernelInitNull` must be called on a kernel before any other kernel control functions may be used. The use of `svm_kernelInitNull` ensures that no kernel control functions are called on a kernel with an undefined status.

Parameters:

k - kernel to be initialized.

Constraints:

k is uninitialized.

Called by:

Controls.

kernelInit

```
void svm_kernelInit(  
    svm_Kernel* k,  
    svm_Processor procLocation,  
    svm_IOBlock scratch,  
    void* extKernelData,  
    size_t extKernelDataSize,  
    bool ctrlReadsKernelArgStruct,  
    svm_ExtKernelWork extKernelWork)
```

Initializes a kernel for execution on a given processor with an optional scratch space for spilling local variables. This function should be called from the initializer for an extension of the base kernel type; it accepts the kernel argument `struct` and the work function of the extended kernel instance. After calling `svm_kernelInit`, the kernel is in the `Unstarted` state.

Parameters:

`k` - kernel to be initialized.
`procLocation` - processor resource on which to execute the kernel (from the PCA Machine Model).
`scratch` - memory block in which to spill variables (optional; an address of zero represents no scratch).
`extKernelData` - structure containing arguments to and results from this kernel.
`extKernelDataSize` - size (in bytes) of this kernel's structure of state variables.
`ctrlReadsKernelArgStruct` – true if control reads any field of `extKernelData` while kernel is paused or after kernel is finished.
`extKernelWork` - work function to be executed for this kernel.

Constraints:

`procLocation` is a literal and is constant across all statements that call `svm_kernelInit` on `k`.
`k` is `Null`, `Unstarted`, or `Finished`.

Called by:

Controls.

5.4. Kernel Control

This section gives detailed descriptions of the SVM API functions that control kernels.

kernelAddDependence

```
void svm_kernelAddDependence(  
    svm_Kernel* k,  
    svm_Kernel* dependsOnKernel)
```

Non-blocking function indicating that kernel `k` may not execute its work function until kernel `dependsOnKernel` has entered the `Finished` state. If `dependsOnKernel` is `Null`, `Unstarted`, or `Finished`, then this function has no effect, which allows dependent kernels to be used in a loop. The dependence is satisfied as soon as `dependsOnKernel` first enters

the `Finished` state, even if `dependsOnKernel` leaves the `Finished` state before `k` can execute; for instance, `dependsOnKernel` could be re-initialized and run again while `k` is still waiting for other kernels.

Note that `kernelAddDependence` does not start kernels, it only prevents them from entering the `Running` state until `dependsOnKernel` has entered the `Finished` state. The `svm_kernelRun` function is used to start kernels.

Parameters:

`k` - kernel that is dependent on the completion of `dependsOnKernel`.
`dependsOnKernel` - kernel that must finish before `k` may execute its work function.

Constraints:

`k` is `Unstarted`.
`dependsOnKernel` has been initialized.
`k` can read from a memory that `dependsOnKernel` can write to. That is, the processor executing `k` and the processor executing `dependsOnKernel` must both be connected to some common memory. This allows `dependsOnKernel` to notify `k` when the dependence is satisfied.

Called by:

Controls.

kernelRun

```
void svm_kernelRun(  
    svm_Kernel* k)
```

Non-blocking function that starts or resumes execution of a kernel. The kernel continues to run until its status is `Paused` or `Finished`.

Parameters:

`k` - kernel to run.

Constraints:

`k` is `Unstarted` or `Paused`.
Of the kernels that are currently `Waiting`, `Running`, or `Paused`, none of them writes to an output stream of `k` or reads from an input stream of `k`.

Called by:

Controls.

kernelPause

```
void svm_kernelPause(  
    svm_Kernel* k)
```

Non-blocking function that interrupts the execution of a kernel and, if the kernel was Running, sets the kernel's status to Paused.

If called from the control thread, the interruption may not be performed immediately because a message will have to be passed to the kernel processor running the kernel. However, the kernel's status must eventually change to either Paused or to Finished in the case where the kernel's work function completes before the interruption is processed. To account for this behavior, the control thread should follow up with a call to `svm_kernelWait`, a blocking function that will not return until the kernel's status is changed to Paused or Finished. Called from the control thread, `svm_kernelPause` is a command not a request and the kernel should pause as soon as possible without regard to state.

If called from within the kernel, the interruption is immediate; no other statement in the kernel's work function is executed before being paused. However, there may still be a delay before the control thread recognizes the Paused state.

If `k` has a status of Paused or Finished, then this function has no effect.

Parameters:

`k` - kernel to pause.

Constraints:

`k` is Running, Paused, or Finished.

If `k` is a predefined kernel, then the processor executing `k` has `mm.Proc.SupportsPredefinedPause` set to true in the PCA Machine Model.

If called from within a kernel, then that kernel is `k`. Kernels may not call `svm_kernelPause` on other kernels.

Called by:

Kernels, Controls.

kernelEnd

```
void svm_kernelEnd(  
    svm_Kernel* k)
```

Non-blocking function that interrupts execution of a kernel and, if the kernel's status is Running, sets it to Finished.

A call to `svm_kernelEnd` is a command, not a request, and the kernel should end as soon as possible without regard to state. If called from the control thread, the interruption may not be performed immediately because a message will have to be passed to the kernel processor running the kernel. However, the kernel's status must eventually change to Finished. This change may occur either because the kernel processed the call to `svm_kernelEnd` or because the kernel's work function completed before the interruption was processed. To

account for this behavior, the control thread should follow up any call to `svm_kernelEnd` with a call to `svm_kernelWait` to ensure that the kernel status changes to `Finished` before performing additional processing.

If called from within the kernel, the interruption is immediate; no other statement of the kernel is executed before the status is changed to `Finished`. However, there may still be a delay before the control thread recognizes the `Finished` state.

If the kernel is `Paused`, then its status is immediately changed to `Finished`, without executing another statement from within the kernel. If the kernel's status is `Finished`, then `svm_kernelEnd` has no effect.

This function is implicitly called when a kernel returns from its work function.

Parameters:

k - kernel to end.

Constraints:

k is `Running`, `Paused`, or `Finished`.

If called from within a kernel, then that kernel is k. Kernels may not call `svm_kernelEnd` on other kernels.

Called by:

Kernels, Controls.

kernelWait

```
void svm_kernelWait(  
    svm_Kernel* k,  
    bool returnKernelArgStruct)
```

Blocking function that does not return until the kernel's status is `Null`, `Unstarted`, `Paused`, or `Finished`.

Parameters:

k - kernel to wait for.

`returnKernelArgStruct` – true if kernel should copy-back the kernel argument struct.

Should only be true if `ctrlReadsKernelArgStruct` was true for the corresponding `svm_kernelInit` call. If true and the kernel is `Finished`, then the run-time system may discard the kernel argument struct after this call.

Constraints:

k has been initialized.

Called by:

Controls.

kernelWaitMultiple

```
void svm_kernelWaitMultiple(  
    size_t n,  
    svm_Kernel* k1,  
    svm_Kernel* k2, ...)
```

Considers the set *K* of *n* kernels (*k*₁, *k*₂, ..., *k*_{*n*}) that have a well-defined status other than *Unstarted* and does not return until one of the following is true: 1) at least one kernel in *K* is *Paused*, 2) all kernels in *K* are *Finished*, or 3) *K* is empty.

`kernelWaitMultiple` is usually used to handle two or more kernels which may pause unexpectedly (to respond to an error condition, for instance). Since it is not known which kernel will pause first, two sequential `kernelWait` calls cannot be used.

Parameters:

n - number of kernels.
*k*₁, *k*₂, ..., *k*_{*n*} - kernels to wait for.

Constraints:

*k*₁, *k*₂, ..., *k*_{*n*} have been initialized.

Called by:

Controls.

kernelGetStatus

```
svm_Kernel_Status svm_kernelGetStatus(  
    svm_Kernel* k)
```

Returns the status of the kernel as currently known by the control thread. The status returned by `svm_kernelGetStatus` may not be the actual status of the kernel, as the status may have changed on the kernel processor and the control thread's state may be stale. Despite this behavior, it is possible to produce correct code because the status returned is always conservative. If a kernel's status is *Null* or *Unstarted* this function will always return the correct status because only the control thread is responsible for setting these states. However, if the status is *Finished* or *Paused*, the function may return *Waiting* or *Running* until the control thread's state is updated. Thus, `svm_kernelGetStatus` will never return a status that would indicate to the control thread that a kernel has completed until it actually has.

Parameters:

k - kernel to inspect.

Returns:

Returns the status of *k* as an `svm_Kernel_Status` type (see Section 5.1).

Constraints:

k has been initialized.

Called by:

Controls.

5.5. User-Defined Kernels

The kernel base data type may be extended to build different kinds of kernels by declaring a new data type for each kernel that includes the kernel base data type as well as an initialization function and a work function that operate on the new data type. Multiple user-defined kernels can be assigned to a single slave processor, but only one of them can be Running; the rest must be Waiting for a previous kernel to complete.

For example, the new data type for an amplifier kernel could be constructed as follows:

```
typedef struct {
    svm_Kernel kernel;
    svm_IStream* in;
    svm_OStream* out;
    int N;
} Amplifier;
```

This Amplifier data type includes the svm_Kernel base data type, an input stream, an output stream, and an argument, N, to be passed into the kernel to indicate the level of amplification. The initialization function is used to assign input and output streams and other arguments to the user-defined kernel. It should also call svm_kernelInit on the kernel base data type to initialize it.

```
void amplifierWork(Amplifier* amp);

void amplifierInit(Amplifier* amp, svm_Processor procLocation
                  svm_IOBlock scratch, svm_IStream* _in,
                  svm_OStream* _out, int _N){
    svm_kernelInit(&amp->kernel, procLocation, scratch, amp,
                  sizeof(Amplifier), (svm_ExtKernelWork) &amplifierWork);
    amp->in = _in;
    amp->out = _out;
    amp->N = _N;
}
```

The work function performs the user-defined computation for the kernel. In this case, each data element popped from the input stream is multiplied by the value of N and pushed onto the output stream. Because the length of the input stream is unknown, it must be checked for an EOS tag before each call to svm_streamPop. If an EOS tag is discovered, the loop terminates, the last item is processed, and the function returns.

```
void amplifierWork(Amplifier* amp) {
    float x;
    while (!streamPeekEOS(amp->in, 0)) {
        svm_streamPop(amp->in, &x);
        x = x * amp->N;
        svm_streamPush(amp->out, &x);
    }
    svm_streamPop(amp->in, &x);
    x = x * amp->N;
    svm_streamPushWithEOS(amp->out, &x);
}
```

If the input stream is known to be infinite, then the work function can be simplified as follows:

```
void amplifierWork(Amplifier* amp) {
    while (1) { // 1 signifies "true"
        float x;
        svm_streamPop(amp->in, &x);
        x = x * amp->N;
        svm_streamPush(amp->out, &x);
    }
}
```

5.6. Pre-Defined Kernels

Special pre-defined kernels are used to move data. While these kernels are intended to correspond to DMA operations, abstracting them as kernels allows an LLC to implement them in software if necessary.

Note that for all the pre-defined kernels only the initialization routine is defined here. Their intended use is to set up a DMA processor; they do not copy data. Like all other kernels, `kernelRun` must be called to begin data movement, as described in Sections 5.1-1.1.

MoveB2B

```
typedef struct {
    svm_Kernel kernel;
    // not exposed
} svm_MoveB2B;

void svm_moveB2BInit(
    svm_MoveB2B* move,
    svm_Processor location,
    svm_IBlock srcBlock,
    svm_OBlock destBlock,
    size_t srcStart,
    size_t destStart,
    size_t length)
```

The primary use of this kernel is for copying data elements between blocks in different memory banks, though it can also be used for copying elements between blocks in a single memory. When the `kernelRun` function is called, the given number (`length`) of elements are read sequentially from `srcBlock` starting at index `srcStart` and written sequentially to `destBlock` starting at index `destStart`.

Parameters:

`move` - kernel to initialize.

`location` - processor resource on which to execute the kernel (from the PCA Machine Model).

`srcBlock` - source block from which elements are read.

`destBlock` - destination block into which elements are written.

`srcStart` - offset into the source block from which the first element is read. Subsequent elements are read sequentially.

`destStart` - offset into the destination block where the first element is to be written.

Subsequent elements are written sequentially.

`length` - number of elements to copy.

Constraints:

`move` is `Null`, `Unstarted`, or `Finished`.

`srcBlock` and `destBlock` have been initialized and both have positive capacity.

`srcBlock` and `destBlock` have the same `elementSize`.

All accesses to `srcBlock` and `destBlock` are within the capacity of each.

Called by:

Controls.

MoveB2S

```
typedef struct {
    svm_Kernel kernel;
    // not exposed
} svm_MoveB2S;

void svm_moveB2SInit(
    svm_MoveB2S* move,
    svm_Processor location,
    svm_IBlock srcBlock,
    svm_OStream* destStream,
    size_t srcStart,
    size_t length,
    int setEOS)
```

The primary use of this kernel is for copying data elements from a block in one memory bank to a stream in another, though it can also be used for copying elements from a block to a stream in a single memory. When the `kernelRun` function is called, the given number (`length`) of elements are read sequentially from `srcBlock` starting at index `srcStart` and pushed onto `destStream`.

Parameters:

`move` - kernel to initialize.

`location` - processor resource on which to execute the kernel (from the PCA Machine Model).

`srcBlock` - source block from which elements are read sequentially.

`destStream` - destination stream onto which elements are pushed.

`srcStart` - offset into the source block from which the first element is read. Subsequent elements are read sequentially.

`length` - number of elements to copy.

`setEOS` - if true (anything except 0), the last element pushed onto `destStream` is given an EOS tag.

Constraints:

move is Null, Unstarted, or Finished.
srcBlock and destStream have been initialized and both have positive capacity.
srcBlock and destStream have the same elementSize.
All accesses to srcBlock are within its capacity.

Called by:

Controls.

MoveS2B

```
typedef struct {
    svm_Kernel kernel;
    // not exposed
} svm_MoveS2B;

void svm_moveS2BInit(
    svm_MoveS2B* move,
    svm_Processor location,
    svm_IStream* srcStream,
    svm_OBlock destBlock,
    size_t destStart,
    size_t length,
    int untilEOS)
```

The primary use of this kernel is for moving data elements from a stream in one memory bank to a block in another, though it can also be used for moving elements from a stream to a block in a single memory. When the kernelRun function is called, the given number (length) of elements are popped from srcStream and written sequentially to destBlock starting at index destStart

Parameters:

move - kernel to initialize.
location - processor resource on which to execute the kernel (from the PCA Machine Model).
srcStream - source stream from which elements are popped.
destBlock - destination block into which elements are written.
destStart - offset into the destination block where the first element is to be written. Subsequent elements are written sequentially.
length - number of elements to move.
untilEOS - if true (anything except 0), ignore the length parameter and continue popping data elements from srcStream until an EOS tag is encountered.

Constraints:

move is Null, Unstarted, or Finished.
srcStream and destBlock have been initialized and both have positive capacity.
srcStream and destBlock have the same elementSize.
All accesses to destBlock are within its capacity.

Called by:

Controls.

MoveS2S

```
typedef struct {
    svm_Kernel kernel;
    // not exposed
} svm_MoveS2S;

void svm_moveS2SInit(
    svm_MoveS2S* move,
    svm_Processor location,
    svm_IStream* srcStream,
    svm_OStream* destStream,
    size_t length,
    int setEOS,
    int untilEOS)
```

The primary use of this kernel is for moving data elements between streams in different memory banks, though it can also be used for moving elements between streams in a single memory. When the kernelRun function is called, the given number (length) elements are popped from srcStream and pushed onto destStream.

Parameters:

move - kernel to initialize.
location - processor resource on which to execute the kernel (from the PCA Machine Model).
srcStream - source stream from which elements are popped.
destStream - destination stream onto which elements are pushed.
length - number of elements to move.
setEOS - if true (anything except 0), the last element pushed onto destStr is given an EOS tag.
untilEOS - if true (anything except 0), ignore the length parameter and continue popping data elements from srcStream until an EOS tag is encountered.

Constraints:

move is Null, Unstarted, or Finished.
srcStream and destStream have been initialized and have positive capacity.
srcStream and destStream have the same elementSize.

Called by:

Controls.

StridedScatterB2B

```
typedef struct {
    svm_Kernel kernel;
    // not exposed
} svm_StridedScatterB2B;

void svm_stridedScatterB2BInit(
    svm_StridedScatterB2B* scatter,
    svm_Processor location,
    svm_IBlock srcBlock,
    svm_OBlock destBlock,
    size_t srcStart,
    size_t destStart,
    size_t length,
    size_t destStride,
    size_t elementsPerStride)
```

Copies segments of `elementsPerStride` data elements from a source block to strided locations in a destination block. When the `kernelRun` function is called, data elements are read sequentially from `srcBlock` starting at index `srcStart` and written to `destBlock` in a strided fashion starting at index `destStart`. The distance between strides in the destination (`destStride`) might be greater than the number of elements copied with each stride (`elementsPerStride`). In this case, the elements moved appear at the beginning of each stride.

Parameters:

- `scatter` - kernel to initialize.
- `location` - processor resource on which to execute the kernel (from the PCA Machine Model).
- `srcBlock` - source block from which elements are read sequentially.
- `destBlock` - destination block into which elements are written in a strided fashion.
- `srcStart` - offset into the source block from which first element is to be read. Subsequent elements are read sequentially.
- `destStart` - offset into the destination block where the first element is to be written. Subsequent elements are written in a strided fashion.
- `length` - total number of strides to perform.
- `destStride` - number of elements between the start of adjacent strides.
- `elementsPerStride` - number of elements that are copied from `srcBlock` into `destBlock` for each stride.

Constraints:

- `scatter` is `Null`, `Unstarted`, or `Finished`.
- `srcBlock` and `destBlock` have been initialized and each has a positive capacity.
- `srcBlock` and `destBlock` have the same `elementSize`.
- All accesses to `srcBlock` and `destBlock` are within the capacity of each.

Called by:

- Controls.

StridedScatterS2B

```
typedef struct {
    svm_Kernel kernel;
    // not exposed
} svm_StridedScatterS2B;

void svm_stridedScatterS2BInit(
    svm_StridedScatterS2B* scatter,
    svm_Processor location,
    svm_IStream* srcStream,
    svm_OBlock destBlock,
    size_t destStart,
    size_t length,
    size_t destStride,
    size_t elementsPerStride,
    int untilEOS)
```

Moves segments of `elementsPerStride` elements from an input stream to strided locations of a destination block. When the `kernelRun` function is called, data elements are popped from `srcStream` and written to `destBlock` in a strided fashion starting at index `destStart`. The distance between strides in the destination (`destStride`) might be greater than the number of elements moved with each stride (`elementsPerStride`). In this case, the elements moved appear at the beginning of each stride.

Parameters:

- `scatter` - kernel to initialize.
- `location` - processor resource on which to execute the kernel (from the PCA Machine Model).
- `srcStream` - source stream from which elements are popped.
- `destBlock` - destination block into which elements are written.
- `destStart` - offset into the destination block where the first element is to be written. Subsequent elements are written in a strided fashion.
- `length` - total number of strides to perform.
- `destStride` - number of elements between the start of adjacent strides.
- `elementsPerStride` - number of elements that are moved from `srcStream` to `destBlock` for each stride.
- `untilEOS` - if true (anything except 0), ignore the `length` parameter and continue popping data elements from `srcStream` until an EOS tag is encountered.

Constraints:

- `scatter` is Null, Unstarted, or Finished.
- `srcStream` and `destBlock` have been initialized and have positive capacities.
- `srcStream` and `destBlock` have the same `elementSize`.
- All accesses to `destBlock` are within its capacity.

Called by:

- Controls.

StridedGatherB2B

```
typedef struct {
    svm_Kernel kernel;
    // not exposed
} svm_StridedGatherB2B;

void svm_stridedGatherB2BInit(
    svm_StridedGatherB2B* gather,
    svm_Processor location,
    svm_IBlock srcBlock,
    svm_OBlock destBlock,
    size_t srcStart,
    size_t destStart,
    size_t length,
    size_t srcStride,
    size_t elementsPerStride)
```

Copies segments of `elementsPerStride` data elements from strided locations of an input block to sequential locations of a destination block. When the `kernelRun` function is called, data elements are read from `srcBlock` in a strided fashion starting at index `srcStart` and written to `destBlock` sequentially starting at index `destStart`. The distance between strides (`srcStride`) might be greater than the number of elements copied from each stride (`elementsPerStride`). In this case, elements are copied from the beginning of the stride.

Parameters:

`gather` - kernel to initialize.
`location` - processor resource on which to execute the kernel (from the PCA Machine Model).
`srcBlock` - source block from which elements are read.
`destBlock` - destination block into which elements are written.
`srcStart` - offset into the source block from which first element is to be read.
 Subsequent elements are read in a strided fashion.
`destStart` - offset into the destination block where the first element is to be written.
 Subsequent elements are written sequentially.
`length` - total number of strides to perform.
`srcStride` - number of elements between the start of adjacent strides.
`elementsPerStride` - number of elements that are copied from `srcBlock` to `destBlock` for each stride.

Constraints:

`gather` is `Null`, `Unstarted`, or `Finished`.
`srcBlock` and `destBlock` have been initialized and have positive capacities.
`srcBlock` and `destBlock` have the same `elementSize`.
 All accesses to `srcBlock` and `destBlock` are within the capacity of each.

Called by:

Controls.

StridedGatherB2S

```
typedef struct {
    svm_Kernel kernel;
    // not exposed
} svm_StridedGatherB2S;

void svm_stridedGatherB2SInit(
    svm_StridedGatherB2S* gather,
    svm_Processor location,
    svm_IBlock srcBlock,
    svm_OStream* destStream,
    size_t srcStart,
    size_t length,
    size_t srcStride,
    size_t elementsPerStride,
    int setEOS)
```

Copies segments of `elementsPerStride` data elements from strided locations of an input block to a destination stream. Elements are read from `srcBlock` in a strided fashion starting at index `srcStart` and pushed onto `destStream`. The distance between strides (`srcStride`) might be greater than the number of elements copied from each stride (`elementsPerStride`). In this case, elements are copied from the beginning of the stride.

Parameters:

`gather` - kernel to initialize.

`location` - processor resource on which to execute the kernel (from the PCA Machine Model).

`srcBlock` - source block from which elements are copied.

`destStream` - destination stream onto which elements are pushed.

`srcStart` - offset into the source block from which first element is to be read. Subsequent elements are read in a strided fashion.

`length` - total number of strides to perform.

`srcStride` - number of elements between the start of adjacent strides.

`elementsPerStride` - number of elements that are copied from `srcBlock` to `destBlock` for each stride.

`setEOS` - if true (anything except 0), then the last element pushed onto `destStream` is given an EOS tag. Otherwise, no element pushed is tagged with an EOS.

Constraints:

`gather` is Null, Unstarted, or Finished.

`srcBlock` and `destStream` have been initialized and have positive capacities.

`srcBlock` and `destStream` have the same `elementSize`.

All accesses to `srcBlock` are within its capacity.

Called by:

Controls.

IndexedScatterB2B

```
typedef struct {
    svm_Kernel kernel;
    // not exposed
} svm_IndexedScatterB2B;

void svm_indexedScatterB2BInit(
    svm_IndexedScatterB2B* scatter,
    svm_Processor location,
    svm_IBlock srcBlock,
    svm_IBlock indexBlock,
    svm_OBlock destBlock,
    size_t srcStart,
    size_t indexStart,
    size_t length,
    size_t elementsPerIndex)
```

This kernel allows irregular scattering of data elements from an input block to an output block using values from an index block as the locations where elements are to be written. When the `kernelRun` function is called, the kernel iterates through `length` indices in `indexBlock` starting at index `indexStart`. For each element, it copies `elementsPerIndex` sequential data elements of the source block starting at index `destStart` and writes them starting at the specified index in `destBlock`.

Parameters:

`scatter` - kernel to initialize.

`location` - processor resource on which to execute the kernel (from the PCA Machine Model).

`srcBlock` - source block from which elements are copied sequentially.

`indexBlock` - index block containing the indices in `destBlock`.

`destBlock` - destination block into which elements are copied at specified indices.

`srcStart` - offset into the source block from which first element is to be read. Subsequent elements are read in a sequential fashion.

`indexStart` - offset into the index block from which first index is to be read. Subsequent indices are read in a sequential fashion.

`length` - total number of indices to be read from the index block.

`elementsPerIndex` - number of elements to move from `srcStream` to `destBlock` for each element of `indexStream`.

Constraints:

`scatter` is `Null`, `Unstarted`, or `Finished`.

`srcBlock`, `indexBlock`, and `destBlock` have been initialized and each has a positive capacity.

`srcBlock` and `destBlock` have the same `elementSize`.

`indexBlock` uses an unsigned representation for the indices.

All accesses to `srcBlock`, `indexBlock`, and `destBlock` are within their capacities.

Called by:

Controls.

IndexedScatterS2B

```
typedef struct {
    svm_Kernel kernel;
    // not exposed
} svm_IndexedScatterS2B;

void svm_indexedScatterS2BInit(
    svm_IndexedScatterS2B* scatter,
    svm_Processor location,
    svm_IStream* srcStream,
    svm_IStream* indexStream,
    svm_OBlock destBlock,
    size_t length,
    size_t elementsPerIndex,
    int untilEOS)
```

This kernel allows irregular scattering of data elements from an input stream to an output block using values from an index stream as the locations where elements are written. When the `kernelRun` function is called, `length` indices are popped off the index stream. For each index popped, `elementsPerIndex` elements of the source stream are popped and written to the destination block starting at that index.

Parameters:

`scatter` - kernel to initialize.
`location` - processor resource on which to execute the kernel (from the PCA Machine Model).
`srcStream` - source stream from which elements are moved.
`indexStream` - index stream containing the indices of `destBlock`.
`destBlock` - destination block into which elements are written at specified indices.
`length` - total number of indices to be read from the index block.
`elementsPerIndex` - number of elements to move from `srcStream` to `destBlock` for each element of `indexStream`.
`untilEOS` - if true (anything except 0), ignore the `length` parameter and continue popping data elements from `srcStream` until an EOS tag is encountered.

Constraints:

`scatter` is Null, Unstarted, or Finished.
`srcStream`, `indexStream`, and `destBlock` have been initialized and each has a positive capacity.
`srcStream` and `destBlock` have the same `elementSize`.
`indexStream` uses an unsigned representation for the indices.
All accesses to `destBlock` are within its capacity.

Called by:

Controls.

IndexedGatherB2B

```

typedef struct {
    svm_Kernel kernel;
    // not exposed
} svm_IndexedGatherB2B;
void svm_indexedGatherB2BInit(
    svm_IndexedGatherB2B* gather,
    svm_Processor location,
    svm_IBlock srcBlock,
    svm_IBlock indexBlock,
    svm_OBlock destBlock,
    size_t indexStart,
    size_t destStart,
    size_t length,
    size_t elementsPerIndex)

```

This kernel allows copying of data elements at irregular locations in an input block to sequential locations in an output block using values from an index block as the locations from which elements are read. When the `kernelRun` function is called, it iterates through `length` indices in `indexBlock` sequentially starting at index `indexStart`. For each index, it copies `elementsPerIndex` data elements of the source block starting at the specified index and writes them sequentially into `destBlock` starting at index `destStart`.

Parameters:

`gather` - kernel to initialize.
`location` - processor resource on which to execute the kernel (from the PCA Machine Model).
`srcBlock` - source block from which elements are copied at specified indices.
`indexBlock` - index block containing the indices in `srcBlock`.
`destBlock` - destination block into which elements are written sequentially.
`indexStart` - offset into the index block from which first index is to be read. Subsequent indices are read in a sequential fashion.
`destStart` - offset into the destination block where the first element is to be written. Subsequent elements are written in a sequential fashion.
`length` - total number of indices to be read from the index block.
`elementsPerIndex` - number of elements to copy from `srcBlock` to `destBlock` for each index read from `indexBlock`.

Constraints:

`gather` is `Null`, `Unstarted`, or `Finished`.
`srcBlock`, `indexBlock`, and `destBlock` have been initialized and each has a positive capacity.
`srcBlock` and `destBlock` have the same `elementSize`.
`indexBlock` uses an unsigned representation for the indices.
All accesses to `srcBlock`, `indexBlock`, and `destBlock` are within their capacities.

Called by:

Controls.

IndexedGatherB2S

```

typedef struct {
    svm_Kernel kernel;
    // not exposed
} svm_IndexedGatherB2S;

void svm_indexedGatherB2SInit(
    svm_IndexedGatherB2S* gather,
    svm_Processor location,
    svm_IBlock srcBlock,
    svm_IStream* indexStream,
    svm_OStream* destStream,
    size_t length,
    size_t elementsPerIndex,
    int setEOS,
    int untilEOS)

```

This kernel copies data elements at irregular locations in an input block to an output stream using values from an index stream as the locations from which elements are read. When the `kernelRun` function is called, it pops `length` indices in `indexStream`, and for each index, it copies `elementsPerIndex` elements of the source block at the specified location and pushes them onto `destStream`.

Parameters:

`gather` - kernel to initialize.
`location` - processor resource on which to execute the kernel (from the PCA Machine Model).
`srcBlock` - source block from which elements are copied.
`indexStream` - index stream containing the indices in `srcBlock` from which elements are to be read.
`destStream` - destination stream onto which elements are pushed.
`length` - total number of indices to be read from the index block.
`elementsPerIndex` - number of elements to copy from `srcBlock` to `destStream` for each element popped from `indexStream`.
`setEOS` - if true (anything except 0), then the last element pushed onto `destStream` is given an EOS tag. Otherwise, no element pushed is tagged with an EOS.
`untilEOS` - if true (anything except 0), ignore the `length` parameter and continue popping data elements from `indexStream` until an EOS tag is encountered.

Constraints:

`gather` is Null, Unstarted, or Finished.
`srcBlock`, `indexStream`, and `destStream` have been initialized and each has a positive capacity.
`srcBlock` and `destStream` have the same `elementSize`.
`indexStream` uses an unsigned representation for the indices.
All accesses to `srcBlock` are within its capacity.

Called by:

Controls.

6. Control and Kernel Threads

An application has a single control thread which is responsible for calling top-level kernels. The control originates with the application's `main()` function and may contain arbitrary C code. Top-level kernels may in turn call other, second-level kernels, and so on.

The following is an example of control code, in this case compressing some data using Run Length Encoding (RLE) and Zip algorithms. For the sake of space, it is assumed these kernels have been defined by the user elsewhere. It is also assumed that memory and processor resources referenced in the example (*e.g.*, GLOBALMEM1) have been defined in the PCA Machine Model.

```
#define TRUE 1
#define SIZE_THRESHOLD 512
int length1, length2;
svm_Stream s0, s1, s2, s4;
svm_Block scratch;
svm_MoveS2S move01, move24;
RLE rle; // Source code for the user-defined Run Length Encoding
        // kernel is not included in this example

// Set the status of all kernels to Null
svm_kernelInitNull (&move01);
svm_kernelInitNull (&move24);
svm_kernelInitNull (&rle.kernel);

// Read file into global memory (not part of SVM API)
length1 = readFile("input.dat", GLOBALMEM1, 0x1000, 1024);

// Create streams and blocks in memory
svm_streamInitWithDataRAM (&s0, GLOBALMEM1, 0x1000, 1024, 4, length1,
                           TRUE, svm_Stream_Never_Wraps);
svm_streamInitRAM (&s1, LOCALMEM1, 0x0, 256, 4,
                  svm_Stream_Unaliased_RAM);
svm_streamInitRAM(&s2, LOCALMEM1, 0x100, 128, 4,
                  svm_Stream_Never_Wraps);
svm_blockInit (&scratch, LOCALMEM1, 0x180, 32, 1);

// Move from memory
svm_moves2SInit(&move01, DMA1, &s0, &s1, 0, TRUE, TRUE);
svm_kernelRun (&move01.kernel);

// Run rle (source for rleInit not included in this example)
rleInit(&rle, PROC1, &scratch, &s1, &s2);
svm_kernelRun (&rle.kernel);

// Get output length
svm_kernelWait (&rle.kernel);
length2 = rle.outputLength;
```

```

// If the output is still too large, run additional compression,
// overwriting s2 in place
if (length2 > SIZE_THRESHOLD) {
    svm_Stream s3;
    svm_Block scratch2;
    Zip zip; // Source code for the user-defined Zip kernel
             // is not included in this example
    svm_kernelInitNull (&zip.kernel);

    svm_streamInitRAM (&s3, LOCALMEM1, 0x100, 128, 4,
                      svm_Stream_Never_Wraps);
    svm_blockInit (&scratch2, LOCALMEM1, 0x180, 32, 1);

    // Run zip kernel (source for zipInit not included in this example)
    // where s2 is the input stream and s3 is the output stream.
    // Note that s3 uses the same memory and address as does s2; this
    // is ok because the zip kernel does data compression and never
    // pushes more elements than have already been consumed.
    zipInit(&zip, PROC1, &scratch2, &s2, &s3);
    svm_kernelRun (&zip.kernel);

    // Get output length
    svm_kernelWait (&zip.kernel);
    length2 = zip.outputLength;
    svm_streamInitWithDataRAM (&s2, LOCALMEM1, 0x100, 128, 4, length2,
                              TRUE, svm_Stream_Never_Wraps);
}

svm_streamInitRAM (&s4, GLOBALMEM1, 0x2000, 1024, 4,
                  svm_Stream_Never_Wraps),

// Move to memory
svm_moveS2SInit(&move24, DMA1, &s2, &s4, 0, TRUE, TRUE);
svm_kernelRun (&move24.kernel);
svm_kernelWait (&move24.kernel);

// Store the result from memory to "output.dat" (not part of API)
writeFile("output.dat", GLOBALMEM1, 0x2000, length2);

```

6.1. Transferring Data Between Kernels

It is often necessary to transfer data between two kernels. Perhaps the most natural way to do this is to use the output stream of one kernel as the input stream for another; in the example above, stream s2 transfers data between multiple kernels.

In the case where only a portion of the data from one stream needs to be transferred, it is possible to alias a new input stream on top of an existing output stream in memory. The new stream may overlap the output stream such that it only transfers a portion of the data. In this case, the control code should indicate that the kernel reading from the new stream is dependent on the kernel that wrote the original stream; this is done by using the reading kernel's `svm_kernelAddDependence` function. The use of `svm_kernelAddDependence` ensures that the LLC is not required to do alias analysis to discover dependencies between kernels.

If data stored in blocks needs to be transferred between two kernels, the output block may be reused as an input block by the receiving kernel or aliasing may be used to transfer only a portion of the data. In both cases, the control code must use the receiving kernel's `svm_kernelAddDependence` function to eliminate the need for alias analysis in the LLC.

6.2. Transferring Data Between Control Code and Kernels

Before `kernelRun` is executed, streams can be initialized with data from the control code. Conversely, the results of a kernel can be used in the control code following the kernel's execution. Data from the kernel is guaranteed to be available to the control thread after `svm_kernelWait` returns. Both of these data transfers are done through memory, either by directly accessing the memory assigned to a stream or block using the move family of kernels, or by calling I/O functions from the control code. The control code can rely on the sequential data layout guaranteed by streams and blocks when managing data transfer (See Section 3). Note that device I/O such as file handling and terminal interaction can be done by the control code and the results can then be moved into streams. However, by describing I/O devices as processors that support a single user-defined kernel, it is possible to eliminate the need to move the results in memory.

It is also possible for the control processor to inspect and modify the state variables of a kernel when it is not in the `Waiting` or `Running` state. Accesses to kernel fields are useful for passing parameters to a kernel or retrieving reduction values from a kernel. For example:

```
#define TRUE 1

// --- Kernel Code ---
typedef struct {
    svm_Kernel kernel;
    svm_IStream* in;
    int sum;
} SumKernel;

void sumKernelWork(SumKernel* k) {
    int x;
    while (!svm_streamPeekEOS (k->in, 0)) {
        svm_streamPop (k->in, &x);
        k->sum += x;
    }
    svm_streamPop(k->in, &x);
    k->sum += x;
}

void sumInit(SumKernel* k, svm_Processor procLocation,
             svm_IOBlock scratch, svm_IStream* _in) {
    svm_kernelInit (&k->kernel, procLocation, scratch, (void*) k,
                   sizeof(SumKernel), (svm_ExtKernelWork) &sumKernelWork);
    k->in = _in;
    k->sum = 0;
}
```

```
// --- Control Code ---
svm_Stream s1;
svm_Block scratch1;
SumKernel sk;
int finalSum;

svm_kernelInitNull (&sk.kernel);
// assumes prior code initializes data in RAM
svm_streamInitWithDataRAM (&s1, LOCALMEM1, 0x100, 128, 4, 128,
    TRUE, 0);
svm_blockInit (&scratch1, LOCALMEM1, 0x180, 16, 1);

// Run sum kernel
sumInit(&sk, PROC1, &scratch1, &s1);
svm_kernelRun (&sk.kernel);

// Get output sum
svm_kernelWait (&sk.kernel);
finalSum = sk.sum;
```

6.3. FIFO Cleanup

It is the responsibility of the HLC to ensure that all elements are drained from a FIFO before a new stream is mapped to it. In some cases, a FIFO is empty when the kernel that it feeds terminates because the kernel executed until it popped the last item, which had an EOS tag. However, in other cases, a kernel might finish before reaching the EOS tag or multiple EOS tags could be present in a stream, and leftover items would need to be cleared from the FIFO before additional kernels could use it.

The HLC can explicitly clear a FIFO using the `svm_streamClearFIFO` command.

7. Machine Model Interaction

This section describes functions for obtaining efficient IDs for hardware resources and configuring polymorphous hardware. It assumes familiarity with the PCA Machine Model [9].

The SVM API does not refer to instances of machine model processors, memories, levels, and morphs using their names as defined by the PCA Machine Model naming convention. Instead, it refers to processors, memories, levels, and morphs using efficient IDs which are implementation-specific types;

```
typedef implementation-specific-type svm_Processor;  
typedef implementation-specific-type svm_Memory;  
typedef implementation-specific-type svm_Level;  
typedef implementation-specific-type svm_Morph;
```

This section describes functions for obtaining the IDs of processors and memories from their machine model names.

The SVM API includes calls to configure morphable hardware resources. When a system boots or starts an SVM component on a subset of its hardware it is assumed to configure the default morphs specified by the machine model and start the `main()` function on the initial processor specified by the machine model. The `main()` function then configures the rest of the available hardware as desired using the functions given in this section.

getProcessor

```
svm_Processor svm_getProcessor(  
    char* processorName)
```

Returns the SVM processor ID corresponding to the specified processor instance name.

Parameters:

processorName - name of the processor instance

Constraints:

processorName must be the name of a valid processor instance.

Called by:

Controls.

getMemory

```
svm_Memory svm_getMemory(  
    char* memoryName)
```

Returns the SVM memory ID corresponding to the specified memory instance name.

Parameters:

memoryName - name of the memory instance

Constraints:

memoryName must be the name of a valid memory instance.

Called by:

Controls.

getLevel

```
svm_Level svm_getLevel(  
    char* levelName)
```

Returns the SVM morph ID corresponding to the specified level instance name.

Parameters:

levelName - name of the level instance

Constraints:

levelName must be the name of a valid level instance.

Called by:

Controls.

getMorph

```
svm_Morph svm_getMorph(  
    char* morphName)
```

Returns the SVM morph ID corresponding to the specified morph instance name.

Parameters:

morphName - name of the morph instance

Constraints:

morphName must be the name of a valid morph instance.

Called by:

Controls.

setConfiguration

```
void svm_setConfiguration(  
    int numMorphs,  
    svmMorph* morphs,  
    int numProcessors,  
    svm_Processor controlProcessor,  
    svm_Memory globalMemory)
```

Configures the system to contain the specified morph instances, and only those instances. If a processor instance configured before the call is not configured after the call, any kernel it is executing is terminated and all state is lost. If a memory instance configured before the call is

not configured after the call, any data it contains is lost. If any resource instance is configured both before and after the call, it is unaffected.

Parameters:

numMorphs – number of morph instances to configure
morphs – array of morphs instances to configure
numProcessors – total number of processors that will be configured
controlProcessor – processor that will execute the control thread
globalMemory – memory that will be used as the global memory

Constraints:

numProcessors must be equal to the total number of processors in the specified morphs.
controlProcessor must be a processor instance in the specified morphs.
globalMemory must be a memory instance in the specified morphs that is accessible by controlProcessor.

Called by:

Controls.

getConfiguration

```
void svm_getConfiguration(  
    int* numMorphs,  
    svmMorph* morphs)
```

Sets numMorphs to the number of morphs currently configured and sets morphs to the morph instances currently configured.

Parameters:

numMorphs – pointer to variable to be set to number of morph instances configured
morphs – pointer to the start of an array to be filled with the list of morph instances configured

Constraints:

morphs must point to a buffer large enough to hold the list of morph instances.

Called by:

Controls.

finalizeConfiguration

```
void svm_finalizeConfiguration()
```

Frees all configured resources.

Parameters:

None.

Constraints:

None .

Called by:

Controls.

7.1. Machine Model Restrictions

Code executed on a processor must conform to the processor's capabilities as described in the machine model. Some processors may not support all opcodes and data types:

1. Supported opcodes are only the logical, arithmetic, and boolean operations found in C. There are no special-purpose DSP operations, although DSP operations may be added at a future date pending further discussion by the Morphware Forum.
2. Supported primitive types are indicated by the set of `mm.Proc.Supports` fields in the PCA Machine Model. For example, the `mm.Proc.SupportsInt` field indicates whether or not integers are supported. There is also support for arrays with a fixed (`int` literal) length and `struct`'s containing members of any other type.

If the processor that will execute the kernel is not connected to the global memory or a data cache backed by the global memory, then the code must not access global data directly and the local stack size must be statically bounded. Practically, this means:

1. No recursive functions (allowing a bounded stack size).
2. No pointers, except for those directly passed to stream, block, and kernel functions as described in this document.
3. No dynamic memory allocation.
4. No accesses to global variables.
5. No calls to functions that violate any of these restrictions.

Non-coherent caches impose an additional restriction:

1. The same cache line must not be written through two different caches.

An architecture should be able to implement any compliant SVM code generated by a valid HLC using a machine model consistent with that architecture.

8. Error Handling

If any of the restrictions in Section 7.1 are violated or if there is an operation that is disallowed by the API (*e.g.*, peeking beyond the capacity of a stream), the behavior is undefined. Error detection, error handling, and error recovery are defined by the implementation of the LLC. It is recommended that each LLC provide a mode where it checks assertions on the restrictions above and fails cleanly if a restriction is violated. However, these assertions could be turned off to obtain higher performance if the LLC trusts that the HLC has generated safe code. Note that there is no way for the HLC to ensure that the original source program is safe.

The potential for undefined behavior is introduced for the sake of performance. If each architecture were required to verify the restrictions above, the overhead would be prohibitive.

References

1. Conte, T.M.; Dubey, P.K.; Jennings, M.D.; Lee, R.B.; Peleg, A.; Rathnam, S.; Schlansker, M.; Song, P.; Wolfe, A. "Challenges to combining general-purpose and multimedia processors." *Computer*, Volume: 30, Issue: 12, December 1997.
2. Diefendorff, K.; Dubey, P.K. "How multimedia workloads will change processor design." *Computer*, Volume: 30, Issue: 9, September 1997.
3. The Polymorphous Computing Architectures (PCA) Program, Information Processing Technology Office (IPTO), Defense Advanced Research Projects Agency (DARPA), <www.darpa.mil/ipto/Programs/pca/index.htm>.
4. The PCA Morphware Forum. <www.morphware.org>.
5. "Introduction to Morphware: Software Architecture for Polymorphous Computing Architectures." Version 1.0, February 23, 2004. Available at <www.morphware.org>.
6. Mattson, P.; Schweitz, E.; Engle, M.; Litvinov, V.; Mackenzie, K. "R-Stream 1.0 Brook Clarification." Reservoir Labs, Inc. November 5, 2003. Available as part of R-Stream 1.0 download package at <www.morphware.org>.
7. "StreamIt Language Specification." Version 2.0, MIT Computer Science and Artificial Intelligence Laboratory. October 17, 2003. <cag.lcs.mit.edu/streamit/papers/streamit-lang-spec.pdf>.
8. R-Stream 2.1 release package. Reservoir Labs, Inc. March 2006. Package may be requested via the Morphware Forum web site <www.morphware.org>.
9. "PCA Machine Model", version 1.2, January 2007. Available at <www.morphware.org>.

Index of SVM Calls and Data Types

Kernel Base Data Type	16
mm.Proc.RequiredMaster	5
mm.Proc.SupportsBlockDMA	5
mm.Proc.SupportsStreamDMA	5
mm.Proc.SupportsUserCode	5
Pre-Defined Kernels.....	27
svm_Block	14, 40
SVM_BLOCK_READ	15
SVM_BLOCK_WRITE.....	15
svm_blockInit	14, 40, 43
svm_finalizeConfiguration	46
svm_getConfiguration.....	46
svm_getLevel.....	45
svm_getMemory	44
svm_getMorph	45
svm_getProcessor	44
svm_IBlock.....	14
svm_IndexedGatherB2B.....	37
svm_IndexedGatherB2S	38
svm_IndexedScatterB2B.....	35
svm_IndexedScatterS2B.....	36
svm_IOBlock	14
svm_IOStream	6, 7
svm_IStream	6, 7
svm_Kernel	16, 26
svm_Kernel_Finished	17
svm_Kernel_Null.....	17
svm_Kernel_Paused.....	17
svm_Kernel_Running	17
svm_Kernel_Status	17

svm_Kernel_Unstarted	17
svm_Kernel_Waiting	17
svm_kernelAddDependence	17, 19, 20, 21 , 41
svm_kernelEnd	18, 23
svm_kernelGetStatus	25
svm_kernelInit	18, 20, 21 , 24, 26, 42
svm_kernelInitNull	17, 20 , 40, 43
svm_kernelPause	17, 23
svm_kernelRun	17, 18, 22 , 40, 43
svm_kernelWait	19, 23, 24 , 40, 41, 42, 43
svm_kernelWaitMultiple	25
svm_Level	44
svm_Memory	44
svm_Morph	44
svm_MoveB2B	27
svm_MoveB2S	28
svm_MoveS2B	29
svm_MoveS2S	30 , 40
svm_OBlock	14
svm_OStream	6 , 7
svm_Processor	44
svm_setConfiguration	45
svm_Stream	6 , 40
svm_Stream_Flags	7
svm_Stream_Never_Wraps	7, 40
svm_Stream_One_EOS	6, 7
svm_Stream_Unaliased_RAM	6, 7, 9, 40
svm_Stream_Unordered	7
svm_streamClearFIFO	10 , 43
svm_streamInitFIFO	9
svm_streamInitRAM	8 , 9, 40
svm_streamInitWithDataRAM	8 , 40, 43

svm_streamPeek	7, 12
svm_streamPeekEOS	6, 7, 9, 13, 42
svm_streamPop	11, 12, 26, 42
svm_streamPush	10
svm_streamPushMulticast	10
svm_streamPushMulticastWithEOS	12
svm_streamPushWithEOS	6, 12
svm_StridedGatherB2B	33
svm_StridedGatherB2S	34
svm_StridedScatterB2B	30
svm_StridedScatterS2B	32
User-Defined Kernels	26



©2007 Georgia Tech Research Corporation, all rights reserved.



APPENDIX E

Last Available Version of the Morphware Stable Interface Document

THREADED VIRTUAL MACHINE – HARDWARE ABSTRACTION LAYER (TVM-HAL) SPECIFICATION

Version 1.0

July 2006

Threaded Virtual Machine – Hardware Abstraction Layer (TVM-HAL) Specification

Version 1.0

July 2006



©2006 Georgia Tech Research Corporation, all rights reserved.

A non-exclusive, non-royalty bearing license is hereby granted to all persons to copy, modify, distribute and produce derivative works for any purpose, provided that this copyright notice and following disclaimer appear on all copies: THIS LICENSE INCLUDES NO WARRANTIES, EXPRESSED OR IMPLIED, WHETHER ORAL OR WRITTEN, WITH RESPECT TO THE SOFTWARE OR OTHER MATERIAL INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF PERFORMANCE OR DEALING, OR FROM USAGE OR TRADE, OR OF NON-INFRINGEMENT OF ANY PATENTS OF THIRD PARTIES. THE INFORMATION IN THIS DOCUMENT SHOULD NOT BE CONSTRUED AS A COMMITMENT OF DEVELOPMENT BY ANY OF THE ABOVE PARTIES.

This material is based in part upon work supported by the U.S. Defense Advanced Research Projects Agency (DARPA) and other agencies of the U.S. Department of Defense (DoD). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or the DoD.

The US Government has a license under these copyrights, and this material may be reproduced by or for the U.S. Government.



Acknowledgements

Authors

The primary author of this document was:

Lance Hammond Stanford University

The author wishes to thank the members of the Morphware Forum who reviewed and contributed to this document. The author also thanks the Defense Advanced Research Projects Agency (DARPA) for its support of this work.

The Morphware Forum

The Morphware Forum is a joint activity of the participants in DARPA's Polymorphous Computing Architectures (PCA) program, as well as other interested developers of embedded computing hardware, software, and application technology. The purpose of the Morphware Forum is to define an open, portable software environment for the development of high performance applications on PCA platforms. Morphware Forum products and information are available at www.morphware.org.

The following organizations are voting members of the Morphware Forum at this writing:

- o Defense Advanced Research Projects Agency
- o BAE Systems
- o Black River Systems Company, Inc.
- o Georgia Institute of Technology
- o Honeywell Space Systems
- o Lockheed Martin Advanced Technology Laboratory (ATL)
- o Lockheed Martin Maritime Systems and Sensors (MS2)
- o Mercury Computer Systems, Inc.
- o Massachusetts Institute of Technology
- o MIT Lincoln Laboratory
- o MITRE
- o National Reconnaissance Office
- o Raytheon Corporation
- o Reservoir Labs, Inc.
- o Rose-Hulman Institute of Technology
- o Stanford University
- o University of Southern California Information Sciences Institute
- o University of Texas, Austin
- o U. S. Air Force Research Laboratory, Eglin AFB Site
- o U. S. Air Force Research Laboratory, Rome NY Site
- o U. S. Air Force Research Laboratory, Wright-Patterson AFB Site

-
- o U.S. Army Tank Automotive Research, Development, and Engineering Center (RDECOM)
 - o U.S. Navy SPAWAR Systems Center

Additional contributing organizations include:

- o Exogi, LLC
- o IBM Austin Research Laboratory
- o Lockheed Martin Aerospace
- o Nallatech Ltd.
- o North Carolina State University
- o Northeastern University
- o Pentum Group, Inc.
- o Protean Devices, Inc.
- o University of Maryland
- o University of Pennsylvania
- o Vanderbilt University

Document Change History

Version 1.0 is the first approved version of the document.

Table of Contents

Acknowledgements.....	i
Authors.....	i
The Morphware Forum.....	i
Document Change History.....	iii
Table of Contents.....	iv
List of Acronyms	vi
1 Introduction.....	1
2 Variable Usage Transformations	5
2.1 Local Variables	5
2.2 Global Variables	6
2.3 Pointer-Based Accesses	7
3 Exception Handling Model	9
3.1 Exception Vector Table System.....	9
3.2 Special Code in Exception Handlers.....	13
3.2.1 Linking of Low-Level Run-time System.....	13
3.2.2 Compiler Register Control.....	14
3.2.3 Register Save/Restore for Context Switches	14
3.2.4 Compiler Stack and Global Pointer Control	15
3.2.5 System Control.....	18
3.3 PCA Processor Control.....	21
3.3.1 Compiler→Processor Control.....	21
3.3.2 Cold-Boot Reset of TVM Nodes	21
3.3.3 Morphing.....	22
3.4 Multithreading Within Cores	23
4 Memory Control Model	26
4.1 Physical Memory View.....	26
4.1.1 Physical Memory Bank Metadata.....	27
4.1.2 Physical Mapping Example	30
4.2 Segment Control	32

4.2.1	Metadata Access	32
4.2.2	Segmentation Control Routines	35
4.2.3	Segmentation Example	37
4.3	Paged Memory Control	38
5	Other Extensions	43
5.1	Processor Identification	43
5.2	Active DMA.....	43
5.3	Multiprocessor Synchronization	45
5.4	Cache Memory Control.....	46
5.5	IEEE 754 Floating Point Control.....	47
5.6	Performance Counter Control and Access.....	48
5.7	Power/Performance Tradeoff Parameters	49
5.8	Data Watchpoint Control	50
5.9	Generic Special Register Access	51
6	TVM-HAL Reference	52
6.1	Compiler Directives	52
6.1.1	Processor Selection	52
6.1.2	Marking Exception Handlers	53
6.1.3	Register File Control.....	53
6.1.4	Variable Access Control Directives.....	54
6.2	HAL Macros (or Functions).....	57
6.2.1	Load and Store Replacements for Pointer References.....	57
6.2.2	Exception and Machine Control Macros	59
6.2.3	Memory Control Macros.....	83
6.2.4	Other Macros	104
6.3	User-Mode HAL routines	128
7	References.....	130
8	Index of HAL Directives, Functions, and Constants	133

List of Acronyms

ANSI	American National Standards Institute
API	Application Programming Interface
BIOS	Basic Input-Output System
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
DARPA	Defense Advanced Research Projects Agency
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
FP	Frame Pointer
GP	Global Pointer
HAL	Hardware Abstraction Layer
HLC	High Level Compiler
IBM	International Business Machines
I/O	Input/Output
ISA	Instruction Set Architecture
LLC	Low Level Compiler
LRU	Least Recently Used
MESI	Modified, Exclusive, Shared, Invalid (cache coherency protocol)
MM	Machine Model
MMX	Intel SIMD instruction set
MSI	Morphware Stable Interface
MTSPR/MFSPR	Move To/From Special-Purpose Register
NOP	No Operation
OS	Operating System
PC	Program Counter
PCA	Polymorphous Computing Architectures
POSIX	Portable Operating System Interface
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
SIMD	Single Instruction Multiple Data
SMP	Shared Memory Processor
SMT	Simultaneously Multithreaded

SP	Stack Pointer
SRAM	Static Random Access Memory
SSE	Streaming SIMD Extensions (Intel)
SVM	Streaming Virtual Machine
TLB	Translation Lookaside Buffer
TVM	Threaded Virtual Machine
UVM	User-level Threaded Virtual Machine
VM	Virtual Machine

1 Introduction

A number of communication-exposed architectures are being developed as part of the Polymorphous Computing Architectures (PCA) initiative [1]. The computing, communication, and memory resources of these machines can be configured to balance the demands of the application and the constraints of the environment. Mapping an application to these architectures is a challenging compiler problem. To obtain good performance, the compiler needs to detect opportunities for data, task, and pipeline parallelism in the source program and calculate a load-balanced mapping onto the processor resources while managing the large array of possible machine configurations.

To address this challenge, the Morphware Forum [2] is defining elements of a portable application development methodology for architectures participating in the PCA program. The methodology includes new or modified source languages, a new application development process, and a framework for expressing system and application metadata. The methodology includes a set of PCA architecture-neutral software development standards referred to as the Morphware Stable Interface (MSI) [3]. One aspect of the MSI is an application component compilation strategy that partitions the compilation process into two stages: a high-level compilation stage and a low-level compilation stage.

The high-level compiler (HLC) accepts application source code and an abstract description of the target architecture, expressed as a PCA Machine Model (MM) [4], and produces a coarse-grain mapping of the application code to the physical resources available in the given architecture. The mapping is expressed as a program expressed in a virtual machine (VM) language based on either threaded or streaming processing. The virtual machine code is subsequently compiled into executable code by architecture-specific low-level compilers (LLCs).

C/C++ is the baseline language for all VM language specifications. The TVM-HAL defines extensions to C. For simplicity, the term “C” will be used to refer to both C and C++ in this document, since the extensions to C proposed here are independent of the object-oriented extensions provided by C++. C is chosen for two major reasons. First, although it is a high-level language, it is easier to understand how code written using C will be converted into machine instructions than is the case with most other high-level languages. Second, C offers direct access through operators to most elemental machine instructions, such as bitwise logic and a variety of integer lengths, that are present on virtually all general-purpose CPUs today, yet are unsupported by many other languages. Consequently, C is well-suited for writing most parts of both user code and the supporting library or run-time code. Many operating systems are written in C in order to achieve portability from one architecture to another.

However, standard C does not possess enough functionality to describe all of the instructions that PCA high level compilers (HLCs) are required to emit. Specifically, certain architectural features of PCA processors require native code generation of instructions that cannot be described using standard C constructs. Therefore, C language extensions must be included in the virtual machines to address these particular code generation requirements. Currently, three main areas are recognized where such extensions may be helpful:

- **Streaming Hardware:** Extensions are needed to generate code for coprocessors optimized to handle vector-like *streams* of data. Adding explicit stream extensions to the

specification allows the high-level compilers to handle the vectorization and coarse grain optimization required for efficient stream processing. These features do not then have to be developed in each of the architecture-specific LLCs.

- **Multimedia Instructions:** C lacks the ability to express operations that are now commonly encoded as multimedia instruction set architecture (ISA) extensions, such as those in Intel's multimedia extensions (MMX) and streaming SIMD extensions (SSE) or Motorola's AltiVec architecture. Many PCA architectures will have functional units for some multimedia operations. Thus, the PCA VMs should include constructs designed to allow the LLC to use these instructions.
- **System Control Instructions:** All architectures possess specialized instructions used by operating system code to control the hardware at a low level in order to control the exception model for the machine, manage memory, synchronize processors, and handle input/output (I/O). Most operating systems written in C access these instructions by calling specialized assembly-language routines or through the use of inline assembly-language instructions. By listing these fundamental functions and implementing them as a standard set of macros or very small functions, a *hardware abstraction layer* can be created to provide the HLC with an architecture-neutral representation of the core functions required by any operating system (OS) or run-time environment.

These three classes of extension are largely orthogonal to one another. Streaming portions of the application use the Streaming Virtual Machine (SVM) Application Programming Interface (API) [5], while threaded parts of the application use an API based on either the User-level Threaded Virtual Machine (UVM) [6] or the low-level Threaded Virtual Machine Hardware Abstraction Layer (TVM-HAL). This document describes the TVM-HAL specification, which provides the functionality required for low-level system control. There is currently no MSI specification for access to multimedia processor instructions.

The TVM-HAL extensions allow the HLCs to generate TVM code for both user threads and most low-level OS or run-time code. The same low-level compiler can then compile both sets of code. User code accesses virtually all machine resources through system calls to an OS support library, while the OS code uses TVM-HAL primitives to directly control the machine. Since most TVM-HAL primitives will trigger privileged instruction exceptions if used in user code, this distinction is enforced at run time. Exceptions to this rule exist for certain multiprocessor synchronization and cache control TVM-HAL functions, which can often be performed without invoking privileged instructions. A consequence of this compilation framework is that threaded code without an OS can be implemented as TVM code that implements the application (like normal user code), while controlling the underlying hardware directly using privileged TVM-HAL primitives (like run-time or OS code). While some applications that target PCA architectures may be written in this manner, most will use at least a minimal OS that adds an interface for ease-of-use and important features such as error checking and memory protection on top of the TVM-HAL primitives.

A consequence of the MSI virtual machine architecture is that most of the TVM-HAL interface is not itself designed to be a target for any HLC, even though it is a part of the VM model that has been designed to be a high-level compiler target. The small subset of TVM-HAL constructs amenable to user code use can be targeted by a high-level compiler. For example, the variable access and global pointer controls **must** generally be used by any high-level compiler targeting

the virtual machine. Also, parallelizing high-level compilers may insert some of the TVM-HAL macros directly instead of using packaged threading libraries like `pthread`s in order to get maximum performance. However, most of the routines defined in this document will be used by authors of OS or run-time code only, as no existing high-level languages known to the authors need to access these functions directly. Hence, TVM-HAL routines will generally be added to OS or run-time code by hand in order to get the proper functionality. This will result in OS or run-time systems made up of two different types of files. The first type, for performing low-level machine interaction, will be made of C manually combined with TVM-HAL primitives. The second, for performing higher-level functions that do not require direct control of special machine functions, can be written in standard C and parsed by a relatively simple C-to-TVM-HAL translator. Figure 1 illustrates how such a translator acts as a TVM-HAL “high level compiler” in the PCA HLC/LLC framework. The role of the “low-level compiler” is implemented with a relatively standard C compiler, augmented with extensions to interpret the various TVM-HAL macros and compiler directives defined in this document. Some extensions may be implemented by simply linking with appropriate architecture-specific libraries. This LLC is sometimes referred to in this document as a “TVM Compiler”.

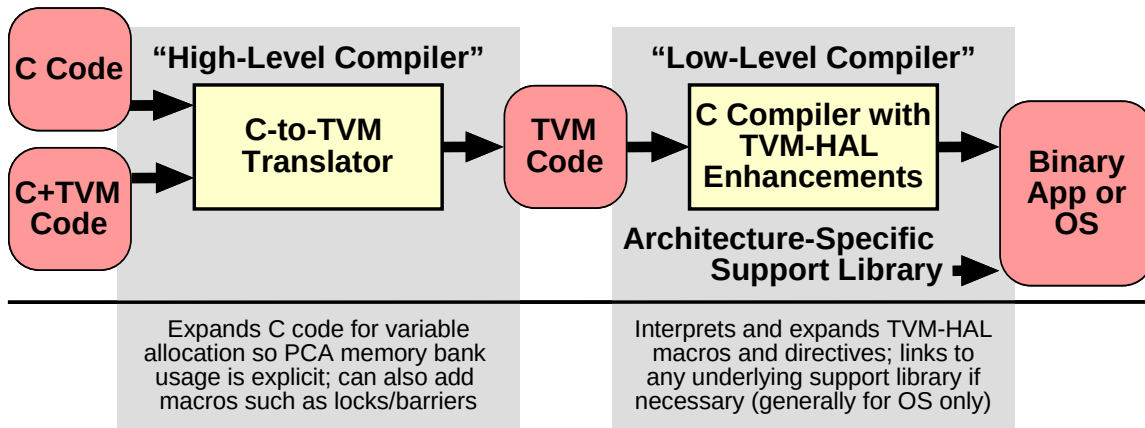


Figure 1. Two-phase PCA compile structure for C code on PCA architectures using the TVM-HAL extensions.

The C-to-TVM-HAL translator tool will take ANSI C or C++ code and convert it to TVM-HAL code. This conversion consists primarily of using the `HAL_LOAD` and `HAL_STORE` primitives defined in Section 2 of this document to add notations to the memory references that specify the physical memory bank of the PCA architecture to be associated with each reference. These notations are required to support the multi-banked memory architectures typical of PCA chips, in contrast to the conventional C model of a single, monolithic memory bank.

The Morphware Forum is working to develop OS and run-time code that will use the TVM-HAL primitives in order to provide a convenient process-and-thread abstraction to user-level code. This effort is complementary to the TVM-HAL API specified in this document, and will result in a series of code layers under typical applications that looks something like the following, from the lowest-level to highest:

1. **Bare Hardware:** The actual machine instructions

2. **The TVM-HAL:** These macros (or small functions) wrapped around the system-specific hardware interfaces, allow all higher layers to be written in a portable manner.
3. **UVM [6]:** A common “kernel” of routines that most full operating systems can be built around. This mostly consists of adding some protection and error-checking code around the TVM-HAL macros, to keep malicious or buggy code from misusing them, making the memory interfaces more multiprocessor-friendly, and adding a scheduling kernel.
4. **The “OS”:** A collection of interfaces that user programs can use to request/control system resources. This could be something complex, like Posix/UNIX, or something much simpler, like an embedded OS. This layer adds the features needed by the current selection of user programs, but are not needed by essentially *all* systems. Some of these features may be simple extensions of the lower-level layers, such as adding control over real-time issues for a real-time OS or more protection code for a system that may be targeted by malicious code. Most of the code at this level, however, will be more “housekeeping” code such as I/O drivers, file system management code, network socket interfaces, and so on.
5. **User Threads:** The actual application programs.

2 Variable Usage Transformations

The most significant difference between normal C code and TVM-HAL code is in the memory access model. C provides three ways of accessing data: explicitly through pointers, and implicitly through named local or global variables. Because PCA architectures have such a complex memory map (discussed in more detail in Section 4), this triad of variable access mechanisms is not sufficient to determine where in memory variables may exist, and therefore must be augmented. This section describes the differences between variable definitions and usages in TVM-HAL code and standard C.

Some TVM-HAL extensions are included primarily to aid optimization of object code. Some PCA systems may require different machine instructions or instruction sequences to access different physical memories. Thus, virtual machine code must generally notify the LLC as to which memory will be accessed by each reference so the machine code generated can be optimized for the indicated memory bank. Several of the functions described in this specification supply compile-time constants to the LLC using a `memory` parameter. The allowable values of this parameter are defined for any particular architecture in its machine model metadata, as numbers assigned to each of the different memory blocks in the system [4]. This metadata is defined more completely in Section 4.1. If the HLC is unable to determine which memory bank may hold a piece of data at compile time, a variable may be assigned to memory bank `HAL_ANY_MEMORY`, a constant (generally zero) that tells the LLC that no memory block has been assigned. Use of this mechanism should be minimized since the HLC must then include code to allow the system to access any of the different memories available in the system.

2.1 Local Variables

Local variables in the TVM-HAL behave in the same manner as in standard C. Upon entry to a function, the compiler generates a stack frame to contain the variables, which can then be referred to by name. An exception to this rule is the beginning of exception handlers, when no stack exists to hold a stack frame for locals. The TVM-HAL provides special routines for establishing a new stack so that normal local variable operation may commence; see Section 3.2.4.

While no routines are required for the use of individual local variables, the following compiler directive indicates the current stack location to the LLC:

```
HAL_STACK_LOCATION(const int memory);
```

This directive can only be set between functions in a `.c` file, and affects all subsequent functions. There may also be a default location for the stack in some systems, especially those in which only one memory bank is appropriate for the task. Otherwise, it is recommended that the HLC insert `HAL_STACK_LOCATION` directives at the beginning of every `.c` file or before every function to ensure that the stack location is always clearly defined.

2.2 Global Variables

Once a stack is allocated, local variables can be defined and allocated by the low-level compiler. In contrast, global variables may be individually placed at arbitrary locations in one or more of the multiple memories of a PCA architecture. As a result, more care is required in their use.

Accesses to global variables are unchanged from C. However, the *definition* of global variables in the TVM-HAL is more complex than in C. The HAL defines several compiler directives that define the origin points of blocks in memory for holding groups of global variables, and then subsequently control the assignment of variables to these blocks:

```
/* Define a "global block" to contain global variables */
HAL_GLOBAL_BLOCK(new identifier, const int memory,
    const HAL_bool grow_upwards );
HAL_GLOBAL_BLOCK_BASE(new identifier, const int memory,
    const HAL_bool grow_upwards, const void *base_absolute_address);

/* Select which "global block" to allocate global variables to */
HAL_GLOBALS(identifier of global block);
/* All globals following this declaration are added to the global
   block. */

HAL_GLOBALS_INIT(identifier of global block);
/* All pre-initialized globals following this declaration are added
   to the global block. Uninitialized globals continue to be placed
   in the block defined with the last HAL_GLOBALS or
   HAL_GLOBALS_UNINIT.*/

HAL_GLOBALS_UNINIT(identifier of global block);
/* All uninitialized globals following this declaration are added
   to the global block. Initialized globals continue to be placed
   in the block defined with the last HAL_GLOBALS or
   HAL_GLOBALS_INIT.*/
```

The parameters of the global block definition include the identity of the physical memory bank that contains the block, the virtual address where the block will start in memory, and whether the block grows up or down from the base address. The base address must be specified using the `_BASE` form of the macro in at least one `.C` file for each global block, but can be omitted in others. In each block, the last parameter is used to determine if data is allocated at (base address +0) and upwards or at (base address -1) and downwards. An upwards and downwards block can therefore have the same starting address without overlapping. The three selection directives choose the global block (or a pair of global blocks, for separate initialized and uninitialized data) into which newly-defined global variables will be placed. These routines allow the LLC to automatically group globals together into one or more large blocks of variables for allocation *en masse*. Globals defined in different `.C` files may share the same identifier in order to be allocated to the same global block by the linker. Figure 2 shows a simple example of allocating global variables in a pair of upward-growing global blocks, using these compiler directives to control the arrangement of data.

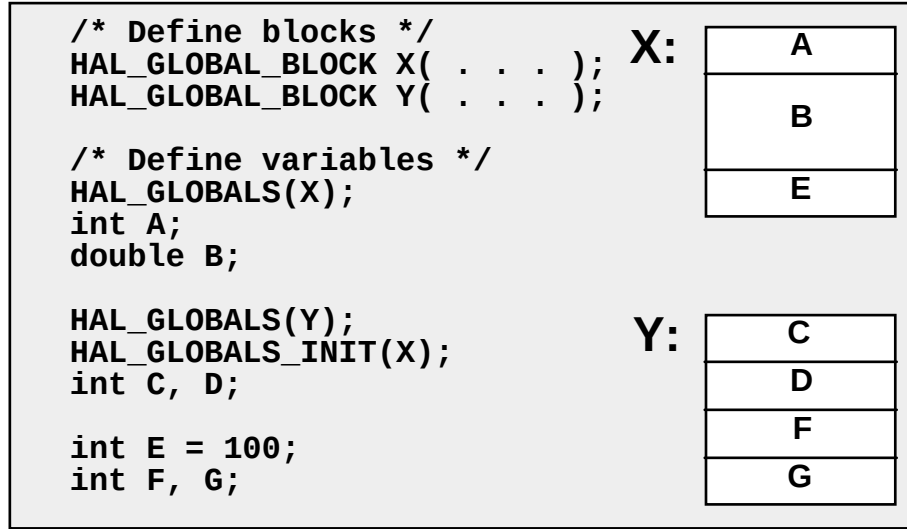


Figure 2: Example of allocation of globals within global blocks. The TVM-HAL code at left produces the memory allocation in blocks X and Y at the right.

In a traditional UNIX process, the linker assembles globals from a variety of .C files into two blocks, one for initialized data and one for uninitialized data. Details of the characteristics of the global blocks, including where they are placed in memory and whether they grow up or down, vary with the architecture and variety of UNIX used. All .C files are compiled with the equivalent of a single `HAL_GLOBALS_INIT` selecting the default initialized data segment and `HAL_GLOBALS_UNINIT` selecting the default uninitialized data segment at the head of the file. In contrast, the more complex memory structure of PCA architectures does not allow these blocks to be defined once. The TVM-HAL global directives may be used to allow the high-level compiler to scatter global variables anywhere they may be needed in any of the possible memory banks.

2.3 Pointer-Based Accesses

Pointer dereferencing, which is usually used to access the heap, is implemented as an explicit load/store mechanism that works with all of the basic C types (`int`, `char`, `double`, etc.). The following description assumes C. Type descriptors are needed as part of the function names to differentiate them properly. If the virtual machine code is based on C++, polymorphism could be used instead of the `_type` descriptors to enable use of the same function name for different data types.

```

type value = HAL_LOAD_type(const int memory, type *pointer)
type value = HAL_LOAD_P_type(const int memory, HAL_phyPtr_type pointer)
HAL_STORE_type(const int memory, type *pointer, type value)
HAL_STORE_P_type(const int memory, HAL_phyPtr_type pointer, type value)

```

It is assumed that these directives will normally compile into simple load and store instructions, respectively. The `memory` input is a physical memory hint that may force the compiler to generate a more complex sequence of instructions for accesses to some memories. Any arithmetic in the `pointer` input field will normally be compiled down to ALU instructions preceding the load or store, but can be computed in the memory instruction itself if an appropriate addressing mode is available for the target architecture. The `_P` versions of the

macros allow direct access to memory using physical addresses instead of virtual ones (see Sections 4.1 and 4.2, respectively, for a description of the difference) by low-level operating system code. This is generally only necessary at system startup time, before the virtual memory map has been established by the routines in Section 4.

The TVM-HAL defines several machine-independent forms of the C fundamental types that can be used with these operations or any other operation requiring a fixed, machine-independent version of a C type:

- HAL_bool 1-bit logical (usually char or int, depending on speed)
 → Enumerated as: HAL_FALSE = 0 and HAL_TRUE = 1
- HAL_boolX 2-bit ternary logical (usually char or int)
 → Enumerated as HAL_bool plus: HAL_DONT_CARE = 2
- HAL_int8 8-bit integer (usually char)
- HAL_int16 16-bit integer (usually short)
- HAL_int32 32-bit integer (usually int or long)
- HAL_int64 64-bit integer (usually long long)
- HAL_int128 128-bit integer (used for vector and similar types)
- HAL_uint8 8-bit unsigned integer (usually unsigned char)
- HAL_uint16 16-bit unsigned integer (usually unsigned short)
- HAL_uint32 32-bit unsigned integer (usually unsigned int or long)
- HAL_uint64 64-bit unsigned integer (usually unsigned long long)
- HAL_uint128 128-bit unsigned integer (usually unsigned vector type)
- HAL_float 32-bit IEEE floating-point (usually float)
- HAL_double 64-bit IEEE floating-point (usually double)
- HAL_longdouble 64-to-96-bit floating-point (as allowed by architecture)
- HAL_ptr Generic void* pointer (usually a 32-bit or 64-bit value)

For example, using these types a load of a char from memory bank #1 becomes HAL_LOAD_int8(1, my_char_ptr). Similarly, a load from a char** from memory bank #1 becomes HAL_LOAD_int8(1, HAL_LOAD_ptr(1, my_char_ptr_ptr)).

3 Exception Handling Model

The core of the TVM-HAL extensions is their exception handling model. TVM-HAL primitives are used extensively in the operating system code, which is always triggered through some form of exceptional condition: an external interrupt, a user instruction performing an illegal operation, or a user instruction that voluntarily raises an exception (a trap or system call on most architectures). As a result, the exception handling model is the gateway to virtually all code that uses other TVM-HAL extensions. This portion of the TVM-HAL interface consists primarily of a mechanism to shift the processor between privileged OS code and normal user code in an architecturally-neutral fashion, including some control routines to specify in advance the behavior of the processor following an exception. The exception handler also includes routines to encapsulate operations that are architecture-dependent, such as register saving during a context switch.

3.1 Exception Vector Table System

The core of the exception handling system is a table of current exception vectors maintained by each processor in a PCA system. This table is responsible for catching every exception that occurs on a PCA processor and vectoring the processor to the appropriate exception handler, composed of compiled TVM-HAL code. Different handlers are selected, dependent upon the source of the exception. This code should be no more than a few instructions long on most processor architectures, typically a table lookup and jump. A high level view of this process is illustrated in Figure 3.

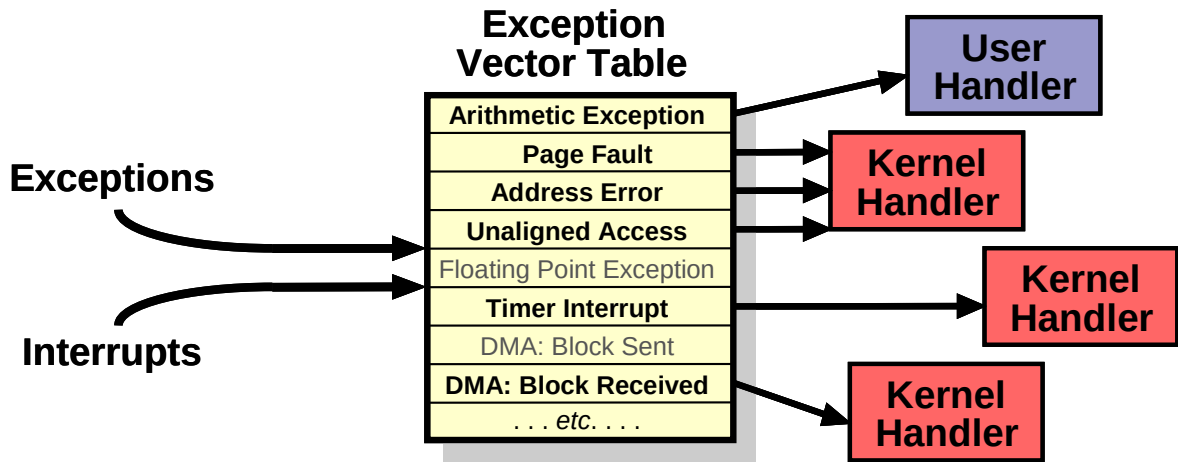


Figure 3: Overview of the TVM-HAL exception handling system. Exceptions or interrupts cause the processor to look up vectors in the table and pass control of execution to a handler. Non-enabled interrupts (such as the non-bold examples above) are handled by a default “unsupported” handler.

Instruction Exceptions		External Interrupts	
Number	Name	Number	Name
0	Unsupported Exception	64	Unsupported Interrupt
1	Illegal/Privileged Instruction	65	Processor Reset
2	Breakpoint instruction	66	Processor Soft Reset
3	Watchpoint break	67	Non-Maskable Interrupt
4	Integer Arithmetic Exception	68-95	— RESERVED —
5	FP Arithmetic Exception	96-125	Maskable I/O Interrupts*
6	FP Unavailable	126	Performance Count Interrupt*
7	FP Denormalized Number	127	Timer Interrupt*
8	Trap instruction	128-255 [Reserved for DMA use]	
9	System call instruction		
16	Page Fault: D-Read		
17	Bus Error: D-Read		
18	Protection Fault: D-Read		
19	Address Error: D-Read		
20	Cache Parity: D-Read		
21	Page Fault: D-Write		
22	Bus Error: D-Write		
23	Protection Fault: I		
24	Address Error: D-Write		
25	Cache Parity: D-Write		
26	Page Fault: I		
27	Bus Error: I		
28	Protection Fault: I		
29	Address Error: I		
30	Cache Parity: I		
10–15, 31–63	— RESERVED —		

Table 1: List of exceptions supported by the exception handler.¹ Interrupts marked with asterisks (96–127) are enabled or disabled by the `HAL_INTERRUPTS_ON` and `HAL_INTERRUPTS_OFF` calls.

¹ This selection of entries is suggested based upon a survey of common processor architectures. Any further common exceptions that are classified in the future should be assigned to some of the currently reserved exception numbers (10-15, or 31–63).

Table 1 gives the current list of exceptions.² The exception vector for each entry in this table consists of a *valid* flag that indicates whether or not it is active, the starting address for the appropriate exception handler, and the privilege level at which the target exception handler should be executed.

Maskable interrupts (#96–127) are prioritized so that lower numbers are assigned to higher-priority interrupts. External hardware such as I/O devices trigger these according to the priority assigned to each device by the system’s interrupt controller. Normally, high-speed devices (such as network interfaces or graphics cards) will be assigned to high priority vectors (lower vector numbers), while lower-speed devices (such as keyboards) are assigned to lower priority vectors (higher vector numbers).

When an internal exception or external interrupt occurs, the exception handler code added to one’s program by the low-level compiler consults the table. If the appropriate entry is enabled, the privilege level is set according to the preset flag and the processor jumps to the handler program counter (PC) stored in the table. Execution then continues within the handler. In the event that the entry is disabled, the “unsupported” handler is started. In general, this handler should kill user threads that trigger unhandled exceptions and ignore all unhandled external interrupts. However, particular application domains may require different implementations.

When an external interrupt is processed, all maskable (low-priority) interrupts are temporarily disabled, by default. Certain critical interrupts, for example `reset`, cannot be disabled. Similarly, instruction exceptions can never be disabled; therefore, interrupt handlers should avoid executing instructions that may cause exceptions when further interrupts cannot be processed. Once interrupts may be processed again, it is the responsibility of the exception handler to re-enable interrupts as soon as possible. Normally, this will occur as soon as key interrupt state has been saved. The TVM-HAL currently guarantees only minimal non-maskable/maskable priority resolution for performance and compatibility reasons. To avoid priority inversion between individual I/O interrupt handlers, it may be necessary to add priority enforcement rules to the code executed at the start of individual handlers.

Privileged OS code controls the contents of the exception table for each processor using calls to the following routines:

```
/* Exception Table Maintenance */
void HAL_EXCEPTION_ENABLE(int entry_number, HAL_codePtr handler_pc,
    int privilege_level);
void HAL_EXCEPTION_REENABLE(int entry_number);
void HAL_EXCEPTION_CLEAR(int entry_number);
HAL_bool HAL_EXCEPTION_USABLE(int entry_number);
HAL_bool HAL_EXCEPTION_GET_ENABLED(int entry_number);
HAL_codePtr HAL_EXCEPTION_GET_HANDLER(int entry_number);
int HAL_EXCEPTION_GET_PRIVILEGE(int entry_number);
```

² Readers not familiar with some of these exception types are referred to the processor architecture books listed in Section 7.

```
/* DMA Exception Control */  
void HAL_DMA_ENABLE(int DMA_controller_num);  
void HAL_DMA_DISABLE(int DMA_controller_num);  
int entry_number = HAL_DMA2VECTORNUM(int DMA_controller_num);  
int DMA_controller_num = HAL_VECTOR2DMANUM(int entry_number);
```

HAL_EXCEPTION_ENABLE enables an exception vector from Table 1 and sets the jump table entry for that vector so that when the exception or interrupt is triggered, the processor will be sent to the proper routine (indicated with an architecturally-specific **HAL_codePtr** pointer—a pointer to a code function using a virtual address) at the proper privilege level. This level should be an integer nominally enumerated to **HAL_PRIVILEGED** = 0 and **HAL_USER** = 1. **HAL_EXCEPTION_CLEAR** clears exceptions that need to be disabled, such as after a context switch to a thread that can only generate a limited set of exceptions, while **HAL_EXCEPTION_REENABLE** re-enables a previously cleared exception. An OS can use **HAL_EXCEPTION_USABLE** to determine whether particular exceptions or interrupts can be triggered on a particular architecture. Finally, the three **HAL_EXCEPTION_GET** routines may be used to examine the current contents of the vector table.

Most external devices trigger a corresponding maskable interrupt that can vector the processor to the appropriate handler. However, DMA controllers utilize a slightly different default way of handling exceptions, in order to allow the use of the **HAL_BLOCKMOVE** routines in Section 5.1. All DMA controllers in a PCA system are assigned a unique **DMA_controller_num** that can be used with the **HAL_DMA** routines above. Initially, all DMA controllers trigger a normal, maskable exception (#96-127) that is set by the standard **HAL_EXCEPTION_ENABLE** routine, like any other interrupt-causing I/O devices. To allow this, the maskable interrupt entry number for a particular DMA controller can be recovered using **HAL_DMA2VECTORNUM** (or vice-versa using **HAL_VECTOR2DMANUM**). The standard maskable interrupt is an effective means for handling DMA interrupts if only one exception routine can be evoked by a particular DMA controller. Consequently, this may be the only DMA interrupt handling technique used by simple operating systems. However, DMA controllers can also be selectively switched to use a special default DMA handler built into each TVM-HAL implementation that can properly parse the vector numbers from an active **BLOCKMOVE**. This functionality is activated by calling **HAL_DMA_ENABLE** with that DMA controller number. The default handler, once enabled and invoked by an interrupt, reads the 7-bit vector number from the outgoing or incoming data block and invokes the appropriate DMA exception (#128-255) enumerated in the **BLOCKMOVE** command for the appropriate processor. This two step exception mechanism (default handler leading to the final DMA exception handler) allows a single DMA controller, in an architecturally neutral way, to launch a wide variety of exception handlers based on the type of data being sent or received. When active moves to the current processor are no longer required from a particular DMA controller, then the **HAL_DMA_DISABLE** routine can be used to deactivate the default handler, returning that controller's DMA processing to either a conventional maskable interrupt handler or deactivating it altogether.

Normal usage of this handler control code is fairly straightforward. Several exception handlers will be set up using **HAL_EXCEPTION_ENABLE** and **HAL_DMA_ENABLE** when the OS boots and then left ready-to-execute so long as that processor is running. Other handlers will dynamically change on a regular basis, modified by calls to system routines such as the UNIX

`signal` calls, or as different processes are switched in and out. The Processor Reset vector, however, requires special treatment. It is normally predefined to a fixed address by hardware that is located in the system's bootstrap ROM. As such, it can never be modified or controlled and is included on the list of exceptions only for reference.

Finally, the two instruction exceptions that can be triggered explicitly by code are triggered by the following two macros:

```
/* Trigger an exception */
void HAL_SYSCALL(int syscall_number);
void HAL_TRAP(int syscall_number, HAL_bool trap_expression);
int syscall_number = HAL_SYSCALL_NUMBER();
```

Both of the first two macros trigger the appropriate exception (#9 or #8, respectively) when executed, but the `HAL_TRAP` macro does so conditionally, only if the `trap_expression` evaluates to `TRUE`. On many architectures, part or all of this expression can be calculated by the trap instruction itself. The exception handler receiving the exception can then recover the system call number using the `HAL_SYSCALL_NUMBER` macro immediately upon starting.

3.2 Special Code in Exception Handlers

To make it possible to write the exception handlers directly in portable TVM-HAL code instead of architecture-specific assembly code, a small number of extensions to C are required. This section describes these extensions.

3.2.1 Linking of Low-Level Run-time System

Although syntactically similar to C functions, most of the TVM-HAL routines are more like macros, and many can in fact be implemented using macros. The use of all capital letters in the naming scheme emphasizes their macro-like nature. The low-level compiler is responsible for converting each routine into a small number of instructions that are placed directly into the object code produced by the compiler. This approach is necessary because a stack and full register set may not be available to support normal function calling conventions at all times. However, some TVM-HAL routines may be too complex to be inlined on some PCA architectures, and must be emulated in software. In these cases, the compiler may emit code that makes calls to a very low-level and architecture-specific run-time system that is only used by the low-level compiler itself and is not visible to user-written TVM-HAL code. This architecture-specific run-time code may also contain code that is not directly called by the user, such as the exception mechanism, default DMA exception handlers, and other key machine-specific operations necessary to support the TVM-HAL model.

For the most part, the presence of the low-level run-time library can be completely ignored by programmers coding directly in TVM-HAL code or by high-level compilers generating this code. However, low-level compilers will require a compiler option to specify when to link the low-level run-time library to the resulting object code, similar to the `-libm` option for current C compilers, as only one copy of the low-level run-time library should be linked in with any particular OS implementation.

3.2.2 *Compiler Register Control*

A key modification required to the TVM compiler (C compiler with TVM-HAL extensions), compared with standard C, is the addition of flags to indicate when the registers contain unsaved user state, making standard register allocation impossible. When these unsafe conditions exist, the TVM compiler must emit code using only a very limited register set (for example, R26-27 on MIPS-ISA processors [17]) or special shadow registers (such as the SPRG registers in PowerPC [15]). The following code illustrates the use of three TVM-HAL compiler directives to address this condition:

```
HAL_EXCEPTION_HDL void ExceptionHandler(void)
{
    << Limited register use, protecting existing state >>
    HAL_OVERWRITE_ON:
        << Normal register use >>
    HAL_OVERWRITE_OFF:
        << Limited register use, can restore state >>
}
```

The `HAL_EXCEPTION_HDL` keyword notifies the compiler that this routine is an exception handler, which can only take and return `void` inputs and outputs because it is not called through a stack frame. As a result, it cannot use local variables of any kind, and must set up a stack using the routines in Section 3.2.4 before it can call other C functions. On routines starting with this keyword, the compiler will also automatically start up in “unsafe registers” mode, to avoid overwriting existing state, and will suppress the generation of routine stack frame code. Within the routine itself, the `HAL_OVERWRITE_ON` and `HAL_OVERWRITE_OFF` directives notify the compiler when it is safe to use the full register set and when it should be constrained again, before the original state is restored. In general, these routines should be kept as close to the top and bottom of the exception handler as possible in order to allow the compiler to use normal register allocation practices for most of the generated code.

3.2.3 *Register Save/Restore for Context Switches*

Routines to save and restore the processor state are complementary to the register control directives above. Since the configuration of the registers is highly variable from one architecture to the next, the TVM-HAL defines a pair of optimized macros to implement this capability, and a macro to prepare memory to hold the processor state:

```
void HAL_SAVE_STATE(const int memory, HAL_ArchitectureSaveBlock *saveTo);
void HAL_RESTORE_STATE(const int memory,
    HAL_ArchitectureSaveBlock *restoreFrom);
void HAL_INIT_STATE(const int memory, HAL_ArchitectureSaveBlock *toInit,
    void *initial_StackPointer, void *initial_GlobalPointer,
    void *initial_FramePointer, HAL_codePtr initial_ProgramCounter);
```

These macros alternately push the current register state out to a block in memory, return it back into the processor, or initialize a newly created `HAL_ArchitectureSaveBlock` record. This record type is an architecture-specific `struct` that is designed to hold the entire state of the processor for a thread. An example `struct` for a simple 64-bit RISC processor is as follows:

```

typedef struct
{
    HAL_int64 integerRegisters[32];
    double fpRegisters[32];
    HAL_int32 processorStatusReg;
    HAL_int32 fpStatusReg;
}
HAL_ArchitectureSaveBlock;

```

Normally, the OS code need only allocate these structures, initialize the major pointers and other information as required with `HAL_INIT_STATE`, and then use the `HAL_SAVE_STATE` and `HAL_RESTORE_STATE` functions in its exception handlers that switch contexts. Thread managers do not normally change the internal register state of the threads they are managing, so therefore it is not necessary for TVM-HAL code to have any knowledge of the internal structure of the `struct`.

The following code is representative of a typical timer interrupt exception handler:

```

HAL_EXCEPTION_HDL void Scheduler(void)
{
    HAL_SAVE_STATE( kMemNum, currentStateBlockPtr );
    HAL_OVERWRITE_ON:
        << Code to select the next thread and put its block pointer
            in the global "currentStateBlockPtr" variable >>
    HAL_OVERWRITE_OFF:
    HAL_RESTORE_STATE( kMemNum, currentStateBlockPtr );
    HAL_EX_RETURN(); /* Return from exception to saved PC */
}

```

In this code, the scheduler code immediately saves the current processor state in the `HAL_ArchitectureSaveBlock` for the currently active thread, which was previously stored in the global `currentStateBlockPtr`, and then frees up the register allocation. It consults its current task queues to determine the next thread to execute, and then sets the `currentStateBlockPtr` with the new thread's state block. Finally, it closes out by restoring the state of the new thread to the processor registers and restarting the thread at its saved program counter upon returning from the exception (see Section 3.2.5 for a description).

3.2.4 Compiler Stack and Global Pointer Control

In order to control the stack and globals used by TVM-HAL code, a few routines are defined specifically to set up the three pointers defined for each thread context (the global pointer (GP), frame pointer (FP), and stack pointer (SP)). The program counter is controlled using different macros, if necessary. These are:

```

/* Macros to get/set pointer values */
void *HAL_GET_SP();
void *HAL_GET_FP();
void *HAL_GET_GP();
/* Normally needed in exception handlers only */
void HAL_SET_SP(void *new_stack_pointer);
void HAL_SET_GP(void *new_global_pointer);
void HAL_SET_FP(void *new_frame_pointer);

```

```
/* Compiler directives: When can SP/FP or GP pointers be used? */
/* Normally needed in exception handlers only */
HAL_STACK_ON:
HAL_STACK_OFF:
/* May be safely used for all C functions to dynamically turn GP on */
HAL_GP_ON:
HAL_GP_OFF:
/* Use to tell compiler where GP is set */
HAL_USE_GP1(identifier_of_global_block, const int offset_into_block);
HAL_USE_GP2(identifier_of_upward_global_block,
            identifier_of_downward_global_block,
            const int offset_into_upward_block);
```

These routines should be used within each exception handler to prepare the system to use the stack and global pointers. The SP and FP are set by giving the system one or more pointers and only then directing it to start using the stack. Prior to the use of `HAL_STACK_ON`, no local variables allocated on the stack may be used. Similarly, no normal C functions may be called from an exception handler until the stack and frame pointers have been set and the stack enabled. However, once a stack has been established by an exception handler, the management of the SP and FP are normally left to the compiler, so these functions are no longer needed.

Global variables, unlike local ones, may be used at *any* time, even if the GP is not set in advance. Without a GP, the address for each global reference is constructed by instructions preceding the memory access. Individual global variables are located at absolute virtual addresses that can be calculated by the linker, given knowledge of the global block in which the variable lies and decisions about how variables will fill that block that are made at compile and link time. As a result, the compiler can insert code to calculate this address at the location of every global access outside of the `HAL_GP_ON` and `HAL_GP_OFF` directives. Between these directives, the compiler attempts to use the global pointer as a base address for calculating the location of globals. This approach may result in fewer machine instructions generated per reference, but is not essential for correct execution.

Correct use of a global pointer can be complex. Instead of an arbitrary address, the GP always points at a predefined global block, or some offset into that block. This offset can be used to move the GP near commonly used variables in a very large global block. Global addresses from that block can then be accessed using the GP. Alternatively, a matched pair of blocks—one going upward and one downward from the same starting address—can be selected as the basis for the GP. The offset is added to the starting address, so a positive offset will be in the upward block while a negative offset will push the GP into the downward block. A common use for this capability is to have uninitialized data in one block and initialized data in the other. A program may also need to access global variables in global blocks other than those pointed to by the GP. In these cases, the low-level compiler will determine that it cannot use the GP and will instead construct the global variable addresses from scratch, as if the GP had not been set. As a result, any variables in global blocks can always be accessed within any function, even if the GP is pointing elsewhere. In practice this should seldom occur, since programs will generally keep most globals in use at any one time together in one location. For accesses to global blocks that are used only infrequently by the program, it may be preferable to use these direct accesses instead of first readjusting the GP.

The GP can be set in two different ways using the TVM-HAL macros. The first method sets the GP dynamically, on a function-by-function basis. In this case, the high-level compiler emits a `HAL_SET_GP/HAL_USE_GP1` or `HAL_SET_GP/HAL_USE_GP2` macro/directive invocation combination and uses `HAL_GP_ON` at the beginning of every C function. This allows a customized GP for every function, and is effective if there are a large number of global variables that cannot all be located close to a single global pointer address. Alternatively, the compiler can put the `HAL_SET_GP` calls only in the OS or thread startup code. In this case, the low-level compiler must know where the GP is pointing while it compiles other functions in the file. To enable this, `HAL_USE_GP1` or `HAL_USE_GP2` compiler directives should be placed at the head of each `.c` file, or before each function if the GP may occasionally be different with some functions. For each function following the directive in the file, these directives tell the low-level compiler what SET call to assume has been invoked prior to the function entry. This approach fixes the GP for many functions at once, and thereby avoids generating GP movement code if it rarely changes.

The following code expands upon the previous exception handler example with SP/FP/GP code:

```
HAL_EXCEPTION_HDL void Scheduler(void)
{
    /* Stack and GP disabled at start of EXCEPTION_HDL */
    /* NO locals allowed in the exception handler */

    /* SAVE_STATE, including SP/FP/GP/PC */
    HAL_SAVE_STATE( kMemNum, currentStateBlockPtr );
    HAL_OVERWRITE_ON:

    /* Set GP to kernel space */
    HAL_SET_GP(kernelUninitBlockPtr, 0);
    HAL_USE_GP2(kernelUninitBlock, kernelInitBlock, 0);
    HAL_GP_ON:

    /* Set up OS-only stack */
    HAL_SET_SP(initialSchedulerSP);
    HAL_SET_FP(initialSchedulerFP);
    HAL_STACK_ON:

    << Code to select the next thread and put its block pointer
        in the global "currentStateBlockPtr" variable >>

    HAL_STACK_OFF:
    HAL_GP_OFF:
    HAL_OVERWRITE_OFF:

    /* RESTORE_STATE, including SP/FP/GP/PC */
    HAL_RESTORE_STATE( kMemNum, currentStateBlockPtr );
    HAL_EX_RETURN();
}
```

These additions to the scheduler example load in initial stack/frame pointers and the global pointer from globals to prepare the system for full C operation in the central part of the routine, where most of the code and calls to normal C functions (using the new stack) can now occur.

The compilation requirements for this model require that no locals be used within the exception handler routine (since there is no stack on which to allocate them at routine entry) and that no calls to other C routines be made outside of the `HAL_STACK_ON/HAL_STACK_OFF` pair. However, three global accesses (italicized in the example above) occur outside of the `HAL_GP_ON/HAL_GP_OFF` pair. These three accesses use direct calculation of the global address, while all others will use the GP loaded in by the `HAL_SET_GP` macro and enabled by the `HAL_USE_GP2` directive.

Large and complex thread management systems may also need to examine or modify the state of the pointers for one or more thread states that have been saved in memory. To facilitate these modifications, the following macros allow direct modifications of pointers within saved thread states:

```
/* Versions of the previous routines to work with saved thread states */
void *HAL_ASB_GET_SP(const int memory, HAL_ArchitectureSaveBlock *block);
void *HAL_ASB_GET_FP(const int memory, HAL_ArchitectureSaveBlock *block);
void *HAL_ASB_GET_GP(const int memory, HAL_ArchitectureSaveBlock *block);
void HAL_ASB_SET_SP(const int memory, HAL_ArchitectureSaveBlock *block,
    void *new_stack_pointer);
void HAL_ASB_SET_FP(const int memory, HAL_ArchitectureSaveBlock *block,
    void *new_stack_pointer);
void HAL_ASB_SET_GP(const int memory, HAL_ArchitectureSaveBlock *block,
    void *new_stack_pointer);
```

Most systems will not need to use these, but they are provided for completeness.

3.2.5 System Control

A small number of low-level macros are needed to allow an OS written in TVM-HAL code to be able to control the exception environments present in typical microprocessors. The most critical macros control the current privilege level of the processor (regarding memory references and instruction execution), the state of memory address translation (if allowed on a per-processor basis instead of a per-memory basis), and the current interrupt masking level:

```
/* Control/monitor current interrupt status */
HAL_bool HAL_INTERRUPTS_ARE_ENABLED();
void HAL_INTERRUPTS_ON();
void HAL_INTERRUPTS_OFF();
int HAL_GET_PRIVILEGE_LEVEL();
void HAL_SET_PRIVILEGE_LEVEL(int new_privilege_level);

/* Turn instruction and data address translation on/off */
HAL_bool HAL_GET_INST_TRANSLATION();
void HAL_SET_INST_TRANSLATION(HAL_bool translate);
HAL_bool HAL_GET_DATA_TRANSLATION();
void HAL_SET_DATA_TRANSLATION(HAL_bool translate);

/* Get/set the masking level for external interrupts */
int HAL_MASK_POINT_AT();
void HAL_MASK_ABOVE(int entry_number);
```

These macros arbitrarily get or set the current interrupt status, the privilege level of the processor for accessing protected memory or instructions (using the same privileges as exception handlers), the on/off status of address translation (by a translation lookaside buffer (TLB)) in the processor, or the external interrupt masking level (all maskable external interrupts from `entry_number` on up are disabled, or all are enabled if `entry_number` is 128 or higher). Note that interrupts are always off upon entering an exception or interrupt handler, and must be turned back on again explicitly by the handler, if this is desired. In processors with controllable TLBs, these are generally also disabled upon entry into exception handlers, and should be turned on again before accessing virtual memory. Instruction exceptions are not necessarily enabled or disabled by these routines, thus it is the responsibility of the exception handler author to avoid instructions that may cause exceptions during critical regions of code when interrupts are disabled. For example, accesses to paged memory (only possible if address translation is active) may cause page faults and should be avoided during code with interrupts disabled.

The following routines handle common interrupt-control operations in exception handlers:

```
/* Return from exception, to last saved position or a new one */
void HAL_EX_RETURN();
void HAL_EX_RETURNX(HAL_codePtr PC, HAL_bool interrupts_enabled,
    int return_privilege_level, HAL_bool inst_translation_on,
    HAL_bool data_translation_on);

/* Recover saved-at-last exception location/status */
HAL_codePtr HAL_WAS_PC();
HAL_bool HAL_INTERRUPTS_WERE_ENABLED();
int HAL_WAS_PRIVILEGE_LEVEL();
HAL_bool HAL_WAS_INST_TRANSLATION_ON();
HAL_bool HAL_WAS_DATA_TRANSLATION_ON();

/* Modify saved-at-last exception location/status */
void HAL_SET_EX_PC(HAL_codePtr new_PC);
void HAL_SET_EX_INTERRUPTS_ON();
void HAL_SET_EX_INTERRUPTS_OFF();
void HAL_SET_EX_PRIVILEGE_LEVEL(int new_level);
void HAL_SET_EX_INST_TRANSLATION(HAL_bool inst_translation_on);
void HAL_SET_EX_DATA_TRANSLATION(HAL_bool data_translation_on);
```

The `HAL_EX_RETURN` macro returns to the location of the previous exception or interrupt, with the processor control state (PC, interrupts enabled/disabled, privilege level, and TLB state) as it was before the exception or interrupt. This routine can be used for simple exception handlers that restart threads without modifying their state. The `HAL_EX_RETURNX` macro or the various save/restore macros can be used for complex handlers that must modify the key machine state of the system. Like exception handler addresses, the return PC is set using an architecturally-specific `HAL_codePtr` pointer—a pointer to code using a virtual address.

The “recover saved status” routines recover the privilege status information that has been saved by the hardware after the last exception or interrupt. This is the same information used by the simple `EX_RETURN` macro when it returns. These recovery routines are provided so that a thread scheduler may read this state before modifying it, for example. As with the routines to modify stack and global pointers, versions of these routines are also provided to read and modify the state of threads after they have been saved to memory in case the thread manager needs to

make extensive changes to many thread states at once. Unlike the interrupt status and privilege level, the masking status cannot be dynamically changed by hardware, and therefore there is no “last mask level” routine.

Much like with the pointer-control macros, there are also versions of the core state control macros which read or write saved thread state in memory. These work in a manner analogous to their “normal” versions, but work with thread state stored in memory instead of within the processor itself:

```
/* Recover saved-at-last exception location/status from memory */
HAL_codePtr HAL_ASB_WAS_PC(const int memory,
    HAL_ArchitectureSaveBlock *block);
HAL_bool HAL_ASB_INTERRUPTS_WERE_ENABLED(const int memory,
    HAL_ArchitectureSaveBlock *block);
int HAL_ASB_WAS_PRIVILEGE_LEVEL(const int memory,
    HAL_ArchitectureSaveBlock *block);
HAL_bool HAL_ASB_WAS_INST_TRANSLATION_ON(const int memory,
    HAL_ArchitectureSaveBlock *block);
HAL_bool HAL_ASB_WAS_DATA_TRANSLATION_ON(const int memory,
    HAL_ArchitectureSaveBlock *block);

/* Modify saved-at-last exception location/status from memory */
void HAL_ASB_SET_EX_PC(const int memory,
    HAL_ArchitectureSaveBlock *block, HAL_codePtr new_PC);
void HAL_ASB_SET_EX_INTERRUPTS_ON(const int memory,
    HAL_ArchitectureSaveBlock *block);
void HAL_ASB_SET_EX_INTERRUPTS_OFF(const int memory,
    HAL_ArchitectureSaveBlock *block);
void HAL_ASB_SET_EX_PRIVILEGE_LEVEL(const int memory,
    HAL_ArchitectureSaveBlock *block, int new_level);
void HAL_ASB_SET_EX_INST_TRANSLATION(const int memory,
    HAL_ArchitectureSaveBlock *block, HAL_bool inst_translation_on);
void HAL_ASB_SET_EX_DATA_TRANSLATION(const int memory,
    HAL_ArchitectureSaveBlock *block, HAL_bool data_translation_on);
```

The last group of macros/routines is used to control the timer interrupt for a processor:

```
void HAL_TIMER_OFF();
void HAL_TIMER_ON();
HAL_bool HAL_TEST_TIMER_SWITCH();
void HAL_TIMER_FREQUENCY_SET(HAL_uint64 frequency_of_interrupts_in_hertz);
HAL_uint64 HAL_TIMER_FREQUENCY_GET();
HAL_bool HAL_TEST_TIMER_SET();
```

These routines can be used to enable or disable the timer interrupts when multitasking is being enabled or disabled. The timer frequency should be set before timer interrupts are enabled, or else the PCA system will default to a 60 Hz timer interrupt. When the frequency is set, the actual clock frequency will be set to the closest hardware-supported frequency equal to or greater than the requested one. These routines normally assume that all processors in a PCA system have their own timers and individual control over which processors are interrupted by their timers. However, some systems may have several processors that either share a common timer or cannot individually turn off timer interrupts. In these systems, these calls may either affect several processors at once (if executed on a master processor that can control the timer handling)

or be ignored (if executed on a slave processor that cannot). The utility of these calls for a particular architecture can be tested with the two `TEST` macros.

3.3 PCA Processor Control

Several additional TVM-HAL features are necessary to control how code is compiled for processors, and then to establish how processors are set up and configured as a PCA system is booted.

3.3.1 Compiler → Processor Control

A PCA system architecture is comprised of many nodes that may not always be identical. Furthermore, processors that do have identical ISAs may not be connected to the same memory system. Unless the system is completely homogeneous, the PCA compiler must know which node(s) to use as its current compiler target. To achieve this, the TVM-HAL adds a compiler directive at the beginning of every `.C` file to specify the target processor(s) that should be used. This directive may also be placed (optionally) between functions to redirect compilation:

```
HAL_PROCESSOR(const int num_processor_targets,
               int processor_targets[]);
```

The processor identifiers provided in these directives are specified in architecture metadata and can be used by the low-level compiler to either verify that all nodes selected for compilation are able to use the same object code, or to determine that multiple object versions of the same source code must be generated. In the latter case, the linker will link separate application binaries for each of the different types of processor cores. Note that these processor object numbers must be encoded in the source code as constants so that the compiler can resolve them at compile time.

3.3.2 Cold-Boot Reset of TVM Nodes

Each TVM-HAL node must support a special, fixed *reset* vector that is used to start up the node from scratch. To a certain extent, this vector acts like the `main` function in a normal C program, but in this case it is for an entire OS instead of a single user program. This reset vector therefore corresponds to the cold-boot entry point into an assembly-language portion of the basic input-output system (BIOS) ROM present on typical computers. The TVM-HAL makes it possible to write this low-level code in C, using code similar to the following example:

```
HAL_RESET_HDL void ColdBoot(void)
{
    /* Stack and GP disabled at start of RESET_HDL */
    /* NO locals are allowed */
    /* Enable overwrite immediately, no register contents */
    HAL_OVERWRITE_ON:

    << Must set up kernel virtual memory mapping first >>
    << All input values should be constants >>
    << Use only physical-memory load/stores >>
    << Use one or more HAL_SET_SEGMENT macros, § 4.2 >>
    << Activate caches with HAL_xCACHE_ENABLE, § 4.2 >>
```



```

/* Set GP to kernel space */
HAL_SET_GP(kernelUninitBlockPtr, 0);
HAL_USE_GP2(kernelUninitBlock, kernelInitBlock, 0);
HAL_GP_ON:

/* Set up OS-only stack */
HAL_SET_SP(initialSP);
HAL_SET_FP(initialFP);
HAL_STACK_ON:

    << Set up rest of memory banks >>
    << Set up the exception vectors/handlers >>
    << Set up timer interrupt >>

HAL_INTERRUPTS_ON();

/* Finish up by starting context switching */
OS_ContextSwitcher();
}

```

The low-level compiler will compile this routine assuming that it will need to start at the fixed Reset vector for the processor, and will therefore be non-relocatable, and in the fixed memory area reserved for use by the core HAL code (with which this code will be directly linked) for load/store optimization purposes. These routines are similar to the standard HAL_EXCEPTION_HDL routines, with the addition of the keyword HAL_RESET_HDL at the beginning. In addition, there generally is no register state to save after a reset, so the HAL_OVERWRITE_ON: flag can generally be used immediately. With this code in place, a typical TVM-HAL node startup process is shown in Figure 4. This process allows for a small amount of architecture-specific ROM code before continuing with the TVM-based reset handler.

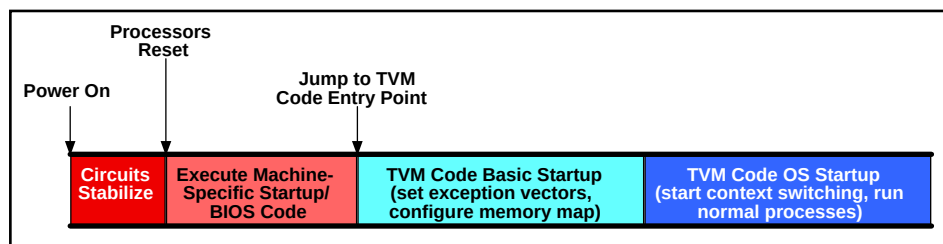


Figure 4: Sequence of events at processor node startup.

3.3.3 Morphing

At the processor level, *morphing* corresponds to simply converting the execution model between threaded and streaming modes.³ Morphing also involves major adjustments to the processor's

³ Other full-processor morphs similar to the streaming mode switch are also possible by adding other varieties of HAL_SWITCH to the HAL.

view of memory, but these are implemented using the standard memory management routines presented in the next section.

The sequence of operations in a morph is similar to a machine reboot. However, so long as the different morph configurations that can be hosted on a processor do not interfere with and modify the memory spaces of other configurations, morphing can be a very fast process. A processor not running TVM-HAL code must first be interrupted externally (probably using active DMA routines from Section 5.1) to switch back to TVM processor mode. TVM mode is the only one capable of running an exception handler, so this is always the “baseline” mode returned to after an interrupt. Once a processor is running TVM code, it can reconfigure as required, using code similar to that in the sample reset handler of Section 3.3.2. Typically, exception vectors will be initialized and memory will be reconfigured to set up the machine for a new run-time or OS environment, and the chosen environment will then be started up. If that environment is a TVM-based threaded processor, the startup morph code jumps to the entry point for the run-time system or OS after configuring the underlying hardware. Otherwise, it must call a TVM-HAL routine to convert itself to a streaming (or other valid type) node. The following routines test to determine whether a switch to a streaming node is possible, and implement the switch:

```
HAL_bool HAL_TEST_FOR_SVM();  
void HAL_SWITCH_TO_SVM(int master_processor);
```

The `HAL_SWITCH_TO_SVM` routine causes the threaded processor to shut down, becoming a slave streaming coprocessor responding to commands from the indicated master processor. Because no further adjustments to the memory system or other OS-like functions are possible after this switch, all memory directly controlled by the streaming node must be set up correctly prior to the switch. The streaming processor continues operating as a slave until it receives an external interrupt, usually through a special DMA message from a threaded processor. Upon receipt of the interrupt, it reverts to single-threaded TVM mode to handle the interrupt and potentially morph to a new state.

3.4 Multithreading Within Cores

Many processors are incorporating the ability for one processor to execute instructions simultaneously from several contexts. This functionality can be implemented at various degrees of granularity, ranging from draining the pipeline on every context switch, through switching on a cycle-by-cycle basis, to simultaneous multithreading of instructions from several threads at once on a wide-issue out-of-order processor engine. Such multithreaded processors are similar to symmetric multiprocessors having a number of processors equal to the number of threads supported by an individual processor. However, the threads in a multithreaded processor interact more than the threads in a typical symmetric multiprocessor due to their close sharing of critical resources such as execution units and local memories. For example, a processor in an SMP running an idle loop will have no impact on the performance of a real thread running on another processor, but the same idle loop running on a multithreaded processor may steal useful CPU cycles away from real tasks. As a result, low-level parts of the OS or run-time library must usually exert more control over multithreaded architectures than they do with typical SMP nodes. The routines in this section add the necessary functionality to the TVM-HAL to allow a task scheduling kernel to properly deal with simple, SMP-like multiple-context processors.

The first group of routines add functionality to allow an OS to enable contexts for instruction scheduling, or disable them if there are too few threads available, thus avoiding the “running idle loop” problem discussed above:

```
/* Get number of contexts available */
int HAL_NUM_CONTEXTS();

/* Start up contexts */
void HAL_INIT_CONTEXT(int context_number, HAL_codePtr initial_PC,
    void *initial_StackPointer, void *initial_GlobalPointer,
    void *initial_FramePointer, HAL_bool interrupts_enabled,
    int privilege_level, HAL_bool inst_translation_on,
    HAL_bool data_translation_on);
void HAL_ACTIVATE(int context_number);

/* Shut down contexts */
void HAL_DEACTIVATE_OTHER(int context_number);
void HAL_DEACTIVATE();    // Deactivate yourself, for I/O blocking
```

A run-time system should use the HAL_NUM_CONTEXTS call early during execution in order to properly configure the context-switching mechanism. HAL_INIT_CONTEXT sets up inactive contexts with key values such as a PC before the new contexts are first activated for scheduling using the HAL_ACTIVATE call. Threads may later be deactivated with either of the latter two calls. The simpler HAL_DEACTIVATE call is used primarily to allow threads to self-block while waiting for I/O. The thread that handles the I/O interrupt can then call HAL_ACTIVATE again to re-enable the stalled thread.

The second group of routines is used to control the scheduling of threads on the processor core:

```
/* Switch to contexts */
void HAL_SWITCH_TO(int context_number);
void HAL_SWITCH();    // Let HAL/hardware choose next thread

/* Scheduler control */
HAL_bool HAL_THREAD_ACTIVE(int context_number);
int HAL_THREAD_PRIORITY_GET(int context_number);
void HAL_THREAD_PRIORITY_SET(int context_number, int priority);
```

The HAL_SWITCH routines are used when threads want to switch out of context voluntarily. This can have great impact on a coarse-grained multithreaded processor, effectively acting as a fast context switch. In contrast, switching is essentially a NOP operation on simultaneously multithreaded processors, since they always execute instructions from all threads. Fine-grained multithreaded processors should instead control the scheduling between different threads using the HAL_THREAD_PRIORITY functions. These set the relative priority used by hardware to schedule instructions from different threads on a cycle-by-cycle basis, if the architecture has a mechanism to flexibly control priority. The exact interpretation of the priority value is hardware-dependent, but for compatibility the macros should be called with values of 0 (lowest priority) to 255 (highest priority). This TVM-HAL priority is then translated by the macro implementation into the appropriate hardware-specific priorities. Finally, the HAL_THREAD_ACTIVE macro can be used to determine which threads are active at a given time.

The final set of TVM-HAL routines for multithreading control how threads generate and interpret external interrupts. These routines set the context(s) that receive interrupts or try to send an interrupt to another context running on the same processor (if allowed). These may not be supported on all architectures. Some multithreaded architectures may be hardwired to always send interrupts to one of the contexts, while other architectures may not be able to redirect timer interrupts and other interrupts separately. Several **TEST** versions of the macros are provided to allow an architecture-neutral OS to determine whether directing or sending interrupts is possible, and whether or not timer and external interrupts may be controlled separately.

```
/* Interrupt Control */
void HAL_INTERRUPTS_TO(int context_number);
HAL_bool HAL_TEST_INTERRUPTS_TO(int context_number);
void HAL_TIMER_TO(int context_number);
HAL_bool HAL_TEST_TIMER_TO(int context_number);
HAL_bool HAL_TEST_TIMER_SEPARATE();
void HAL_SEND_INTERRUPT_TO(int context_number, int interrupt_vector);
HAL_bool HAL_TEST_SEND_INTERRUPT_TO(int context_number);
```

Because these routines are not guaranteed to be supported on all architectures, OS or run-time code that attempts to use them must always check the system metadata or use a **TEST** macro to ensure that they are valid before calling them.

4 Memory Control Model

PCA architectures are composed of many processors and memories. While higher-level metadata processing functions can view these as a complex graph of interconnected nodes, routines at the TVM-HAL level are mostly concerned with a simpler view of the system: a segmented address space, controlled using the memory control model described in this section. This model defines how OS or run-time code views the complicated memory structure in a PCA architecture by assigning all usable blocks of memory in the architecture to *memory segments*. Once the *view* that software has of memory is defined by this segmentation, the software can selectively control memory management features provided by architectures for memory protection, paging, and other functions on a segment-by-segment basis.

Pointers in the remainder of this section are typed in several ways. Special pointers used by low-level code that point directly at physical memory locations are marked by variations on the `HAL_phyPtr` type. Some of these are extended to indicate a physical address pointer that points only at a certain type of value, such as a `HAL_codePhyPtr` (instruction code only). Conventional C pointers always refer to virtual addresses that have been mapped from the physical address space to a user-visible virtual address space. In a few cases, the virtual nature of pointers is emphasized by marking them as `HAL_ptr`s instead of `void *` pointers. This occurs only for memory mapping calls where both physical and virtual addresses are used, so that it is critical to clearly differentiate between the two address spaces. Much like standard `void *` pointers, the internal definition of each pointer type is implementation-dependent. In most cases the pointer type will be `HAL_uint32` or `HAL_uint64`. All pointers can always be loaded as `HAL_ptr` types.

4.1 Physical Memory View

The memory model defines two levels of address encoding visible to software. The lower of the two levels is the physical memory address space. This level, defined by the PCA machine model of an architecture, presents a view of the memory within a PCA architecture to run-time or OS code. The actual memory banks in a PCA architecture are viewed as small parts of a single, large physical address space. Small banks of SRAM near processors are typically located in one part of the address space, while large banks of off-chip DRAM are grouped elsewhere. While each memory array within the architecture is generally located at a unique address range within the physical space, memories that can never be accessed from the same processor may be aliased to the same addresses. As a result, the machine model also usually defines a debug memory space that does not permit aliasing of any kind. This address space is typically used only for hardware debugging and test purposes, since it is never visible to any software running on the system.

The physical memory level is primarily used only by OS or run-time code, which maps the physical address space to virtual addresses that can be used by programs. However, many TVM-HAL and SVM routines have constant memory hints included that are added by the high-level compiler to supplement any pointers in the macro definition or function call. The most common of these is the `HAL_LOAD_type(constant_physical_bank_number_hint, pointer)` macro defined in Section 2.3. These constant hints indicate to the low-level

compiler which bank of physical memory will be responsible for holding a data structure indicated using a pointer. These hints allow the low-level compiler to generate code optimized for the unique characteristics of the specified memory bank, even if the physical bank number cannot be determined using the virtual address in the pointer field, for example if it is a variable. With the exception of these hints, the physical memory bank structure is generally not visible above the level of the segmentation management routines, detailed in the next subsection.

4.1.1 Physical Memory Bank Metadata

Each physical memory bank has a number of attributes and parameters, defined using metadata, that control how it can be used by the segmentation management routines.⁴ The metadata about each memory bank that is HAL-accessible can be described using several groups of related values. The first is just basic statistics about the memory block, including its “name” and size:

- **Component Identifier** (int): Each bank of memory in the system is assigned a unique identifier, or “object number,” for use throughout the HAL metadata routines.
- **Debugging Base Address** (HAL_phyPtr): This attribute is the base address of the memory bank in the “debug address space” used for debugging purposes (such as testing the on-chip memory to look for production errors). It will often be identical to the standard physical address. *This will generally be ignored by all software!*
- **Physical Base Address** (HAL_phyPtr): This attribute is the address of the beginning of the address range used to access the memory bank by processors running low-level, privileged code. It will usually be the same as the debugging address, except when an *equivalent-access memory bank group* is formed consisting of several banks of memory that are all accessed using the same physical address ranges, but on different processors. This will often be the case with small memories attached to identical processor nodes. In that case, the banks will be given different debug addresses to allow off-chip probes to see them separately at debug time, but identical physical addresses to allow the same code to run on any of the identical processor nodes and use the nearest local memories, without modification.
- **Size** (HAL_int64): The size of the memory bank, in bytes.

The second group of metadata values summarize which processors in the system can access the memory bank, and how:

- **Optimization Number** (int): Each bank of memory in the system is assigned an “optimization number” that is used in the constant bank hints associated with loads and stores (as mentioned previously) to aid the low-level compiler in optimization of accesses to that bank. Since different memory banks in an architecture may be accessed with considerably different sequences of instructions, or at least different numbers of load delay slots, such optimization is often crucial to ensure good performance. Unlike

⁴ The caching metadata listed here provides a simple specification about what connections can be made with any particular memory bank. The information provided by these should be sufficient at run time, but it may also be helpful to augment it with further figures or to provide access to the full topological views of memory defined in Section 5 of the PCA Machine Model [4].

component numbers, memory banks that can be accessed equally well with identical instruction streams can share the same optimization numbers.

- **Accessible by count** (int): The number of processors that can access the associated memory bank directly.
- **Accessible by** (int []): Identifiers of processors that can access the associated memory.

The third group of metadata values describes the access capabilities of the memory bank and how it may be controlled by the OS to allow address translation and memory protection:

- **Memory capabilities** (int): This attribute indicates to the OS what functions can be assigned to this memory bank. This is a bit vector created by ORing together integers representing each of the possible memory capabilities: HAL_MC_R_BIT = normal RAM access, HAL_MC_F_BIT = FIFO access, HAL_MC_I_BIT = can be instruction cache, HAL_MC_D_BIT = can be data cache, and HAL_MC_U_BIT = can be unified cache. Common configurations are defined directly, including: HAL_MC_R = normal RAM memory only, HAL_MC_I = must be I-cache, HAL_MC_D = must be D-cache, and HAL_MC_RIDU = can be anything but a FIFO.
- **Shiftable** (HAL_bool): This attribute indicates that addresses to this bank are translated in hardware by adding a base value to the virtual address, so that the virtual addresses and the physical access address do not need to be identical. If this is HAL_FALSE, then this memory bank can only be assigned to segments so that the virtual addresses and the physical access addresses are identical.
- **Boundable** (HAL_bool): This attribute indicates that the addresses fed to the memory are checked with base-and-bound logic, allowing multiple different segments from this physical bank to be mapped to the processor's virtual address space separately. If this is HAL_FALSE, then this memory bank is always translated as a single, monolithic block, since there is no way for hardware to enforce an end to each of the individual segments.
- **Protectable** (int): Accesses to this bank are screened through a protection mechanism to prevent stray references. This is a bitvector that specifies the types of accesses that can be selectively denied from accessing the memory bank: HAL_USER_EXECUTABLE, HAL_USER_READABLE, HAL_USER_WRITABLE, HAL_PRIV_EXECUTABLE, HAL_PRIV_READABLE, or HAL_PRIV_WRITABLE. The protection modes are discussed further below. Accesses without the appropriate attribute bit set can only feed addresses straight into the memory without any checking. Blocks that are unprotectable from user accesses should not be trusted to maintain data across context switches in a multitasking environment.

The fourth group of metadata values describes how the bank can be used as a cache for other memory banks, if possible, and for which ones. Similarly, it also describes which memory banks in the system can act as a cache for this one.

- **Cacheable** (HAL_bool): This attribute indicates to the OS that this bank of memory can be cached in other, more local memories. In general, a memory bank will be either cache-capable (and local) or cacheable (and remote), but not both.
- **Cache coherent** (HAL_bool): This bank has hardware to enforce coherence with other banks, or is the highest-level memory shared among processors, and is therefore coherent by definition.

- **Cache below count** (int): The size of the array of processors and caches that this bank can be below.
- **Cache below** (int []): Object identifiers for the processor(s) or other memory bank(s) that can use this bank of memory as a cache. Processors listed here use the cache bank as a primary cache, while memory banks listed here can use the bank as an L(n+1) backing cache or RAM.
- **Cache above count** (int): The size of the array of memory banks that this bank can be above.
- **Cache above** (int []): Identifiers for the other memory bank(s) that can have their contents cached by this bank. These banks should either be straight RAM banks or L(n+1) backing caches.
- **Cache linesize count** (int): The number of valid line sizes for cache-capable banks.
- **Cache linesize** (int []): The valid line size(s) for cache-capable memory banks.
- **Cache gang count** (int): The number of other processors that can join with this one.
- **Cache ganging** (int []): Object identifiers for other memory banks that can combine with this one to form a single, larger cache.

The fifth group of metadata values describes the legal ways in which the addresses in this memory bank can be remapped by paged memory management units within the processors to implement a paged virtual memory space:

- **Mappable** (HAL_bool): This attribute indicates to the OS that the associated bank of memory can safely be mapped using paged memory management mechanisms. By definition, a *mappable* region is essentially also *shiftable*, because the various pages may be mapped virtually anywhere into a shared virtual address space.
- **Map pagesize count** (int): The number of legal page sizes.
- **Map pagesizes** (int []): The supported page size(s) for mappable memories. There will typically be one or two small page sizes (4-16KB) used for mapping virtual memory that is dynamically paged, and a few large page sizes (1MB-1GB) used for statically mapping any pages of memory that are actually statically allocated.

Finally, there may also be cases when an architecture has several memory banks that can work both independently or together as a single, large bank. In the latter mode, the various memory banks can often be interleaved together at one or more levels of granularity. This can be handled most easily by defining an additional physical memory block that has its own unique physical address and memory bank object number, but actually corresponds to a group of smaller memory banks when “ganged” together into a super-bank. Such combination banks need no debugging address, since the individual banks can be tested separately. A combination bank can be identified by checking to see that the debugging base address is zero. Permitting *combination memory blocks* requires the following additional attributes:

- **Interleave Factor Count** (int): The number of different legal interleave factors. A combination memory bank that cannot be interleaved has a factor count of 0.
- **Interleave Factors** (int []): The legal granularities at which the individual memory banks making up a combination memory bank may be interleaved by hardware, in address bits. For example, an array of interleaved memories may alternate at each 64-bit word

(granularity = $\log_2 8 = 3$) or be limited to interleaving by 128-byte cache lines (granularity = $\log_2 128 = 7$). An interleave factor of 0 indicates that “not interleaved” is also possible.

4.1.2 Physical Mapping Example

Figure 5 shows a schematic example of physical memory mapping in a PCA architecture. While simple, this example illustrates some of the key concepts from this section. At the top of the figure is a diagram of the actual architecture being analyzed. The heart of this system is composed of four processors on a single die, with each of the processors containing a small local memory. There is also an on-chip memory shared by all of the four processors. Each processor possesses hardware that allows the software-visible addresses of the local memories to be varied, but the shared memory can only be accessed using fixed addresses. Finally, there is a shared bank of off-chip main memory, much like that in conventional PC systems today.

On the lower left is a sample “debugging” address space mapping the six RAM memories present in this system into a simple, flat 64-bit address space. This raw mapping is used only for verification of the chip during the manufacturing process, and is never directly used by software running on any of the chip’s processors. On the lower right, these six memories are remapped into a physical address space that is visible to a TVM-HAL-based run-time system or OS. Generally, this mapping is the same as the debugging address space, but some minor variations are clearly evident in the figure.

The four local SRAMs associated with each processor have a more complex physical mapping. They are usually used only as local scratchpad memory for their associated processor. In order to make all four processors appear identical to software, the physical mapping shifts all of the local memories to share the same physical addresses, as an “equivalent-access memory bank group.” Using these address ranges, each processor can only access the small memory that is actually attached to it. While this remapping may seem counterintuitive at first, software running on *any* of these nodes can access the local memories using the *same* physical addresses, and can therefore run unchanged, without regard to which node it is running on. Using these addresses will automatically access the memory bank that is physically attached to the processor executing the code. The example physical mapping also offers an alternate way to use these memories, in a manner more amenable to streaming execution. The four memories may be viewed together as a single, large “combination” memory block. This block appears as a monolithic bank of memory in the physical map, but is actually made up by interleaving the four real RAM blocks at a number of bytes set by the combination bank’s “interleave factor.”

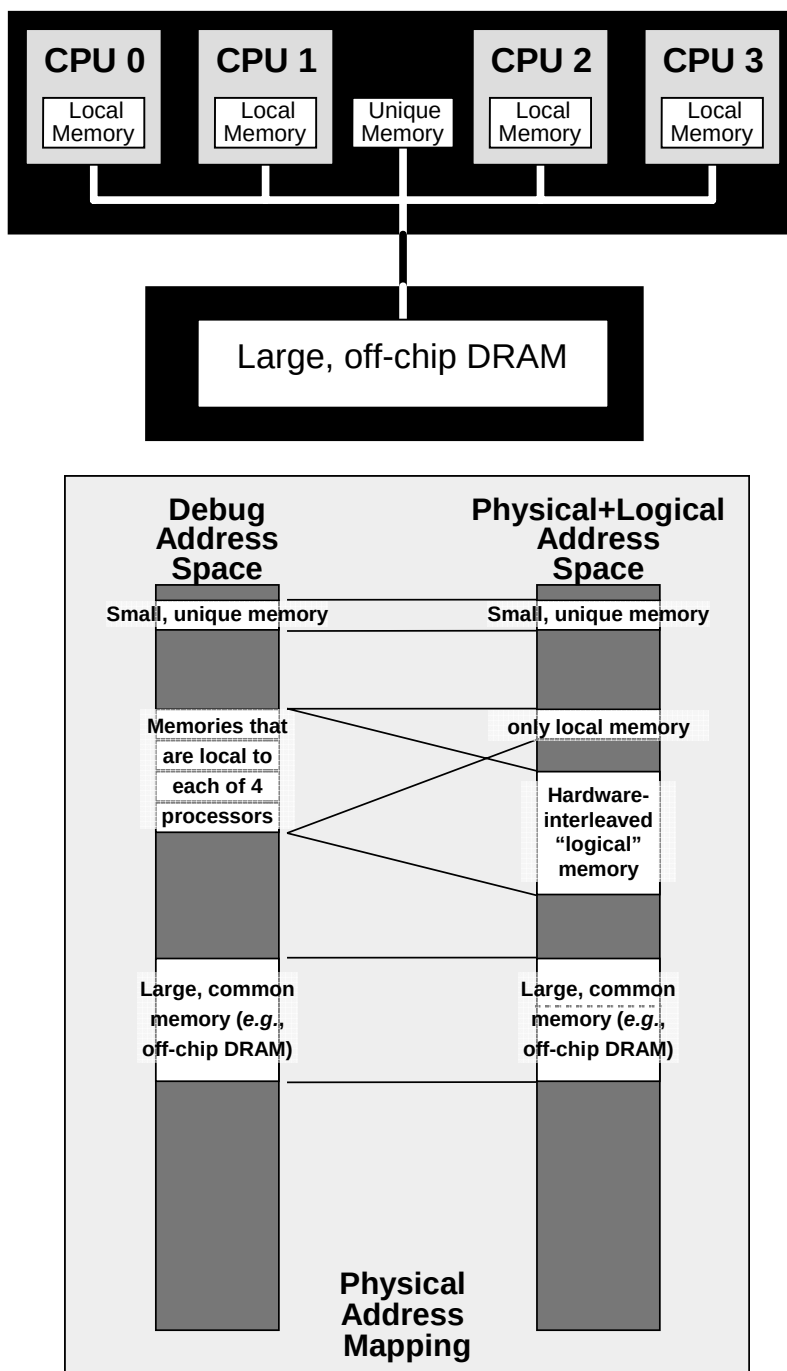


Figure 5: The Physical Address Mapping Process.

Note that these mappings are set entirely by the architecture's PCA machine model. The debugging address space represents the actual RAMs in the chip, while the physical addressing is recorded in the architectural metadata that is created by the architecture designers and input to the compilers. TVM-HAL software cannot affect these mappings, since they represent the memory baseline that can be controlled by software using the segment control code in Section 4.2.

4.2 Segment Control

Executing processors typically view only a subset of the physical address space at any one time. The TVM-HAL segmentation control routines enable an OS or run-time system to correctly manage the “views” of memory provided to running tasks. The OS or run-time system may use the following family of privileged macros to control how the various portions of physical memory are assembled together into a virtual address space usable by threads running on a particular processor. They will typically be called at system startup to do initial memory allocation and then during context switches or morphs to rearrange memory allocations from one process or OS to the next.

4.2.1 Metadata Access

Before it can segment memory for use by user code, an OS must first be able to query a simplified form of the system metadata to get a view of the physical memory map provided by the underlying architecture. One way that this can be performed is by “walking” through the entire hierarchical tree of physical objects contained in the system, starting with the “root” object. As the tree is “walked,” each item can be queried to determine other, related objects in the tree and the characteristics of each object passed. In the case of memory banks, each object has all of the characteristics listed in Section 4.1.1. Other objects in the system also have characteristics of interest, as listed in the reference section of this document, but these are of less importance for code using TVM-HAL calls. The first metadata routines permit “walking” of the tree and listing of characteristics:

```
/* HAL-level metadata information interrogation */
int root_object_identifier = HAL_ROOT_OBJ();
HAL_uint64 characteristic_as_int64 = HAL_GET_OBJ_CHARACTERISTIC(
    int object_identifier, int characteristic_num,
    int array_offset /* 0 if none */);
int object_type = HAL_OBJ_TYPE(int object_identifier);
```

The first routine returns the object identifier, an enumeration of the physical object names in the system, for the “root” object at the base of the system’s object tree. The `HAL_GET_OBJ_CHARACTERISTIC` routine can then be used to determine both characteristics of the item in question and its sub-objects, simply by varying wide range of `characteristic_num` numbers (and `array_offset` numbers for figures that require more than one value to describe them, such as lists of valid cache line sizes). Each object also has a type — root (`HAL_MM_TYPE_ROOT`), level (`HAL_MM_TYPE_LEVEL`), processor (`HAL_MM_TYPE_PROC`), memory (`HAL_MM_TYPE_MEM`), dynamic network (`HAL_MM_TYPE_NET`), or point-to-point link (`HAL_MM_TYPE_LINK`) — that can be determined using the `HAL_OBJ_TYPE` function. Once the type is known, the user can choose characteristics from the memory characteristics described previously or the following lists for other objects.

The root object contains characteristics that describe the nominal word lengths of the system, variable alignment, and other figures that are mostly of use during code optimization and adjustment:

- **Address Size (int):** Bit width of pointers.

- **Word Size (int):** Bit width of the architecture’s natural “word.”
- **Big Endian? (HAL_bool):** Is the machine big endian?
- **Char Signed? (HAL_bool):** Are 8-bit variables nominally signed?
- **Char Size (int):** Bit width of C char type (usually 8).
- **Short Size (int):** Bit width of C short type (usually 16).
- **Int Size (int):** Bit width of C int type (usually 32 or 64).
- **Long Size (int):** Bit width of C long type (usually 32 or 64).
- **Long Long Size (int):** Bit width of C long long type (usually 64).
- **Float Size (int):** Bit width of C float type (usually 32).
- **Double Size (int):** Bit width of C double type (usually 64).
- **LongDouble Size (int):** Bit width of C longdouble type (usually 64–128).
- **Char Alignment (int):** Bit alignment of C char type (usually 8).
- **Short Alignment (int):** Bit alignment of C short type (usually 16).
- **Int Alignment (int):** Bit alignment of C int type (usually 32 or 64).
- **Long Alignment (int):** Bit alignment of C long type (usually 32 or 64).
- **Long Long Alignment (int):** Bit alignment of C long long type (usually 64).
- **Float Alignment (int):** Bit alignment of C float type (usually 32).
- **Double Alignment (int):** Bit alignment of C double type (usually 64).
- **LongDouble Alignment (int):** Bit alignment of C longdouble type (usually 64–128).
- **Root Level (int):** Object identifier of the bottom level object.

Level objects group together other objects into a flat “level” of objects in the hierarchical object tree. Sublevels are then used to group together progressively more tightly-coupled groups of components. Hierarchy-walking code spends most of its time walking through level objects.

- **Processor Count (int):** Number of processors in the level.
- **Processor List (int []):** Object identifiers of processors in the level.
- **Memory Count (int):** Number of memories in the level.
- **Memory List (int []):** Object identifiers of memories in the level.
- **Network Count (int):** Number of networks in the level.
- **Network List (int []):** Object identifiers of networks in the level.
- **Link Count (int):** Number of links in the level.
- **Link List (int []):** Object identifiers of links in the level.
- **Level Count (int):** Number of sub-levels in the level.
- **Level List (int []):** Object identifiers of sub-levels in the level.

Processor objects mostly indicate where processors are located. Most of the other information in these objects may be recovered by other HAL functions, at least for the processor making that HAL call. This metadata is mostly for use if a processor needs to know the status of a *different* processor in the system:

- **Identity (int):** Sequential (0, 1, 2, . . .) number assigned to each processor that is returned by the HAL_PROCESSOR() macro (from Section). Unlike the non-sequential object identifier, this number is useful for tasks such as dividing up work among processors.
- **Number of contexts (int):** The number of contexts, for multi-context processors.

- **Number of performance counters (int):** The number of performance counters usable via HAL performance counter routines on this processor.
- **Have a TLB? (HAL_bool):** Whether or not this processor has a TLB attached. If not, then it cannot use paged memory.
- **Number of preset segments (int):** The number of processor segments that are preset to fixed-function banks of memory for this processor.
- **Preset segment numbers (int []):** Segment numbers of the preset segments.
- **Number of SVM Masters (int):** Number of processors that can be this processor's master when it switches to streaming mode using HAL_SWITCH_TO_SVM.
- **SVM Masters (int []):** Object identifiers of the processors that can be this processor's master when it switches to streaming mode using HAL_SWITCH_TO_SVM.

Network objects do not have any HAL-visible metadata characteristics, while link objects have only a few:

- **Sender (int):** Object identifier of the object at the “head” of the link.
- **Receiver (int):** Object identifier of the object at the “tail” of the link.
- **Bidirectional (HAL_bool):** Indicates that this is a bidirectional link. Otherwise, data flows only from the sender to the receiver.

While “walking” through levels from the root node is a good way to get an overview of the system, often OS code executing TVM-HAL calls will just want to know what kinds of memories are attached, or some other simple analysis. For this purpose, the HAL also includes simpler “iterator” access routines that just sequentially return objects of a given type, allowing a simple loop to walk through all objects of that type while ignoring system structure:

```
/* Object listing routines */
int level_object_identifier = HAL_LEVEL_OBJ_ITERATE(int iterator_count);
int level_object_identifier = HAL_PROC_OBJ_ITERATE(int iterator_count);
int level_object_identifier = HAL_MEM_OBJ_ITERATE(int iterator_count);
int level_object_identifier = HAL_NET_OBJ_ITERATE(int iterator_count);
int level_object_identifier = HAL_LINK_OBJ_ITERATE(int iterator_count);
int number_of_levels = HAL_NUM_LEVELS();
int number_of_procs = HAL_NUM_PROCS();
int number_of_mems = HAL_NUM_MEMS();
int number_of_nets = HAL_NUM_NETS();
int number_of_links = HAL_NUM_LINKS();
```

These routines are used simply by supplying an `iterator_count` ranging from 0 to (`number_of_x - 1`), and it returns the object identifiers of all objects of that type, one by one.

Once it has interrogated the metadata model of the memories that are to be segmented, the OS must determine certain attributes of the processor on which it is executing before it can usefully segment memory, including whether or not it has a TLB (and can therefore use dynamic paging within memory segments) and what “segments” are permanently in use by the processor:

```
/* Processor segmentation characteristic interrogation */
HAL_bool mapping_possible = HAL_HAS_TLB();
int a_preset_segment_number = HAL_PRESET_SEGMENT_NUM(int iterator_count);
int number_of_preset_segments = HAL_NUM_PRESET_SEGMENTS();
```

The preset segments returned iteratively by `HAL_PRESET_SEGMENT_NUM` will usually be for fixed blocks of memory, such as caches, which cannot be selectively reconfigured by the HAL system. These routines provide a way for HAL routines to get the segment numbers for these segments so that they may at least be queried for their characteristics, even if they cannot be modified using HAL calls.

4.2.2 Segmentation Control Routines

Once an OS or run-time system has a view of how the memory system is constructed, it can create a virtual address space for each processor composed of several distinct *segments* of memory. Some of these, especially ones like L1 caches, may be preset by hardware and can only be examined by TVM-HAL code using macros such as `HAL_PRESET_SEGMENT_NUM`. In PCA architectures, however, others should be configurable to allow different computation models to take advantage of different memory layouts. The TVM-HAL model of this mapping is that each processor maintains a hardware table of active segment entries. Each entry in this table can then be assigned arbitrarily to ranges of virtual and physical memory, only limited by the capabilities of the segmentation table control and underlying memory hardware. The following macros are used to manage the current segment table on a processor:

```
/* Memory segment control */
void HAL_CLEAR_SEGMENT(int segment_number);
int HAL_SET_SEGMENT(void *starting_virtual_address,
    HAL_phyPtr starting_physical_address, HAL_uint64 size_in_bytes,
    HAL_bool cacheable, HAL_bool cache_coherent, HAL_bool userExecutable,
    HAL_bool userReadable, HAL_bool userWritable, HAL_bool osExecutable,
    HAL_bool osReadable, HAL_bool osWritable, HAL_bool mapped,
    HAL_uint64 map_physical_size, int interleave_factor);
```

The various parameters to `HAL_SET_SEGMENT` control how the segment assigned to a slot in the table views the physical address space, and how it is seen by any code running on the processor:

- **Starting virtual address:** The address where this segment will appear to start to programs executing on the processor.
- **Starting physical address:** The physical access address in the target memory bank where this segment will actually start. This implicitly determines the memory component number for the `HAL_SET_SEGMENT` call. Note that this must be the same as the “starting virtual address” parameter for non-shiftable memories.
- **Size in bytes:** The size of this segment, in bytes. Note that this will always be the rest of the physical block in memory for non-boundable memories. For page-mapped segments, this is the size of the virtual memory seen by the programs executing on the processor, and can be much larger than the actual physical memory block.
- **Cacheable and cache coherent:** These parameters set whether the segment can be safely cached by local memories and whether or not the low-level compiler should ensure that all accesses are made in a cache-coherent manner. These have no meaning for local memories, but the cacheable flag will often be set to `TRUE` for remote memories (except special cases, such as memory-mapped I/O). The separate cache coherent flag should

only be set when shared memory is absolutely essential, as it may cause severe performance problems on some architectures.

- **Executable, readable, and writable:** These set the permissions for the memory segment. A segment must be executable in order to be fetchable into an instruction cache, readable to accept loads, and/or writable in order for stores to work. These are set separately for user and privileged code, much like UNIX file permissions. However, some of these parameters may not be controllable for all processors. Privileged code, in particular can often ignore permissions, so the OS permissions should be construed as guidelines for accessing memory, and not as strict rules. Some architectures may not differentiate between executable, readable, and/or writable independently in hardware. In these cases, executable and readable may be combined together into a single “readable” flag or all three flags may be combined together into a single “writable” flag.
- **Mapped and map physical size:** These parameters indicate that a large virtual segment is to be mapped by a paged memory management unit onto a much smaller physical segment, indicated by the `map_physical_size` field. An OS that uses this option must use the page table management routines in the next section to control the operation of this functionality.
- **Interleave factor:** This parameter indicates how the bytes of the individual memory blocks within a combo physical memory should be interleaved as they are addressed by this segment.

Once segments have been created (or, more likely, for the preset segments), they may be queried in a manner analogous to metadata objects in order to determine their characteristics using the following routine:

```
HAL_uint64 HAL_GET_SEGMENT_CHARACTERISTIC(int segment_number,  
int characteristic_num);
```

The characteristics that may be read using this call match those programmed in with the `HAL_SET_SEGMENT` call on a one-for-one basis.

Finally, there are several functions to configure, enable, and disable caches. These are designed primarily to be used during system bootup and memory morphing, in order to bypass the caches while any necessary software setup of those memory regions occurs:

```
/* Cache segment control */  
/* "x" in routine names is one of I, D, or U (unified) */  
HAL_xCACHE_ENABLE(int num_components_in_cache,  
int memory_component_ids[], int num_components_below,  
int components_below[], int num_components_above,  
int components_above[], int linesize);  
HAL_xCACHE_ON();  
HAL_xCACHE_OFF();  
HAL_xCACHE_ON_SEG(int segment_number);  
HAL_xCACHE_OFF_SEG(int segment_number);
```

The cache configuration routines currently require three lists of memory bank components and a linesize:

- **Cache Components:** The memory bank components in this list are used to construct the cache.

- **Components Below:** These component identifier(s) identify the processors or $L(n-1)$ -level cache banks that forward misses to this L_n -level cache.
- **Components Above:** The component identifier(s) of any $L(n+1)$ -level caches or RAM memory banks to which misses occurring in the current L_n -level cache should be forwarded.
- **Linesize:** The cache linesize to use for this bank. It must be one of the sizes physically supported by this particular memory bank, or the linesize will be undefined.

Once caches are enabled with these parameters set, the various **ON/OFF** macros may be used to enable or disable the configured caches. The basic version of these macros enables or disables all caches for a processor at once, while the `_SEG` versions selectively control the caches associated with particular memory segments. For example, the latter functionality allows one to turn off a data cache associated with one memory bank while leaving an instruction cache holding instructions from another enabled.

4.2.3 Segmentation Example

This section continues the example started in Section 4.1.2. Figure 6 illustrates one way that the physical memory-mapped in the earlier example can be mapped by TVM-HAL code into a virtual address space for use by applications. Since segmentation is controlled by software, it should be emphasized that this is only one of many ways that the same physical memory could be segmented. An ambitious OS could remap the segments dynamically for different applications which had different memory needs.

The segmentation of the small, SRAM memories is fairly simple in this example. The unique memory shared by all processors is not pageable and is not shiftable, so it is mapped directly to the virtual address space as a segment that appears identical to the physical mapping. The four identical, local memories are shiftable, however, so they are mapped to more convenient virtual addresses that will be translated to the appropriate physical addresses by logic in the load/store units of each processor. In this example, both the equivalent-access view and the combination view of these RAM banks have been mapped using different segments. In practice, an OS would typically map only one of these views of the same RAM to active segments in order to force user code to view these memories either as four separate local memory banks or as a single, large interleaved bank, but not both. These local memories might also be enabled as caches for their associated processors using `HAL_UCACHE_ENABLE`, in order to speed access to the slower, off-chip DRAM bank. In this latter case, they would not be allocated to any virtual address segment, as caches are automatically associated with the segments of the components they are caching.

In this half of the example, the interesting part of the segmentation is in the DRAM. Unlike the other memory banks, this bank is split into two separate segments, using hardware-enforced segment bounds checking to determine which of the two segments is being used by any particular memory reference. The smaller segment is address-shifted, much like the SRAM blocks. As a result, it is treated like normal RAM by software accessing it. The larger region is allocated to a *memory-mapped* segment. In the virtual address space, this segment is much larger than the original DRAM segment holding it, since the actual DRAM only acts as a page cache for the segment's virtual address space. Every access to this segment's virtual addresses is translated through the TLB (and, automatically through the associated page table in memory if

the TLB misses) in the processor core making the access. The result of this translation is then used to access the DRAM bank. Any misses to the page cache in the DRAM trigger an exception, which uses the mechanisms described in Section 3 to transfer control to a page fault handler written in TVM-HAL code. This faulting process and the associated TVM-HAL routines are described more fully in the next section.

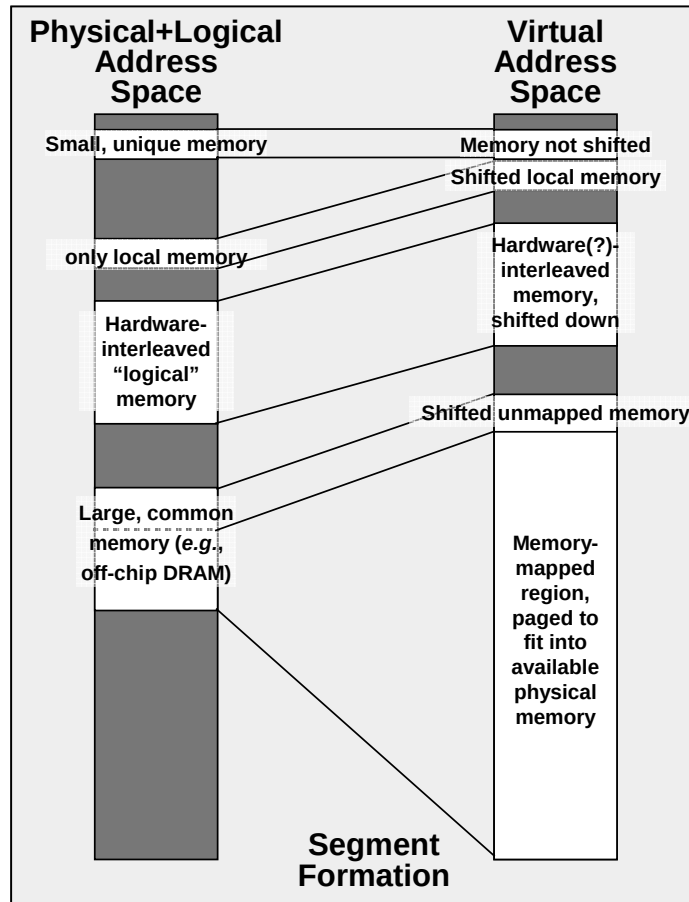


Figure 6: The Segment Mapping Process.

4.3 Paged Memory Control

Any segments that require paging of memory to/from a backing store in order to emulate a large virtual address space in a smaller physical one require some special routines to allow the OS to control the necessary page tables. The TVM-HAL handles TLB refills automatically, because these must be performed with hardware or highly optimized assembly code to avoid a negative impact on performance. However, page faults are much less frequent and can therefore invoke less highly optimized TVM-based code to control how the page table is updated, or to determine if a virtual address error has occurred. This section describes the routines that allow page fault exception handlers written in TVM-HAL code to perform the required functions. In particular, functions to access the page tables themselves are given, as they may be arranged differently (in structure and entry format) for each processor architecture. Page versions of `HAL_phyPtrs` and `HAL_ptr`s (or just `void *`s) are used throughout this section to indicate pointers to the starting addresses of pages in the physical and virtual address spaces, respectively.

The first few functions for managing paged memory control the TLB and memory exception logic within the processor core executing the TVM-HAL code:

```
/* Hardware control */
int HAL_FAULT_TYPE();
int HAL_FAULT_SIZE();
HAL_ptr HAL_GET_FAULT_ADDRESS();
void HAL_LOAD_TABLE_BASE(HAL_ptr new_table_base);

/* TLB Flushing */
void HAL_TLB_INVALIDATE(HAL_ptr virtual_address_of_entry_to_kill);
void HAL_TLB_INVALIDATE_ALL();
```

`HAL_FAULT_TYPE`, `HAL_FAULT_SIZE`, `HAL_GET_FAULT_ADDRESS`, and `HAL_TLB_INVALIDATE` are used together in page fault handlers. The first three routines recover the address of the faulting load or store and determine what it was attempting to do when it faulted. For `HAL_FAULT_TYPE` the return value indicates what type of instruction was executing: `HAL_FAULT_LOAD` = load, `HAL_FAULT_STORE` = store, `HAL_FAULT_IFETCH` = instruction fetch, or `HAL_FAULT_OTHER` = other (usually a synchronization instruction) caused the last page fault. `HAL_FAULT_SIZE` indicates the size of the data word being read or written, in bytes. Together, these two macros can tell the fault handler the opcode of the faulting instruction in an architecturally-neutral manner. Once the handler has determined a victim page to discard from memory, `HAL_TLB_INVALIDATE` can be used to clear the old page entry from the TLB. This routine can also be used to handle TLB shutdown in multiprocessor environments.

`HAL_LOAD_TABLE_BASE` and `HAL_TLB_INVALIDATE_ALL` are used during context switches to quickly switch to an entirely different page table, associated with another process. The former routine tells the hardware which table to switch to, while the latter clears all TLB entries at once so that no entries from the old page table are inadvertently used during execution of the new process (if the underlying architecture requires this). In extreme cases, this routine can also be used for TLB shutdown.

Before the previous routines can be used, a page table must be established in memory for any process using mapped memory. The following routine, `HAL_INIT_PAGETABLE`, creates a stump page table having only a few entries and allocates memory for this table using the memory fetch function supplied by the calling OS. The routine then returns a pointer to the base of the page table that can be used in virtually all other routines in this section. Meanwhile, in order to allow the OS to manage its own page table memory, it records a pointer to an OS-supplied memory fetch/free function for later use, should the table need to expand or shrink at some point. It also records a 16-bit process ID to associate with this page table. This can be a UNIX process ID or just an arbitrary number, as long as each page table present simultaneously has a unique number. When a process is complete, the page table may be eliminated with a call to the matching `HAL_FREE_PAGETABLE` function.

```
/* Page table startup */
typedef HAL_ptr (*TableExpandCodePtr)(HAL_int64 num_bytes_to_allocate,
    int align);
typedef void (*HAL_TableShrinkCodePtr)(HAL_ptr address);
HAL_ptr HAL_INIT_PAGETABLE(int segment_number, int target_segment,
    HAL_uint16 process_id, HAL_int64 basic_page_size, TableExpandCodePtr
    memfetcher_function, HAL_TableShrinkCodePtr memfree_function);
void HAL_FREE_PAGETABLE(HAL_ptr page_table);
```

The memory page control functions clear or set entries in an existing page table. Clearing page table entries upon page eviction is a fairly straightforward process, just requiring a call to the `HAL_CLEAR_PAGE` function. Adding a new page to the table requires a bit more sophistication, however. All three routines must be used in a cycle by the OS to: 1) determine which conflicting pages must be removed from the page table (if it has a limited size), using `HAL_TEST_SET_PAGE`, then 2) remove up to `HAL_MAXIMUM_CONFLICTING_PAGES` conflicting pages using `HAL_CLEAR_PAGE`, before finishing up with 3) a `HAL_SET_PAGE` to lock in the new page. Sample pseudo-code for this operation is given at the end of this section.

```
/* Page table entry control */
void HAL_CLEAR_PAGE(HAL_ptr page_table_base,
    HAL_PagePtr starting_virtual_address);
int HAL_TEST_SET_PAGE(HAL_ptr page_table_base, HAL_PagePtr
    starting_virtual_address, HAL_PagePhyPtr starting_physical_address,
    HAL_int64 page_size, const int memory, int *num_page_conflicts,
    HAL_ptr *conflicts_page_table_base_array,
    HAL_PagePtr *conflicts_virtual_addr_array);
void HAL_SET_PAGE(HAL_ptr page_table_base,
    HAL_PagePtr starting_virtual_address,
    HAL_PagePhyPtr starting_physical_address, HAL_int64 page_size,
    HAL_boolX cacheable, HAL_boolX cache_coherent, HAL_boolX userExecutable,
    HAL_boolX userReadable, HAL_boolX userWritable, HAL_boolX osExecutable,
    HAL_boolX osReadable, HAL_boolX osWritable);
int characteristic_controllable = HAL_PAGING_CHECK(int protection_type);
```

Not all architectures will support all of the page protection and control flags supported here. Because unsupported flags will be ignored, page-level hardware support for each flag should be tested by architecture-neutral software with `HAL_PAGING_CHECK` (which can test all of the possible flags using `protection_type` values of `HAL_CACHEABLE`, `HAL_CACHE_COHERENT`, `HAL_USER_EXECUTABLE`, `HAL_USER_READABLE`, `HAL_USER_WRITABLE`, `HAL_PRIV_EXECUTABLE`, `HAL_PRIV_READABLE`, or `HAL_PRIV_WRITABLE`). Flags that are supported but need to be left in the segment default states can be set as “don’t cares,” as the `HAL_boolX` type is an extended, ternary boolean type (with legal values of `HAL_FALSE`, `HAL_TRUE`, and `HAL_DONT_CARE`).

The page table itself must exist in a mapped or unmapped memory in a different segment from the one being mapped. For example, a user memory table is likely to be in mapped kernel space, while a kernel memory map will be in unmapped memory. If more entries need to be allocated to the table, the TVM-HAL uses the memory fetcher function (set at table initialization) associated with this page table to allocate more page entries for the table. The memory fetcher function is a C function, written in low-level TVM-HAL code, that allocates more space for the page table in the appropriate memory bank. `HAL_TEST_SET_PAGE` automatically calls this

routine when it needs to expand the page tables. The routine acts much like `malloc`, returning a pointer to the allocated memory block, or `NULL` if the requested memory block could not be allocated. However, unlike `malloc` it must only allocate memory blocks with starting addresses that are evenly divisible by the `align` parameter, as most page table implementations require that their table sections align with page boundaries. If simply allocating more memory cannot get the necessary amount of room in the page table, then `HAL_TEST_SET_PAGE` chooses a candidate list of pages to discard from the page table in order to make more room. It is the responsibility of the calling OS to clear (at least some of) these page table entries before they are eliminated permanently by the subsequent call to `HAL_SET_PAGE` that actually loads the new page into the page table.

Finally, the paging software uses these page lookup/status functions to perform utility functions at clock interrupts. All are used when deciding how to discard or flush pages in memory when pages need to be freed to make space for more. In addition, run-time systems with more capable page management algorithms may occasionally use the `HAL_GET_NEXT_PAGE` (for walking page tables) and `HAL_GET_RESET_PAGE_TOUCHED` (for checking and clearing page status) routines to track recently-used pages in order to avoid flushing them.

```
/* Page table lookup & walking */
HAL_bool HAL_PAGE_PRESENT(HAL_ptr page_table_base,
    HAL_PagePtr starting_virtual_address);
HAL_PagePhyPtr HAL_GET_PHY_PAGE(HAL_ptr page_table_base,
    HAL_PagePtr starting_virtual_address);
void HAL_GET_PREVIOUS_PAGE(const int memory, HAL_ptr *page_table_base,
    HAL_PagePtr *starting_virtual_address);
void HAL_GET_NEXT_PAGE(const int memory, HAL_ptr *page_table_base,
    HAL_PagePtr *starting_virtual_address);

/* Page status */
HAL_bool HAL_GET_PAGE_TOUCHED(HAL_ptr page_table_base,
    HAL_PagePtr start_virtual_address);
HAL_bool HAL_GET_RESET_PAGE_TOUCHED(HAL_ptr page_table_base,
    HAL_PagePtr start_virtual_address);
HAL_bool HAL_GET_PAGE_MODIFIED(HAL_ptr page_table_base,
    HAL_PagePtr start_virtual_address);
HAL_bool HAL_GET_RESET_PAGE_MODIFIED(HAL_ptr page_table_base,
    HAL_PagePtr start_virtual_address);
```

Following is the sequence of events in a typical page fault, using these TVM-HAL routines:

- The page fault is received by the OS when the exception handler, previously set using the TVM-HAL exception control routines, is triggered by an instruction exception on an instruction fetch, load, or store.
- The faulting address is recovered using `HAL_GET_FAULT_ADDRESS`, and the address is checked for errors. If the address is valid and from a page that has been swapped to disk or simply removed from the page table due to a conflict, then the fault handler continues. If the address is invalid, then appropriate error-handling mechanisms can be invoked, instead.
- An invalid physical page is selected to hold the new page. If none are available, a victim page in memory is chosen from among those in the page table using a user-defined

algorithm. Most such algorithms work by scanning through pages using `HAL_GET_NEXT_PAGE` and some combination of the `HAL_GET_PAGE_TOUCHED` and/or `HAL_GET_PAGE_MODIFIED` routines during timer interrupts in order to intelligently maintain lists of pages that have not been used recently.

- The status of the page being forced out of memory is recorded (particularly modified status using `HAL_GET_PAGE_MODIFIED`). The old page is then removed using `HAL_TLB_INVALIDATE` (to clear the TLB entry) and `HAL_CLEAR_PAGE` (to remove the victim page from the page table).
- If the victim page was modified, then it is flushed out to disk using run-time I/O routines. This long-duration task can be delayed to the end of the process simply by having the OS always ensure that it has some unmodified pages handy at all times which may be overwritten without bothering with this step.
- Run-time I/O routines are used to fetch the page containing the faulting address into memory at the physical address of the old victim page.
- When I/O completes, the new page is added to the page table with the page table management functions, using code similar to the following pseudo-code:

```
int must_remove, can_remove;
HAL_ptr ptbs[HAL_MAXIMUM_CONFLICTING_PAGES], victim_ptb;
HAL_PagePtr vas[HAL_MAXIMUM_CONFLICTING_PAGES], victim_va;

must_remove = HAL_TEST_SET_PAGE( <<new page info>> , &(can_remove),
    &(ptbs[0]), &(vas[0]));
while (must_remove > 0)
{
    << Choose a candidate page from among the can_remove pages in the
        ptbs/vas lists here, put in victim_ptb/victim_va >>
    HAL_CLEAR_PAGE(victim_ptb, victim_va);
    must_remove -= 1;
}
HAL_SET_PAGE( <<new page info>>, <<new page access flags>> );
```

- Execution resumes. Usually, an immediate TLB refill will be triggered to load the new page table entry into the TLB before the process may continue.
- While execution of the faulting process is resuming, if any of the victim page(s) cleared out during the page table update were modified, then they may be flushed out to disk using run-time I/O routines in the background.

Because all of these functions are complex, the complete page fault process may require many thousands of processor instructions.

5 Other Extensions

Most low-level functions under typical OSES involve exceptions or memory, so those are the focus of the majority of the TVM-HAL macros. However, many other features are commonly found in processors that require small sequences of unique, architecture-specific instructions to access or control them. This section describes a variety of TVM-HAL routines that are included to handle these other, often more minor functions.

5.1 Processor Identification

It can sometimes be useful for a program to execute different code depending upon which processor it is being executed. For instance, a processor on the edge of a processor grid may choose to execute I/O code for the other processors, or the code might select different memory configurations based on the current processor ID. This macro returns the current processor's ID number, which is numbered sequentially beginning at 0. The processor ID can easily be converted to an object identification number using `HAL_PROC_OBJ_ITERATE`.

```
int processor_ident = HAL_PROCESSOR_IDENT();
```

5.2 Active DMA

The low-level DMA model supported directly in the TVM-HAL is a fairly simple block DMA transfer mechanism that can work by either blocking calling threads or through active notifications passed through the TVM-HAL exception model. The sender, receiver, or both can be notified when a DMA operation is complete, and can then use this signal to trigger further operations. No significant error checking or memory protection functions are encapsulated within these routines. While they return error codes, the values permitted will most likely be simple options such as 0=successful, 1=failure. More detailed error analysis must be included in any OS or run-time library wrapper layers, if it is desired. With the following set of exception handlers and OS layers, fairly complex high-level network protocols can be built upon this foundation

```
/* Thread-blocking block moves */
int error_code = HAL_BLOCKMOVE_P(const int DMA_controller_num,
    const int memory_source, const int memory_dest,
    HAL_phyPtr source_phy_ptr, HAL_phyPtr dest_phy_ptr, int num_bytes);
int error_code = HAL_BLOCKMOVE(const int DMA_controller_num,
    const int memory_source, const int memory_dest,
    void *source_ptr, void *dest_ptr, int num_bytes);

/* Exception-triggering block moves */
void HAL_ACTIVE_BLOCKMOVE_P(const int DMA_controller_num,
    const int memory_source, const int memory_dest,
    HAL_phyPtr source_phy_ptr, HAL_phyPtr dest_phy_ptr, int num_bytes,
    int source_vector, const int dest_processor_num,
    int destination_vector, const int memory_error_code,
    HAL_phyPtr put_error_code_here_mem_ptr);
```

```
void HAL_ACTIVE_BLOCKMOVE(const int DMA_controller_num,
    const int memory_source, const int memory_dest,
    void *source_ptr, void *dest_ptr, int num_bytes,
    int source_vector, const int dest_processor_num,
    int destination_vector, const int memory_error_code,
    int *put_error_code_here_ptr);
```

The first pair of block moves cause the entire threaded processor to block and wait until the block move is complete. As a result, they will mostly be of use only in simple systems that run without an operating system (and therefore do not have other threads available to run on the processor while waiting for DMA to occur).

The second set of block moves should be called by any OS or run-time libraries that control on-chip networks (such as RapidIO or VIA) or DMA engines. The major difference between these block moves and the first pair of block moves is that they run in the background and then trigger interrupts at the DMA initiator, the specified receiver node, or both when the DMA operation completes. The call may optionally indicate 7-bit exception vectors (but not a PC for an exception handler directly) for the source and/or destination processors to take when the operation is complete. This provides a balance between fast interrupt handling and secure functionality. The vector numbers should be left equal to zero if no interrupt is desired for the source and/or destination. Finally, the DMA vector numbers should be in the range of 128-255 (or 0, for disabled). This allows the DMA vectors to share the basic exception vector table. Any other values will result in no interrupt, although all but zero should be considered reserved. Thus, a vector number of 0 should be used if no interrupt is desired, rather than a random number. The associated interrupt vector must be set up in advance by the run-time system working on that particular processor in order for the interrupt to be handled properly. This setup should also include establishing any necessary protocols (using variables in directly shared memory or a protocol established in advance) so that the receiving processor knows what block of data has arrived when it is interrupted, since the **BLOCKMOVE** calls do not provide any inherent mechanism for transmitting this information. If the vector has not been prepared, then the interrupt will generally be ignored. Also, note again that these routines perform no inherent error-checking in order to maximize performance, and could therefore result in undefined behavior if used by an OS that sends out-of-range or improper vector numbers (such as the Reset vector, for example).

When a DMA controller completes an active block move, it can send interrupts to the affected processors. If the maskable interrupt (#96-126) triggered by the DMA controller handling this block move (specified by the first parameter) has been either assigned to an exception handler (with **HAL_EXCEPTION_ENABLE**) or has had the default DMA exception handler enabled (with **HAL_DMA_ENABLE**), then the interrupt will be taken as soon as possible (*i.e.*, when it is not masked by a higher-priority interrupt). A user-defined exception handler attached directly to the DMA controller can be invoked, but in general it is better to use the default handlers. The primary reason is to maintain portability of the OS code, even with interrupt controllers that use varying DMA interrupt semantics. The default DMA handler reads the vectors sent with the block DMA call and vectors the processor to the exception specified in the **source_vector** (for the sender) or the **destination_vector** (for the receiver) fields of the block move. This mechanism is often more powerful than simply attaching a single exception handler to a

DMA controller, since it allows many different responses to block DMA operations to be handled by the controller while maintaining a relatively simple API interface for all calls.

Both forms of block move can return an error code indicating the outcome of the DMA transfer. For the active block move, this error code is stored into an address provided by the sender (or omitted if this pointer is NULL).⁵ Compile-time errors may also occur. Some source or destination processors may not be able to accept interrupts from a particular DMA controller in a PCA architecture, because of limited interconnection network availability. In these situations, the low-level compiler should return errors when it tries to compile block moves requesting interrupts on these missing links.

5.3 Multiprocessor Synchronization

This specification adopts a POSIX pthreads model [7] for handling multiprocessor synchronization issues. Most pthreads calls translate easily into low-level macros with a small amount of wrapper code to match pthreads requirements. To keep the underlying TVM-HAL as simple as possible, however, only the atomic memory references or potentially hardware-assisted functions at the core of the pthreads routines are implemented. The functions implemented are as follows:

```
/* Atomic test-and-set for locks and condition variables */
int HAL_MUTEX_INIT_P(const int memory,
    HAL_phyPtr_mutex_t phy_lock_ptr);
int HAL_MUTEX_TRYLOCK_P(const int memory,
    HAL_phyPtr_mutex_t phy_lock_ptr);
int HAL_MUTEX_UNLOCK_P(const int memory,
    HAL_phyPtr_mutex_t phy_lock_ptr);
int HAL_MUTEX_INIT (const int memory, HAL_mutex_t *lock_ptr);
int HAL_MUTEX_TRYLOCK(const int memory, HAL_mutex_t *lock_ptr);
int HAL_MUTEX_UNLOCK(const int memory, HAL_mutex_t *lock_ptr);

/* Barrier synchronization, direct physical memory version */
int HAL_BARRIER_GLOBAL_INIT_P(const int memory,
    HAL_phyPtr_barrier_t phy_barrier, int proc_count);
int HAL_BARRIER_LOCAL_INIT_P(const int memory,
    HAL_phyPtr_barrier_local_t phy_barrier);
int HAL_BARRIER_ENTER_P(const int memory,
    HAL_phyPtr_barrier_t phy_barrier,
    HAL_phyPtr_barrier_local_t phy_barrier_local);

/* Barrier synchronization, normal version */
int HAL_BARRIER_GLOBAL_INIT(const int memory, HAL_barrier_t *barrier,
    int proc_count);
int HAL_BARRIER_LOCAL_INIT(const int memory,
    HAL_barrier_local_t *barrier);
int HAL_BARRIER_ENTER(const int memory, HAL_barrier_t *barrier,
    HAL_barrier_local_t *barrier_local);
```

⁵ The full enumeration of potential error codes is still to be determined.

The TRYLOCK routine is a wrapper around a generic atomic test-and-set that can be used to implement a variety of pthreads-like lock and condition variable functions. The various barrier wrappers hew more closely to the input API because some or all of these may be assisted by barrier-control hardware. Like the DMA functions, these routines also have memory-type hints to aid the low-level compiler during code generation. Both define versions that access physical memory directly (for some low-level OS or run-time system use) and through virtual addresses (for most use). Both forms of synchronization also define structs to contain the objects in memory: `HAL_mutex_ts` and `HAL_barrier_ts/HAL_barrier_local_ts`. These architecturally-defined types contain enough information for the HAL to build a mutex element or barrier, respectively, on a given architecture. The barrier types are split into two halves, one global (one per barrier) and one local (repeated on each processor participating in the barrier). The latter allows each processor to maintain some private state associated with each barrier in order to allow for more efficient implementations.

5.4 Cache Memory Control

Cache control functions allow the program to explicitly control the processor's load/store unit and cache memory coherence to facilitate the building of additional, software-defined synchronization primitives. These functions are especially useful if incoherently cached memory is used for synchronization. Examples of useful primitives are memory sync, prefetch operations, and cache flushing. The following API encapsulates these operations:

```
/* Memory barrier operations */
void HAL_MEMSYNC();           // Blocking
HAL_bool state = HAL_MEMSYNC_TEST(); // Nonblocking

/* Memory barrier for a particular address */

/* Physical addressing versions (privileged only) */
HAL_MEMFENCE_P( const int memory, HAL_phyPtr phy_address );
HAL_bool state = HAL_MEMFENCE_TEST_P( const int memory,
    HAL_phyPtr phy_address );

/* Normal versions */
HAL_MEMFENCE( const int memory, void *address );
HAL_bool state = HAL_MEMFENCE_TEST( const int memory, void *address );

/* Cache control operations, physical addressing versions (privileged
only) */

/* Make MESI = I, discard any modified data */
void HAL_CACHE_INVALID_P(const int memory, HAL_phyPtr phy_address,
    int num_bytes);
/* Make MESI = I, save any modified data */
void HAL_CACHE_FLUSH_P(const int memory, HAL_phyPtr phy_address,
    int num_bytes);
/* Make MESI = S, prefetching data in "shared" state (for reading) */
void HAL_PREFETCH_READ_P(const int memory, HAL_phyPtr phy_address,
    int num_bytes);
/* Make MESI = E, prefetching data in "exclusive" state (for writing) */
```

```

void HAL_PREFETCH_WRITE_P(const int memory, HAL_phyPtr phy_address,
    int num_bytes);
/* Make MESI = M, clearing out line to 0's (without fetch, if possible) */
void HAL_TOUCH_WRITE_P(const int memory, HAL_phyPtr phy_address,
    int num_bytes);
/* Invalidate instructions */
void HAL_CACHE_INVALID_INST_P(const int memory, HAL_phyPtr phy_address,
    int num_bytes);
/* Prefetch instructions */
void HAL_PREFETCH_INST_P(const int memory, HAL_phyPtr phy_address,
    int num_bytes);

/* Normal versions */
void HAL_CACHE_INVALID( const int memory, void *address, int num_bytes);
void HAL_CACHE_FLUSH( const int memory, void *address, int num_bytes );
void HAL_PREFETCH_READ( const int memory, void *address, int num_bytes);
void HAL_PREFETCH_WRITE(const int memory, void *address, int num_bytes);
void HAL_TOUCH_WRITE( const int memory, void *address, int num_bytes );
void HAL_CACHE_INVALID_INST( const int memory, void *address,
    int num_bytes);
void HAL_PREFETCH_INST( const int memory, void *address, int num_bytes);

```

The memory barrier operations either force all load/store operations for the entire thread (or a particular address accessed by that thread) to go to completion, or check on the current status of load/store operations for that thread (in the non-blocking form). Cache operations prefetch or flush out all of the cache lines in a range of memory, and leave the MESI cache coherence bits for shared memory (if present) in the indicated state. Direct physical addressing versions are provided for low-level OS or run-time memory management, while virtual address versions are provided for more general use.

The invalidate operation blindly invalidates all data from the cache without writeback, even if it has been modified, while the flush operation forces a writeback if necessary. On incoherently cached regions of memory, they are functionally identical. Similarly, for incoherent memory the two data prefetch operations are functionally identical. The TOUCH operation is an alternative to prefetching before writes that clears out a range of cache memory instead of loading the current state into the cache. On some hardware, this can be performed without actually consuming bandwidth fetching the current state out of memory.

5.5 IEEE 754 Floating Point Control

C provides access to floating point arithmetic through its normal mathematical operators (+, -, *, and /) and standard mathematics library (`math.h`). However, it does not provide any access to the additional math functions provided in the IEEE 754 floating point arithmetic standard, and as a result they have been very difficult to use in the past, typically accessible only through architecture-specific control routines. The ANSI C99 specification has solved this problem with the addition of the `fenv.h` header file. Note that OSes enabling the use of these functions will still need to add appropriate floating point exception handlers, using HAL routines, in order to properly direct any floating point exceptions that may result.

5.6 Performance Counter Control and Access

Many modern processors include monitor hardware designed to aid the tuning of high-performance software. The TVM-HAL provides access to the monitor hardware with the following set of routines:

```
/* Reset counters */
void HAL_PERF_RESET_ALL();
void HAL_PERF_RESET(int type_select, int counter_select);
HAL_uint64 HAL_PERF_RESET_EXCEPTION(int type_select, int counter_select,
    HAL_uint64 except_count);

/* Read counters */
HAL_uint64 HAL_PERF_READ(int counter_select);
HAL_uint64 HAL_PERF_READ_RESET(int counter_select);

/* Test available counters */
HAL_bool HAL_PERF_PRESENT(int type_select, int counter_select);
HAL_bool HAL_PERF_EXCEPTION_OK(int type_select, int counter_select);
HAL_uint64 HAL_PERF_MAX_COUNT(int type_select, int counter_select);
```

These routines reset, read, or read and reset the current value of any hardware performance counters (as 64-bit integers). If the counters in a particular architecture are only 32 bits long, the upper 32 bits of the returned result are set to zero. `HAL_PERF_RESET` chooses one of the performance counters, selected using `counter_select`, and tasks it to count a particular type of performance statistic using `type_select`, which chooses a statistic from the list of possible statistics in Table 2. Note that not all counters may be able to count all different types of statistics on some architectures. If an architecture does not provide enough data to calculate a particular performance counter statistic, an attempt to count it always returns a performance count of 0. In order to avoid this problem, it is recommended that users use the various “test” routines to determine the capabilities of the underlying performance measurement hardware before actually using it.

The `HAL_PERF_RESET_EXCEPTION` call allows an alternate mode of performance counting. With it, the user can request that a performance interrupt (#126) be triggered when the count reaches a certain preset amount. Using this mechanism, an OS no longer needs to poll the performance counters, but can just take statistics when interrupted. If an appropriate trigger count can be determined in advance, this can be a good alternative. However, since this requires more complex hardware and software, fewer processors implement it effectively. As a result, the simpler polling of performance counters is generally a better choice.

Number	Description
0	Processor cycles (time)
1	RESERVED
2	Number of instructions executed
3	Number of cache misses that have occurred (I)
4	Number of TLB misses that have occurred (I)
5	Number of loads executed
6	Number of cache misses that have occurred (D: read)
7	Number of TLB misses that have occurred (D: read)
8	Number of stores executed
9	Number of cache misses that have occurred (D: write)
10	Number of TLB misses that have occurred (D: write)
11	Number of loads + stores executed
12	Number of cache misses that have occurred (D: read + write)
13	Number of TLB misses that have occurred (D: read + write)
14	Number of correctly predicted branches executed
15	Number of branch mispredictions

Table 2: Generic performance counters supported by the HAL model.

5.7 Power/Performance Tradeoff Parameters

The TVM-HAL can provide access to hardware functions that allow precise control over processor aspects that affect power usage. Most of these TVM-HAL routines are accessible only by privileged code. The routines provided currently for power control test whether the processor clock frequency can be modified, and read and modify the processor clock frequency if so:

```
/* Clock speed variation */
HAL_uint64 HAL_GET_CLOCK_FREQ();
void HAL_SET_CLOCK_FREQ(HAL_uint64 frequency);
HAL_bool HAL_TEST_SET_CLOCK_FREQ();
```

The clock frequency is specified in Hertz. The frequency may be decreased by OS software to a low frequency to save power if the workload permits, or increased to a higher frequency when more performance is required.

The following routines completely shut down a processor (or all processors on a PCA chip) during time periods when there is no work to be done:

```
/* Processor sleep-until-interrupt */
void HAL_DOZE();
void HAL_DOZE_ALL();
void HAL_SLEEP();
void HAL_SLEEP_ALL();

/* Test for processor sleep-until-interrupt */
HAL_bool HAL_TEST_DOZE();
HAL_bool HAL_TEST_DOZE_ALL();
HAL_bool HAL_TEST_SLEEP();
HAL_bool HAL_TEST_SLEEP_ALL();
```

The `HAL_DOZE` command is used to turn off the processor in such a way that it may be restarted immediately after an interrupt. For typical processors, this will mean turning off the clock to the processor itself while keeping the PLL driving the clock and bus logic running. For longer shutdowns, `HAL_SLEEP` sets the processor to a quiescent state from which a restart may require up to approximately a millisecond. Depending on the architecture, this may allow the system to completely shut the chip down, turning off the PLL and on-chip memories entirely. The various “test” versions of the macros allow OSes to determine whether or not a processor supports particular power-down modes.

Another factor that is helpful for power management is temperature measurements, which are primarily used to avoid overheating today’s hot and fast processors. The following two macros allow an OS to measure the current processor temperature, and to determine whether or not the measurement is legitimate.

```
/* Temperature measurement */
HAL_int32 HAL_GET_TEMPERATURE();
HAL_bool HAL_TEST_GET_TEMPERATURE();
```

The returned integer value is somewhat unusual for this particular macro. Instead of just returning the current processor temperature, it returns the (processor temperature in °C)•(256). This allows fractional temperatures down to 1/256 of a degree to be returned, if the measurement hardware in the system is capable of such precision.

5.8 Data Watchpoint Control

Debuggers often need to monitor particular instructions or data items at runtime to determine when the instructions are executed or the data is modified. On the instruction side, it is a fairly straightforward process for a debugger to simply replace a “watched” instruction with a breakpoint or system call instruction that triggers an exception every time that the processor attempts to execute the original instruction. This software-only approach works well, and can be implemented without any hardware support other than the system call instruction itself. On the other hand, using software to “watch” for reads and writes of a data item can be costly and difficult, as there are often many loads and stores in a program that may all attempt to read or write the variable. Marking and monitoring all of these with software is usually impractical.

As a result, many processors today implement *watchpoint* registers, which hold one or more addresses and trigger exceptions when loads and/or stores access those addresses. This provides a much more reasonable and low-overhead way for debuggers to monitor a small number of data addresses without slowing down execution of the program dramatically. The TVM-HAL

provides a small set of routines to access any watchpoint registers in a target processor in an architecturally-neutral manner:

```
/* Capability test macros */
int HAL_WATCHPOINT_COUNT();
HAL_bool HAL_WATCHPOINT_TYPE_CHECK(int type);

/* Watchpoint control macros */
void HAL_SET_WATCHPOINT(int number, int type, void *address);
void HAL_CLEAR_WATCHPOINT(int number);

/* Watchpoint information macros */
int HAL_GET_WATCHPOINT_NUMBER();
void *HAL_GET_WATCHPOINT(int number);
void *HAL_GET_WATCH_ADDRESS();
```

HAL_WATCHPOINT_COUNT and HAL_WATCHPOINT_TYPE_CHECK should be used first by any debugger to verify the capabilities of the underlying hardware. _COUNT returns the number of watchpoint registers that are available, and _TYPE_CHECK returns whether or not the registers are capable of operating in various modes: exception on read (HAL_WATCH_LOAD), exception on write (HAL_WATCH_STORE), and exception on both read and write (HAL_WATCH_LOAD_STORE). After the debugger establishes the hardware capabilities, then it can use HAL_SET_WATCHPOINT to start hardware monitoring of an address with one of the watchpoint registers, or HAL_CLEAR_WATCHPOINT to end monitoring. Lastly, watchpoint exception handlers (#3) can use HAL_GET_WATCHPOINT_NUMBER and HAL_GET_WATCHPOINT to determine the address of the data address that caused the exception and HAL_GET_WATCH_ADDRESS to get the address of the load or store that triggered the fault.

5.9 Generic Special Register Access

The TVM-HAL provides a generic mechanism to read and write special registers. This allows programmers to access special registers within a processor directly, without the normal architecture-neutralization TVM-HAL layers. This functionality is provided in both 32-bit and 64-bit variants, although typically only one will work on a particular architecture (the one that matches the “natural” size of the general-purpose registers, usually):

```
HAL_uint32 HAL_GENERIC_READ32(int register_select);
HAL_uint64 HAL_GENERIC_READ64(int register_select);
HAL_GENERIC_WRITE32(int register_select, HAL_uint32 new_value);
HAL_GENERIC_WRITE64(int register_select, HAL_uint64 new_value);
```

The interpretation of `register_select` is architecture-dependent. For example, on a MIPS processor `register_select` would use the values 0 to 31 to represent coprocessor 0 control registers 0-31 and a value of 32 to represent the floating point control/status register, while progressively higher numbers could be used to map other special purpose registers. For a PowerPC, the various special purpose registers could be accessed using these macros according to the register number used with the PowerPC’s MTSPR/MFSPR instructions. Other architectures, including PCA systems, would have their own unique mapping tables.

6 TVM-HAL Reference

This section lists the TVM-HAL routines with a summary of all inputs and outputs for each. Throughout the section, all parameters to directives or macros that must be compile-time constants are marked with the `const` keyword. Parameters to macros that can dynamically vary at run time are not marked with this keyword. This nomenclature is used to clearly distinguish those parameters must be set by the high-level compiler in the two-level compilation framework, during compilation, from those that can remain variable at runtime.

6.1 Compiler Directives

This section summarizes the compiler directives defined as a part of the TVM-HAL.

6.1.1 Processor Selection

HAL_PROCESSOR

```
HAL_PROCESSOR (  
    const int num_processor_targets,  
    const int processor_targets[]);
```

This compiler directive controls which processor core(s) within a particular PCA chip should be targeted by the low-level compiler as it compiles the following code. All PCA chips are multiprocessors of some sort. If all processors are identical, and can all run any code, then this directive is unnecessary. Otherwise, the directive specifies which subset of processors within a heterogeneous PCA chip should be considered valid targets for the functions following the directive. The low-level compiler should respond by producing different versions of object code for each of the specified processors that require it.

Inputs:

const int num_processor_targets: The number of elements in the `processor_targets` array.

const int processor_targets: An array containing a list of processor object identifiers (as specified in architecture metadata) for processors that may be required to execute the functions located in the TVM source after this directive. Because this is a compiler directive, these must be actual constants in the source code text.

Used by:

All TVM code, before or between functions, may be automatically generated.

Described in:

Section 3.3.1

6.1.2 Marking Exception Handlers

HAL_EXCEPTION_HDL ...

HAL_RESET_HDL ...

Exception and reset handlers are marked with the following directives at the beginning of the handlers. Since the handlers are C functions invoked by the hardware at interrupts, when there is no call stack functioning, these are used where one would typically find a return value at the beginning of normal functions, in place of a keyword like `void`.

- **HAL_EXCEPTION_HDL**: Indicates that the following routine is an exception handler, and will be entered with no valid stack or global pointers and a register state that is volatile and should therefore not be overwritten until the **HAL_OVERWRITE_ON** macro is used to indicate that it is safe for normal register allocation to commence.
- **HAL_RESET_HDL**: Indicates that the following routine is the reset handler, and will be entered directly from the basic, architecture-specific initialization code after system reset. This code has many restrictions upon it, as the memory state within the system is assumed to be completely scrambled upon entry except for the small ROM segment containing this handler alone as segment #0. This routine should organize the memory for the system, and then commence with normal execution of a core boot routine like a thread switcher.

Used by:

Exception or cold-reset handlers, always manually inserted.

Described in:

Sections 3.2.2 and 3.3.2

6.1.3 Register File Control

HAL_OVERWRITE_ON:

HAL_OVERWRITE_OFF:

HAL_STACK_ON:

HAL_STACK_OFF:

HAL_GP_ON:

HAL_GP_OFF:

The following directives, used in a manner like C *labels* in any threaded code, notify the compiler as to how it may currently use processor registers (the whole register file, the stack pointer, and the global pointer, in particular) while generating code:

- **HAL_OVERWRITE_ON**: Normal register allocation may commence after this directive.
- **HAL_OVERWRITE_OFF**: After this point, only a very limited register set may be used, as the program is entering a context-switch zone of code. On typical architectures, it will usually be one or two OS-only registers or a set of special shadow, OS-only registers.
- **HAL_STACK_ON**: The compiler may assume that the stack pointer is now valid, and use the C stack in a normal manner to allocate local variables and stack frames for function calls from this point onwards.

- **HAL_STACK_OFF:** The stack pointer should now be regarded as invalid. No local variables may be used or function calls may be made until the stack is re-enabled.
- **HAL_GP_ON:** The global pointer should be considered valid now, allowing for optimized address calculation of global variables in one or a pair of global blocks referenced by the last **HAL_USE_GP1** or **HAL_USE_GP2** directive. Other globals may still be accessed following this directive, but require slower, “from scratch” address calculations before each access.
- **HAL_GP_OFF:** The global pointer should not be used, so global variables must always be accessed by calculating their location in memory “from scratch” before accessing them. This is slower, but the variables may still be used as necessary.

Used by:

While these may all be used anywhere, in practice the **OVERWRITE** and **STACK** directives will only be used by exception or cold-reset handlers, while the **GP** directives may be used in any TVM code that moves the GP, and may therefore be automatically generated.

Described in:

Section 3.2.2 and 3.2.4

6.1.4 Variable Access Control Directives

HAL_STACK_LOCATION

```
HAL_STACK_LOCATION(const int memory);
```

Notifies the compiler which physical memory bank is being used to hold the stack for the following C function(s), so that memory accesses to the stack may be compiled correctly.

Input:

const int memory: The physical memory bank (identified by optimization number) containing the stack for the following function(s).

Used by:

All TVM code, before or between functions; may be automatically generated.

Described in:

Section 2.1

HAL_GLOBAL_BLOCK**HAL_GLOBAL_BLOCK_BASE**

```
HAL_GLOBAL_BLOCK(  
    new_identifier,  
    const int memory,  
    const HAL_bool grow_upwards);  
HAL_GLOBAL_BLOCK_BASE(  
    new_identifier,  
    const int memory,  
    const HAL_bool grow_upwards,  
    const void *base_absolute_address);
```

Define an empty global block to contain global variables. This is a range of memory that starts at the *base_absolute_address* (where “starts” can be defined as either the top *or* bottom). At least one file of a linked set used to build a binary should use the `_BASE` version of the directive in order to set an origin point for the block to grow from. Others may use the simpler form, which allows them to add variables to a global block that has been defined externally in another file. After a global block has been defined, with either directive, globals may be added to the block using the `HAL_GLOBALS`, `HAL_GLOBALS_INIT`, and `HAL_GLOBALS_UNINIT` directives.

Inputs:

- new_identifier*: The name of the global block to be defined. This must be unique for each global block defined within a linked-together binary file.
- const int memory*: The physical memory bank (identified by optimization number) containing the global block.
- const HAL_bool grow_upwards*: A boolean value that, if true, indicates the block grows upwards (new global variables are allocated at increasing addresses) from its base address. Otherwise, the block grows downwards in memory from the base address.
- const void *base_absolute_address*: The absolute virtual (post-segmentation) address where this global block starts growing (its origin point). This only needs to be defined in one source file from among a linked set using the same global block, but if the `_BASE` form of the directive is used in multiple files then the value must be the same everywhere. For downward-growing blocks, the first byte allocated is actually (base – 1), and not the base itself, in order to allow separate upward-growing and downward-growing blocks to share the same origin.

Used by:

All TVM code, with the global variable declarations at the beginning of C files; may be automatically generated for a few basic global blocks (such as the standard UNIX-C pair or others used by particular compiler and/or OS frameworks).

Described in:

Section 2.2

HAL_GLOBALS**HAL_GLOBALS_INIT****HAL_GLOBALS_UNINIT**

```
HAL_GLOBALS(identifier of global block);
HAL_GLOBALS_INIT(identifier of global block);
HAL_GLOBALS_UNINIT(identifier of global block);
```

These directives select global blocks that have been defined previously with the `HAL_GLOBAL_BLOCK` directives for allocation of global variables subsequently defined in the code. Following one of these directives, which should be scattered among the global variable declarations at the beginning of legal TVM-HAL code, any global variables that are present in the file will be allocated according to the following rules:

- **HAL_GLOBALS**: All globals following this declaration, until another one of these directives is invoked, are added to the single global block selected here.

- **HAL_GLOBALS_INIT:** All *pre-initialized* globals following this declaration, until the next **HAL_GLOBALS** or **HAL_GLOBALS_INIT** invocation, are added to the global block selected here. Allocation of uninitialized globals is unaffected.
- **HAL_GLOBALS_UNINIT:** All *uninitialized* globals following this declaration, until the next **HAL_GLOBALS** or **HAL_GLOBALS_UNINIT** invocation, are added to the global block selected here. Allocation of initialized globals is unaffected.

Finally, it should be noted that declaring a global variable before using any of these directives is an error in TVM-HAL code, because the low-level compiler will not be able to determine where to allocate the global (as HAL code is designed to run on a system with complex, segmented memories).

Input:

identifier of global block: The identifier of the global block to which subsequent global variables are to be attached.

Used by:

All TVM code, before or among global variable declarations; may be automatically generated.

Described in:

Section 2.2

HAL_USE_GP1**HAL_USE_GP2**

```
HAL_USE_GP1(  
    identifier_of_global_block,  
    const int offset_into_block);  
HAL_USE_GP2(  
    identifier_of_upward_global_block,  
    identifier_of_downward_global_block,  
    const int offset_into_upward_block);
```

Unlike the previous directives, which are used only in the variable declaration parts of C files, these may either be used before/between function calls *or* within functions. In the latter case, they should always be paired with **HAL_SET_GP** macro, which does the actual loading of the GP register. They are used to notify the low-level compiler which global blocks may be addressed in an accelerated manner by using the GP register as a base address. If used in the middle of functions, these should only be used while the code is under the influence of a **HAL_GP_OFF** directive, and the global variable access acceleration will only take effect after the next **HAL_GP_ON** directive.

The **HAL_USE_GP1** version should be used if the GP is placed at the base address of a single global block, while the **HAL_USE_GP2** version should be used if the GP is set between two global blocks, one grown upwards and one grown downwards from the GP address. This is common when one block is filled with initialized data and the other with uninitialized data.

Inputs:

- identifier_of_global_block*: This is the name (a C identifier) of the global block that the GP is currently pointed at.
- identifier_of_upward_global_block*: This is the name (a C identifier) of the upward-growing global block of a downward/upward pair that the GP is currently pointed at.
- identifier_of_downward_global_block*: This is the name (a C identifier) of the downward-growing global block of a downward/upward pair that the GP is currently pointed at. This block must share the same origin address as the upward-growing block, or a compiler error will occur.
- const int offset_into_block*: This is a positive integer offset specifying how far into the block from the origin the GP is set (note that it is subtracted from the origin for downward-growing blocks, but the input value should still be positive). This is used to indicate when the GP is being moved around within a large block of globals, because many processor architectures can access variables close to a base address more efficiently than ones farther away.
- const int offset_into_upward_block*: Same as previous, but for the upward block of the two blocks. If this value is negative, then it specifies an offset into the paired downward block instead.

Used by:

All TVM code, before or between functions alone, or within functions when used along with `HAL_USE_GP1` or `HAL_USE_GP2` macros that adjust the GP, may be automatically generated as necessary.

Described in:

Section 3.2.4

6.2 HAL Macros (or Functions)

This section summarizes the various macros that are defined as a part of the TVM-HAL. While most of these should be implemented using fairly simple macros that compile down to a few assembly-language instructions, some low-level compilers may actually execute some of these using function calls to more complex emulation routines that are encapsulated in a low-level code library that is not visible outside of the compiler. In general, they work the same either way, except when no stack frame is available to support function calls (for example, at the very beginning of an exception handler). In this case, the LLC may generate errors if it cannot generate a desired HAL call exclusively using one or more macros.

6.2.1 Load and Store Replacements for Pointer References

HAL_LOAD
HAL_LOAD_P
HAL_STORE
HAL_STORE_P

```
type HAL_LOAD_type(const int memory, type *pointer);
type HAL_LOAD_P_type(const int memory, HAL_phyPtr_type pointer);
HAL_STORE_type(const int memory, type *pointer, type value);
```

```
HAL_STORE_P_type(const int memory, HAL_phyPtr_type pointer, type value);
```

These macros perform a load or store operation, given a pointer (to a virtual address or physical address, depending upon the version) and a memory space. Their main enhancement over a basic pointer dereference operation in C is that they also specify a memory space. In general, these should compile down into a single machine instruction (load or store), but for some more remote or I/O-like memory spaces they may become much longer sequences of instructions. Low-level compiler designers should note that some arithmetic in the calculation of the *pointer* input may often be merged in with the load or store instruction's address calculation. Finally, there are several varieties of these macros, for various data types:

```
HAL_bool      1-bit logical (usually char or int, depending on speed)
    → Enumerated as: HAL_FALSE = 0 and HAL_TRUE = 1
HAL_boolX     2-bit ternary logical (usually char or int)
    → Enumerated as HAL_bool plus: HAL_DONT_CARE = 2
HAL_int8      8-bit integer (usually char)
HAL_int16     16-bit integer (usually short)
HAL_int32     32-bit integer (usually int or long)
HAL_int64     64-bit integer (usually long long)
HAL_int128    128-bit integer (used for vector types and such)
HAL_uint8     8-bit unsigned integer (usually unsigned char)
HAL_uint16    16-bit unsigned integer (usually unsigned short)
HAL_uint32    32-bit unsigned integer (usually unsigned int or long)
HAL_uint64    64-bit unsigned integer (usually unsigned long long)
HAL_uint128   128-bit unsigned integer (usually unsigned vector type)
HAL_float     32-bit IEEE floating-point (usually float)
HAL_double    64-bit IEEE floating-point (usually double)
HAL_longdouble 64-to-96-bit floating-point (as allowed by architecture)
HAL_ptr       Generic void* pointer (usually a 32-bit or 64-bit value)
```

All instances of *type* in the prototype above should be replaced with the desired HAL_type identifier, such as HAL_int8 HAL_LOAD_int8(const int memory, HAL_int8 *pointer). Note that the word appended to the HAL_LOAD or HAL_STORE omits the HAL part, since it would be redundant. The high-level compiler should choose the appropriate type for the data being loaded or stored, and convert the pointer dereference into the appropriate macro.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the variable being accessed.

*type *pointer*: The address of the variable being accessed (virtual address).

*HAL_phyPtr_type *pointer*: The address of the variable being accessed (physical address).

type value: The value to store.

Returns:

(*type*) The value of the variable being loaded (HAL_LOAD only).

Used by:

Virtual address versions: All TVM code, in functions; will almost always be automatically generated.

Physical address versions: Low-level OS code; manual placement only.

Described in:
Section 2.3

6.2.2 Exception and Machine Control Macros

This section summarizes the macros that are typically used by exception handlers to do tasks such as interrupt control and context switching in a relatively architecture-neutral way.

6.2.2.1 EXCEPTION HANDLER CONTROL MACROS

HAL_EXCEPTION_ENABLE **HAL_EXCEPTION_REENABLE**

```
void HAL_EXCEPTION_ENABLE(
    int entry_number,
    HAL_codePtr handler_pc,
    int privilege_level);
void HAL_EXCEPTION_REENABLE(int entry_number);
```

Enables an exception vector in the exception vector table for the processor that makes the call. Each processor in a multiprocessor system maintains its own table. The **ENABLE** form assigns an exception handler and escape privilege level before actually enabling the vector, while **REENABLE** turns the vector back on with its previously used handler and privilege level intact.

Inputs:

int entry_number: The entry in the current exception vector table to enable, from 0-255. Several of these numbers are enumerated for convenience in the following table:

Enumerated Name	Number
HAL_EXCEPTION_UNSUPPORTED	0
HAL_EXCEPTION_ILLEGAL_PRIVILEGED_INST	1
HAL_EXCEPTION_BREAKPOINT_INST	2
HAL_EXCEPTION_WATCHPOINT	3
HAL_EXCEPTION_INTEGER_ARITHMETIC	4
HAL_EXCEPTION_FP_ARITHMETIC	5
HAL_EXCEPTION_FP_UNAVAILABLE	6
HAL_EXCEPTION_FP_DENORMALIZED	7
HAL_EXCEPTION_TRAP	8
HAL_EXCEPTION_SYSCALL	9
HAL_EXCEPTION_PAGE_FAULT_DREAD	16
HAL_EXCEPTION_BUS_ERROR_DREAD	17
HAL_EXCEPTION_PROTECTION_FAULT_DREAD	18

Enumerated Name	Number
HAL_EXCEPTION_ADDRESS_ALIGNMENT_DREAD	19
HAL_EXCEPTION_CACHE_PARITY_DREAD	20
HAL_EXCEPTION_PAGE_FAULT_DWRITE	21
HAL_EXCEPTION_BUS_ERROR_DWRITE	22
HAL_EXCEPTION_PROTECTION_FAULT_DWRITE	23
HAL_EXCEPTION_ADDRESS_ALIGNMENT_DWRITE	24
HAL_EXCEPTION_CACHE_PARITY_DWRITE	25
HAL_EXCEPTION_PAGE_FAULT_I	26
HAL_EXCEPTION_BUS_ERROR_I	27
HAL_EXCEPTION_PROTECTION_FAULT_I	28
HAL_EXCEPTION_ADDRESS_ALIGNMENT_I	29
HAL_EXCEPTION_CACHE_PARITY_I	30
HAL_INTERRUPT_UNSUPPORTED	64
HAL_INTERRUPT_HARD_RESET	65
HAL_INTERRUPT_SOFT_RESET	66
HAL_INTERRUPT_NON_MASKABLE	67
HAL_INTERRUPT_PERFORMANCE	126
HAL_INTERRUPT_TIMER	127

HAL_codePtr handler_pc: A pointer to an exception-handler function to load into the selected exception vector.

int privilege_level: The privilege level that the system should be set at before it enters the handler_pc function. This is nominally either HAL_PRIVILEGED = 0 or HAL_USER = 1.

Used by:

OS startup and exception handling code.

Described in:

Section 3.1

HAL_EXCEPTION_CLEAR

```
void HAL_EXCEPTION_CLEAR(int entry_number);
```

Clears an exception vector in the master exception vector table so that the processor no longer responds to that exception or interrupt with a handler. If used improperly, this can hang up the machine if a critical exception occurs but cannot trigger an exception handler.

Input:

int entry_number: The entry in the current exception vector table to clear, from 0-255.

Used by:

OS startup and exception handling code.

Described in:

Section 3.1

HAL_EXCEPTION_USABLE

```
HAL_bool HAL_EXCEPTION_USABLE(int entry_number);
```

Tests to see whether or not one of the exceptions in the architecture-neutral exception vector table can actually be triggered on the underlying architecture. If this returns **FALSE**, then the underlying OS software does not need worry about scheduling this vector.

Input:

int entry_number: The entry in the current exception vector table to test, from 0-255.

Returns:

(*HAL_bool*) A Boolean indicating whether or not this exception vector is usable on the underlying architecture. A **FALSE** response indicates that either the interrupt hardware cannot send an interrupt when the condition occurs or that the condition has no meaning on the architecture.

Used by:

OS startup and exception handling code.

Described in:

Section 3.1

HAL_EXCEPTION_GET_ENABLED**HAL_EXCEPTION_GET_HANDLER****HAL_EXCEPTION_GET_PRIVILEGE**

```
HAL_bool HAL_EXCEPTION_GET_ENABLED(int entry_number);
HAL_codePtr HAL_EXCEPTION_GET_HANDLER(int entry_number);
int HAL_EXCEPTION_GET_PRIVILEGE(int entry_number);
```

These return the currently set values from the exception vector table for a given entry number.

Input:

int entry_number: The entry in the current exception vector table to test, from 0-255.

Returns:

(*HAL_bool*), (*HAL_codePtr*), (*int*) Each returns a value from the current exception table entry. Respectively, these are whether or not the vector is enabled (**HAL_TRUE** = enabled), the current pointer to the vector's exception handler (a **HAL_codePtr**), or its privilege level. The latter is nominally either **HAL_PRIVILEGED** = 0 or **HAL_USER** = 1.

Used by:

OS startup and exception handling code.

Described in:

Section 3.1

6.2.2.2 SYSTEM CALL MACROS

HAL_SYSCALL**HAL_TRAP**

```
void HAL_SYSCALL(int syscall_number);
void HAL_TRAP(
    int syscall_number,
    HAL_bool trap_expression);
```

These routines may be used by any code to trigger a system call (#9) or trap (#8) exception, respectively. However, they will generally be used only by user code to trigger a voluntary raising of the privilege level to kernel. The `HAL_SYSCALL` macro triggers an exception unconditionally when called, while the `HAL_TRAP` macro triggers an exception only when the `trap_expression` Boolean expression evaluates to `HAL_TRUE`.

Inputs:

int number: The system call or trap number to record (usually in a register) for the exception handler to recover with a subsequent `HAL_SYSCALL_NUMBER` macro.

HAL_bool trap_expression: An arbitrary expression that should be evaluated to determine whether or not to take a trap. On many architectures, part or all of this expression can be calculated by the trap instruction itself.

Used by:

Any code, but most often within user code I/O libraries such as `libc`.

Described in:

Section 3.1

HAL_SYSCALL_NUMBER

```
int HAL_SYSCALL_NUMBER();
```

Returns the system call number recorded by the `HAL_SYSCALL` or `HAL_TRAP` macro that was just invoked.

Returns:

(int) The last recorded trap or syscall number.

Used by:

Exception handlers #8 or #9.

Described in:

Section 3.1

6.2.2.3 CONTEXT MANAGEMENT MACROS

HAL_SAVE_STATE

HAL_RESTORE_STATE

HAL_INIT_STATE

```
void HAL_SAVE_STATE(
    const int memory,
    HAL_ArchitectureSaveBlock *saveTo);
void HAL_RESTORE_STATE(
    const int memory,
    HAL_ArchitectureSaveBlock *restoreFrom);
void HAL_INIT_STATE(
    const int memory,
    HAL_ArchitectureSaveBlock *toInit,
    void *initial_StackPointer,
    void *initial_GlobalPointer,
    void *initial_FramePointer,
    HAL_codePtr initial_ProgramCounter);
```

These routines are used by context management functions within the OS to juggle the register state associated with every active thread context in the system, each of which is stored in an architecture-specific `HAL_ArchitectureSaveBlock` structure. As each thread is created, `HAL_INIT_STATE` is used to initialize the block with basic values that are needed by the architecture—usually a stack pointer, global pointer, (often) a frame pointer, and the initial program counter associated with the thread. As each context is activated, its state is loaded into the active register file by the context switch code using `HAL_RESTORE_STATE` macro (except for the program counter, which is only invoked at the subsequent return-from-exception). Later, the context may be removed from the processor using `HAL_SAVE_STATE` (which saves the program counter from when the last exception was triggered). The cycle of these two functions continues over the life of the thread until it finally completes and the `HAL_ArchitectureSaveBlock` can be deleted. See Section 3.2.3 for an example of how all of this works together.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the *saveTo/restoreFrom/toInit* `HAL_ArchitectureSaveBlock` record.

*HAL_ArchitectureSaveBlock *saveTo/restoreFrom/toInit*: A pointer to an implementation-dependent `HAL_ArchitectureSaveBlock` record that contains the processor state to save/restore/initialize. Depending upon which `HAL_ArchitectureSaveBlock` is passed into the call, different contexts may be saved or restored.

*void *initial_StackPointer*: The initial value of the stack pointer register to place in an empty `HAL_ArchitectureSaveBlock` record. If omitted, the stack pointer can be initialized later by the context itself.

*void *initial_GlobalPointer*: The initial value of the global pointer register to place in an empty `HAL_ArchitectureSaveBlock` record. If omitted, the global pointer can be initialized later by the context itself.

*void *initial_FramePointer:* The initial value of the frame pointer register to place in an empty `HAL_ArchitectureSaveBlock` record. This may not be used on all architectures.

*void *initial_ProgramCounter:* The initial value of the program counter to place in an empty `HAL_ArchitectureSaveBlock` record. The new context will start executing at this PC only after it is 1) loaded into a processor using `HAL_RESTORE_STATE` and 2) the processor returns from an exception using one of the `HAL_EX_RETURN` macros.

Used by:

OS context switching code.

Described in:

Section 3.2.3

HAL_GET_SP**HAL_GET_FP****HAL_GET_GP**

```
void *HAL_GET_SP();
void *HAL_GET_FP();
void *HAL_GET_GP();
```

These macros return the current value of the stack, frame, or global pointers.

Returns:

(*void **) The current value of the stack, frame, or global pointers.

Used by:

OS interrupt handling or context switching code.

Described in:

Section 3.2.4

HAL_SET_SP**HAL_SET_FP****HAL_SET_GP**

```
void HAL_SET_SP(void *new_stack_pointer);
void HAL_SET_FP(void *new_frame_pointer);
void HAL_SET_GP(void *new_global_pointer);
```

These macros save a new value into the current stack, frame, or global pointers.

Modifications to the stack frame and/or global pointers should only occur when use of those pointers is prohibited to the compiler using a `HAL_XP_OFF` directive.

Input:

*void *new_stack_pointer:* The new value of the stack pointer.

*void *new_global_pointer:* The new value of the global pointer.

*void *new_frame_pointer:* The new value of the frame pointer.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:
Section 3.2.4

HAL_ASB_GET_SP
HAL_ASB_GET_FP
HAL_ASB_GET_GP

```
void *HAL_ASB_GET_SP(
    const int memory,
    HAL_ArchitectureSaveBlock *block);
void *HAL_ASB_GET_FP(
    const int memory,
    HAL_ArchitectureSaveBlock *block);
void *HAL_ASB_GET_GP(
    const int memory,
    HAL_ArchitectureSaveBlock *block);
```

These macros return saved values of the stack, frame, or global pointers.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the `HAL_ArchitectureSaveBlock` record.
*HAL_ArchitectureSaveBlock *block*: A pointer to an implementation-dependent `HAL_ArchitectureSaveBlock` record with the processor state to read from.

Returns:

(*void **) The current value of the stack, frame, or global pointers.

Used by:

OS interrupt handling or context switching code.

Described in:
Section 3.2.4

HAL_ASB_SET_SP
HAL_ASB_SET_FP
HAL_ASB_SET_GP

```
void HAL_ASB_SET_SP(
    const int memory,
    HAL_ArchitectureSaveBlock *block,
    void *new_stack_pointer);
void HAL_ASB_SET_FP(
    const int memory,
    HAL_ArchitectureSaveBlock *block,
    void *new_frame_pointer);
void HAL_ASB_SET_GP(
    const int memory,
    HAL_ArchitectureSaveBlock *block,
    void *new_global_pointer);
```

These macros save a new value into the stack, frame, or global pointers for a previously saved thread. Modifications to the stack frame and/or global pointers should only occur when use of those pointers is prohibited to the compiler using a `HAL_xP_OFF` directive.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the `HAL_ArchitectureSaveBlock` record.
*HAL_ArchitectureSaveBlock *block*: A pointer to an implementation-dependent `HAL_ArchitectureSaveBlock` record with the processor state to write to.
*void *new_stack_pointer*: The new value of the stack pointer.
*void *new_global_pointer*: The new value of the global pointer.
*void *new_frame_pointer*: The new value of the frame pointer.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.4

HAL_INTERRUPTS_ARE_ENABLED

```
HAL_bool HAL_INTERRUPTS_ARE_ENABLED();
```

Tests to see whether or not interrupts are currently enabled.

Returns:

(*HAL_bool*) TRUE if interrupts are currently enabled, FALSE otherwise.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.5

HAL_INTERRUPTS_ON
HAL_INTERRUPTS_OFF

```
void HAL_INTERRUPTS_ON();  
void HAL_INTERRUPTS_OFF();
```

Turns interrupts on or off for this processor.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.5

HAL_GET_PRIVILEGE_LEVEL

```
int HAL_GET_PRIVILEGE_LEVEL();
```

Gets the current processor privilege level.

Returns:

(*int*) The current privilege level, which is nominally either HAL_PRIVILEGED = 0 or HAL_USER = 1.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.5

HAL_SET_PRIVILEGE_LEVEL

```
void HAL_SET_PRIVILEGE_LEVEL(int new_privilege_level);
```

Sets a new processor privilege level.

Input:

int privilege_level: The privilege level that the system should switch to. This is nominally either HAL_PRIVILEGED = 0 or HAL_USER = 1.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.5

HAL_GET_INST_TRANSLATION**HAL_GET_DATA_TRANSLATION**

```
HAL_bool HAL_GET_INST_TRANSLATION();  
HAL_bool HAL_GET_DATA_TRANSLATION();
```

Returns whether or not address translation (*i.e.* processor TLBs) are enabled for instruction fetches or data accesses, respectively. On processors with address translation permanently enabled (albeit generally only for a portion of the virtual address space), this always returns HAL_TRUE.

Returns:

(*HAL_bool*) Returns HAL_TRUE if translation is currently enabled.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.5

HAL_GET_INST_TRANSLATION**HAL_SET_DATA_TRANSLATION**

```
void HAL_SET_INST_TRANSLATION(HAL_bool translate);  
void HAL_SET_DATA_TRANSLATION(HAL_bool translate);
```

Activates or deactivates address translation (*i.e.* processor TLBs) for instruction fetches or data accesses, respectively. On processors with address translation permanently enabled (albeit generally only for a portion of the virtual address space), these are ignored.

Input:

HAL_bool translate: Activates translation on an input of HAL_TRUE.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.5

HAL_MASK_POINT_AT

```
int HAL_MASK_POINT_AT();
```

Returns the current external interrupt mask point, the number of the highest enabled external interrupt from among the set of maskable interrupts (96-127, see Section 3.1). This is often used by a context switch handler to save the state of the processor as it was previously running, but may also be used to check on the automatically-adjusted mask point after an interrupt.

Returns:

(*int*) The value of the highest unmasked external interrupt, in the range of 96-127 (the external/maskable interrupts and timer interrupt). Anything above this value will be masked.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.1 and 3.2.5

HAL_MASK_ABOVE

```
void HAL_MASK_ABOVE(int entry_number);
```

Sets the current external interrupt mask point, the number of the highest enabled external interrupt from among the set of maskable interrupts (numbers 96-127, see Section 3.1). This can be used to adjust the interrupt masking level after an interrupt is handled, or to set it for a new context on switches.

Input:

int entry_number: The highest external interrupt, in the range of 96-127 (the external/maskable interrupts and timer interrupt), that should be unmasked after the call. Afterwards, all higher-value entry numbers will be masked off.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.1 and 3.2.5

HAL_EX_RETURN
HAL_EX_RETURNX

```
void HAL_EX_RETURN();  
void HAL_EX_RETURNX(  
    HAL_codePtr PC,  
    HAL_bool interrupts_enabled,  
    int return_privilege_level,  
    HAL_bool inst_translation_on,  
    HAL_bool data_translation_on);
```

Return to a previous context after an exception or interrupt has been handled. The basic form is used for most exception handling, when the program can simply return to the location of the last exception within a thread or to the instruction just prior to the interrupt. The more complex X form is used when the context-switching mechanism must make significant modifications of the thread's core state before returning.

Inputs:

HAL_codePtr PC: The address (new program counter) to return to.

HAL_bool interrupts_enabled: Whether or not interrupts should be enabled after returning.

int return_privilege_level: The privilege level that the system should switch to upon returning. This is nominally either `HAL_PRIVILEGED = 0` or `HAL_USER = 1`.

HAL_bool inst_translation_on: This controls whether or not address translation is enabled for instructions after returning: `HAL_TRUE` enables, and `HAL_FALSE` disables.

HAL_bool data_translation_on: This controls whether or not address translation is enabled for data after returning: `HAL_TRUE` enables, and `HAL_FALSE` disables.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.5

HAL_WAS_PC

```
HAL_codePtr HAL_WAS_PC();
```

Returns the program counter where the processor was executing when it encountered the most recent exception or interrupt. The PC returned will be of the instruction that caused an exception or the next instruction to be executed in an interrupted context. Note that in architectures with branch delay slots this will always point at the branch, even if the exception occurred in the "delay slot" instruction.

Returns:

(*HAL_codePtr*) The program counter (PC) of the instruction that caused an exception or the next instruction to be executed in an interrupted context. Note that in architectures with branch delay slots this will always point at the branch, even if the exception occurred in the "delay slot" instruction.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.5

HAL_INTERRUPTS_WERE_ENABLED

```
HAL_bool HAL_INTERRUPTS_WERE_ENABLED();
```

Returns the external interrupt enable status that was active when the processor encountered the most recent exception or interrupt.

Returns:

(*HAL_bool*) TRUE if interrupts were enabled in the previously excepted/interrupted context, or FALSE otherwise.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.5

HAL_WAS_PRIVILEGE_LEVEL

```
int HAL_WAS_PRIVILEGE_LEVEL();
```

Returns the privilege level that was active when the processor encountered the most recent exception or interrupt.

Returns:

(*int*) The privilege level of the previously excepted/interrupted context, which is nominally either HAL_PRIVILEGED = 0 or HAL_USER = 1.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.5

HAL_WAS_INST_TRANSLATION_ON**HAL_WAS_DATA_TRANSLATION_ON**

```
HAL_bool HAL_WAS_INST_TRANSLATION_ON();
```

```
HAL_bool HAL_WAS_DATA_TRANSLATION_ON();
```

Returns the status of whether or not address translation was enabled for instructions or data, respectively, just prior to the last interrupt or exception.

Returns:

(*HAL_bool*) Whether or not address translation was enabled: HAL_TRUE is returned if address translation was enabled, and HAL_FALSE if disabled.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:
Section 3.2.5

HAL_SET_EX_PC

```
void HAL_WAS_PC(HAL_codePtr new_PC);
```

Sets the PC at which the processor will restart execution after returning from the current exception handler.

Inputs:
HAL_codePtr PC: The address (new program counter) to return to.

Used by:
OS startup, interrupt handling, or context switching code.

Described in:
Section 3.2.5

HAL_SET_EX_INTERRUPTS_ON
HAL_SET_EX_INTERRUPTS_OFF

```
void HAL_SET_EX_INTERRUPTS_ON();  
void HAL_SET_EX_INTERRUPTS_ON();
```

Turn interrupts on or off after the processor returns from the next exception.

Used by:
OS startup, interrupt handling, or context switching code.

Described in:
Section 3.2.5

HAL_SET_EX_PRIVILEGE_LEVEL

```
void HAL_WAS_PRIVILEGE_LEVEL(int new_level);
```

Sets the privilege level that will be active after the processor returns from this handler.

Inputs:
int return_privilege_level: The privilege level that the system should switch to upon returning. This is nominally either HAL_PRIVILEGED = 0 or HAL_USER = 1.

Used by:
OS startup, interrupt handling, or context switching code.

Described in:
Section 3.2.5

HAL_SET_EX_INST_TRANSLATION
HAL_SET_EX_DATA_TRANSLATION

```
void HAL_SET_EX_INST_TRANSLATION(HAL_bool inst_translation_on);  
void HAL_SET_EX_DATA_TRANSLATION(HAL_bool data_translation_on);
```

Sets the status of whether or not address translation will be enabled for instructions or data, respectively, after the processor returns from this exception.

Inputs:

HAL_bool inst_translation_on: This controls whether or not address translation is enabled for instructions after returning: `HAL_TRUE` enables, and `HAL_FALSE` disables.

HAL_bool data_translation_on: This controls whether or not address translation is enabled for data after returning: `HAL_TRUE` enables, and `HAL_FALSE` disables.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.5

HAL_ASB_WAS_PC

```
HAL_codePtr HAL_ASB_WAS_PC(const int memory,  
    HAL_ArchitectureSaveBlock *block);
```

Returns the program counter where the processor was executing just prior to the last interrupt or exception before the thread state was saved to memory. The PC returned will be of the instruction that caused an exception or the next instruction to be executed in an interrupted context. Note that in architectures with branch delay slots this will always point at the branch, even if the exception occurred in the “delay slot” instruction.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the `HAL_ArchitectureSaveBlock` record.

*HAL_ArchitectureSaveBlock *block:* A pointer to an implementation-dependent `HAL_ArchitectureSaveBlock` record with the processor state to read from.

Returns:

(*HAL_codePtr*) The program counter (PC) of the instruction that caused an exception or the next instruction to be executed in an interrupted context. Note that in architectures with branch delay slots this will always point at the branch, even if the exception occurred in the “delay slot” instruction.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.5

HAL_ASB_INTERRUPTS_WERE_ENABLED

```
HAL_bool HAL_ASB_INTERRUPTS_WERE_ENABLED(const int memory,  
    HAL_ArchitectureSaveBlock *block);
```

Returns the external interrupt enable status that was active just prior to the last interrupt or exception before the thread state was saved to memory.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the `HAL_ArchitectureSaveBlock` record.
*HAL_ArchitectureSaveBlock *block*: A pointer to an implementation-dependent `HAL_ArchitectureSaveBlock` record with the processor state to read from.

Returns:

(*HAL_bool*) TRUE if interrupts were enabled in the previously excepted/interrupted context, or FALSE otherwise.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.5

HAL_ASB_WAS_PRIVILEGE_LEVEL

```
int HAL_ASB_WAS_PRIVILEGE_LEVEL(const int memory,
    HAL_ArchitectureSaveBlock *block);
```

Returns the privilege level that was active just prior to the last interrupt or exception before the thread state was saved to memory.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the `HAL_ArchitectureSaveBlock` record.
*HAL_ArchitectureSaveBlock *block*: A pointer to an implementation-dependent `HAL_ArchitectureSaveBlock` record with the processor state to read from.

Returns:

(*int*) The privilege level of the previously excepted/interrupted context, which is nominally either `HAL_PRIVILEGED = 0` or `HAL_USER = 1`.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.5

HAL_ASB_WAS_INST_TRANSLATION_ON
HAL_ASB_WAS_DATA_TRANSLATION_ON

```
HAL_bool HAL_ASB_WAS_INST_TRANSLATION_ON(const int memory,
    HAL_ArchitectureSaveBlock *block);
HAL_bool HAL_ASB_WAS_DATA_TRANSLATION_ON(const int memory,
    HAL_ArchitectureSaveBlock *block);
```

Returns the status of whether or not address translation was enabled for instructions or data, respectively, just prior to the last interrupt or exception before the thread state was saved to memory.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the `HAL_ArchitectureSaveBlock` record.
*HAL_ArchitectureSaveBlock *block*: A pointer to an implementation-dependent `HAL_ArchitectureSaveBlock` record with the processor state to read from.

Returns:

(*HAL_bool*) Whether or not address translation was enabled: `HAL_TRUE` is returned if address translation was enabled, and `HAL_FALSE` if disabled.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.5

HAL_ASB_SET_EX_PC

```
void HAL_ASB_WAS_PC(const int memory,  
    HAL_ArchitectureSaveBlock *block, HAL_codePtr new_PC);
```

Sets the PC at which the processor will restart execution after the thread state being examined is restored and the processor returns from an exception.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the `HAL_ArchitectureSaveBlock` record.
*HAL_ArchitectureSaveBlock *block*: A pointer to an implementation-dependent `HAL_ArchitectureSaveBlock` record with the processor state to write to.
HAL_codePtr PC: The address (new program counter) to return to.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.5

HAL_ASB_SET_EX_INTERRUPTS_ON
HAL_ASB_SET_EX_INTERRUPTS_OFF

```
void HAL_ASB_SET_EX_INTERRUPTS_ON(const int memory,  
    HAL_ArchitectureSaveBlock *block);  
void HAL_ASB_SET_EX_INTERRUPTS_OFF(const int memory,  
    HAL_ArchitectureSaveBlock *block);
```

Turn interrupts on or off after the thread state being examined is restored and the processor returns from an exception.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the `HAL_ArchitectureSaveBlock` record.
*HAL_ArchitectureSaveBlock *block*: A pointer to an implementation-dependent `HAL_ArchitectureSaveBlock` record with the processor state to write to.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.5

HAL_ASB_SET_EX_PRIVILEGE_LEVEL

```
void HAL_ASB_WAS_PRIVILEGE_LEVEL(const int memory,  
    HAL_ArchitectureSaveBlock *block, int new_level);
```

Sets the privilege level that will be active after the thread state being examined is restored and the processor returns from an exception.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the HAL_ArchitectureSaveBlock record.

*HAL_ArchitectureSaveBlock *block*: A pointer to an implementation-dependent HAL_ArchitectureSaveBlock record with the processor state to write to.

int return_privilege_level: The privilege level that the system should switch to upon returning. This is nominally either HAL_PRIVILEGED = 0 or HAL_USER = 1.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.5

HAL_ASB_SET_EX_INST_TRANSLATION**HAL_ASB_SET_EX_DATA_TRANSLATION**

```
void HAL_ASB_SET_EX_INST_TRANSLATION(const int memory,  
    HAL_ArchitectureSaveBlock *block, HAL_bool inst_translation_on);  
void HAL_ASB_SET_EX_DATA_TRANSLATION(const int memory,  
    HAL_ArchitectureSaveBlock *block, HAL_bool data_translation_on);
```

Sets the status of whether or not address translation will be enabled for instructions or data, respectively, after the thread state being examined is restored and the processor returns from an exception.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the HAL_ArchitectureSaveBlock record.

*HAL_ArchitectureSaveBlock *block*: A pointer to an implementation-dependent HAL_ArchitectureSaveBlock record with the processor state to write to.

HAL_bool inst_translation_on: This controls whether or not address translation is enabled for instructions after returning: HAL_TRUE enables, and HAL_FALSE disables.

HAL_bool data_translation_on: This controls whether or not address translation is enabled for data after returning: HAL_TRUE enables, and HAL_FALSE disables.

Used by:

OS startup, interrupt handling, or context switching code.

Described in:

Section 3.2.5

6.2.2.4 TIMER INTERRUPT CONTROL

HAL_TIMER_OFF**HAL_TIMER_ON**

```
void HAL_TIMER_OFF();
```

```
void HAL_TIMER_ON();
```

Turn timer interrupts (usually used for preemptive context switching) on and off. Usually they are turned on at startup and never disabled.

Used by:

OS startup or context switching code.

Described in:

Section 3.2.5

HAL_TIMER_FREQUENCY_SET

```
void HAL_TIMER_FREQUENCY_SET(HAL_uint64 frequency_of_interrupts_in_hertz);
```

Attempts to set the timer interrupt frequency to a new value. The value is set to 60 Hz by default, but in many architectures it may be adjusted. The macro will set the timer interrupt to the closest frequency supported by the architecture that is *greater than* the requested frequency. If this is not possible, the frequency will remain unchanged.

Input:

int frequency_of_interrupts_in_hertz: The new clock frequency to set timer interrupts to.

Used by:

OS startup or context switching code.

Described in:

Section 3.2.5

HAL_TIMER_FREQUENCY_GET

```
HAL_uint64 HAL_TIMER_FREQUENCY_GET();
```

Returns the current timer interrupt frequency, in Hz.

Returns:

(int) The current timer interrupt frequency, in Hz.

Used by:

OS startup or context switching code.

Described in:

Section 3.2.5

HAL_TEST_TIMER_SWITCH**HAL_TEST_TIMER_SET**

```
HAL_bool HAL_TEST_TIMER_SWITCH();
HAL_bool HAL_TEST_TIMER_SET();
```

Test to see whether or not the timer interrupt can be enabled/disabled or have its frequency set from this processor.

Returns:

(*HAL_bool*) Whether or not the timer ON/OFF macros or SET macro will work, respectively.

Used by:

OS startup or context switching code.

Described in:

Section 3.2.5

6.2.2.5 SVM INITIALIZATION MACROS

HAL_TEST_FOR_SVM

```
HAL_bool HAL_TEST_FOR_SVM();
```

Checks to see if the current processor can switch to streaming mode at all, or if it is only capable of running threaded code.

Returns:

(*HAL_bool*) TRUE if this processor can be switched to streaming mode, FALSE otherwise.

Used by:

Any TVM code.

Described in:

Section 3.3.3

HAL_SWITCH_TO_SVM

```
void HAL_SWITCH_TO_SVM(int master_processor);
```

Ends threaded execution as it switches the processor to streaming mode, as a slave worker processor dependent upon the indicated master processor. There is no coming back unless the processor is interrupted. At that point, it reverts to threaded execution to handle the interrupt and may choose not to go back to SVM code if necessary.

Input:

int master_processor: The TVM processor that will issue stream operations to this processor after it switches to streaming mode. In other words, the processor that this streaming processor will listen to.

Used by:

Any TVM code.

Described in:
Section 3.3.3

6.2.2.6 MULTITHREADING CONTROL MACROS

HAL_NUM_CONTEXTS

```
int HAL_NUM_CONTEXTS();
```

Returns the number of hardware contexts supported by this processor, and is generally only for use during startup.

Returns:
(*int*) The number of hardware contexts supported by this processor.

Used by:
OS startup or context switching code.

Described in:
Section 3.4

HAL_INIT_CONTEXT

```
void HAL_INIT_CONTEXT(  
    int context_number,  
    HAL_codePtr initial_PC,  
    void *initial_StackPointer,  
    void *initial_GlobalPointer,  
    void *initial_FramePointer,  
    HAL_bool interrupts_enabled,  
    int privilege_level,  
    HAL_bool inst_translation_on,  
    HAL_bool data_translation_on);
```

Much like HAL_INIT_STATE, this routine loads up a hardware context with the information necessary to get it started. As the state is being loaded directly into a hardware context, and cannot be changed before execution commences, the core state information set using HAL_EX_RETURNX is also set here. When a new context is spawned, this routine is used to load the infant hardware context. When initialization is complete, the context does not compete for processor resources until the first HAL_ACTIVATE.

Inputs:
int context_number: The number of the hardware context that should be initialized.
HAL_codePtr initial_PC: A pointer to the first instruction that should be executed by the new context when it is first activated (its initial PC, or program counter).
*void *initial_StackPointer*: The initial value of the stack pointer register to place in the new hardware context, or nothing if the context will set the value itself.
*void *initial_GlobalPointer*: The initial value of the global pointer register to place in the new hardware context, or nothing if the context will set the value itself.
*void *initial_FramePointer*: The initial value of the frame pointer register to place in the new hardware context, or nothing if the context will set the value itself.

HAL_bool interrupts_enabled: Whether or not interrupts should be enabled when the new hardware context is executing—can it be interrupted?

int privilege_level: The privilege level that the system should be at when it starts executing the new context. This is nominally either `HAL_PRIVILEGED = 0` or `HAL_USER = 1`.

HAL_bool inst_translation_on: This controls whether or not address translation is enabled for instructions after returning: `HAL_TRUE` enables, and `HAL_FALSE` disables.

HAL_bool data_translation_on: This controls whether or not address translation is enabled for data after returning: `HAL_TRUE` enables, and `HAL_FALSE` disables.

Used by:

OS startup or context switching code.

Described in:

Section 3.4

HAL_ACTIVATE

```
void HAL_ACTIVATE(int context_number);
```

Frees a hardware context that had been deactivated (either because it is new or was blocked) to execute code and compete for resources within the processor core.

Input:

int context_number: The number of the hardware context that should be activated.

Used by:

OS startup or context switching code.

Described in:

Section 3.4

HAL_DEACTIVATE_OTHER HAL_DEACTIVATE

```
void HAL_DEACTIVATE_OTHER(int context_number);  
void HAL_DEACTIVATE();
```

Deactivates another hardware context (or the current context, in the simple version), if the context needs to be stopped (killed) or stalled temporarily (such as while waiting for I/O). After being deactivated, the context is still present within the processor, but does not execute any instructions. It may be replaced with a different context at this point.

Input:

int context_number: The number of the hardware context that should be deactivated.

Used by:

OS startup or context switching code.

Described in:

Section 3.4

**HAL_SWITCH_TO
HAL_SWITCH**

```
void HAL_SWITCH_TO(int context_number);  
void HAL_SWITCH();
```

Cause the processor to switch to a new context, when the currently running processor wants to voluntarily give up processor resources. A particular other context can be specified, or the choice of “next” can be left to the OS or run-time system, with the simpler form above. Note that these macros will only have an effect in a fairly coarse-grained multithreading environment where the program is mostly running instructions from one thread at a time. In a simultaneously multithreaded (SMT) system, this call will have little impact, since all contexts already issue instructions simultaneously.

Input:

int context_number: The number of the hardware context that the processor should switch to next, if the context has a scheduling preference.

Used by:

OS startup or context switching code.

Described in:

Section 3.4

HAL_THREAD_ACTIVE

```
HAL_bool HAL_THREAD_ACTIVE(int context_number);
```

Tests to see whether or not a context is active, *i.e.*, containing a thread context that is capable of execution. It can be used to determine if other contexts on the processor are blocked or simply haven't been used yet.

Input:

int context_number: The number of the hardware context that should be checked to see whether or not it is active.

Returns:

(*HAL_bool*) TRUE if the thread is currently active, FALSE if not.

Used by:

OS startup or context switching code.

Described in:

Section 3.4

HAL_THREAD_PRIORITY_GET

```
int HAL_THREAD_PRIORITY_GET(int context_number);
```

Gets the current hardware thread context priority, for systems that support a priority mechanism to favor scheduling of one thread context over another. If there is no support for priority, then all calls to this routine will return 0.

Input:

int context_number: The number of the hardware context whose priority is being checked.

Returns:

(*int*) An integer representing the current scheduling priority, from 0=low to 255=high.

Used by:

OS startup or context switching code.

Described in:

Section 3.4

HAL_THREAD_PRIORITY_SET

```
void HAL_THREAD_PRIORITY_SET(int context_number, int priority);
```

Sets the current hardware thread context priority in systems that support a priority scheme. Priorities are limited to 8-bit unsigned values, from 0-255. The higher a number in comparison with other threads, the more cycles that thread will tend to get when scheduled on a processor. On architectures that do not support thread priority, this call has no effect.

Inputs:

int context_number: The number of the hardware context whose priority is being set.

int priority: An integer representing the desired scheduling priority for this context, from 0=low to 255=high.

Used by:

OS startup or context switching code.

Described in:

Section 3.4

HAL_INTERRUPTS_TO**HAL_TIMER_TO**

```
void HAL_INTERRUPTS_TO(int context_number);
```

```
void HAL_TIMER_TO(int context_number);
```

Select contexts within a processor that are the designated external interrupt catchers. When external interrupts arrive, they interrupt the selected context (instead of one at random, or always context #0). Not all architectures will support this feature, so one should check first using the HAL_TEST_INTERRUPTS_TO and HAL_TEST_TIMER_TO macros.

Inputs:

int context_number: The number of the hardware context that should receive all external interrupts.

Used by:

OS startup code.

Described in:

Section 3.4

HAL_SEND_INTERRUPT_TO

```
void HAL_SEND_INTERRUPT_TO(  
    int context_number,  
    int interrupt_vector);
```

Sends an interrupt from one context within a processor to another, using an arbitrary interrupt vector from the interrupt vector table. This can be used to cause related contexts to sync up at appropriate times. The `HAL_TEST_SEND_INTERRUPT_TO` macro should be used beforehand, however, to verify that that this will actually work.

Inputs:

int context_number: The number of the hardware context that should receive the interrupt.

int interrupt_vector: The number of the exception vector that the receiving processor should use to select an interrupt handler.

Used by:

Any TVM code, but usually OS.

Described in:

Section 3.4

HAL_TEST_INTERRUPTS_TO**HAL_TEST_TIMER_TO****HAL_TEST_SEND_INTERRUPT_TO**

```
HAL_bool HAL_TEST_INTERRUPTS_TO(int context_number);  
HAL_bool HAL_TEST_TIMER_TO(int context_number);  
HAL_bool HAL_TEST_SEND_INTERRUPT_TO(int context_number);
```

Test to see whether or not the underlying architecture supports direction of interrupts to another process. The first routine checks to see if general external interrupts can be redirected, the second checks to see if timer interrupts can be redirected, and the third checks to see if this context can direct an interrupt to the target context using a `HAL_SEND_INTERRUPT_TO` operation.

Input:

int context_number: The number of the hardware context being tested.

Returns:

(*HAL_bool*) TRUE if the interrupt reassignment or communication can be done, FALSE otherwise.

Used by:

Any TVM code, but usually OS.

Described in:

Section 3.4

HAL_TEST_TIMER_SEPARATE

```
HAL_bool HAL_TEST_TIMER_SEPARATE();
```

Checks to see if the timer interrupt can be controlled separately from the other external interrupts using `HAL_TIMER_TO`, or if it must always be clumped in with the rest of the external interrupts.

Returns:

(*HAL_bool*) TRUE if timer interrupts can be controlled separately from other external interrupts, FALSE otherwise

Used by:

Any TVM code, but usually OS.

Described in:

Section 3.4

6.2.3 Memory Control Macros

This section summarizes the macros that are used by explicit startup/morphing code to modify the segmented memory space used by TVM-HAL code.

6.2.3.1 METADATA ACCESS MACROS

HAL_ROOT_OBJ

```
int HAL_ROOT_OBJ();
```

Returns the object identifier of the single “root” object in the system, which contains basic, system-wide stats and the identifier of the root “level” object.

Returns:

(*int*) The root object identifier.

Used by:

OS startup/morphing code.

Described in:

Section 4.2.1

HAL_GET_OBJ_CHARACTERISTIC

```
HAL_uint64 HAL_GET_OBJ_CHARACTERISTIC(  
    int object_identifier,  
    int characteristic_num,  
    int array_offset);
```

This is the basic function that architecture-neutral startup code can use to interrogate the TVM-HAL and determine key system parameters. Using this call, startup code can determine what the capabilities of the physical memory layout on this system are in a very precise way. Facts like the size and physical location of memory blocks, and the ability to use virtual memory to map flexible memory spaces onto the physical memory are all described by various calls to this function.

Inputs:

int object_identifier: The object identifier of the object to be queried. The interpretation of this field is system-dependent.

int characteristic_num: The characteristic (from the lists in Section 4.1.1 or 4.2.1) that is being queried from the selected object. These lists are enumerated below for ease of reference in typical code:

HAL Root Object Characteristics:

Enumerated Name	Description
HAL_MM_ROOT_ADDRESS_SIZE	int: Bit size of addresses
HAL_MM_ROOT_WORD_SIZE	int: Bit size of typical “word”
HAL_MM_ROOT_BIG_ENDIAN	HAL_bool: Big endian
HAL_MM_ROOT_CHAR_SIGNED	HAL_bool: Char type signed ?
HAL_MM_ROOT_SIZE_CHAR	int: Bit size of C char type
HAL_MM_ROOT_SIZE_SHORT	int: Bit size of C short type
HAL_MM_ROOT_SIZE_INT	int: Bit size of C int type
HAL_MM_ROOT_SIZE_LONG	int: Bit size of C long type
HAL_MM_ROOT_SIZE_LONGLONG	int: Bit size of C longlong type
HAL_MM_ROOT_SIZE_FLOAT	int: Bit size of C float type (32)
HAL_MM_ROOT_SIZE_DOUBLE	int: Bit size of C double type (64)
HAL_MM_ROOT_SIZE_LONGDOUBLE	int: Bit size of C longdouble type
HAL_MM_ROOT_ALIGN_CHAR	int: Bit alignment of C char type
HAL_MM_ROOT_ALIGN_SHORT	int: Bit alignment of C short type
HAL_MM_ROOT_ALIGN_INT	int: Bit alignment of C int type
HAL_MM_ROOT_ALIGN_LONG	int: Bit alignment of C long type
HAL_MM_ROOT_ALIGN_LONGLONG	int: Bit alignment of C longlong type
HAL_MM_ROOT_ALIGN_FLOAT	int: Bit alignment of C float type (32)
HAL_MM_ROOT_ALIGN_DOUBLE	int: Bit alignment of C doubles (64)
HAL_MM_ROOT_ALIGN_LONGDOUBLE	int: Bit alignment of C longdoubles
HAL_MM_ROOT_LEVEL_OBJ	Root level object number

HAL Processor Object Characteristics:

Enumerated Name	Description
HAL_MM_PROC_IDENTITY	int: Sequential number for this processor
HAL_MM_PROC_NUM_MASTERS	int: Number of master processors
HAL_MM_PROC_MASTERS	Object identifiers of processors that can be masters to this one
HAL_MM_PROC_HAS_TLB	Bool: Does processor have a TLB?
HAL_MM_PROC_NUM_PRESET_SEGMENTS	int: Number of preset segments
HAL_MM_PROC_PRESET_SEGMENTS	Segment numbers of preset segs.
HAL_MM_PROC_NUM_CONTEXTS	int: Number of contexts supported
HAL_MM_PROC_PERF_COUNTERS	int: Number of performance counters

HAL Memory Object Characteristics:

Enumerated Name	Description
HAL_MM_MEM_DEBUG_ADDRESS	HAL_PhyPtr: Base “debug” address
HAL_MM_MEM_PHYSICAL_ADDRESS	HAL_PhyPtr: Software-visible physical addr.
HAL_MM_MEM_OPTIMIZATION_NUM	int: “memory” code # for compilers
HAL_MM_MEM_SIZE	HAL_uint64: Memory size
HAL_MM_MEM_SHIFTABLE	HAL_bool: Can segment be address-shifted?
HAL_MM_MEM_BOUNDBLE	HAL_bool: Can segment be bounded?
HAL_MM_MEM_CAPABILITIES	int: Enumerated memory abilities
HAL_MM_MEM_CACHE_BELOW_COUNT	int: Number of “cache below” objects
HAL_MM_MEM_CACHE_BELOW	Obj. #s: Processors or other memories that can use this memory as a higher-level cache
HAL_MM_MEM_CACHE_ABOVE_COUNT	int: Number of “cache above” memories
HAL_MM_MEM_CACHE_ABOVE	Obj. #s: Other memories that can use this memory as a lower-level cache
HAL_MM_MEM_CACHE_LINESIZE_COUNT	int: Number of possible line sizes
HAL_MM_MEM_CACHE_LINESIZES	ints: Possible line sizes
HAL_MM_MEM_CACHE_GANG_COUNT	int: Number of gang-capable memories
HAL_MM_MEM_CACHE_GANGS	Obj. #s: Other memories that can join with this into a single cache
HAL_MM_MEM_CACHEABLE	bool: Can this memory be cached by others?
HAL_MM_MEM_PROTECTABLE	int: Enumerated protection bits
HAL_MM_MEM_MAPPABLE	bool: PMMU mechanism works?
HAL_MM_MEM_PAGESIZE_COUNT	int: Number of possible page sizes
HAL_MM_MEM_PAGESIZES	ints: Possible page sizes
HAL_MM_MEM_NUM_ACCESSIBLE	int: Number of processors that can access
HAL_MM_MEM_ACCESSIBLE	Obj. #s: Processors that can LD/ST here, which may be none if it’s cache-only
HAL_MM_MEM_CACHE_COHERENT	bool: Memory can be kept coherent w/ others
HAL_MM_MEM_INTERLEAVE_COUNT	int: Number of legal interleave factors
HAL_MM_MEM_INTERLEAVES	ints: Possible memory interleave factors

HAL Link Object Characteristics:

Enumerated Name	Description
HAL_MM_LINK_SENDER	Object # of “sender” end of link
HAL_MM_LINK_RECEIVER	Obj. # of “receiver” end of link
HAL_MM_LINK_BIDIRECTIONAL	Boolean: Bidirectional link?

int array_offset: The offset of the element that is being read within array-based characteristics, such as the list of sizes of memory pages that a memory bank supports. This should be set to 0 for parameters that are not array-based.

Returns:

(*HAL_int64*) Requested characteristic for the selected object, cast to a `HAL_int64` type variable, which will always be large enough to hold the requested value. Depending upon the requested value, this should usually be cast back to another type. The two memory characteristics that are “enumerated” integers are actually bit vectors of flags created by logically OR-ing together several separate constant flags. These separate flags all have constants to define and describe them, as described here:

HAL Memory Capability Flags:

Enumerated Name	Description
HAL_MC_R_BIT	Bit indicates “RAM-capable” memory
HAL_MC_F_BIT	Bit indicates “FIFO-capable” memory
HAL_MC_I_BIT	Bit indicates I-cache capable memory
HAL_MC_D_BIT	Bit indicates D-cache capable memory
HAL_MC_U_BIT	Bit indicates U-cache capable memory

HAL Common Memory Capability Combinations:

Enumerated Name	Description
HAL_MC_R	Normal RAM only
HAL_MC_F	FIFO-only memory
HAL_MC_RF	RAM or FIFO memory
HAL_MC_RI	RAM or instruction cache
HAL_MC_RD	RAM or data cache
HAL_MC_RID	RAM, I-cache, or D-cache
HAL_MC_RU	RAM or unified cache
HAL_MC_RIDU	RAM, I-cache, D-cache, or U-cache
HAL_MC_I	Must be instruction cache
HAL_MC_D	Must be data cache
HAL_MC_U	Must be unified cache

HAL Controllable Memory Protection Flags:

Enumerated Name	Description
HAL_USER_EXECUTABLE	User-executable privilege selectable
HAL_USER_READABLE	User-read privilege selectable
HAL_USER_WRITABLE	User-write privilege selectable
HAL_PRIV_EXECUTABLE	Privileged mode-executable selectable
HAL_PRIV_READABLE	Privileged mode-readable selectable
HAL_PRIV_WRITABLE	Privileged mode-writable selectable

Used by:

OS startup/morphing code

Described in:

Section 4.2.1

HAL_GET_OBJ_TYPE

```
int HAL_GET_OBJ_TYPE(int object_identifier);
```

This is the basic function that architecture-neutral startup code can use to interrogate the TVM-HAL and determine the object type of any object identifiers returned by other functions, such as HAL_GET_OBJ_CHARACTERISTIC.

Inputs:

int object_identifier: The object identifier of the object to be queried. The interpretation of this field is system-dependent.

Returns:

(*int*) Type of the chosen object, as defined from the following enumerated list:

root (HAL_MM_TYPE_ROOT),
 level (HAL_MM_TYPE_LEVEL),
 processor (HAL_MM_TYPE_PROC),
 memory (HAL_MM_TYPE_MEM),
 dynamic network (HAL_MM_TYPE_NET), or
 point-to-point link (HAL_MM_TYPE_LINK).

If the object given to the call does not exist, then HAL_MM_TYPE_NONE is returned.

Used by:

OS startup/morphing code

Described in:

Section 4.2.1

HAL_LEVEL_OBJ_ITERATE**HAL_PROC_OBJ_ITERATE****HAL_MEM_OBJ_ITERATE****HAL_NET_OBJ_ITERATE****HAL_LINK_OBJ_ITERATE**

```
int HAL_LEVEL_OBJ_ITERATE(int iterator_count);
int HAL_PROC_OBJ_ITERATE(int iterator_count);
int HAL_MEM_OBJ_ITERATE(int iterator_count);
int HAL_NET_OBJ_ITERATE(int iterator_count);
int HAL_LINK_OBJ_ITERATE(int iterator_count);
```

These functions return all object identifiers for a particular type of object sequentially, so that a caller may easily step through all objects of that type (usually memory objects).

Inputs:

int iterator_count: A value between 0 and the total number of objects of the given type in the system, minus one, which specifies which object identifier from within a type should be returned to the caller.

Returns:

(*int*) Object identifier of the n^{th} object of the type selected by the routine called.

Used by:

OS startup/morphing code.

Described in:

Section 4.2.1

HAL_NUM_LEVELS**HAL_NUM_PROCS****HAL_NUM_MEMS****HAL_NUM_NETS****HAL_NUM_LINKS**

```
int HAL_NUM_LEVELS();
int HAL_NUM_PROCS();
int HAL_NUM_MEMS();
int HAL_NUM_NETS();
int HAL_NUM_LINKS();
```

Determine the total number of objects for a particular type of object, allowing the caller to determine how many steps will be required in any iteration loop to get all objects of that type.

Returns:

(*int*) Total number of objects of the type selected by the routine called.

Used by:

OS startup/morphing code.

Described in:

Section 4.2.1

HAL_HAS_TLB

```
HAL_bool HAL_HAS_TLB();
```

Tests to see whether or not a processor supports hardware mapping on a page-by-page level within each segment using a TLB.

Returns:

(*HAL_bool*) HAL_TRUE if this processor has a TLB and can map paged memory,
HAL_FALSE if not.

Used by:

OS startup/morphing code.

Described in:

Section 4.2.1

HAL_PRESET_SEGMENT_NUM

```
int HAL_PRESET_SEGMENT_NUM(int iterator_count);
```

This is the equivalent of the preceding object iteration routines, which convert an iterator into an object number, but for preset segments within a processor. It is used by startup code to determine the ID numbers for these segments, as necessary, so that their characteristics may be queried.

Inputs:

int iterator_count: A value between 0 and the total number of preset segments for this processor, minus one, which specifies which segment identification number should be returned to the caller.

Returns:

(*int*) Segment number of the n^{th} preset segment on this particular processor.

Used by:

OS startup/morphing code.

Described in:

Section 4.2.1

HAL_NUM_PRESET_SEGMENTS

```
int HAL_NUM_PRESET_SEGMENTS();
```

Returns the number of preset segments present on the processor, giving a clear endpoint for users looping through segments using the preceding HAL_PRESET_SEGMENT_NUM routine.

Returns:

(*int*) Number of preset segments on this particular processor.

Used by:

OS startup/morphing code.

Described in:
Section 4.2.1

6.2.3.2 MEMORY SEGMENTATION MACROS

HAL_CLEAR_SEGMENT

```
void HAL_CLEAR_SEGMENT(int segment_number);
```

Unmaps a current segment, eliminating the segment from the list of memory segments that can be addressed by software. It is mostly of use when morphing to a new configuration that needs to use the available segmentation control hardware for different configurations.

Input:
int segment_number: This selects one of the current segments for clearing.

Used by:
OS startup/morphing code.

Described in:
Section 4.2.2

HAL_SET_SEGMENT

```
int HAL_SET_SEGMENT(  
    void *starting_virtual_address,  
    HAL_phyPtr starting_physical_address,  
    HAL_uint64 size_in_bytes,  
    HAL_bool cacheable,  
    HAL_bool cache_coherent,  
    HAL_bool userExecutable,  
    HAL_bool userReadable,  
    HAL_bool userWritable,  
    HAL_bool osExecutable,  
    HAL_bool osReadable,  
    HAL_bool osWritable,  
    HAL_bool mapped,  
    HAL_uint64 map_physical_size,  
    int interleave_factor);
```

Sets up a segment-control register within the processor to map in a segment of memory. In the process, a wide range of access characteristics can be assigned to each memory block. See the list of inputs for the full range of characteristics that can be assigned with this call. It may be necessary to add more options in the future, as well.

Inputs:
*void *starting_virtual_address*: The base address in virtual (software-visible) memory where this segment will be visible to TVM or SVM code.
HAL_phyPtr starting_physical_address: The base address in physical (TVM-HAL only) memory where memory accesses to the segment actually map to.
HAL_int64 size_in_bytes: The size of the mapped segment, in bytes, in the virtual address space that is visible to software.

HAL_bool cacheable: Whether or not this segment should be cached in local memories that have been preselected using the `HAL_xCACHE_ENABLE` macro.

HAL_bool cache_coherent: This indicates whether or not the user expects cached copies to be maintained in a coherent manner with other processors' caches. If `TRUE`, the first preference will be to enable hardware cache coherency mechanisms. Otherwise, software coherence protocols will be introduced on each load and/or store to this segment, which may impose a significant performance penalty. As a result, this should only be used if it is considered absolutely necessary (and if one is fairly certain that hardware support will be present on the target PCA architecture).

HAL_bool userExecutable: Flag indicating that memory in this segment can contain user-level executable program code.

HAL_bool userReadable: Flag indicating that memory in this segment can be read by the processor using load accesses while in user mode. This should generally be executable, too.

HAL_bool userWritable: Flag indicating that memory in this segment can be written by the processor using store accesses while in user mode. This should generally be executable and readable, too.

HAL_bool osExecutable: Flag indicating that memory in this segment can contain privileged executable program code.

HAL_bool osReadable: Flag indicating that memory in this segment can be read by the processor using load accesses while in privileged mode. This should generally be executable, too.

HAL_bool osWritable: Flag indicating that memory in this segment can be written by the processor using store accesses while in privileged mode. This should generally be executable and readable, too.

HAL_bool mapped: This determines whether or not the memory within the segment is broken up into pages by hardware and mapped further using a TLB.

HAL_int64 map_physical_size: The size of the physical memory block that is actually mapped to represent a much larger virtual address space. Pages are mapped within this space and made visible to the programmer within the entire virtual address space specified using the *size_in_bytes* parameter.

int interleave_factor: This selects a combo-block interleave factor from the list of possible, hardware-supported factors given in memory characteristic #24.

Returns:

(*int*) If positive, this is the segment number for the new segment. It can be used in subsequent calls to clear the segment or examine its characteristics. If negative, it indicates that the `HAL_SET_SEGMENT` call was unable to allocate hardware resources capable of mapping the new segment for one of many possible reasons (if more than one, only the first error is returned), as enumerated below:

Enumerated Name	Description
HAL_SEG_ERR_VIRTUAL	Bad starting virtual address
HAL_SEG_ERR_PHYSICAL	Bad starting physical address
HAL_SEG_ERR_SIZE	Bad segment size in bytes
HAL_SEG_ERR_CACHEABLE	Illegal cacheable setting
HAL_SEG_ERR_COHERENT	Illegal cache coherent setting
HAL_SEG_ERR_USER_EXECUTABLE	Illegal executable (user code) setting
HAL_SEG_ERR_USER_READABLE	Illegal readable by user code setting
HAL_SEG_ERR_USER_WRITABLE	Illegal writable by user code setting
HAL_SEG_ERR_PRIV_EXECUTABLE	Illegal executabl (system code) setting
HAL_SEG_ERR_PRIV_READABLE	Illegal readable by system code setting
HAL_SEG_ERR_PRIV_WRITABLE	Illegal writable by system code setting
HAL_SEG_ERR_MAPPED	Illegal mapped by a TLB setting
HAL_SEG_ERR_MAP_PHYS_SIZE	Bad actual physical size of mapped memory
HAL_SEG_ERR_INTERLEAVE	Bad segment interleave factor
HAL_SEG_ERR_TABLE_FULL	Legal segment not allocated because insufficient hardware resources are available, most likely among segmentation control registers
HAL_SEG_ERR_ILLEGAL	General “illegal segment” error

Used by:

OS startup/morphing code.

Described in:

Section 4.2.2

HAL_GET_SEGMENT_CHARACTERISTIC

```
HAL_uint64 HAL_GET_SEGMENT_CHARACTERISTIC(  
    int segment_number,  
    int characteristic_num);
```

This is a function that architecture-neutral startup code can use to interrogate the TVM-HAL and determine key system parameters. Using this call, startup code can determine what the statistics of existing program segments are.

Inputs:

int segment_number: The segment number of the segment to be queried. The interpretation of this field is system-dependent.

int characteristic_num: The characteristic (from the list in Section 4.2.2) that is being queried from the selected segment. These lists are enumerated below for ease of reference in typical code:

Enumerated Name	Description
HAL_SEG_START_VIRTUAL	void *: Starting virtual address
HAL_SEG_START_PHYSICAL	HAL_phyPtr: Starting physical address
HAL_SEG_SIZE	HAL_uint64: Segment size in bytes
HAL_SEG_CACHEABLE	HAL_bool: Cacheable
HAL_SEG_CACHE_COHERENT	HAL_bool: Cache coherent
HAL_SEG_USER_EXECUTABLE	HAL_bool: Executable, for user code
HAL_SEG_USER_READABLE	HAL_bool: Readable by user code
HAL_SEG_USER_WRITABLE	HAL_bool: Writable by user code
HAL_SEG_PRIV_EXECUTABLE	HAL_bool: Executable, for system code
HAL_SEG_PRIV_READABLE	HAL_bool: Readable by system code
HAL_SEG_PRIV_WRITABLE	HAL_bool: Writable by system code
HAL_SEG_MAPPED	HAL_bool: Mapped by a TLB
HAL_SEG_MAP_PHYS_SIZE	HAL_uint64: Actual physical size of mapped memory
HAL_SEG_INTERLEAVE	int: Segment interleave factor

Returns:

(*HAL_int64*) Requested memory characteristic for the selected segment cast to a HAL_int64 type variable, which will always be large enough to hold the requested value. Depending upon the requested value, this should usually be cast back to another type.

Used by:

OS startup/morphing code.

Described in:

Section 4.2.2

6.2.3.3 CACHE CONTROL MACROS

The following macros may be applied to the instruction cache (I), data cache (D), or unified cache (U, which applies to both I and D if the hardware is actually split). The *x* in each macro name should be replaced with the appropriate letter to select the correct macro.

HAL_xCACHE_ENABLE

```
void HAL_xCACHE_ENABLE(
    int num_components_in_cache,
    int memory_component_idents[],
    int num_components_below,
    int components_below[],
    int num_components_above,
    int components_above[],
    int linesize);
```

Configures a physical memory bank to act as a cache memory for other memory banks instead of as an independently addressable memory bank. In order to allow complex

hierarchies of caches, it allows the specification of one or more components above it (level N-1 cache blocks or processors) and one or more components below (level N+1 cache blocks or addressable memory banks). After enabling, caches are turned on or off on a segment-by-segment basis by PCA processors using the next macros.

Inputs:

- int num_components_in_cache*: The number of memory components that are ganged together to act as a cache.
- int memory_component_ids[]*: The object identifiers for the memories participating in the cache structure. The length of this array should be specified using the *num_components_in_cache* parameter.
- int num_components_below*: The number of memory components that are part of the next level in the hierarchy (the N+1 level cache or actual memory being cached).
- int components_below[]*: The object identifiers for the memories participating in the next level of the cache hierarchy. The length of this array should be specified using the *num_components_below* parameter.
- int num_components_above*: The number of memory components that are part of the previous level in the hierarchy (the N-1 level cache or processors that can use the cache).
- int components_above[]*: The object identifiers for the memories participating in the previous level of the cache hierarchy. The length of this array should be specified using the *num_components_above* parameter.
- int linesize*: The linesize used by the cache at this level of the hierarchy. This should be chosen from among the available linesizes listed in the metadata for the various memory blocks participating in the cache.

Used by:

OS startup/morphing code.

Described in:

Section 4.2.2

HAL_xCACHE_ON**HAL_xCACHE_OFF****HAL_xCACHE_ON_SEG****HAL_xCACHE_OFF_SEG**

```
void HAL_xCACHE_ON();
void HAL_xCACHE_OFF();
void HAL_xCACHE_ON_SEG(int segment_number);
void HAL_xCACHE_OFF_SEG(int segment_number);
```

Enables or disables the selected type of caching (instruction, data, or unified) for this processor. If desired, caching can also be selectively enabled on a segment-by-segment basis.

Inputs:

- int segment_number*: The identifier number of the segment returned by `HAL_SET_SEGMENT` or `HAL_PRESET_SEGMENT_NUM` for which caching should be enabled or disabled.

Used by:

OS startup/morphing code.

Described in:

Section 4.2.2

6.2.3.4 VIRTUAL MEMORY CONTROL MACROS

HAL_FAULT_TYPE

```
int HAL_FAULT_TYPE();
```

Returns a code indicating the cause of the last page fault, allowing a single fault handler to selectively deal with several types of faults in an appropriate manner.

Returns:

(*int*) An enumerated value: HAL_FAULT_LOAD=load, HAL_FAULT_STORE=store, HAL_FAULT_IFETCH=instruction fetch, or HAL_FAULT_OTHER=other (usually a synchronization instruction) caused the last page fault.

Used by:

OS page fault handling code.

Described in:

Section 4.3

HAL_FAULT_SIZE

```
int HAL_FAULT_SIZE();
```

Returns the size of the data access during the last page fault, a parameter that is useful if the handler needs to simulate part of the load or store itself.

Returns:

(*int*) The size of the data word being read from or written to memory, in bytes, when the last page fault occurred.

Used by:

OS page fault handling code.

Described in:

Section 4.3

HAL_GET_FAULT_ADDRESS

```
HAL_ptr HAL_GET_FAULT_ADDRESS();
```

Returns the address of the data value being read or written when the last page fault occurred.

Returns:

(*HAL_ptr*) The virtual address being accessed when the last page fault occurred.

Used by:

OS page fault handling code.

Described in:
Section 4.3

HAL_LOAD_TABLE_BASE

```
void HAL_LOAD_TABLE_BASE(HAL_ptr new_table_base);
```

Loads a new virtual memory table base pointer into the TLB-control hardware registers or software TLB refill control data. This call should generally be combined with a `HAL_TLB_INVALIDATE_ALL` and used whenever contexts need to switch.

Input:

HAL_ptr new_table_base: This is the virtual address of the base of the new page table (as returned by `HAL_INIT_PAGETABLE`) that should be used to refill the TLB, when necessary.

Used by:

OS startup/morphing/context switching code.

Described in:
Section 4.3

HAL_TLB_INVALIDATE
HAL_TLB_INVALIDATE_ALL

```
void HAL_TLB_INVALIDATE(HAL_ptr virtual_address_of_entry_to_kill);  
void HAL_TLB_INVALIDATE_ALL();
```

These macros invalidate one entry from the TLB or the entire TLB at once. The first version is usually used when a page is evicted from main memory before another is loaded in from the disk. The latter is usually used when a context is being swapped off of the processor, and therefore all of its TLB entries need to be removed so that they may not be accidentally reused by another context.

Input:

Vptr virtual_address_of_entry_to_kill: This is the virtual address being mapped by a single entry in the TLB that needs to be eliminated, usually due to flushing of the page from memory.

Used by:

OS page fault handling or morphing/context switching code.

Described in:
Section 4.3

HAL_INIT_PAGETABLE

```
HAL_ptr HAL_INIT_PAGETABLE(  
    int segment_number,  
    int target_segment,  
    HAL_uint16 process_id,  
    HAL_int64 basic_page_size,  
    HAL_TableExpandCodePtr memfetcher_function,  
    HAL_TableShrinkCodePtr memfree_function);
```

Creates a new pagetable in memory that can subsequently be used to map virtual memory on a page-by-page basis using the other paged memory management macros. Usually, one page table per process is sufficient, but more may be required in PCA architectures to manage all of the different memories effectively. Because the memory requirements for a page table are highly variable, the function requires that the user supply customized `malloc` and `free` routines to it so that it may allocate or deallocate additional memory on an as-needed basis.

Inputs:

int segment_number: The number of the memory segment (in terms of hardware register segment number) that will be holding the page table itself.

int target_segment: The segment (in terms of hardware register segment number) that is to be mapped with this page table. This chooses the type of table structure correctly for the TLB and memories participating in the mapped memory.

HAL_uint16: The process ID associated with this memory space. Essentially, this is a memory space ID that can be used by the HAL to organize the multiple page tables that it may be managing with a unique ID associated with each.

HAL_int64 basic_page_size: The size of pages within this page table. This should obviously be a size supported by the TLB and memories that are participating.

TableExpandCodePtr memfetcher_function: A pointer to the caller-provided `malloc` function that can be used by the page table management routines to expand the table automatically, when necessary. This function should be of the format: `HAL_ptr TableMalloc(HAL_int64 num_bytes_to_allocate, int align);` and is essentially a `malloc` function with the additional restraint that the returned blocks may need to be aligned to be even with every *align* bytes of memory. Also, this routine *must* allocate memory in a different segment than the one that is mapped by the page table itself, since a table cannot map its own memory segment. In general, user memory tables may be in mapped OS memory segments and OS memory tables must be in unmapped memory, but some architectures with smaller page tables require *all* tables to be in unmapped segments.

TableShrinkCodePtr memfree_function: A pointer to the caller-provided `free` function that can be used by the page table management routines to shrink the table automatically, when necessary. This function should be of the format: `void TableFree(HAL_ptr address);` and is just like a normal `free` function in almost every respect.

Returns:

(*HAL_ptr*) Virtual address of the base of the newly-created page table that can be used in several other routines.

Used by:

OS context initialization code

Described in:

Section 4.3

HAL_FREE_PAGETABLE

```
void HAL_FREE_PAGETABLE(HAL_ptr page_table_base);
```

Frees up a pagetable in memory, performing any necessary cleanup tasks and deallocating the memory using the `memfree_function` for the page table.

Inputs:

HAL_ptr page_table_base: Base address of the page table to eliminate.

Used by:

OS context destruction code.

Described in:

Section 4.3

HAL_CLEAR_PAGE

```
void HAL_CLEAR_PAGE(  
    HAL_ptr page_table_base,  
    HAL_PagePtr starting_virtual_address);
```

Evicts a single entry from a page table, on the basis of its virtual address. It should be used when a page is flushed out of memory.

Inputs:

HAL_ptr page_table_base: Base address of the page table to modify.

HAL_PagePtr starting_virtual_address: Virtual address of the beginning of the page to evict from the page table.

Used by:

OS page fault handling code.

Described in:

Section 4.3

HAL_TEST_SET_PAGE

```
int HAL_TEST_SET_PAGE(  
    HAL_ptr page_table_base,  
    HAL_PagePtr starting_virtual_address,  
    HAL_PagePhyPtr starting_physical_address,  
    HAL_int64 page_size,  
    const int memory,  
    int *num_page_conflicts,  
    HAL_ptr *conflicts_page_table_base_array,  
    HAL_PagePtr *conflicts_virtual_addr_array);
```

Prepares the page table for accepting a new page. On architectures that have unlimited-size page tables (*i.e.* they simply allocate more memory whenever necessary), this routine will usually do nothing. On architectures that have limited-size page tables, however, this routine “reserves” a spot in the page table and returns the number of pages that must be removed from the page table before the new page may be loaded. In order to provide the OS with as much choice as possible, a list of candidate victim pages is supplied that is equal to or potentially larger than the number of pages that must be removed. These pages can then be removed with `HAL_CLEAR_PAGE` before the new page is officially loaded into the page table with `HAL_SET_PAGE`.

Inputs:

HAL_ptr page_table_base: Base address of the page table to modify.

HAL_PagePtr starting_virtual_address: The base address in virtual (software-visible) memory where the new page will be visible to TVM or SVM code.

HAL_PagePhyPtr starting_physical_address: The base address in physical (TVM-HAL only) memory where the page actually resides.

HAL_int64 page_size: The size of the page, in bytes. This factor will normally be constant for a given page table, but some large superpages may also be used from time to time.

const int memory: The memory bank optimization number for the segment containing the three “output” items, below.

*int *num_page_conflicts*: Pointer to an integer output buffer. On output, this contains the number of choices of pages to be removed from the page table by the OS presented in the list given in the next two output variables.

*HAL_ptr *conflicts_page_table_base_array*: Pointer to an array of pointers output buffer. On output, this contains an array of *num_page_conflicts* pointers to the page table bases associated with the pages presented back to the OS for possible eviction. This buffer should be able to hold at least `HAL_MAXIMUM_CONFLICTING_PAGES` entries.

*HAL_PagePtr *conflicts_virtual_address_array*: Pointer to an array of page pointers output buffer. On output, this contains an array of *num_page_conflicts* virtual addresses of pages presented back to the OS for possible eviction. This buffer should be able to hold at least `HAL_MAXIMUM_CONFLICTING_PAGES` entries.

Returns:

(*int*) Number of pages that *must* be cleared out from the page table from among the list of possible candidate pages supplied by this function, in order to make room for the new page. This number will be equal to or less than the number of possible candidate pages supplied.

Used by:

OS page fault handling code.

Described in:

Section 4.3

HAL_SET_PAGE

```
void HAL_SET_PAGE(  
    HAL_ptr page_table_base,  
    HAL_PagePtr starting_virtual_address,  
    HAL_PagePhyPtr starting_physical_address,  
    HAL_int64 page_size,  
    HAL_boolX cacheable,  
    HAL_boolX cache_coherent,  
    HAL_boolX userExecutable,  
    HAL_boolX userReadable,  
    HAL_boolX userWritable,  
    HAL_boolX osExecutable,  
    HAL_boolX osReadable,  
    HAL_boolX osWritable);
```

Loads a page entry into the page table, and should be called as soon as possible after a corresponding `HAL_TEST_SET_PAGE`, because the HAL will usually be holding a lock on the page table during the time between these two calls. It will generally be used whenever a page fault occurs to load in the characteristics of the newly-mapped page of memory. Much like `HAL_SET_SEGMENT`, there are many options that control how the page may be accessed. See the list of inputs for a full selection of these options. Note that all of the protection parameters can be set using the `HAL_boolX` type, which specifies a ternary Boolean value—`HAL_FALSE`, `HAL_TRUE`, and `HAL_DONT_CARE` (*i.e.*, use the segment defaults).

Inputs:

HAL_ptr page_table_base: Base address of the page table to modify.

HAL_PagePtr starting_virtual_address: The base address in virtual (software-visible) memory where this page will be visible to TVM or SVM code.

HAL_PagePhyPtr starting_physical_address: The base address in physical (TVM-HAL only) memory where the page actually resides.

HAL_int64 page_size: The size of the page, in bytes. This factor will normally be constant for a given page table, but some large superpages may also be used from time to time.

HAL_boolX cacheable: Whether or not this page should be cached in local memories that have been preselected using the `HAL_XCACHE_ENABLE` macro.

HAL_boolX cache_coherent: This indicates whether or not the user expects cached copies to be maintained in a coherent manner with other processors' caches. If `TRUE`, the first preference will be to enable hardware cache coherency mechanisms. Otherwise, software coherence protocols will be introduced on each load and/or store to this segment, which may impose a significant performance penalty. As a result, this should only be used if it is considered absolutely necessary (and if one is fairly certain that hardware support will be present on the target PCA architecture).

HAL_boolX userExecutable: Flag indicating that memory in this page can contain user-level executable program code.

HAL_boolX userReadable: Flag indicating that memory in this page can be read by the processor using load accesses while in user mode. This memory should generally be executable also.

HAL_boolX userWriteable: Flag indicating that memory in this page can be written by the processor using store accesses while in user mode. This should also generally be both executable and readable.

HAL_boolX osExecutable: Flag indicating that memory in this page can contain privileged executable program code.

HAL_boolX osReadable: Flag indicating that memory in this page can be read by the processor using load accesses while in privileged mode. This should also generally be executable.

HAL_boolX osWriteable: Flag indicating that memory in this page can be written by the processor using store accesses while in privileged mode. This should also generally be both executable and readable.

Used by:

OS page fault handling code.

Described in:

Section 4.3

HAL_PAGING_CHECK

```
int HAL_PAGING_CHECK(int protection_type);
```

Tests to see if the architecture actually supports selective memory protection according to the UNIX-like varieties supported by the HAL (executable, readable, writable) on a page-by-page level. Many architectures will have only some of these flags supported in hardware (for example, readable but not executable, or only user-level memory protection flags but not privileged-mode memory protection). It can also be used to test whether the hardware supports pages that may be selectively cacheable or cache-coherent.

Input:

int protection_type: This selects one of the desired protection or cache coherence flags to see if the hardware actually enforces this type of memory protection or coherence at the selected privilege level. This integer selects the protection type from among the following enumerated list of values: `HAL_CACHEABLE`, `HAL_CACHE_COHERENT`, `HAL_USER_EXECUTABLE`, `HAL_USER_READABLE`, `HAL_USER_WRITABLE`, `HAL_PRIV_EXECUTABLE`, `HAL_PRIV_READABLE`, or `HAL_PRIV_WRITABLE`.⁶

Returns:

(*int*) An enumerated type that specifies the flexibility: `HAL_SUPPORTED` if this processor supports the selected caching or memory protection mode (*i.e.*, can selectively block or cache references in this manner), `HAL_IGNORED` if the mode is ignored on this architecture. For protection modes, this means that it will be either not protected and freely executable/readable/writable, or this form of protection is controlled only as a side effect of setting another, higher-priority type of protection. In the latter case, writable is given the highest priority to be supported, then readable, and finally executable.

⁶ If other privilege levels exist in the system, then three flags will generally exist for each additional privilege state, too.

Used by:

OS page fault handling initialization code.

Described in:

Section 4.3

HAL_PAGE_PRESENT

```
HAL_bool HAL_PAGE_PRESENT(  
    HAL_ptr page_table_base,  
    HAL_PagePtr starting_virtual_address);
```

Tests to determine whether or not a page, identified by its starting virtual address, is present within a page table.

Inputs:

HAL_ptr page_table_base: Base address of the page table to check for the page.

HAL_PagePtr starting_virtual_address: The base address in virtual (software-visible) memory of the page to check.

Returns:

(*HAL_bool*) TRUE if the page is present in the page table, or FALSE if not.

Used by:

OS page fault handling or timer interrupt code.

Described in:

Section 4.3

HAL_GET_PHY_PAGE

```
HAL_PagePhyPtr HAL_GET_PHY_PAGE(  
    HAL_ptr page_table_base,  
    HAL_PagePtr starting_virtual_address);
```

Performs a virtual-to-physical memory translation so that page fault handling code can find a page in physical memory prior to flushing it out to disk, and so that it knows where in physical memory to load the new page.

Inputs:

HAL_ptr page_table_base: Base address of the page table to use for lookup.

HAL_PagePtr starting_virtual_address: The base address in virtual (software-visible) memory of the page to perform the virtual-to-physical mapping on.

Returns:

(*HAL_PagePhyPtr*) The physical address of the mapped page, or NULL if it is not in memory.

Used by:

OS page fault handling or timer interrupt code.

Described in:

Section 4.3

HAL_GET_PREVIOUS_PAGE**HAL_GET_NEXT_PAGE**

```

void HAL_GET_PREVIOUS_PAGE(
    const int memory,
    HAL_ptr *page_table_base,
    HAL_PagePtr *starting_virtual_address);
void HAL_GET_NEXT_PAGE(
    const int memory,
    HAL_ptr *page_table_base,
    HAL_PagePtr *starting_virtual_address);

```

These routines return the page table base address and starting virtual addresses of the previous and next pages that are actually present within the page table. Because two values must be returned, they are passed by reference and changed in-place. The exact definition of “previous” and “next” is implementation-dependent, but a page-aging algorithm (such as the clock algorithm) is guaranteed to walk through *all* of the pages in the table if it calls these functions enough times.

Inputs:

const int memory: The memory optimization number of the memory containing the buffers for the next two inputs.

*HAL_ptr *page_table_base*: A pointer to a buffer containing the base address of the page table to walk. Upon return, this may be modified if the “next” page in physical memory is from a different page table.

*HAL_PagePtr *starting_virtual_address*: A pointer to a buffer containing the base address in virtual (software-visible) memory of the page to start walking from. Upon return, this buffer contains the pointer to the previous or next page, instead.

Used by:

OS page fault handling or timer interrupt code.

Described in:

Section 4.3

HAL_GET_PAGE_TOUCHED**HAL_GET_RESET_PAGE_TOUCHED****HAL_GET_PAGE_MODIFIED****HAL_GET_RESET_PAGE_MODIFIED**

```

HAL_bool HAL_GET_PAGE_TOUCHED(
    HAL_ptr page_table_base,
    HAL_PagePtr start_virtual_address);
HAL_bool HAL_GET_RESET_PAGE_TOUCHED(
    HAL_ptr page_table_base,
    HAL_PagePtr start_virtual_address);
HAL_bool HAL_GET_PAGE_MODIFIED(
    HAL_ptr page_table_base,
    HAL_PagePtr start_virtual_address);
HAL_bool HAL_GET_RESET_PAGE_MODIFIED (
    HAL_ptr page_table_base,
    HAL_PagePtr start_virtual_address);

```

These routines are used by the OS's page-aging or flushing code to check on the status of pages. The OS may read whether or not pages have been touched (read or written) or modified (written) since the last time the page was examined. This is useful for determining which pages have been used recently (touched) or are dirty and need to be flushed to disk (modified). In addition, the RESET versions of the calls clear out the bits when the current values are no longer needed (*i.e.*, after the aging algorithm is done with the page or after the data is flushed).

Inputs:

HAL_ptr page_table_base: Base address of the page table to examine and/or modify.

HAL_PagePtr start_virtual_address: The base address in virtual (software-visible) memory of the page to examine and/or modify the status of.

Returns:

(*HAL_bool*) The current state of the touched (read or written) or modified (written) bits for the page being examined.

Used by:

OS page fault handling or timer interrupt code.

Described in:

Section 4.3

6.2.4 Other Macros

This section summarizes the macros that do not obviously fit into either the exception handler category or the memory control category. Most are described further in Section 5.

6.2.4.1 PROCESSOR IDENTIFICATION

HAL_PROCESSOR_IDENT

```
int HAL_PROCESSOR_IDENT();
```

Returns the processor identification number for the current processor. This number is a sequential value unique to each processor, from 0 onwards. It can easily be converted to an object identification number using `HAL_PROC_OBJ_ITERATE`. This can be used by operating system code to specialize the configuration or function of the processor, *i.e.*, to select different memory configurations based on the current processor.

Returns:

(*int*) This processor's sequential ID number.

Used by:

OS startup/morphing code, or any core code which must differentiate processors.

Described in:

Section 5.1

6.2.4.2 ACTIVE DMA MACROS

HAL_BLOCKMOVE_P

HAL_BLOCKMOVE

```
int HAL_BLOCKMOVE_P(
    const int DMA_controller_num,
    const int memory_source,
    const int memory_dest,
    HAL_phyPtr source_phy_ptr,
    HAL_phyPtr dest_phy_ptr,
    int num_bytes);
int HAL_BLOCKMOVE(
    const int DMA_controller_num,
    const int memory_source,
    const int memory_dest, void *source_ptr,
    void *dest_ptr,
    int num_bytes);
```

These operations move a block of data from one place in a PCA architecture to another, on the same or different memory blocks, using a DMA controller. These basic versions of the routines move the data blindly from one place to another and block the calling processor while they are doing it. When the operation is complete, an error code⁷ notifies the caller as to success or failure of therequested operation.

Inputs:

const int DMA_controller_num: The DMA controller (number-to-hardware mapping is defined on an implementation-specific basis) that is to manage this block-move operation.

const int memory_source: The physical memory bank (identified by optimization number) containing the source (input) buffer.

const int memory_dest: The physical memory bank (identified by optimization number) containing the destination (output) buffer.

HAL_phyPtr source_phy_ptr: The starting address of the source (input) buffer, in physical addresses. This should only be used by the low-level OS to move data around directly from one memory bank to another.

*void *source_ptr*: The starting address of the source (input) buffer, in the virtual address space. This is the normal way of using the function.

HAL_phyPtr dest_phy_ptr: The starting address of the destination (output) buffer, in physical addresses.

*void *dest_ptr*: The starting address of the destination (output) buffer, in virtual addresses.

int num_bytes: The number of bytes to transfer with this operation.

Returns:

(*int*) DMA operation error code.

⁷ The error code will usually be limited to 0=success and 1=failure. The list of codes may be more extensive if the architectures supports reporting of more types of low-level errors.

Used by:

Any TVM code (`_P` restricted to OS).

Described in:

Section 5.2

HAL_ACTIVE_BLOCKMOVE_P**HAL_ACTIVE_BLOCKMOVE**

```
void HAL_ACTIVE_BLOCKMOVE_P(  
    const int DMA_controller_num,  
    const int memory_source,  
    const int memory_dest,  
    HAL_phyPtr source_phy_ptr,  
    HAL_phyPtr dest_phy_ptr,  
    int num_bytes,  
    int source_vector,  
    const int dest_processor_num,  
    int destination_vector,  
    const int memory_error_code,  
    HAL_phyPtr put_error_code_here_mem_ptr);  
void HAL_ACTIVE_BLOCKMOVE(  
    const int DMA_controller_num,  
    const int memory_source,  
    const int memory_dest,  
    void *source_ptr,  
    void *dest_ptr,  
    int num_bytes,  
    int source_vector,  
    const int dest_processor_num,  
    int destination_vector,  
    const int memory_error_code,  
    int *put_error_code_here_ptr);
```

These operations move a block of data from one place in a PCA architecture to another, on the same or different memory blocks, using a DMA controller. These active versions of the **BLOCKMOVE** routines still move the data blindly from one place to another, but are non-blocking, and let the processor continue to execute code while the transfer completes. At the end of the transfer, an error code is stored in an integer-size buffer supplied by the call, and the DMA controller can interrupt the source processor, the destination processor, or both. The caller may specify specific DMA interrupt vectors right in the transfer, in order to allow a customized interrupt-handling response to each kind of packet transfer. It should also be noted that the destination processor is indicated within the call itself, so this may be selected in a somewhat arbitrary manner, a useful technique if one wants to send an interrupt over to another processor.

Inputs:

const int DMA_controller_num: The DMA controller (number-to-hardware mapping is defined on an implementation-specific basis) that is to manage this block-move operation.

const int memory_source: The physical memory bank (identified by optimization number) containing the source (input) buffer.

const int memory_dest: The physical memory bank (identified by optimization number) containing the destination (output) buffer.

HAL_phyPtr source_phy_ptr: The starting address of the source (input) buffer, in physical addresses. This should only be used by the low-level OS to move data around directly from one memory bank to another.

*void *source_ptr*: The starting address of the source (input) buffer, in the virtual address space. This is the normal way of using the function.

HAL_phyPtr dest_phy_ptr: The starting address of the destination (output) buffer, in physical addresses.

*void *dest_ptr*: The starting address of the destination (output) buffer, in virtual addresses.

int num_bytes: The number of bytes to transfer with this operation.

int source_vector: The exception vector to trigger on the source processor when the transfer is complete. This should be in the range of #128–255. When the DMA processor triggers an interrupt at the end of the handler, then the processor is automatically vectored off to the desired exception vector. If no source interrupt is desired, then leave this equal to 0.

const int dest_processor_num: The destination processor, which should be interrupted when the transfer is complete. Note that in a PCA architecture with complex memory interconnections, this may not even be the processor that is closest to the memory!

int destination_vector: The exception vector to trigger on the destination processor when the transfer is complete. Restrictions are identical to the *source_vector*.

const int memory_error_code: The physical memory bank (identified by optimization number) containing the error code buffer.

HAL_phyPtr put_error_code_here_mem_ptr: The location where the error code should be placed when the transfer concludes, in physical addresses that should only be used by low-level OS code.

*int *put_error_code_here_ptr*: The location where the error code should be placed when the transfer concludes, in virtual addresses for use by all.

Returns:

(*int*) DMA error code, through the integer pointed at by *put_error_code_here_ptr*. This value is guaranteed to NOT be –1, so that is a good value to pre-load into the error code buffer if any processors want to be able to poll on the completion of the DMA (unlikely, but possible).

Used by:

Any TVM code (*_P* restricted to OS).

Described in:

Section 5.2

HAL_DMA_ENABLE
HAL_DMA_DISABLE

```
void HAL_DMA_ENABLE(int DMA_controller_num);  
void HAL_DMA_DISABLE(int DMA_controller_num);
```

These routines enable or disable the standard DMA interrupt handler (that accepts the hardware-specified #96-126 interrupt and re-vectors to #128-255, depending upon the packet contents) for a specified DMA controller. This allows both simple, custom interrupt handlers for some DMA controllers and full-fledged active vectoring for others.

Input:

int DMA_controller_num: The DMA controller (number-to-hardware mapping is defined on an implementation-specific basis) that is to have the standard DMA interrupt handler enabled or disabled.

Used by:

OS initialization or context switch code.

Described in:

Sections 3.1 and 5.2

HAL_DMA2VECTORNUM

```
int HAL_DMA2VECTORNUM(int DMA_controller_num);
```

Translates from DMA controller numbers to their associated hardware interrupt vector (#96-126, in the exception vector table). This routine will usually only be needed if an OS programmer wants to substitute a customized DMA interrupt handler for the standard one.

Input:

int DMA_controller_num: The DMA controller (number-to-hardware mapping is defined on an implementation-specific basis) for which to return its hardware exception number.

Returns:

(*int*) Hardware-specified #96-126 vector associated with the DMA controller.

Used by:

OS initialization or context switch code.

Described in:

Sections 3.1 and 5.2

HAL_VECTOR2DMANUM

```
int HAL_VECTOR2DMANUM(int entry_number);
```

Performs the inverse operation of HAL_DMA2VECTORNUM, returning a DMA controller number for a given exception vector. If a hardware exception number that is not associated with a DMA controller (either because it's assigned to different hardware or is unused) is supplied, then the routine will always return 0.

Input:

int entry_number: The hardware exception number for which to return its assigned DMA controller mapping.

Returns

(*int*) DMA controller number associated with the hardware-specified #96-126 hardware interrupt vector, if there is one (0 otherwise).

Used by:

OS initialization or context switch code.

Described in:

Sections 3.1 and 5.2

6.2.4.3 MULTIPROCESSOR SYNCHRONIZATION MACROS

HAL_MUTEX_INIT_P

HAL_MUTEX_INIT

```
int HAL_MUTEX_INIT_P(
    const int memory,
    HAL_phyPtr_mutex_t phy_lock_ptr);
int HAL_MUTEX_INIT(
    const int memory,
    HAL_mutex_t *lock_ptr);
```

These routines initialize a pthreads-style mutex (lock) object in physical or virtual memory, respectively, so that it may be used with the HAL_MUTEX_TRYLOCK and HAL_MUTEX_UNLOCK macros.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the mutex object.

HAL_phyPtr_mutex_t phy_lock_ptr: The address of the mutex to initialize, located directly in physical memory for low-level use.

*HAL_mutex_t *lock_ptr*: The virtual address of the mutex to initialize.

Returns:

(*int*) Error code from release. Generally 0 = no error.

Used by:

Any TVM code (_P restricted to OS).

Described in:

Section 5.3

HAL_MUTEX_TRYLOCK_P
HAL_MUTEX_TRYLOCK

```
int HAL_MUTEX_TRYLOCK_P(  
    const int memory,  
    HAL_phyPtr_mutex_t phy_lock_ptr);  
int HAL_MUTEX_TRYLOCK(  
    const int memory,  
    HAL_mutex_t *lock_ptr);
```

These macros perform an atomic test-and-set on the lock value at the core of the mutex object supplied either using a physical or virtual address. If successful, the lock is acquired and the value **TRUE** is returned. Otherwise, the calls return with a value of **FALSE**. This is identical to the standard pthreads interpretation, except for the additional *memory* specifier.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the mutex object.
HAL_phyPtr_mutex_t phy_lock_ptr: The address of the mutex to try and lock, located directly in physical memory for low-level use.
*HAL_mutex_t *lock_ptr*: The virtual address of the mutex to try to lock.

Returns:

(*int*) **TRUE** if the lock is successfully acquired, or **FALSE** if still waiting.

Used by:

Any TVM code (*_P* restricted to OS).

Described in:

Section 5.3

HAL_MUTEX_UNLOCK_P
HAL_MUTEX_UNLOCK

```
int HAL_MUTEX_UNLOCK_P(  
    const int memory,  
    HAL_phyPtr_mutex_t phy_lock_ptr);  
int HAL_MUTEX_UNLOCK(  
    const int memory,  
    HAL_mutex_t *lock_ptr);
```

These macros release a lock that was previously acquired using **HAL_MUTEX_TRYLOCK**.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the mutex object.
HAL_phyPtr_mutex_t phy_lock_ptr: The address of the mutex to release, located directly in physical memory for low-level use.
*HAL_mutex_t *lock_ptr*: The virtual address of the mutex to release.

Returns:

(*int*) Error code from initialization. Generally 0 = no error.

Used by:

Any TVM code (`_P` restricted to OS).

Described in:

Section 5.3

HAL_BARRIER_GLOBAL_INIT_P**HAL_BARRIER_GLOBAL_INIT**

```
int HAL_BARRIER_INIT_P(
    const int memory,
    HAL_phyPtr_barrier_t phy_barrier,
    int proc_count);
int HAL_BARRIER_INIT(
    const int memory,
    HAL_barrier_t *barrier,
    int proc_count);
```

These routines initialize a global barrier object for use on a PCA system, in physical or virtual memory, respectively. These barriers may be built up from software primitives or use explicit hardware barrier acceleration in order to enhance performance as much as possible. This “half” of the initialization should be called by only one of the processors that will be participating in the barrier, before any of the processors try and enter the barrier.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the barrier object.

HAL_phyPtr_barrier_t phy_barrier: The address of the barrier to initialize, located directly in physical memory for low-level use.

*HAL_barrier_t *barrier*: The virtual address of the barrier to initialize.

int proc_count: The number of processors that will be participating in the barrier. This value cannot be changed once it is set at barrier initialization.

Returns:

(*int*) Error code from initialization. Generally 0 = no error.

Used by:

Any TVM code (`_P` restricted to OS).

Described in:

Section 5.3

HAL_BARRIER_LOCAL_INIT_P**HAL_BARRIER_LOCAL_INIT**

```
int HAL_BARRIER_LOCAL_INIT_P(
    const int memory,
    HAL_phyPtr_barrier_local_t phy_barrier);
int HAL_BARRIER_LOCAL_INIT(
    const int memory,
    HAL_barrier_local_t *barrier);
```

These routines initialize a local barrier object for use on a PCA system, in physical or virtual memory, respectively. These barriers may be built up from software primitives or use explicit hardware barrier acceleration in order to enhance performance as much as possible. This “half” of the initialization should be called by each of the processors that will be participating in the barrier (each using their own local barrier object) before that processor enters the barrier for the first time.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the barrier object.

HAL_phyPtr_barrier_local_t phy_barrier: The address of a local part of the barrier to initialize, located directly in physical memory for low-level use.

*HAL_barrier_local_t *barrier*: The virtual address of a local part of the barrier to initialize.

Returns:

(*int*) Error code from initialization. Generally 0 = no error.

Used by:

Any TVM code (*_P* restricted to OS).

Described in:

Section 5.3

HAL_BARRIER_ENTER_P**HAL_BARRIER_ENTER**

```
int HAL_BARRIER_ENTER_P(  
    const int memory,  
    HAL_phyPtr_barrier_t phy_barrier,  
    HAL_phyPtr_barrier_local_t phy_barrier_local);  
int HAL_BARRIER_ENTER(  
    const int memory,  
    HAL_barrier_t *barrier,  
    HAL_barrier_local_t *barrier_local);
```

These routines should be called when a processor reaches the barrier and needs to wait there. The processor is added to the list of waiting processors and delayed until all processors reach the barrier or an interrupt occurs.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the barrier object.

HAL_phyPtr_barrier_t phy_barrier: The address of the global portion of the barrier to enter, located directly in physical memory for low-level use.

*HAL_barrier_t *barrier*: The virtual address of the global portion of the barrier to enter.

HAL_phyPtr_barrier_local_t phy_barrier: The address of this processor’s local portion of the barrier to enter, located directly in physical memory for low-level use.

*HAL_barrier_local_t *barrier*: The virtual address of this processor’s local portion of the barrier to enter.

Returns:

(*int*) Error code from starting. Generally 0 = no error.

Used by:

Any TVM code (*_P* restricted to OS).

Described in:

Section 5.3

6.2.4.4 LOW-LEVEL SYNCHRONIZATION MACROS

HAL_MEMSYNC

```
void HAL_MEMSYNC();
```

Forces the processor to block until all memory references have propagated out of private buffers (such as write buffers) and are now visible to any other cache-coherent processors in the system.

Used by:

Any TVM code.

Described in:

Section 5.4

HAL_MEMSYNC_TEST

```
HAL_bool HAL_MEMSYNC_TEST();
```

This is similar to `HAL_MEMSYNC`, except that it does not block. Instead, a Boolean value is returned that indicates whether or not the flushing has completed yet. If there is no non-blocking memory synchronization operation available on the underlying architecture, then the call will be converted into a blocking `MEMSYNC` by the low-level compiler.

Returns:

(*HAL_bool*) TRUE if memory is now synchronized (all private buffers flushed), or FALSE otherwise.

Used by:

Any TVM code.

Described in:

Section 5.4

HAL_MEMFENCE_P**HAL_MEMFENCE**

```
void HAL_MEMFENCE_P(  
    const int memory,  
    HAL_phyPtr phy_address);  
void HAL_MEMFENCE(  
    const int memory,  
    void *address);
```

These routines perform a `HAL_MEMSYNC` operation, waiting while buffers are flushed out to memory that is visible to other processors, but only need to wait for all accesses to the supplied address to complete, instead of *all* accesses to *all* addresses. If no instruction allowing flushing on a per-address basis is present in the underlying architecture, then this is automatically converted into a `HAL_MEMSYNC` by the low-level compiler.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the address that needs to be flushed out.

HAL_phyPtr address: The address of the address to flush, located directly in physical memory for low-level use.

*void *address*: The virtual address of the address to flush.

Used by:

Any TVM code (`_P` restricted to OS).

Described in:

Section 5.4

HAL_MEMFENCE_TEST_P**HAL_MEMFENCE_TEST**

```
HAL_bool HAL_MEMFENCE_TEST_P(  
    const int memory,  
    HAL_phyPtr phy_address);  
HAL_bool HAL_MEMFENCE_TEST(  
    const int memory,  
    void *address);
```

These routines perform a `HAL_MEMSYNC_TEST` operation, returning a Boolean value indicating whether or not buffers have been flushed out to memory that is visible to other processors, but only need to wait for all accesses to the supplied address to complete, instead of *all* accesses to *all* addresses. If no instruction allowing flushing on a per-address basis is present in the underlying architecture, then this is automatically converted into a `HAL_MEMSYNC_TEST` by the low-level compiler.

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the address that needs to be flushed out.

HAL_phyPtr address: The address of the address to flush, located directly in physical memory for low-level use.

*void *address*: The virtual address of the address to flush.

Returns:

(*HAL_bool*) TRUE if memory is now synchronized for the address in question (all private buffers flushed), or FALSE otherwise.

Used by:

Any TVM code (`_P` restricted to OS).

Described in:
Section 5.4

HAL_CACHE_INVALID_P

HAL_CACHE_INVALID

```
void HAL_CACHE_INVALID_P(
    const int memory,
    HAL_phyPtr phy_address,
    int num_bytes);
void HAL_CACHE_INVALID(
    const int memory,
    void *address,
    int num_bytes);
```

These routines take a region of cached memory and force it to become invalid. Because any modified data in the cache will be discarded as the lines are forced into the invalid state, these macros should be used with exceptional care. Versions are provided for managing areas using virtual addresses or direct physical addresses (low-level OS code only).

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the address where the cache-management operation will occur.

HAL_phyPtr address: The address of the region of memory where cache management should start, located directly in physical memory for low-level use.

*void *address:* The virtual address of the region of memory where cache management should start.

int num_bytes: The length of the region that needs to be managed. This may span across several cache lines, which will result in multiple low-level cache operations being initiated by a single macro invocation.

Used by:

Any TVM code (*_P* restricted to OS).

Described in:
Section 5.4

HAL_CACHE_FLUSH_P

HAL_CACHE_FLUSH

```
void HAL_CACHE_FLUSH_P(
    const int memory,
    HAL_phyPtr phy_address,
    int num_bytes);
void HAL_CACHE_FLUSH(
    const int memory,
    void *address,
    int num_bytes);
```

These routines take a region of cached memory and force it to become invalid. However, any modified data in the memory region is written back to main memory as the line is forced out of the cache. As a result, this is a sort of programmer-timed cache line eviction that may

be of use when transient data should be pushed out of a processor so that another may use it with less cache coherence overhead required. Versions are provided for managing areas using virtual addresses or direct physical addresses (low-level OS code only).

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the address where the cache-management operation will occur.

HAL_phyPtr address: The address of the region of memory where cache management should start, located directly in physical memory for low-level use.

*void *address:* The virtual address of the region of memory where cache management should start.

int num_bytes: The length of the region that needs to be managed. This may span across several cache lines, which will result in multiple low-level cache operations being initiated by a single macro invocation.

Used by:

Any TVM code (`_P` restricted to OS).

Described in:

Section 5.4

HAL_PREFETCH_READ_P**HAL_PREFETCH_READ**

```
void HAL_PREFETCH_READ_P(  
    const int memory,  
    HAL_phyPtr phy_address,  
    int num_bytes);  
void HAL_PREFETCH_READ(  
    const int memory,  
    void *address,  
    int num_bytes);
```

These routines take a region of cached memory and prefetch it into the local cache memory. This can greatly accelerate later loads that need to access the data. If other processors are holding the line, then it will be loaded in in the S (shared) state, but if no other processor is holding the line then it is loaded in as E (exclusive). Versions are provided for managing areas using virtual addresses or direct physical addresses (low-level OS code only).

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the address where the cache-management operation will occur.

HAL_phyPtr address: The address of the region of memory where cache management should start, located directly in physical memory for low-level use.

*void *address:* The virtual address of the region of memory where cache management should start.

int num_bytes: The length of the region that needs to be managed. This may span across several cache lines, which will result in multiple low-level cache operations being initiated by a single macro invocation.

Used by:

Any TVM code (`_P` restricted to OS).

Described in:

Section 5.4

HAL_PREFETCH_WRITE_P**HAL_PREFETCH_WRITE**

```
void HAL_PREFETCH_WRITE_P(  
    const int memory,  
    HAL_phyPtr phy_address,  
    int num_bytes);  
void HAL_PREFETCH_WRITE(  
    const int memory,  
    void *address,  
    int num_bytes);
```

These routines are the same as the `HAL_PREFETCH_READ` ones, except that they always load the lines into the local cache in the E (exclusive) state, knocking the region out of other processors' caches in the process. Hence, the lines are loaded in the cache and ready to be modified by stores without any further cache coherence protocol events. Versions are provided for managing areas using virtual addresses or direct physical addresses (low-level OS code only).

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the address where the cache-management operation will occur.

HAL_phyPtr address: The address of the region of memory where cache management should start, located directly in physical memory for low-level use.

*void *address*: The virtual address of the region of memory where cache management should start.

int num_bytes: The length of the region that needs to be managed. This may span across several cache lines, which will result in multiple low-level cache operations being initiated by a single macro invocation.

Used by:

Any TVM code (`_P` restricted to OS).

Described in:

Section 5.4

HAL_TOUCH_WRITE_P
HAL_TOUCH_WRITE

```
void HAL_TOUCH_WRITE_P(  
    const int memory,  
    HAL_phyPtr phy_address,  
    int num_bytes);  
void HAL_TOUCH_WRITE(  
    const int memory,  
    void *address,  
    int num_bytes);
```

These routines act much like `HAL_PREFETCH_WRITE`, except that the old state of the region is not actually loaded into the cache. Instead, other processors are forced to give up the region and a zeroed-out region set to the M (modified) state is loaded directly into the cache. This is a potentially much faster version of `HAL_PREFETCH_WRITE` for cases when the program does not care about the previous contents of the region, knowing in advance that the entire region will be overwritten by new data. Versions are provided for managing areas using virtual addresses or direct physical addresses (low-level OS code only).

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the address where the cache-management operation will occur.

HAL_phyPtr address: The address of the region of memory where cache management should start, located directly in physical memory for low-level use.

*void *address*: The virtual address of the region of memory where cache management should start.

int num_bytes: The length of the region that needs to be managed. This may span across several cache lines, which will result in multiple low-level cache operations being initiated by a single macro invocation.

Used by:

Any TVM code (`_P` restricted to OS).

Described in:

Section 5.4

HAL_CACHE_INVALID_INST_P
HAL_CACHE_INVALID_INST

```
void HAL_CACHE_INVALID_INST_P(  
    const int memory,  
    HAL_phyPtr phy_address,  
    int num_bytes);  
void HAL_CACHE_INVALID_INST(  
    const int memory,  
    void *address,  
    int num_bytes);
```

These routines invalidate a block of instructions into the instruction cache, which is useful when implementing self-modifying code such as a just-in-time compiler. Versions are

provided for managing areas using virtual addresses or direct physical addresses (low-level OS code only).

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the address where the cache-management operation will occur.

HAL_phyPtr address: The address of the region of memory where cache management should start, located directly in physical memory for low-level use.

*void *address:* The virtual address of the region of memory where cache management should start.

int num_bytes: The length of the region that needs to be managed. This may span across several cache lines, which will result in multiple low-level cache operations being initiated by a single macro invocation.

Used by:

Any TVM code (*_P* restricted to OS).

Described in:

Section 5.4

HAL_PREFETCH_INST_P HAL_PREFETCH_INST

```
void HAL_PREFETCH_INST_P(
    const int memory,
    HAL_phyPtr phy_address,
    int num_bytes);
void HAL_PREFETCH_INST(
    const int memory,
    void *address,
    int num_bytes);
```

These routines prefetch a block of instructions into the instruction cache. Versions are provided for managing areas using virtual addresses or direct physical addresses (low-level OS code only).

Inputs:

const int memory: The physical memory bank (identified by optimization number) containing the address where the cache-management operation will occur.

HAL_phyPtr address: The address of the region of memory where cache management should start, located directly in physical memory for low-level use.

*void *address:* The virtual address of the region of memory where cache management should start.

int num_bytes: The length of the region that needs to be managed. This may span across several cache lines, which will result in multiple low-level cache operations being initiated by a single macro invocation.

Used by:

Any TVM code (*_P* restricted to OS).

Described in:
Section 5.4

6.2.4.5 PERFORMANCE COUNTER MACROS

HAL_PERF_RESET_ALL **HAL_PERF_RESET**

```
void HAL_PERF_RESET_ALL();  
void HAL_PERF_RESET(  
    int type_select,  
    int counter_select);
```

These routines reset one or all of the performance counters on the processor to zero, in anticipation of taking measurements later. The single-counter version also configures the selected counter to count events of a particular type, and should thus be called at least once for each counter during initial configuration.

Input:

int type_select: This integer value chooses a performance counter in an architecture-neutral manner, using the values in Table 2. For convenience, constants are supplied:

Enumerated Name	Description
HAL_PERF_PROCESSOR_CYCLES	Processor cycles (time)
HAL_PERF_INSTRUCTION_COUNT	Instructions executed
HAL_PERF_I_CACHE_MISSES	Instruction cache misses
HAL_PERF_I_TLB_MISSES	Instruction TLB misses
HAL_PERF_LOAD_COUNT	Loads performed
HAL_PERF_DREAD_CACHE_MISSES	Data cache misses (load)
HAL_PERF_DREAD_TLB_MISSES	Data TLB misses (load)
HAL_PERF_STORE_COUNT	Stores performed
HAL_PERF_DWRITE_CACHE_MISSES	Data cache misses (store)
HAL_PERF_DWRITE_TLB_MISSES	Data TLB misses (store)
HAL_PERF_LDST_COUNT	Loads + stores performed
HAL_PERF_DATA_CACHE_MISSES	Data cache misses (all)
HAL_PERF_DATA_TLB_MISSES	Data TLB misses (all)
HAL_PERF_BRANCH_GOOD_PREDICT	Correctly predicted branches
HAL_PERF_BRANCH_MISPREDICT	Mispredicted branches

int counter_select: This integer value chooses from among multiple counters on a processor, if it contains more than one performance counter.

Used by:
Any TVM code.

Described in:
Section 5.6

HAL_PERF_RESET_EXCEPTION

```
HAL_uint64 HAL_PERF_RESET_EXCEPTION(
    int type_select,
    int counter_select,
    HAL_uint64 except_count);
```

Resets a performance counter on the processor so that a performance exception (#127) is triggered after *except_count* instances of the performance counter event have occurred (or the value returned, if *except_count* instances cannot be counted for some reason). It is otherwise like HAL_PERF_RESET.

Input:

int type_select: This integer value chooses a performance counter in an architecture-neutral manner, using the values in Table 2.

int counter_select: This integer value chooses from among multiple counters on a processor, if it contains more than one performance counter.

HAL_uint64 except_count: The number of events that should be counted before triggering a performance exception.

Returns:

(*HAL_uint64*) The actual number of events that will be counted before an exception occurs, which may be different than *except_count* if the performance counter cannot handle the given input value. Most frequently, it may be truncated down to a smaller value if the counter is too small. 0 is returned if exceptions are not possible on this architecture or with the selected type/counter combination.

Used by:

Any TVM code.

Described in:

Section 5.6

HAL_PERF_READ**HAL_PERF_READ_RESET**

```
HAL_uint64 HAL_PERF_READ(int counter_select);
HAL_uint64 HAL_PERF_READ_RESET(int counter_select);
```

These read one performance counter attached to a PCA processor, if present. The RESET variant resets the counter immediately afterwards so it may continue counting again starting at zero.

Input:

int counter_select: This integer value chooses from among multiple counters on a processor, if it contains more than one performance counter.

Returns:

(*HAL_uint64*) The current value of the performance counter, as a 64-bit unsigned number.

Used by:

Any TVM code.

Described in:
Section 5.6

HAL_PERF_PRESENT
HAL_PERF_EXCEPTION_OK

```
HAL_bool HAL_PERF_PRESENT(int type_select, int counter_select);  
HAL_bool HAL_PERF_EXCEPTION_OK(int type_select, int counter_select);
```

Verifies that a performance counter of interest is actually present on an architecture before code attempts to use it to take performance measurements, and verifies that the selected performance counter can actually count/cause an exception for the desired statistic, since some processors have heterogeneous sets of counters.

Input:

int type_select: This integer value chooses a performance counter in an architecture-neutral manner, using the values in Table 2.

int counter_select: This integer value chooses from among multiple counters on a processor, if it contains more than one performance counter.

Returns:

(*HAL_bool*) TRUE if this performance counter is present and can count/cause and exception for the desired statistic in the current architecture, FALSE if not.

Used by:

Any TVM code.

Described in:
Section 5.6

HAL_PERF_MAX_COUNT

```
HAL_uint64 HAL_PERF_MAX_COUNT(int type_select, int counter_select);
```

Determines the maximum number of events that can be counted by a particular performance counter.

Input:

int type_select: This integer value chooses a performance counter in an architecture-neutral manner, using the values in Table 2.

int counter_select: This integer value chooses from among multiple counters on a processor, if it contains more than one performance counter.

Returns:

(*HAL_uint64*) The maximum number of events from the selected type that can be counted by this particular performance counter.

Used by:

Any TVM code.

Described in:
Section 5.6

6.2.4.6 POWER CONTROL MACROS

HAL_GET_CLOCK_FREQ

```
HAL_uint64 HAL_GET_CLOCK_FREQ();
```

This gets the current processor clock frequency of the entire PCA chip.

Returns:

(*HAL_uint64*) The current processor clock value in Hertz.

Used by:

OS power-control routines.

Described in:

Section 5.7

HAL_SET_CLOCK_FREQ

```
void HAL_SET_CLOCK_FREQ(HAL_uint64 frequency);
```

Sets the current processor clock frequency of the entire PCA chip. It will set the actual hardware clock to the closest hardware-supported value that is *higher* than the value requested, so that the processor will deliver at least as much performance as was requested by the call.

Input:

HAL_uint64 frequency: This integer value chooses a new clock frequency (in Hertz) for the processor.

Used by:

OS power-control routines.

Described in:

Section 5.7

HAL_TEST_SET_CLOCK_FREQ

```
HAL_bool HAL_TEST_SET_CLOCK_FREQ();
```

Tests to see if the HAL_SET_CLOCK_FREQ call will work if executed on this processor.

Returns:

(*HAL_bool*) TRUE if this processor can set the clock frequency.

Used by:

OS power-control routines.

Described in:

Section 5.7

HAL_DOZE**HAL_DOZE_ALL****HAL_SLEEP****HAL_SLEEP_ALL**

```
void HAL_DOZE();
void HAL_DOZE_ALL();
void HAL_SLEEP();
void HAL_SLEEP_ALL();
```

These routines turn off the processor (or the entire PCA chip) completely until it receives an external interrupt. The **DOZE** command only turns off the processor to a limited extent. In this mode, the clocks and state of the system are maintained so that the processor can wake up immediately (on the order of ~10 clock cycles) after an interrupt is received. **SLEEP** lets the system shut down further, allowing clocks to completely turn off and state within the processor to be lost. In this extreme power-saving mode, the processor may require up to ~1 ms or so to restart, so this should only be used for long-duration shutdowns while no interrupts requiring a low response latency are expected. If processor-by-processor dozing or sleeping is not available, then processors will be put into idle loops by the single-processor versions.

Used by:

OS power-control routines.

Described in:

Section 5.7

HAL_TEST_DOZE**HAL_TEST_DOZE_ALL****HAL_TEST_SLEEP****HAL_TEST_SLEEP_ALL**

```
HAL_bool HAL_TEST_DOZE();
HAL_bool HAL_TEST_DOZE_ALL();
HAL_bool HAL_TEST_SLEEP();
HAL_bool HAL_TEST_SLEEP_ALL();
```

Tests to see if the corresponding sleep/doze routines work on the underlying processor architecture.

Returns:

(*HAL_bool*) TRUE if the corresponding power management routine is functional.

Used by:

OS power-control routines.

Described in:

Section 5.7

HAL_GET_TEMPERATURE

```
HAL_int32 HAL_GET_TEMPERATURE();
```

Measures the temperature of the processor chip, in order to allow software to make adjustments to the system cooling.

Returns:

(*HAL_uint32*) The current processor temperature in degrees Celsius multiplied by 256 (to allow measurements in increments of down to 1/256 of a degree, if the hardware supports resolution finer than a degree).

Used by:

OS power-control routines.

Described in:

Section 5.7

HAL_TEST_GET_TEMPERATURE

```
HAL_bool HAL_TEST_GET_TEMPERATURE();
```

Tests to see if the HAL_GET_TEMPERATURE call will work if executed on this processor.

Returns:

(*HAL_bool*) TRUE if this processor can get the temperature.

Used by:

OS power-control routines.

Described in:

Section 5.7

6.2.4.7 WATCHPOINT CONTROL MACROS

HAL_WATCHPOINT_COUNT

```
int HAL_WATCHPOINT_COUNT();
```

Returns the number of watchpoint registers that are available on this processor core.

Returns:

(*int*) The number of watchpoint registers available.

Used by:

Debuggers.

Described in:

Section 5.8

HAL_WATCHPOINT_TYPE_CHECK

```
HAL_bool HAL_WATCHPOINT_TYPE_CHECK(int type);
```

Checks to see whether or not the hardware watchpoint registers can handle one of three types of watchpoints: load, store, or load/store.

Input:

int type: This integer value chooses one of three types of watchpoint register modes to check for: load-only (HAL_WATCH_LOAD), store-only (HAL_WATCH_STORE), and load-and-store (HAL_WATCH_LOAD_STORE).

Returns:

(*HAL_bool*) Whether or not the hardware watchpoint registers support the selected type of operation.

Used by:

Debuggers.

Described in:

Section 5.8

HAL_SET_WATCHPOINT
HAL_CLEAR_WATCHPOINT

```
void HAL_SET_WATCHPOINT(int number, int type, void *address);  
void HAL_CLEAR_WATCHPOINT(int number);
```

These macros set or clear hardware watchpoints.

Input:

int number: Which watchpoint to set or clear, for architectures with multiple ones.

int type: This integer value chooses one of three types of watchpoint register modes to use for this watchpoint: load-only (HAL_WATCH_LOAD), store-only (HAL_WATCH_STORE), and load-and-store (HAL_WATCH_LOAD_STORE).

*void *address*: Pointer to the address to be “watched.” Some low-order bits may be truncated off of this address by the hardware if it is incapable of watching at a high enough resolution.

Used by:

Debuggers.

Described in:

Section 5.8

HAL_GET_WATCHPOINT_NUMBER

```
int HAL_GET_WATCHPOINT_NUMBER();
```

Returns the number of the watchpoint register that triggered the most recent watchpoint exception.

Returns:

(*int*) Number of the latest watchpoint register to trigger an exception.

Used by:

Debugger exception handlers.

Described in:

Section 5.8

HAL_GET_WATCHPOINT

```
void *HAL_GET_WATCHPOINT(int number);
```

Returns the address currently being “watched” by a watchpoint register. While it can be used anywhere, it is primarily of use during the handling of watchpoint exceptions.

Input:

int number: Which watchpoint to read, for architectures with multiple ones.

Returns:

(*void **) Address currently being “watched” by the selected register.

Used by:

Debugger exception handlers.

Described in:

Section 5.8

HAL_GET_WATCH_ADDRESS

```
void *HAL_GET_WATCH_ADDRESS();
```

Returns the address of the load or store instruction that triggered the latest watchpoint exception.

Returns:

(*void **) Address of the load or store that triggered the latest watch exception.

Used by:

Debugger exception handlers.

Described in:

Section 5.8

6.2.4.8 GENERIC REGISTER ACCESS MACROS

HAL_GENERIC_READ32**HAL_GENERIC_READ64**

```
HAL_uint32 HAL_GENERIC_READ32(int register_select);
```

```
HAL_uint64 HAL_GENERIC_READ64(int register_select);
```

These routines perform direct reads of the various special-purpose registers that are present on any architecture (segment registers on x86, CP registers on MIPS, sprs on PowerPC, *etc.*), so that any unusual registers not addressed by the previously-specified APIs can still be used in legal TVM-HAL code. Note that TVM-HAL code that utilizes these calls will not be portable. Essentially, these routines provide a guarantee that the TVM-HAL code can be used in cases where one or two architecture-specific details that cannot be abstracted by other HAL macros also cannot be avoided by low-level code that needs to perform a particular task.

Input:

int register_select: The number of the generic, architecture-specific register to read.

Because most of these registers are unusual, and may not have a “number” per se,

each implementation of the TVM-HAL will have an enumerated list of possible registers that can be accessed.

Returns:

(*HAL_uint32/64*) Contents of the processor-specific register, as a 32-bit or 64-bit integer, as per the original request from the programmer (which should match the natural size of the registers in the architecture).

Used by:

Any architecture-specific TVM code (mostly OS)

Described in:

Section 5.9

HAL_GENERIC_WRITE32**HAL_GENERIC_WRITE64**

```
HAL_GENERIC_WRITE32(  
    int register_select,  
    HAL_uint32 new_value);  
HAL_GENERIC_WRITE64(  
    int register_select,  
    HAL_uint64 new_value);
```

These are the opposite of the `HAL_GENERIC_READ`, writing new data into architecture-specific registers when necessary.

Inputs:

int register_select: The number of the generic, architecture-specific register to write. Because most of these registers are unusual, and may not have a “number” per se, each implementation of the TVM-HAL will have an enumerated list of possible registers that can be accessed.

HAL_uint32/64 new_value: The value to store into the architecture-specific register.

Used by:

Any architecture-specific TVM code (mostly OS)

Described in:

Section 5.9

6.3 User-Mode HAL routines

All HAL routines from the previous section are usable within privileged or supervisor-level code, but only a small subset is usable within normal user code. This section lists those exceptions in one convenient location.

- **HAL_LOAD_type, HAL_STORE_type**: These are usable at any time, whenever a pointer must be dereferenced by any code. (Sections 2.3 and 6.2.1)
- **HAL_SYSCALL, HAL_TRAP**: These are generally *only* used in user code, in order to switch to privileged execution mode when necessary (generally within library code). (Sections 3.1 and 6.2.2.2)

- **HAL_GET/SET_SP/FP/GP:** These are usable at any time, whenever one of the pointers needs to be modified. However, in most user code, they will probably only be generated automatically by the compiler. Any manual use can be very dangerous for user code, as the user-specified actions may conflict with compiler-generated ones. (Sections 3.2.4 and 6.2.2.3)
- **HAL Metadata Routines:** These are designed for use by low-level software, such as the OS, and therefore implementations may access protected memory or otherwise trigger faults if they are used directly in user code. However, if the OS wants to make the information visible to user applications, then the values provided by these calls could easily be passed up to user code with only a thin layer of code around the HAL calls. (Sections 4.2.1 and 6.2.3.1)
- **HAL_MUTEX Routines:** These can be used for user-level locks as well as OS ones. (Sections 5.3 and 6.2.4.3)
- **HAL_BARRIER Routines:** These can be used for user-level barriers as well as OS ones. (Sections 5.3 and 6.2.4.3)
- **Cache Line Control Macros:** These are usable at any time, whenever a program needs to directly control its caches (for prefetching, invalidation, *etc.*). User-level code may only be able to affect lines from its own address space, however. (Sections 5.4 and 6.2.4.4)
- **HAL_PERF_READ:** While most performance counter operations are limited to supervisor code, reading of the counters can often be done “on-the-fly” during user code. This may not work on all architectures, however, and should not be relied upon to work on all systems without an exception. (Sections 5.6 and 6.2.4.5)
- **HAL_GENERIC_READ/WRITE:** While most registers accessed by these functions will probably be privileged-mode only, there may be a few that are user-visible, such as floating point status registers. (Sections 5.9 and 6.2.4.8)

Functionality from other HAL routines might be accessed by user code through the system call interface, so that the OS can check for errors and apply any necessary inter-thread protection before making the HAL call for the user. An examples might be to allow some user control over the multiple-context processor routines or the `HAL_BLOCKMOVE` and `HAL_ACTIVE_BLOCKMOVE` routines. The latter would be very useful to user-level applications, but will require significant protection checking to ensure that buggy user code cannot cause the DMA system to hang up the system.

7 References

The ideas in the HAL relied upon input from four major types of sources: the previous TVM/SVM documents produced by the Morphware Forum, academic research into architectures and operating systems, books on commercial operating systems, and books on commercial architectures. The TVM/SVM documents set the stage for this work, the academic documents provided some ideas for extensions, and the commercial publications provided hard requirements that any potentially successful environment would have to meet. The references are listed by type in order to make the distinction about the nature of the various documents clear.

Morphware Forum Documents

Most of these documents are published by the PCA Morphware Forum [2] and are available at the Morphware web site at www.morphware.org.

1. The Polymorphous Computing Architectures (PCA) Program, Information Processing Technology Office (IPTO), Defense Advanced Research Projects Agency (DARPA), <www.darpa.mil/ipto/Programs/pca/index.htm>.
2. The PCA Morphware Forum. <www.morphware.org>.
3. “Introduction to Morphware: Software Architecture for Polymorphous Computing Architectures.” Version 1.0, February 23, 2004. Available at <www.morphware.org>.
4. “PCA Machine Model,” Version 1.2, July 2006.
5. “Streaming Virtual Machine Specification,” Version 1.2, July 2006.
6. “User-level TVM (UVM) Interface Specification,” September 3, 2003 revision, TRIPS Project, Department of Computer Sciences, The University of Texas at Austin.

Standards

7. “Portable Operating System Interface (POSIX)”, IEEE Standard 1003.1, 2004 Edition, Available at <www.ieee.org>.

Academic Documents

The following work most directly impacted the TVM-HAL model (in particular, with the default DMA handlers), and merits inclusion here.

8. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer, “Active Messages: A mechanism for integrated communication and computation,” *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, 1992, pp. 256–266.

Other academic work has had more indirect influence on this specification, but not enough to be listed.

Operating System / Run-time Architecture Books

The largest number of concepts encompassed in the HAL design come from the basic design of operating systems. The following selection describes some of the basic books on the topic, and a few more specialized ones that helped in the design of the HAL.

9. A. Silberschatz, G. Gagne, and P. B. Galvin, *Operating System Concepts*, Sixth Edition, Addison-Wesley, 2002.
10. A. S. Tanenbaum and A. S. Woodhull, *Operating Systems: Design and Implementation*, Second Edition, Prentice-Hall, 1997.
11. M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley, 1996.
12. M. Beck, H. Böhm, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner, *Linux Kernel Internals*, Addison-Wesley, 1996.
13. J. Labrosse, *MicroC/OS-II: The Real-Time Kernel*, Lawrence, KS: R&D Books, 1999.
14. E. Farquhar and P. Bunce, *The MIPS Programmers Handbook*, San Francisco: Morgan Kaufmann, 1994.

Examination of source code for some UNIX variants also contributed to the design of the TVM-HAL. Reference [13] noted above describes a simple real-time OS, essentially the only open-source real-time kernel available. Examination of this system made it evident that the low-level machine interfaces in real-time OSes were essentially identical to normal operating systems like Linux. The real differences were in what the scheduler *does* at every timer interrupt. This helped influence the level where the TVM-HAL is set, because a design constraint was to allow both normal and real-time operating systems to be able to use it.

Processor Architecture Books

Along with operating system architecture, one needs to understand a fair amount about processor architecture, and especially the privileged instructions available on processor architectures, in order to understand the HAL. Here are a few suggestions, but the list is far from all-inclusive (in particular, there are untold numbers of books on the x86 architecture):

15. Motorola and IBM Microelectronics, *The PowerPC Microprocessor Family: The Programming Environments*, Motorola Databook, 1994.
16. Motorola, *AltiVec™ Technology Programming Environments Manual*, Motorola Databook, 2001.
17. J. Heinrich, *MIPS R4000 Microprocessor User's Manual*, Englewood Cliffs, NJ: Prentice-Hall, 1993.
18. R. Sites, *Alpha Architecture Reference Manual*, Digital Equipment Corporation, 1992.
19. Motorola, *MC68000 Family Reference Manual*, Motorola Databook, 1990.

The various architectures described in these books agree on many features, but some differences were present that the TVM-HAL model tries to isolate as much as possible. While the specification mostly focused on the common features supported by several RISC architectures, some comparisons were made to requirements of older, CISC architectures to test how flexible the specification could be. The 68000 was chosen as a typical CISC, because it is a fairly clean CISC architecture that is still common in embedded environments. In particular, hardware-prioritized interrupts were supported much better on CISC architectures like the 68K than in most RISC architectures prevalent today. As a result, prioritized interrupts are included in the

TVM-HAL specification so they may be used when available, because emulating them in the HAL macros is not too expensive even if they are not inherently supported.

8 Index of HAL Directives, Functions, and Constants

HAL_ACTIVATE	24, 78, 79
HAL_ACTIVE_BLOCKMOVE	44, 106
HAL_ACTIVE_BLOCKMOVE_P	43, 106
HAL_ANY_MEMORY	5
HAL_BARRIER_INIT	45, 111
HAL_BARRIER_INIT_P	45, 111
HAL_BARRIER_START	45, 112
HAL_BARRIER_START_P	45, 112
HAL_BLOCKMOVE	12, 43, 105
HAL_BLOCKMOVE_P	43, 105
HAL_CACHE_COHERENT	40, 101
HAL_CACHE_FLUSH	47, 115
HAL_CACHE_FLUSH_P	46, 115
HAL_CACHE_INVALID	47, 115
HAL_CACHE_INVALID_P	46, 115
HAL_CACHEABLE	40, 101
HAL_CC_D	28
HAL_CC_I	28
HAL_CC_R	28
HAL_CC_RD	28
HAL_CC_RI	28
HAL_CC_RID	28
HAL_CC_RIDU	28
HAL_CC_RU	28
HAL_CC_U	28
HAL_CLEAR_PAGE	40, 42, 97, 98
HAL_CLEAR_SEGMENT	35, 90
HAL_DEACTIVATE	24, 79
HAL_DEACTIVATE_OTHER	24, 79
HAL_DMA_DISABLE	12, 108

HAL_DMA_ENABLE	12, 44, 108
HAL_DMA2VECTORDNUM.....	12, 108
HAL_DOZE.....	50, 124, 126, 127
HAL_DOZE_ALL	50, 124, 126
HAL_EX_RETURN	19, 69, 70, 71, 73, 74, 75
HAL_EX_RETURNX	15, 17, 19, 69, 70, 71, 73, 74, 75
HAL_EXCEPTION_CLEAR.....	11, 12, 60
HAL_EXCEPTION_ENABLE	11, 12, 44, 59
HAL_EXCEPTION_GET_ENABLED.....	11, 61
HAL_EXCEPTION_GET_HANDLER	11, 61
HAL_EXCEPTION_GET_PRIVILEGE.....	11, 61
HAL_EXCEPTION_HDL	14, 15, 17, 22, 53
HAL_EXCEPTION_PC	19, 20, 69, 71, 72, 74
HAL_EXCEPTION_REENABLE	11, 12, 59
HAL_EXCEPTION_USABLE.....	11, 12, 61
HAL_FAULT_IFETCH	39, 95
HAL_FAULT_LOAD	39, 95
HAL_FAULT_OTHER	39, 95
HAL_FAULT_SIZE	39, 95
HAL_FAULT_STORE.....	39, 95
HAL_FAULT_TYPE	39, 95
HAL_GENERIC_READ	128
HAL_GENERIC_READ32	51, 127
HAL_GENERIC_READ64	51, 127
HAL_GENERIC_WRITE32	51, 128
HAL_GENERIC_WRITE64	51, 128
HAL_GET_CLOCK_FREQ.....	49, 123, 124, 125
HAL_GET_FAULT_ADDRESS	39, 41, 95
HAL_GET_FP	15, 63, 64, 65
HAL_GET_GP	15, 62, 63, 64, 65
HAL_GET_MEM_CHARACTERISTIC_BY_NUM.....	32, 36, 83, 87, 92
HAL_GET_NEXT_PAGE	41, 103

HAL_GET_PAGE_MODIFIED.....	41, 42, 103
HAL_GET_PAGE_TOUCHED	41, 42, 103
HAL_GET_PHY_PAGE	41, 102
HAL_GET_PREVIOUS_PAGE.....	41, 103
HAL_GET_PRIVILEGE_LEVEL	18, 66
HAL_GET_RESET_PAGE_MODIFIED	41, 103
HAL_GET_RESET_PAGE_TOUCHED.....	41, 103
HAL_GET_SP	15, 62, 63, 64, 65
HAL_GLOBAL_BLOCK.....	6, 54, 55
HAL_GLOBAL_BLOCK_BASE	6, 54
HAL_GLOBALS.....	6, 55, 56
HAL_GLOBALS_INIT	6, 7, 55, 56
HAL_GLOBALS_UNINIT	6, 7, 55, 56
HAL_GP_OFF.....	16, 17, 18, 53, 56
HAL_GP_ON	16, 17, 18, 22, 53, 56
HAL_HAS_TLB.....	34, 89
HAL_IGNORED	101
HAL_INIT_CONTEXT.....	24, 78
HAL_INIT_PAGETABLE	40, 96
HAL_INIT_STATE.....	14, 15, 63, 78
HAL_INTERRUPTS_ARE_ENABLED	18, 66
HAL_INTERRUPTS_OFF.....	10, 18, 66, 67
HAL_INTERRUPTS_ON	10, 18, 22, 66, 67
HAL_INTERRUPTS_TO.....	25, 81
HAL_INTERRUPTS_WERE_ENABLED	19, 20, 70, 72
HAL_LOAD	3, 7, 8, 26, 57, 58
HAL_LOAD_P	7, 57
HAL_LOAD_TABLE_BASE	39, 96
HAL_MASK_ABOVE	18, 68
HAL_MASK_POINT_AT.....	18, 68
HAL_MEMFENCE	46, 113
HAL_MEMFENCE_P.....	46, 113

HAL_MEMFENCE_TEST.....	46, 114
HAL_MEMFENCE_TEST_P	46, 114
HAL_MEMSYNC	46, 113, 114
HAL_MEMSYNC_TEST.....	46, 113, 114
HAL_MUTEX_INIT	45, 109
HAL_MUTEX_INIT_P	45, 109
HAL_MUTEX_TRYLOCK	45, 109, 110
HAL_MUTEX_TRYLOCK_P	45, 110
HAL_MUTEX_UNLOCK	45, 109, 110
HAL_MUTEX_UNLOCK_P	45, 110
HAL_NUM_CONTEXTS	24, 78
HAL_NUM_MEMORIES.....	83, 87, 92
HAL_OVERWRITE_OFF	14, 15, 17, 53
HAL_OVERWRITE_ON	14, 15, 17, 21, 22, 53
HAL_PAGE_PRESENT.....	41, 102
HAL_PAGING_CHECK.....	40, 101
HAL_PERF_PRESENT	48
HAL_PERF_READ.....	48, 121, 122
HAL_PERF_READ_RESET.....	48, 121, 122
HAL_PERF_RESET.....	48, 120, 121
HAL_PERF_RESET_ALL.....	48, 120, 121
HAL_PREFETCH_INST	47, 118, 119
HAL_PREFETCH_INST_P	47, 118, 119
HAL_PREFETCH_READ	47, 116, 117
HAL_PREFETCH_READ_P	46, 116
HAL_PREFETCH_WRITE.....	47, 117, 118
HAL_PREFETCH_WRITE_P	47, 117
HAL_PRIV_EXECUTABLE.....	28, 40, 101
HAL_PRIV_READABLE.....	28, 40, 101
HAL_PRIV_WRITABLE.....	28, 40, 101
HAL_PRIVILEGED.....	12, 60, 61, 67, 68, 69, 70, 71, 72, 73, 74, 75, 79
HAL_PROCESSOR	21, 52

HAL_PROCESSOR_IDENT	43, 104
HAL_RESET_HDL	21, 22, 53
HAL_RESTORE_STATE	14, 15, 17, 63
HAL_SAVE_STATE	14, 15, 17, 63
HAL_SEND_INTERRUPT_TO	25, 82
HAL_SET_CLOCK_FREQ	49, 123, 125
HAL_SET_FP	15, 17, 22, 64, 65
HAL_SET_GP	15, 17, 18, 22, 56, 64, 65
HAL_SET_PAGE	40, 98, 100
HAL_SET_PRIVILEGE_LEVEL	18, 67
HAL_SET_SEGMENT	21, 35, 90, 94, 100
HAL_SET_SP	15, 17, 22, 64, 65
HAL_SLEEP	50, 124
HAL_SLEEP_ALL	50, 124
HAL_STACK_LOCATION	5, 54
HAL_STACK_OFF	16, 17, 18, 53
HAL_STACK_ON	16, 17, 18, 22, 53
HAL_STORE	3, 7, 57, 58
HAL_STORE_P	7, 57
HAL_SUPPORTED	101
HAL_SWITCH	24, 80
HAL_SWITCH_TO	24, 80
HAL_SWITCH_TO_SVM	23, 77
HAL_TEST_FOR_SVM	23, 77
HAL_TEST_INTERRUPTS_TO	25, 81, 82
HAL_TEST_SEND_INTERRUPT_TO	25, 82
HAL_TEST_SET_CLOCK_FREQ	49, 123, 125
HAL_TEST_TIMER_SEPARATE	25, 82
HAL_TEST_TIMER_SET	20, 77
HAL_TEST_TIMER_SWITCH	20, 77
HAL_TEST_TIMER_TO	25, 81, 82
HAL_THREAD_ACTIVE	24, 80

HAL_THREAD_PRIORITY_GET.....	24, 80
HAL_THREAD_PRIORITY_SET.....	24, 81
HAL_TIMER_FREQUENCY_GET	20, 76
HAL_TIMER_FREQUENCY_SET.....	20, 76
HAL_TIMER_OFF.....	20, 76
HAL_TIMER_ON	20, 76
HAL_TIMER_TO.....	25, 81, 83
HAL_TLB_INVALIDATE	39, 42, 96
HAL_TLB_INVALIDATE_ALL.....	39, 96
HAL_TOUCH_WRITE.....	47, 118
HAL_TOUCH_WRITE_P.....	47, 118
HAL_USE_GP1.....	16, 17, 54, 56, 57
HAL_USE_GP2.....	16, 17, 18, 22, 54, 56, 57
HAL_USER	12, 60, 61, 67, 69, 70, 71, 72, 73, 74, 75, 79
HAL_USER_EXECUTABLE.....	28, 40, 101
HAL_USER_READABLE.....	28, 40, 101
HAL_USER_WRITABLE	28, 40, 101
HAL_VECTOR2DMANUM.....	12, 108
HAL_WAS_PRIVILEGE_LEVEL	19, 20, 70, 71, 73, 75
HAL_xCACHE_ENABLE	21, 36, 91, 93, 100
HAL_xCACHE_OFF	36, 88, 94
HAL_xCACHE_OFF_SEG.....	36, 88, 94
HAL_xCACHE_ON	36, 88, 94
HAL_xCACHE_ON_SEG	36, 88, 94

APPENDIX F

Last Available Version of the Morphware Stable Interface Document

USER-LEVEL TVM (UVM) INTERFACE SPECIFICATION

Version 0.9a

March 16, 2005

User-level TVM (UVM) Interface Specification

Version 0.9a

March 16, 2005



©2004 Georgia Tech Research Corporation, all rights reserved.

A non-exclusive, non-royalty bearing license is hereby granted to all persons to copy, modify, distribute and produce derivative works for any purpose, provided that this copyright notice and following disclaimer appear on all copies: THIS LICENSE INCLUDES NO WARRANTIES, EXPRESSED OR IMPLIED, WHETHER ORAL OR WRITTEN, WITH RESPECT TO THE SOFTWARE OR OTHER MATERIAL INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF PERFORMANCE OR DEALING, OR FROM USAGE OR TRADE, OR OF NON-INFRINGEMENT OF ANY PATENTS OF THIRD PARTIES. THE INFORMATION IN THIS DOCUMENT SHOULD NOT BE CONSTRUED AS A COMMITMENT OF DEVELOPMENT BY ANY OF THE ABOVE PARTIES.

This material is based in part upon work supported by the U.S. Defense Advanced Research Projects Agency (DARPA) and other agencies of the U.S. Department of Defense (DoD). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or the DoD.

The US Government has a license under these copyrights, and this material may be reproduced by or for the U.S. Government.



This page deliberately left blank.

Acknowledgements

Authors

The primary authors of this document were:

Doug Burger	The University of Texas at Austin
Mike Dahlin	The University of Texas at Austin

The authors wish to thank the members of the Morphware Forum who reviewed and contributed to this document. The authors also thank the Defense Advanced Research Projects Agency (DARPA) for their support of this work.

The Morphware Forum

The Morphware Forum is a joint activity of the participants in DARPA's Polymorphous Computing Architectures (PCA) program, as well as other interested developers of embedded computing hardware, software, and application technology. The purpose of the Morphware Forum is to define an open, portable software environment for the development of high performance applications on PCA platforms. Morphware Forum products and information are available at www.morphware.org.

The following organizations are voting members of the Morphware Forum at this writing:

- o Defense Advanced Research Projects Agency
- o Georgia Institute of Technology
- o Lockheed Martin Advanced Technology Laboratory
- o Mercury Computer Systems, Inc.
- o Massachusetts Institute of Technology
- o MIT Lincoln Laboratory
- o MPI Software Technology, Inc.
- o Protean Devices, Inc.
- o Reservoir Labs, Inc.
- o Stanford University
- o University of Texas at Austin
- o University of Southern California Information Sciences Institute

Additional participating organizations include:

- o Air Force Research Laboratory
- o Applied Photonics
- o BAE Systems
- o Brigham Young University
- o California Institute of Technology
- o George Mason University
- o IBM Austin Research Laboratory
- o IBM T. J. Watson Research Center
- o Lockheed Martin Aerospace
- o Lockheed Martin NE&SS
- o Los Alamos National Laboratory
- o Mississippi State University
- o North Carolina State University
- o Northrop Grumman
- o Raytheon
- o Sandia National Laboratories
- o Southwest Research Institute
- o Space and Naval Warfare Systems Center San Diego
- o University of California, Berkeley
- o University of California, Irvine
- o University of Illinois at Urbana-Champaign
- o University of Maryland
- o University of Pennsylvania
- o Vanderbilt University
- o Vermont University
- o VLSI Photonics

Document Change History

Version 1.0 is the first approved version of the document.

Table of Contents

Acknowledgements	iii
Authors	iii
The Morphware Forum	iii
Document Change History	v
Table of Contents	vi
Table of Acronyms	vii
1. Introduction	1
2. Variable usage transformations.....	3
3. Processor control and exception handling	4
3.1. Exception vector table system	4
3.2. Special code in exception handlers.....	4
3.3. Morphing in this model.....	4
3.4. Scheduler activations.....	4
3.5. Slot Scheduling	7
4. Memory control model.....	13
4.1. Physical memory view	13
4.2. Segment control.....	13
4.3. Paged memory control.....	17
5. Other Extensions.....	18
5.1. Active DMA	18
5.2. Multiprocessor synchronization.....	18
5.3. Cache Memory Control	21
5.4. IEEE 754 Floating Point Control	21
5.5. Performance Counters/Run-time Metadata.....	21
5.6. Generic Special Register Access.....	21
6. UVM Reference.....	22
6.1. Scheduler Functions	23
6.2. Segment Control Functions.....	34
7. References.....	40

Table of Acronyms

API	Application Programming Interface
DARPA	Defense Advanced Research Projects Agency
DMA	Direct Memory Access
HAL	Hardware Abstraction Layer
I/O	Input/Output
MSI	Morphware Stable Interface
PA	Physical Address
PCA	Polymorphous Computing Architectures
POSIX	Portable Operating System Interface
PID	Process Identification (ID)
RAM	Random Access Memory
SA	Scheduler Activation
SVM	Streaming Virtual Machine
TLB	Translation Lookaside Buffer
TVM	Threaded Virtual Machine
UVM	User-Level TVM
VA	Virtual Address

1. Introduction

A number of communication-exposed architectures are being developed as part of the Polymorphous Computing Architectures (PCA) initiative [1]. These machines can be configured to meet the demands of the application and the constraints of the environment. Mapping an application to these architectures is a challenging compiler problem. To obtain good performance, the compiler needs to detect data, task, and pipeline parallelism in the source program and calculate a load-balanced mapping onto the processor resources, all while taking into account the large array of possible streaming morphs.

To address this challenge, the Morphware Forum [2] is defining elements of a portable application development methodology for architectures of the PCA program that includes new source languages, a new application development process, and a framework for expressing system and application metadata. One element of the methodology, referred to as the Morphware Stable Interface (MSI) [3], is a strategy that partitions the compilation process into two stages: a high-level compilation stage and a low-level compilation stage.

The high-level compilers accept application source code and the description of an architecture via a PCA Machine Model [x], and produce a mapping to the physical resources available in the given architecture. The mapping is expressed as a virtual machine language based for either threaded or streaming processing. Streaming parts of the application use the Streaming Virtual Machine (SVM) Application Programming Interface (API), while threaded parts of the application using an API based on either a user-level Threaded Virtual Machine (UVM) or a low-level Threaded Virtual Machine Hardware Abstraction Layer (TVM-HAL). This virtual-machine code is later compiled into executable code by architecture-specific low-level compilers.

This document describes a “virtual machine” interface for user-level threaded PCA libraries and programs. This document complements the TVM-HAL Specification [4], which provides a similar interface for privileged or operating system run-time system code.

This interface differs slightly from the TVM-HAL interface in that it does not assume complete access to all machine resources. As a result, it provides interfaces that allow schedulers to react when resources are taken away, and it provides interfaces for mapping memory when other processes may also be doing the same thing. Of course, if a process is lucky enough to run in an environment where it is the only process on the machine, that’s fine too. Thus, applications written to this interface should be portable to both “thin” run-time systems that dedicate an entire machine to one process and “thicker” operating systems that multiplex a machine across processes as well as “thin kernel/fat library” systems that provide minimal multiplexing mechanisms while leaving policy decisions to application libraries.

Note that although this UVM interface will typically be built over the TVM-HAL interface, this UVM interface is still quite low level. Generally, applications programmers will not access these functions directly. Instead libraries that provide higher-level functionality will generally access these functions. For example, a user-level scheduler library might run above the low-level scheduler activations interface described here.

Thus, for portability, we anticipate UVM run-time systems and libraries being built on the TVM-HAL and exporting the interfaces defined here. Then user-level libraries are built on these UVM interfaces and user-level programs run over these library interfaces.

This document's organization follows the organization of the TVM-HAL specification (August 2004 Revision). In each section, we note which HAL interfaces are exported unchanged to the UVM, which are changed, and which are unavailable. We also note any related new interfaces provided by the UVM.

2. Variable usage transformations

The TVM-HAL specifies ways that code should refer to local variables (Section TVM-HAL-2.1), global variables (Section TVM-HAL-2.2), and pointer-based access (Section TVM-HAL-2.3). These interfaces can be thought of as “macros” that the compiler outputs to annotate code. These TVM-HAL interfaces are available unchanged to user-level processes on the UVM.

3. Processor control and exception handling

The TVM-HAL document defines an exception-handling model in Section 3. Similar low-level interfaces are available for user-level programs. In addition, *scheduler activations* provide a slightly higher-level interface for coping with and influencing processor-scheduling decisions by the run-time system.

3.1. Exception vector table system

Section 3.1 of the TVM-HAL document defines a set of up to 255 exception vectors and a way to enable/disable delivery of these exceptions. The same interfaces are available to user-level programs via the UVM interface with one small change: the arguments to `HAL_EXCEPTION_ENABLE(int entry_number, CodeVPtr handler_pc)` do not include the third argument (`int privilege_level`).

3.2. Special code in exception handlers

Section 3.2 of the TVM-HAL document defines a set of compiler macros and run-time hooks to provide methods for saving/restoring processor state in exception handlers. The same interfaces are available to user-level programs via the UVM interface except that `HAL_SET_PRIVILEGE_LEVEL( )` is not available to UVM applications.

Note that system control (TVM-HAL Section 3.2.5) is per-processor and per-process. For example, `HAL_INTERRUPTS_ON( )` and `HAL_INTERRUPTS_OFF( )` turn interrupts on/off only for the current processor and the calls for finding out about the “last” or “most recent” interrupt (e.g., the `HAL_EXCEPTION_PC( )` return information that is maintained on a per-processor basis).

Note that run-time systems that implement thread migration must ensure threads that use processor context-dependent facilities are non-migratable before the function call, and remain non-migratable until any context-dependent work associated with the call is complete. If this atomicity is violated it could result in errors; for instance failing to restore interrupt handling on processors where interrupts were meant to be temporarily inhibited.

3.3. Morphing in this model

The TVM-HAL document describes a `HAL_SWITCH_TO_SVM( )` procedure that allows a processor to convert itself to streaming mode under control of a specified master. This interface is also available to user-level programs.

3.4. Scheduler activations

Scheduler activations [5] provide a communication interface between the kernel scheduler (if any) and the user-level scheduler to allow user-level schedulers to react to changes in the number of available processors, runnable threads, etc. This interface allows user-level libraries and applications to react to changes in allocations by the kernel scheduler; this interface is therefore applicable only to the user-level interface.

The scheduler activations interface is organized around four upcalls from the kernel scheduler to the user-level-scheduler code. These upcalls are indexed as follows:

```
typedef enum {
    uvm_saHasBeenPreempted = 0,
    uvm_saAddProcessor = 1,
    uvm_saHasBlocked = 2,
    uvm_saHasUnblocked = 3
} uvm_saUpcallIndex;
```

The upcall signatures are described in Table 1.

Number	Symbolic	Call signature	When Called
0	<code>uvm_saHasBeenPreempted</code>	<code>has_been_preempted(int preemptedProcessor, uvm_saArchitectureSaveBlock *preemptedThreadState)</code>	A processor has been taken away from the process
1	<code>uvm_saAddProcessor</code>	<code>add_processor(int processorID, uvm_saArchitectureSaveBlock *bufferThatWillBeUsed)</code>	A processor has been given to the process
2	<code>uvm_saHasBlocked</code>	<code>has_blocked(int interruptedProcessor, uvm_saThreadID token)</code>	A thread has blocked in the kernel (e.g., for I/O)
3	<code>uvm_saHasUnblocked</code>	<code>has_unblocked(uvm_threadID token uvm_saArchitectureSaveBlock *threadState)</code>	A previously blocked thread is runnable

Table 1 Scheduler activation kernel upcalls to user-level scheduler

In addition, there are nine user-level-scheduler initiated calls

```
/* Set up handlers for upcalls */
int uvm_saHandlerSet(uvm_saUpcallIndex entryNumber, CodeVPtr handlerPC);
int uvm_saSetMaxProcessors(int max);
int uvm_saStateBufferSet(int procID,
    uvm_saArchitectureSaveBlock *buffer);

/* Enable/disable SA upcalls */
int uvm_saEnable(int procID);
int uvm_saDisable(int procID);

/* Ask kernel to allocate/deallocate processors to this process */
int uvm_saRequestProcessor (int procID);
int uvm_saReleaseProcessor();
```

```
/* Manage and use uvm_saArchitectureSaveBlock tokens */  
size_t uvm_saArchitectureSaveBlockSize()  
void uvm_saResumeThread(uvm_saArchitectureSaveBlock *buffer)
```

The type `CodeVPtr` is defined in the TVM-HAL document.

A `uvm_saArchitectureSaveBlock` is an opaque, implementation-dependent data type that stores the architectural state of a thread (or, in some implementations, is a reference to the stored architectural state of a thread); its size is `uvm_architecturalSaveBlockSize()` and the only functions operating on this object are defined above.

A `uvm_saThreadID` is an implementation-dependent integer type used as an identifier for a suspended thread. A test-for-equality of thread ID's may be done using the equality operator.

Initialization. When a process is created, all handlers are null and all state buffers are null. As long as *any* handler or state buffer (with ID < max processors requested) is null, scheduler activations are not enabled, and the kernel will silently select, run, and suspend a process's kernel threads according to its scheduling policy. Once all are set, a process may call `uvm_saEnable` to enable scheduler activations.

The function `uvm_saHandlerSet()` is called to set pointers for each of the handler functions. It returns 0 on success or a negative value on error. The error codes are `uvm_e_saBadRange` if the `entryNumber` is not a legal value, or `uvm_e_saNeedDisable` if this call is made when scheduler activations are enabled at any processor for the process.

The function `uvm_saSetMaxProcessors()` is called to specify the maximum number of kernel threads (aka user-level-scheduler instances) that a process plans to use. It returns 0 on success or a negative value on error. The error codes are `uvm_e_saBadRange` if the max is not at least 1, or `uvm_e_saNeedDisable` if this call is made when scheduler activations are enabled at any processor for the process.

The function `uvm_saStateBufferSet()` is called to provide buffers to which callbacks can save the state of preempted threads. A process must allocate one `uvm_saArchitectureSaveBlock` buffer of size `uvm_saArchitectureSaveBlockSize` bytes per kernel thread up to the maximum number of kernel threads (processors) the process will use, and it must bind each buffer to a kernel thread using the `uvm_saStateBufferSet()` call. If `procID` is `uvm_saProcAny = -1` then the buffer is globally visible and can be bound by the run-time system to any processor. If `procID` is a non-negative integer, then the buffer is only visible to the specified processor. The function returns 0 on success or a negative value on error. The error codes are `uvm_e_saBadRange` if the ID is negative and not `uvm_saProcAny`, or `uvm_e_saNeedDisable` if this call is made when scheduler activations are enabled at any processor for the process.

Enable/disable. Enable/disable calls are used for initialization or reconfiguration (e.g., changing the maximum number of kernel threads a processor may want or changing a handler.) They are also used for deadlock avoidance on handler calls. When a process is created, scheduler activations are globally disabled for the process. Once the handlers are initialized (see above), a process can globally enable scheduler activations for the process. When scheduler activations are

globally enabled, callbacks can be disabled on a per-processor basis. This is done to avoid deadlocks between callbacks from the kernel that may access locks in the user-level scheduler and other calls that access those locks [1]. When a callback does occur, the state is disabled for the scheduler instance upon entry to the callback.

The function `uvm_saEnable()` is called with argument `uvm_saGlobal = -2` or with a non-negative value. In the first case, we are activating scheduler activations for a process and in the second we are reactivating SA for a particular processor. The call returns 0 on success or a negative value on error. The error codes are `uvm_e_saNotInitialized` if any of the handlers or buffers have not yet been initialized, or `uvm_e_saBadRange` if the `procID` is negative and not `uvm_saGlobal`.

The function `uvm_saDisable()` is called with argument `uvm_saGlobal` or with non-negative value. In the first case, we are deactivating scheduler activations for a process. In the second case, we are deactivating scheduler activations for one processor. The call returns 0 on success or a negative value on error. The error `uvm_e_saBadRange` is returned if the `procID` is negative and not `uvm_saGlobal`.

Request/yield processors. These calls communicate to the kernel scheduler the desire of a process to take more/fewer processors. Note that this interface is intended as a simple initial resource management facility in keeping with the classic scheduler activations interface. Additional facilities may be needed to provide user-level schedulers more precise influence over the decisions of kernel schedulers in order to facilitate gang scheduling and real time scheduling.

The function `uvm_saRequestProcessor()` is called to indicate that the process could make productive use of an additional physical processor. If a processor is or becomes available and if fewer than max kernel threads are active for the process the kernel scheduler MAY subsequently invoke the `add_processor()` upcall (defined above.) The `procID` argument is a hint expressing a desire for a particular processor; the special value `uvm_saProcAny` may be included to indicate no preference. The kernel scheduler is not obligated to obey this hint. The call returns 0 on success or a negative value on error. The error codes are `uvm_e_saBadRange` if the `procID` is neither `uvm_saProcAny` nor the identifier of any processor, or `uvm_saBadBuffer` if there is not currently a buffer that can be legally mapped to the requested processor.

The function `uvm_saReleaseProcessor()` is called to indicate that the calling kernel thread no longer needs the processor. Normally, it does not return. If it returns, there has been an error and a negative value indicating the nature of the error is returned. No error codes have been specified at this time.

Resume thread. After a `uvm_saHasUnblocked` call indicates that a thread is runnable, a user-level scheduler may call `uvm_saResumeThread()` to resume execution of a previously suspended thread. This call does not normally return; it only returns on error.

3.5. Slot Scheduling

To facilitate the coexistence of multiple classes of user level schedulers, such as gang and real time schedulers, the UVM model includes a *slot scheduling* interface. Slot scheduling provides a way for applications to precisely specify both when and where they should be selected to run by

a kernel or other privileged scheduler. The system maintains a globally visible schedule of future time-slice reservations (slots) for each processor. Each slot records a decision made in advance about which process will run during the corresponding time slice. Applications can view which slots are available across the system and place requests for slots in which they would like to run.

The slot scheduling interface consists of the slot array data structures, three calls that directly manipulate the slot schedule and three calls controlling allocation policy. Three additional system calls are defined to provide for the kernel to report to the user program the address and length of the read-only slot schedule array (`uvm_CpuSlot *slotArray`), to allow the user program to query the kernel for the index of the next slot in the schedule, and to allow the user to determine the length of a time slice.

```
/* Individual slot data structure */
typedef struct {
    int ownerPID;
    int scheduleePID;
    int priority;
} uvm_CpuSlot;

/* Spec for individual slot request */
typedef struct {
    int processorID;
    int index;
    int scheduleePID;
    int priority;
} uvm_CpuSlotRequest;

/* System-defined constants */
const int uvm_maxTokenPriority;

/* Scheduling system calls */
int uvm_claimCpuSlots(
    uvm_CpuSlotRequest *slotSpec,
    int *length,
    bool notifyOwnerOnPreempt,
    CodeVPtr handler);

int uvm_freeCpuSlots(
    uvm_CpuSlotRequest *slotSpec,
    int *length);

int uvm_reserveCpuSlots(
    int processorID,
    int numerator,
    int denominator,
    bool immediate);

/* Banking system calls */
int uvm_getTokenCounts(
    int processID,
    int *tokenCount);
```

```
int uvm_donateTokens(
    int recipientPID,
    int count,
    int priority);

int uvm_changeTokenCount(
    int processID,
    int count,
    int priority,
    bool absolute);

/* Miscellaneous system calls */
uvm_CpuSlot *uvm_getCpuSlotArray(
    int *length);

int uvm_getNextCpuSlotIndex();
int uvm_getCpuSlotDurationMs();
```

3.5.1 Slot arrays and kernel scheduler

For each processor, the kernel (or whichever privileged entity is responsible for low-level scheduling decisions) maintains an array of `uvm_CpuSlots` representing consecutive future timeslices. Each slot records the PID of the process that currently owns it (`ownerPID`) and the PID of a process that the owner has nominated to run during the corresponding timeslice (`scheduleePID`). Decisions about which threads should run during the slots assigned to schedulees are the responsibility of user level thread schedulers and do not directly involve the slot scheduler. The `uvm_CpuSlot` data structure also indicates the level of token with which the owner purchased its slot (priority); tokens and resource accounting are discussed more fully in the section “Banking system calls”.

Rather than providing applications with a narrow system call interface for querying the current state of the schedule, the actual slot arrays are stored in shared memory regions, to which all processes may obtain read-only access using the `uvm_getCpuSlotArray` call (described in the section “Miscellaneous system calls”). This gives user level schedulers the ability to quickly and easily scan the slot schedule in order to figure out how they can satisfy their particular scheduling constraints. Multiple processes may read the slot schedule concurrently but only one process at a time may succeed in claiming a particular slot. Races among equal-priority claims that overlap in time are resolved by the system within the `uvm_claimCpuSlots` call.

All slots are equal in duration; applications can find out the duration value in milliseconds with the `uvm_getCpuSlotDurationMs` call. Since system resources are finite, the length of the slot arrays is fixed and the kernel scheduler walks through them in periodic round-robin fashion. Slot claims persist through repeated cyclings of the schedule and change only in response to user requests or certain system events. For example, a process claiming Slot #42 in a CPU array that is 100 timeslices long can expect to have its schedulee run at timeslices 42, 142, 242, etc. To support execution patterns such as gang scheduling, the system should also guarantee that slot boundaries are synchronized across processors.

When the slot schedule does not provide a valid candidate for execution, either because the current slot is unclaimed or because its schedulee is not runnable, the system will fall back to some sort of best-effort or proportional share scheduling among the existing runnable processes.

3.5.2 *Scheduling system calls*

These calls directly influence the state of the slot schedule and are the primary interface applications uses for implementing their own scheduling policies.

The function `uvm_claimCpuSlots()` is used to both request new slots for the calling process and to change the parameters of existing claims. The caller passes in a variable length `slotSpec` array whose length is specified by the `length` argument. Each `uvmCpuSlotRequest` structure in the `slotSpec` array has an `index` field which specifies the target slot in the `uvmCpuSlot` array and a `processorID` field which specifies the target processor. Values for the `index` field may range from 0 to one less than the length returned from the `uvm_getCpuSlotArray` call. The caller must order the `slotSpec` array so that the `processorID` field is in increasing order and, for a particular processor, the `index` field is in increasing order. This ordering means that if N is the length returned from the `uvm_getCpuSlotArray` call, then $(N * \text{processorID} + \text{index})$ will be ordered and increasing through the `slotSpec` Array. The kernel passes back a possibly shortened version of the `slotSpec` array, with the value of `length` adjusted appropriately, to report which slots were successfully claimed or modified. All slots are purchased using the caller's available token funds at the given priorities, and any slots successfully claimed will be tagged with the caller's PID in the `ownerPID` field of the corresponding `uvm_CpuSlots`. If `notifyOwnerOnPreempt` is true then the system will inform the caller if any of the specified claims are preempted or revoked in the future via a callback to the `handler` in the caller's address space. This call will return 0 if there were no problems with the request or a negative value otherwise. The error codes are `uvm_e_badProcessor` if the call does refer to a valid processor; `uvm_e_badSlotSpec` if any `slotSpec` values were out of range or not strictly increasing; `uvm_e_badLen` if the `slotSpec` length was negative or greater than the total number of slots in the schedule; `uvm_e_badProcessID` if the schedulee argument is invalid; `uvm_e_badPriority` if the priority argument is negative or greater than `uvm_maxTokenPriority`; `uvm_e_insufficientTokens` if the caller does not possess adequate funds to purchase all the slots requested. In the cases of `uvm_e_badSlotSpec` and `uvm_e_insufficientTokens`, the kernel may still succeed in claiming some slots for the caller and these successes will be reported via the `slotSpec` and `length` parameters.

The function `uvm_freeCpuSlots()` relinquishes slots previously claimed by the caller. The semantics of the arguments are the same as those for the function `uvm_claimCpuSlots()`, except that the kernel does not pass back information reporting which slots were successfully freed. This call will return 0 if there were no problems with the request or a negative value otherwise. The error codes are `uvm_e_badProcessor` if the call did not refer to a valid processor; `uvm_e_badSlotSpec` if any `slotSpec` values were out of range or not strictly increasing; `uvm_e_badLen` if the `slotSpec` length was negative or greater than the total number of slots in the schedule. In the case of `uvm_e_badSlotSpec`, the kernel may still succeed in freeing some slots for the caller.

The function `uvm_reserveCpuSlots()` globally constrains applications' ability to claim slots in a particular slot array. This call is restricted to privileged system users. For each interval in the array specified by `processorID` containing denominator slots, the system will ensure that numerator of those slots remain unclaimed by any process, thus giving non-slot scheduled applications the chance to regularly receive processor time via the fallback scheduling algorithm. If `immediate` is true, the system will forcibly revoke existing slot claims so that the numerator/denominator constraint is satisfied immediately; if `immediate` is false, the system will satisfy the constraint lazily by only barring new claims, leaving existing claims untouched. This call will return 0 on success or a negative value on error. The error codes are `uvm_e_badProcessor` if the call did not refer to a valid processor; `uvm_e_badNumDen` if numerator is negative or greater than denominator, or if denominator is non-positive or greater than `uvm_cpuSlotArrayLength`; or `uvm_e_access` if the caller does not have sufficient privilege to perform this operation.

3.5.3 *Banking system calls*

These calls provide a flexible way for the system to carry out policies regarding applications' usage of the slot schedule. Each process possibly has a number of tokens which must be used in order to purchase slots in the scheduling grid. All tokens are equivalent in terms of the number of slots they can buy: each slot costs exactly one token. However, tokens may be of differing priorities, allowing the system to empower some processes with greater purchasing privileges than others. A token of priority x can be used to purchase either an unclaimed slot or a slot that was claimed using a token of priority less than x . In the latter case, the previous owner's claim is preempted and the owner is possibly notified via an upcall.

The function `uvm_getTokenCounts()` queries the system for the amounts of tokens currently available to the process identified by `processID`. Processes can always query the system to find out their own token counts but may be restricted from finding out the available token counts of other processes or other users. The argument `tokenCount` must point to storage in the caller's address space large enough to contain an integer array of length $(\text{uvm\_maxTokenPriority} + 1)$. On return this array will be filled with the token counts at the various priority levels. The call returns 0 on success or a negative value on error. The error codes are `uvm_e_badProcessID` if the process argument is invalid, or `uvm_e_access` if the caller does not have sufficient privilege to look up the token counts for the given process.

The function `uvm_donateTokens()` transfers tokens of the given priority from the caller's funds to the funds of the process identified by `recipientPID`. This call will return 0 on success or a negative value on error. The error codes are `uvm_e_badProcessID` if the recipient argument is invalid; `uvm_e_badPriority` if the priority argument is negative or greater than `uvm_maxTokenPriority`; or `uvm_e_insufficientTokens` if the caller does not have the specified number of tokens. In the case of `uvm_e_insufficientTokens`, the system will transfer all the tokens the caller possesses at the given priority.

The function `uvm_changeTokenCount()` tweaks the available token funds of the process identified by `processID` at the given priority. This call is restricted to privileged system users. If `absolute` is true, the caller's token count is directly set to count; if `absolute` is false, the count argument is interpreted as a delta change and is added to the caller's current token count.

This call will return 0 on success or a negative value on error. The error codes are `uvm_e_badProcessID` if the process argument is invalid; `uvm_e_badPriority` if the priority argument is negative or greater than `uvm_maxTokenPriority`; or `uvm_e_access` if the caller does not have sufficient privilege to perform this operation.

3.5.4 Miscellaneous system calls

The function `uvm_getCpuSlotArray` returns the memory address of a read-only segment in the caller's address space where the slot arrays have been mapped. The length of the slot array is returned in the `length` argument. The total length of the slot array is the number of processors times the number of slots (returned in `length`) scheduled. The major direction with unit stride is slots for a particular processor. The minor direction with stride `length` is along processors. If a two dimensional C data array were defined to index the slot array the row index would be the processor number and the column index would be the slot number. So consecutive elements of slot schedule are laid one after the other; that is, if `addr` is the return value of a call to `uvm_getCpuSlotArray`, the slot array for processor `i` is located at `addr + i*length`. Note that the processor index `i` starts at zero for the first processor in the array. There are no error conditions defined for this function.

The function `uvm_getNextCpuSlotIndex` returns the index of the slot following the currently scheduled slot. Values for this index range from 0 to one less than the length returned from the `uvm_getCpuSlotArray` call. This call is intended as a means for user schedulers to locate a starting point for walking the slot arrays. There are no error conditions defined for this function.

The function `uvm_getCpuSlotDurationMs` returns the length of individual slots in milliseconds. There are no error conditions defined for this function.

4. Memory control model

The memory control interface available to user-level libraries (or applications) is similar to the interface available to privileged/kernel code defined in the TVM-HAL document.

The main difference is support for multiprogramming—whereas a kernel memory manager is omnipotent and knows about all memory allocation decisions in the system, a user-level memory management library is parochial and may not know what allocation decisions are being made outside of that user-level process. Therefore, the user-level interface differs from the privileged interface in that (a) there is a way to learn about what memory resources are currently available (b) requests for a given memory resource can be rejected (e.g., because that resource is not currently available or because the kernel is limiting how many resources the user-level process can grab), and (c) for portability, mapping (`uvm_setSegment()`) and unmapping (`uvm_clearSegment()`) requests include the *view* that they affect. A *view* is a set of hardware threads or processors that are constrained to share the same virtual address to physical address (VA to PA) mappings. For some architectures, each physical processor has a separately-controlled TLB, allowing each processor to have a different view of memory. In other architectures, several hardware threads on the same physical processor might share a TLB and share a view. Finally, some architectures (e.g., a system running over POSIX for backwards compatibility) may not permit different processors to have different mappings of memory; in this case, the only view is `uvm_ptovAll`.

4.1. Physical memory view

Section 4.1 of the TVM-HAL document describes the model of physical memory.

4.2. Segment control

Section 4.2 of the TVM-HAL specification defines the *interrogation calls* `HAL_NUM_MEMORIES()`, `HAL_GET_MEM_CHARACTERISTIC_BY_NUM(...)`, `HAL_GET_MEM_CHARACTERISTI_BY_IDENT(...)` and the *cache segment control calls* `HAL_ICACHE_ON(...)`, `HAL_ICACHE_OFF()`, `HAL_DCACHE_ON(...)`, and `HAL_DCACHE_OFF()`. These calls are also available to user-level code. Note that at the user level, the cache control calls are advisory only and may be ignored by the system.

Note that the type `PhyPtr` is defined in the TVM-HAL document, and the type `int64` is defined in the TVM-HAL document as `HAL_int64`.

The user-level interfaces for *memory segment control* differ slightly from the TVM-HAL:

```
int numSegmentsPossiblePerProcessor = uvm_numSegments();

int uvm_processorsPerView();

int viewForThisProcessor = uvm_viewId();

int uvm_clearSegment(int viewId, int segmentNumber);

/* The type int64 is equivalent to HAL_int64 */
/* The type bool false is 0; true is !0 */
int uvm_setSegment(int viewId,
                  int segmentNumber,
                  void *startingVirtualAddress,
                  int64 vsize,
                  int bank,
                  int64 offset,
                  bool cachable,
                  bool cacheCoherent,
                  bool executable,
                  bool executeOnly,
                  bool writable,
                  bool mapped,
                  int64 mapPhysicalSize,
                  int interleaveSize,
                  void **returnVirtualAddress);
```

We also include three calls not included in the TVM-HAL:

```
int uvm_pinSegment(int viewId, int segmentNumber);
int uvm_unPinSegment(int viewId, int segmentNumber);

int uvm_getStartPA(int viewId, int segmentNumber,
                  PhyPtr *resultPtr);
```

The function `uvm_numSegments()` returns the number of segments that a physical processor can map into virtual memory.

The function `uvm_processorsPerView()` returns the number of processors that are restricted to share the same view of memory. A return value of 1 means that each physical processor can have a different VA/PA mapping; a return value of, say, 4, means that groups of four hardware threads must share the same VA/PA mapping; a return value of `uvm_ptovAll = -2` means that all processors are restricted to see the same VA/PA mapping (e.g., legacy implementations over POSIX.)

The function `uvm_viewId()` returns the ID of the view that is visible from the current processor.

The function `uvm_clearSegment()` releases a mapping and the associated physical memory back to the run-time system. After this call is made, the user-level process no longer can access the indicated segment. The view specified by `viewId` must be visible from the current processor, and the segment specified by `segmentNumber` must match a currently mapped segment. This call returns 0 on success. On error it returns the error code `uvm_e_wrongView`

if the view ID refers to a view other than the one visible from the processor from which the function was called; or `uvm_e_badSeg` if the specified segment number does not correspond to a mapping. Note that before making the call the run-time system may need to ensure that the calling thread cannot be migrated to another processor during the call.

The function `uvm_setSegment ( )` maps a range of physical memory to a virtual address for the calling process. If the mapping succeeds, the call returns 0. Otherwise it returns a negative error code.

- The `viewId` must be the view that is visible from the current processor. Otherwise, the function returns the error code `uvm_e_wrongView`. Note that a run-time system may need to ensure that the calling thread cannot be migrated to another processor during the call.
- The `segmentNumber` must correspond to an unused segment number for the view. The call returns `uvm_e_badSeg` if the specified segment number already corresponds to a mapping.
- The `startingVirtualAddress` and `vsize` defines the virtual address range for the mapping. The argument `startingVirtualAddress` specifies either a virtual address or `(void*)NULL` to allow the run-time system to choose the address. The argument `vsize` specifies the size of the virtual address range. If the specified virtual address range overlaps with any other mapping then the error code `uvm_e_badVrange` is returned by the function. The starting virtual address actually assigned (or `NULL` on error) is placed in `**returnVirtualAddress`.
- The `bank` and `offset` (and if mapped, `mapPhysicalSize`) specifies the physical resources to use for this memory. The `bank` argument specifies a memory bank and the `offset` specifies an offset in bytes from the start of the memory bank to the start of the physical memory to be mapped.

Note that `HAL_GET_MEM_CHARACTERISTIC_BY_NUM( . . . )` may be used to get the physical address of the start of the bank. Memory banks may be referenced either by an integer number or by a string. The TVM-HAL supports both. The UVM only references a bank by its number.

If `bank = uvm_pmemBankAny = -1`, then `offset` must be `uvm_pmemOffsetAny = -1`, and the system can choose any free physical memory of the requisite size that supports the specified mode.

If `bank ≠ uvm_pmemBankAny` but `offset = uvm_pmemOffsetAny`, then the system can choose any free physical memory in the specified bank of the requisite size that supports the specified mode.

If neither `bank` nor `offset` is a wild card, then the system may map only the specified address and no other.

On error the call returns a negative value. If the bank is a wild card but the offset is not, the system returns `uvm_e_offset`. If no region of the specified size in the specified bank is available, the call returns `uvm_e_psize`. If the specified bank and/or offset are not available (e.g. in use by another process) the call returns `uvm_e_inUse`. If the physical address specified by the bank and offset is not valid the call returns `uvm_e_badPA`. If the specified bank is not visible to the specified view (e.g., because the system does not support globally addressable memory for some or all physical addresses), the call returns `uvm_e_bankNotVisible`.

- `cachable`, `cacheCoherent`, `executable`, `executeOnly`, `writable` specify the mode bits that must be set for the mapped segment. The call returns `uvm_e_mode` if one or more of the specified bits is not available with the bank requested, or if `bank` is `uvm_pmemBankAny` the call returns `uvm_e_mode` if no bank supports the specified combination.
- `mapped` specifies a mapping when a large virtual range is mapped to a small physical range.

The function `uvm_pinSegment()` tells the run-time system that the application plans to refer to a region of memory by its physical address (e.g., for Active DMA) and that the run-time system should not change the VA-to-PA mapping for that segment. We anticipate that few PCA run-time systems will initially do any dynamic remapping of VA-to-PAs, but this call allows programs to explicitly note when they depend on that assumption. This call returns 0 on success or a negative error code on failure. Error codes include `uvm_e_perm` if you do not have permission to pin the specified region, or `uvm_e_badArg` if the specified view/segment is not a mapped view/segment. Run-time systems can refuse to allow an application to pin a segment for any reason or no reason. To minimize the likelihood of a pin request being rejected, applications should generally pin (a) small regions and (b) direct-mapped (e.g., non-interleaved) regions. The function `uvm_unpinSegment()` tells the operating system that the application no longer depends on knowing the physical address of a segment. This call returns 0 on success and a negative value on error; at present no error conditions are defined.

The function `uvm_getStartPA()` returns the physical base address for a segment. This call is needed for two reasons. First, because applications can supply wild-card physical addresses to `setSegment` calls, they don't always know which physical regions were assigned to them; if they later need to know (e.g., for IO such as Active DMA), this call gives them a way to find out. Second, it is possible that in the future some PCA run-time systems may remap VA to PA regions. This call allows an application to find out about such things. This call only makes sense for segments in the *pinned* state. This call returns 0 and sets `*PhyPtr` to the requested value on success. On failure it returns a negative error code `uvm_e_badArg` if the specified segment or view are not mapped, or `uvm_e_unpinnedMem` if the segment is not pinned.

Note that the “proper” way to use a physical pointer for IO is to first pin a segment, then look up its physical address, then use it for IO. It is unwise to skip the “look up the address” step since future PCA run-time systems may allow the mappings to change over time.

4.3. *Paged memory control*

Section 4.3 of the TVM-HAL document describes a paged memory control interface that is applicable to kernel/privileged code but not to user code.

5. Other Extensions

5.1. Active DMA

The TVM-HAL document describes an Active DMA interface in section 5.2. This interface is available to user-level programs as well.

In some implementations, there may be no protection or virtualization of the Active DMA interface; in such systems, any application can access the DMA controller hardware directly – there is no protection but there is also very little overhead. In other implementations, the hardware and run-time system may virtualize the DMA controller so that different processes cannot interfere with one another. Either way, the same interface is used.

Note that this user-level interface uses physical addresses for addressing. The addresses must be *pinned* (see Section 4.2) before use and may not be *unpinned* until all pending DMAs have completed. An attempt to access unpinned memory triggers the error code `uvm_e_unpinnedMem`.

5.2. Multiprocessor synchronization

Section 5.3 Multiprocessor Synchronization of the TVM-HAL specification specifies an atomic read-modify-write primitive and a barrier-synchronization primitive. These interfaces are available to user-level applications.

Below, we also define a higher-level mutex primitive and a condition variable primitive. The functions described below represent a subset of POSIX functionality; each function is identical to an existing POSIX function, but some of the POSIX configuration options for some arguments are not supported by the UVM. Below, we define the interfaces that act upon the types `pthread_mutex_t`, `pthread_mutexattr_t`, and `pthread_cond_t`; additional details about these types may be found in the POSIX spec.

5.2.1 Mutex lock

A mutex object must first be created using the `pthread_mutex_init()` API. As an optimization, a mutex can be initialized with default behavior by simply assigning it the value `PTHREAD_MUTEX_INITIALIZER`. `pthread_mutex_lock()` locks the given mutex. If the mutex is currently unlocked, it becomes locked and owned by the calling thread, and `pthread_mutex_lock()` returns immediately. If the mutex is already locked by another thread, `pthread_mutex_lock()` suspends the calling thread until the mutex is unlocked. `pthread_mutex_trylock()` behaves identically to `pthread_mutex_lock()`, except that it does not block the calling thread if the mutex is already locked by another thread. Instead, `pthread_mutex_trylock()` returns immediately with the error code `EBUSY`. `pthread_mutex_unlock()` unlocks the given mutex. The mutex is assumed to be locked and owned by the calling thread on entrance to `pthread_mutex_unlock()`. `pthread_mutex_destroy()` destroys a mutex object, freeing the resources it might hold. The mutex must be unlocked on entrance.

POSIX defines three flavors of mutexes: *fast mutexes*, which give good performance but will deadlock if the same thread attempts to lock a mutex it already holds; *error-checking* mutexes

detect double lock situations and return an error code, and *recursive mutexes* maintain a lock count to allow threads to nest lock obtains and releases to arbitrary depths. While error-checking and recursive mutexes would add safety and flexibility, we do not have concrete evidence that they are required. Therefore only fast mutexes are currently supported in the UVM API.

```
int pthread_mutex_init(pthread_mutex_t *mp, const pthread_mutexattr_t
*attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_destroy(pthread_mutex_t *mp);

int pthread_mutex_lock(pthread_mutex_t *mp);
int pthread_mutex_trylock(pthread_mutex_t *mp);
int pthread_mutex_unlock(pthread_mutex_t *mp);
```

If successful, all of these functions return 0; otherwise, an error number is returned. `pthread_mutex_trylock()` returns 0 if a lock on the mutex object referenced by `mp` is obtained; otherwise, an error number is returned. These functions fail and return the corresponding value if any of the following conditions are detected:

- EFAULT** `mp` or `attr` points to an illegal address.
- EINVAL** `pthread_mutex_init()` determined that the value specified by `mp` or `attr` is invalid.
- EBUSY** `pthread_mutex_trylock()` determined that the mutex pointed to by `mp` was already locked.

A `pthread_mutexattr_t` specifies whether a thread is private to a process (default) or can be shared across processes.

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);
```

Where `pshared` can take the values

- PTHREAD_PROCESS_PRIVATE** – The mutex can synchronize threads within this process (default).
- PTHREAD_PROCESS_SHARED** – The mutex can synchronize threads in this process and other processes. Only one process should initialize the mutex.

If successful, all of these functions return 0; otherwise, a nonzero error number is returned.

- EINVAL** The value specified by the `attr` pointer or the `pshared` flag is invalid.
- ENOMEM** There is insufficient memory for `pthread_mutexattr_init` to complete.

5.2.2 Condition variables

A condition (short for “condition variable”) is a synchronization object that allows threads to suspend execution and relinquish the processors until some predicate on shared data is satisfied. The basic operations on conditions are (1) signal the condition (when the predicate becomes true), and (2) wait for the condition (suspending the thread execution until another thread signals the condition.) A condition variable must always be associated with a mutex, to avoid the race

condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it.

The function `pthread_cond_init()` initializes the condition variable `cond`, using the condition attributes specified in `cond_attr`. Variables of type `pthread_cond_t` can also be initialized statically, using the constant `PTHREAD_COND_INITIALIZER`. The function `pthread_cond_destroy()` destroys a condition variable, freeing the resources it might hold. No threads must be waiting on the condition variable on entrance to `pthread_cond_destroy()`.

The function `pthread_cond_signal()` restarts one of the threads that are waiting on the condition variable `cond`. If no threads are waiting on `cond`, nothing happens. If several threads are waiting on `cond`, exactly one is restarted, but it is not specified which. The function `pthread_cond_broadcast()` restarts all the threads that are waiting on the condition variable `cond`. Nothing happens if no threads are waiting on `cond`.

The function `pthread_cond_wait()` atomically unlocks the mutex (as per `pthread_unlock_mutex()`) and waits for the condition variable `cond` to be signaled. The thread execution is suspended and does not consume any CPU time until the condition variable is signaled. The mutex must be locked by the calling thread on entrance to `pthread_cond_wait()`. Before returning to the calling thread, `pthread_cond_wait()` re-acquires the mutex (as per `pthread_lock_mutex()`).

Unlocking the mutex and suspending on the condition variable is done atomically. Thus, if all threads always acquire the mutex before signaling the condition, this guarantees that the condition cannot be signaled (and thus ignored) between the time a thread locks the mutex and the time it waits on the condition variable.

The function `pthread_cond_timedwait()` behaves identically to `pthread_cond_wait()`, except that it also bounds the duration of the wait. If `cond` has not been signaled by the time specified by `abstime`, the mutex `mutex` is reacquired and `pthread_cond_timedwait()` returns the error `ETIMEDOUT`. The `abstime` parameter specifies an absolute time, with the same origin as `time(2)` and `gettimeofday(2)`.

```
int pthread_cond_init(pthread_cond_t *cond,
                     const pthread_condattr_t *attr);
pthread_cond_t cond= PTHREAD_COND_INITIALIZER;
int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_condattr_init(pthread_condattr_t *attr);
int pthread_condattr_destroy(pthread_condattr_t *attr);

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
                          pthread_mutex_t *mutex,
                          const struct timespec *abstime);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

If successful, all of these functions return 0; otherwise, an error number is returned.

EINVAL	The value specified by <code>cond</code> or <code>mutex</code> is invalid.
ETIMEDOUT	The condition variable was not signaled prior to the timeout specified to <code>pthread_cond_timed_wait</code> .

5.3. Cache Memory Control

Section 5.4 of the TVM-HAL document defines a collection of interfaces for cache memory control. Most of these interfaces are also available to user-level programs. The exceptions are the physical-address versions of the FENCE and cache control operations (see the TVM-HAL document for details.)

5.4. IEEE 754 Floating Point Control

Section 5.5 of the TVM-HAL document defines a few routines to standardize the interface for IEEE 754 floating-point control. These routines are also available to user-level programs.

5.5. Performance Counters/Run-time Metadata

Section 5.6 of the TVM-HAL document defines a set of routines for a standard interface to processor supported performance-monitoring counters. These interfaces are also available to user-level programs.

5.6. Generic Special Register Access

Section 5.8 of the TVM-HAL document defines a generic mechanism to read and write special registers. This special register access is also available to user-level programs. Use of these registers is architecture dependence and is not standard. Use of these registers will not produce portable code. Consequently code using special register access functions will need to be reviewed and modified when porting to a new architecture.

6. UVM Reference

This section is a reference for information already presented above. New functions defined in this document for scheduler activation and for memory segment control are referenced below. Functions that are defined by the POSIX specification [6] and included in this document, and functions mentioned in this document that are defined in the TVM-HAL document are not included below.

Also referenced below are UVM specific error codes and type definitions.

6.1. Scheduler Functions

6.1.1 Scheduler Activations

The following constants and types are defined for this section.

```
#define uvm_saProcAny -1
#define uvm_saGlobal -2

/* This structure is implementation-dependent */
struct myArchitectureSaveBlock;
typedef myArchitectureSaveBlock uvm_saArchitectureSaveBlock;

typedef enum {
    uvm_saHasBeenPreempted = 0,
    uvm_saAddProcessor = 1,
    uvm_saHasBlocked = 2,
    uvm_saHasUnblocked = 3
} uvm_saUpcallIndex;
```

uvm_saHandlerSet

Called to give the kernel the pointer to a handler function for upcalls from the kernel to the user code.

Prototype

```
int uvm_saHandlerSet(
    uvm_saUpcallIndex entryNumber,
    CodeVPtr handlerPC);
```

Arguments

`entryNumber` indicates the proper call signature for the handler. See Table 1 for defined handler prototypes.

`handlerPC` is the address of the handler function.

Returns

An integer equal to 0 on success or a negative value on error.

Errors

`uvm_e_saBadRange` if `entryNumber` is not a legal value.

`uvm_e_saNeedDisable` if scheduler activations are enabled at any processor for the process.

Described in

Under initialization in Section 3.4

uvm_saSetMaxProcessors

A scheduler activations initialization routine to select the maximum number of kernel threads that a process plans to use.

Prototype

```
int uvm_saSetMaxProcessors(int max);
```

Arguments

max is the maximum number of kernel threads that a process plans to use.

Returns

An integer equal to 0 on success or a negative value on error.

Errors

uvm_e_saBadRange if max is not at least one.

uvm_e_saNeedDisable if scheduler activations are enabled at any processor for the process.

Described in

Under initialization in Section 3.4

uvm_saStateBufferSet

Provides the kernel a pointer to a buffer. The state of preempted threads are saved in the buffer.

Prototype

```
int uvm_saStateBufferSet(int procID, uvm_saArchitectureSaveBlock *buffer);
```

Arguments

procID is either a positive integer indicating a specified processor or is uvm_saProcAny indicating a buffer that is globally visible and can be bound by the runtime system to any processor.

Returns

An integer equal to 0 on success or a negative value on error.

Errors

uvm_e_saBadRange if the ID is negative and not uvm_saProcAny.

uvm_e_saNeedDisable if this call is made when scheduler activations are enabled at any processor for the process.

Described in

Under *Initialization* in Section 3.4

uvm_saEnable

Activates or reactivates scheduler activations.

Prototype

```
int uvm_saEnable(int procID);
```

Arguments

`procID` is either a positive integer to reactivate scheduler activations for a particular instance of a user-level scheduler on a particular processor; or `uvm_saGlobal` to activate scheduler activations for a process.

Returns

An integer equal to 0 on success or a negative value on error.

Errors

`uvm_e_saNotInitialized` if any of the handlers or buffers have not yet been initialized.

`uvm_e_saBadRange` if `procID` is negative and not `uvm_saGlobal`.

Described in

Under *Enable/disable* in Section 3.4

uvm_saDisable

This function deactivates scheduler activations either for a process or for one instance of a user level scheduler.

Prototype

```
int uvm_saDisable(int procID);
```

Arguments

`procID` is either positive to disable a single instance of a user level scheduler or `uvm_saGlobal` to deactivate all scheduler activations for a process.

Returns

An integer equal to 0 on success or a negative value on error.

Errors

`uvm_e_saBadRange` if `procID` is negative and not `uvm_saGlobal`.

Described in

Under *Enable/disable* in Section 3.4

uvm_saRequestProcessor

A process calls this function to indicate it would like an additional physical processor. The kernel scheduler invokes the `add_processor()` upcall defined in Table 1 if the request is granted. The minimum conditions for a processor to be added are that a processor is or becomes available and that fewer than the maximum kernel threads (set using `uvm_saSetMaxProcessors`) are active for the process.

Prototype

```
int uvm_saRequestProcessor(int procID);
```

Arguments

The `procID` is a positive integer which is a hint to request a particular processor from the kernel scheduler, or it is `uvm_saProcAny` indicating no processor preference. The kernel is not required to return a particular processor indicated by a hint.

Returns

An integer equal to 0 on success or a negative value on error.

Errors

`uvm_e_saBadRange` if `procID` is neither `uvm_saProcAny` nor the identifier of any processor.

`uvm_saBadBuffer` if there is not currently a buffer that can be legally mapped to the requested processor.

Described in

Under *Request/yield processors* in Section 3.4

uvm_saReleaseProcessor

A thread calls this function to tell the kernel it no longer needs its processor.

Prototype

```
int uvm_saReleaseProcessor();
```

Returns

Normally does not return. A return would be an error.

Errors

No error codes have been specified

Described in

Under *Request/yield processors* in Section 3.4

uvm_saArchitectureSaveBlockSize

This function returns the size in bytes of an `uvm_saArchitectureSaveBlock` type.

Prototype

```
size_t uvm_saArchitectureSaveBlockSize()
```

Returns

The size of data in bytes that must be allocated for instantiation of an instance of the opaque type `uvm_saArchitectureSaveBlock`.

Described in

Section 3.4

uvm_saResumeThread

After an upcall from the kernel (`uvm_saHasUnblocked`) indicating a blocked thread is runnable this call will resume execution of the blocked thread.

Prototype

```
void uvm_saResumeThread(uvm_saArchitectureSaveBlock *buffer)
```

Arguments

`buffer` is a pointer to a previously created and instantiated buffer of type `uvm_saArchitectureSaveBlock`.

Returns

This call does not normally return, except on error.

Described in

Under *Resume thread* in section 3.4

6.1.2 Slot Scheduling Functions

The following structures are defined for slot scheduler use.

```
typedef struct {
    int ownerPID;
    int scheduleePID;
    int priority;
} uvm_CpuSlot;

typedef struct {
    int processorID;
    int index;
    int scheduleePID;
    int priority;
} uvm_CpuSlotRequest;
```

There is also a system-defined constant that defines the maximum token priority value.

```
const int uvm_maxTokenPriority
```

uvm_claimCpuSlots

This function is used to request new slots for the calling process or to change the parameters of an existing claim.

Prototype

```
int uvm_claimCpuSlots(  
    uvm_CpuSlotRequest *slotSpec,  
    int *length,  
    bool notifyOwnerOnPreempt,  
    CodeVPtr handler);
```

Arguments

`slotSpec` is an array of type `uvm_CpuSlotRequest` of length `length`

`length` is the number of elements in the `slotSpec` array.

`notifyOwnerOnPreempt` is a Boolean. If true the system is to notify the caller if any of the specified claims are preempted or revoked in the future using the handler pointed to by `handler`.

`handler` is a callback the system uses to notify a process that one or more of its specified slot requests was preempted.

Returns

An integer equal to 0 on success or a negative value on error.

Errors

`uvm_e_badProcessor` if the call does refer to a valid processor

`uvm_e_badSlotSpec` if any `slotSpec` values were out of range or not strictly increasing.

`uvm_e_badLen` if the `slotSpec` length was negative or greater than the total number of slots in the schedule.

`uvm_e_badProcessID` if the schedulee argument is invalid.

`uvm_e_badPriority` if the priority argument is negative or greater than `uvm_maxTokenPriority`.

`uvm_e_insufficientTokens` if the caller does possess adequate funds to purchase all the slots requested.

Described in

Section 3.5.1 and 3.5.2

uvm_freeCpuSlots

This function releases slots previously claimed by the caller.

Prototype

```
int uvm_freeCpuSlots(  
    uvm_CpuSlotRequest *slotSpec,  
    int *length);
```

Arguments

`slotSpec` is an array of type `uvm_CpuSlotRequest` of length `length`

`length` is the length of the `slotSpec` array.

Returns

An integer equal to 0 on success or a negative value on error.

Errors

`uvm_e_badProcessor` if the call did not refer to a valid processor.

`uvm_e_badSlotSpec` if any `slotSpec` values were out of range or not strictly increasing.

`uvm_e_badLen` if the `slotSpec` length was negative or greater than the total number of slots in the schedule.

Described in

Section 3.5.2

uvm_reserveCpuSlots

This function globally constrains an applications' ability to claim slots in a particular slot array, and is reserved for privileged system users.

Prototype

```
int uvm_reserveCpuSlots(  
    int processorID,  
    int numerator,  
    int denominator,  
    bool immediate);
```

Arguments

`processorID` specifies the slot array.

`numerator` is the number of slots in the array specified by `processorID` which may not be claimed by any process.

`denominator` specifies the total number of slots in the array specified by `processorID`.

`immediate` is a Boolean. If `immediate` is true the system will forcibly revoke existing slots to satisfy the `numerator` criteria; and if false the system will satisfy the `numerator` criteria by recovering slots as they are released by owning processes.

Returns

An integer equal to 0 on success or a negative value on error.

Errors

`uvm_e_badProcessor` if the call did not refer to a valid processor.

`uvm_e_badNumDen` if `numerator` is negative or greater than `denominator`, or if `denominator` is non-positive or greater than `uvm_cpuSlotArrayLength`.

`uvm_e_access` if the caller does not have sufficient privilege to perform this operation.

Described in

Section 3.5.2

uvm_getTokenCounts

This function queries the system for the amounts of tokens currently available to a specified process.

Prototype

```
int uvm_getTokenCounts(  
    int processID,  
    int *tokenCount);
```

Arguments

`processID` is the process for which the available token count array is being requested.

`tokenCount` is a pointer to an integer array (in the callers address space) of length `uvm_maxTokenPriority + 1`.

Returns

An integer equal to 0 on success or a negative value on error.

Errors

`uvm_e_badProcessID` if the process argument is invalid.

`uvm_e_access` if the caller does not have sufficient privilege to look up the token counts for the given process.

Described in

Section 3.5.3

uvm_donateTokens

This function transfers tokens of a given priority from the calling process to a selected recipient process.

Prototype

```
int uvm_donateTokens(  
    int recipientPID,  
    int count,  
    int priority);
```

Arguments

`recipientPID` is the ID of the process the tokens are to be donated to.

`count` is an integer indicating the number of tokens to be donated.

`priority` is the priority level of the tokens to be donated.

Returns

An integer equal to 0 on success or a negative value on error.

Errors

`uvm_e_badProcessID` if the recipient argument is invalid.

`uvm_e_badPriority` if the priority argument is negative or greater than `uvm_maxTokenPriority`.

`uvm_e_insufficientTokens` if the caller does not have the specified number of tokens of the proper priority.

Described in

Section 3.5.3

uvm_changeTokenCount

This function will adjust the available token funds at a selected priority of a selected process to `count` if `absolute` is true, or will adjust the available token funds at the selected priority by an increment of `count` if `absolute` is false. This is a privileged function.

Prototype

```
int uvm_changeTokenCount(  
    int processID,  
    int count,  
    int priority,  
    bool absolute);
```

Arguments

`processID` identifies the process that will have its token count adjusted.

`count` is an integer indicating the number of tokens the process indicated by `processID` will have of priority `priority` if `absolute` is true, or the change in tokens of priority `priority` if `absolute` is false.

`priority` is the priority level of the tokens to be awarded.

`absolute` is a Boolean indicating if the token count is to be set to `count` (true) or if the token count is to be adjusted by `count` (false).

Returns

An integer equal to 0 on success or a negative value on error.

Errors

`uvm_e_badProcessID` if the process ID is not valid.

`uvm_e_badPriority` if less than zero or greater than `uvm_maxTokenPriority`.

`uvm_e_access` implies the caller does not have sufficient privileges.

Described in

Section 3.5.3

uvm_getCpuSlotArray

This function is used to get the address of the read-only memory segment where the slot array is stored. The number of slots (per processor) is returned as the `length` argument. The address returned is the beginning of the slot array associated with processor zero. If more than one processor is associated with the slot array then the address associated with each processor is the processor number times the length address plus the address returned by this function.

Prototype

```
uvm_cpuSlot *uvm_getCpuSlotArray(  
    int *length);
```

Arguments

`length` is used to return the length of the slot schedule array returned by the call.

Returns

A pointer to an array of type `uvm_cpuSlot` or NULL if no array exists or there is an error.

Errors

None defined.

Described in

Section 3.5.4

uvm_getNextCpuSlotIndex

This function returns the index of the slot following the currently scheduled slot.

Prototype

```
int uvm_getNextCpuSlotIndex();
```

Arguments

None

Returns

Index of next slot to be executed.

Errors

None defined

Described in

Section 3.5.4

uvm_getCpuSlotDurationMS

This function returns the length of individual slots in milliseconds.

Prototype

```
int uvm_getCpuslotDurationMS();
```

Arguments

None

Returns

The length of individual slots in milliseconds.

Errors

None defined

Described in

Section 3.5.4

6.2. Segment Control Functions

The following constants and types are defined for this section. The `bool` type is generally 0 for false and 1 (!0) for true.

```
#define uvm_pmemBankAny -1
#define uvm_pmemOffsetAny -1
#define uvm_ptovAll -2
typedef HAL_int64 int64;
```

uvm_numSegments()

A function to obtain the number of segments that a physical processor can simultaneously map into virtual memory.

Prototype

```
int uvm_numSegments()
```

Returns

An integer (`numSegmentsPossiblePerProcessor`) equal to the number of segments that a physical processor can map into virtual memory.

Described in

Section 4.2

Also see `HAL_NUM_SEGMENTS( )` in the TVM-HAL document.

uvm_processorsPerView()

A function to obtain the number of processors that are restricted to share the same view of memory.

Prototype

```
int uvm_processorsPerView();
```

Returns

An integer equal to the number of processors that are restricted to share the same view of memory, or `uvm_ptovAll` indicating all processors are restricted to the same VA/PA mapping.

Described in

Section 4.2

uvm_viewId();

A function to obtain the view ID visible from the current processor.

Prototype

```
int uvm_viewId();
```

Returns

An integer (`viewForThisProcessor`) that is the ID for the view that is visible from the current processor.

Described in

Section 4.2

uvm_clearSegment

Release physical memory back to the runtime system.

Prototype

```
int uvm_clearSegment(int viewId, int segmentNumber);
```

Arguments

The `viewId` is the view visible from the current processor

The `segmentNumber` is a currently mapped segment.

Returns

An integer equal to 0 on success or a negative value on error.

Errors

`uvm_e_wrongView` if the view ID refers to a view other than the one visible from the processor from which the function was called

`uvm_e_badSeg` if the specified segment number does not correspond to a mapping

Described in

Section 4.2

Also see `HAL_CLEAR_SEGMENT ( )` in the TVM-HAL document.

uvm_setSegment

Request the kernel map a range of physical memory to a virtual address.

Prototype

```
int uvm_setSegment(  
    int viewId,  
    int segmentNumber,  
    void *startingVirtualAddress,  
    int64 vsize,  
    int bank,  
    int64 offset,  
    bool cachable,  
    bool cacheCoherent,  
    bool executable,  
    bool executeOnly,  
    bool writable,  
    bool mapped,  
    int64 mapPhysicalSize,  
    int interleaveSize,  
    void **returnVirtualAddress);
```

Arguments

The `viewId` is the view visible from the current processor.

The `segmentNumber` corresponds to an unused segment number for the view.

The `startingVirtualAddress` is a starting virtual address or `(void*)NULL` to allow the kernel to select the address.

The `vsize` indicates the size of the virtual address in bytes whose starting point is defined by `startingVirtualAddress`.

The `bank` is an integer indicating a memory bank to map, or `uvm_pmemBankAny` to indicate the system can choose a bank.

The `offset` is an integer indicating an offset in bytes into the memory bank, or `uvm_pmemOffsetAny` to allow the system to choose any free physical memory in the bank. If `bank` is `uvm_pmemBankAny` then `offset` must be `uvm_pmemOffsetAny`.

The `cachable` is a Boolean bit for mode control.

The `cacheCoherent` is a Boolean bit for mode control.

The `executable` is a Boolean bit for mode control

The `executeOnly` is a Boolean bit for mode control

The `writable` argument is a Boolean bit for mode control.

The `mapped` argument is a Boolean where *True* indicates a large virtual range is mapped to a small physical range.

The `mapPhysicalSize` is the size in bytes of physical memory being requested. The virtual address range may be much greater.

The `interleaveSize` suggests a combo-block interleave factor to the kernel. See the TVM-HAL document for more information on interleave factors and combo physical memory configurations.

The `returnVirtualAddress` is a pointer to a (`void*`) where the virtual address assigned by the kernel is returned.

Returns

An integer equal to 0 on success or a negative value on error.

Errors

`uvm_e_wrongView` if the `viewId` is not visible from the current processor

`uvm_e_badSeg` if the specified segment number is not available.

`uvm_e_badVrange` if the range specified by the `startingVirtualAddress` and the `vsize` arguments overlaps with any other mapping for the view.

`uvm_e_offset` if the memory bank is specified to be `uvm_pmemBankAny` and the `offset` is not specified to be `uvm_pmemOffsetAny`.

`uvm_e_bankNotVisible` if the memory bank is not visible from the view associated with the process.

`uvm_e_badPA` if the physical address is not valid.

`uvm_e_inUse` if the specified physical address using `bank` and `offset` are in use by another process.

`uvm_e_psize` if no region of the specified size in the specified bank is available.

`uvm_e_mode` if the combination of mode bits for `cachable`, `cacheCoherent`, `executable`, `executeOnly`, and `writable` are not supported.

Described in

Section 4.2

Also see `HAL_SET_SEGMENT(...)` in the TVM-HAL document.

`uvm_pinSegment`

This function requests that a runtime system not change the VA to PA mapping for a segment of memory.

Prototype

```
int uvm_pinSegment(int viewId, int segmentNumber);
```

Arguments

`viewId` is the ID of the view available for the current processor.

segmentNumber of a region of memory.

Returns

An integer equal to 0 on success or a negative error code on failure.

Errors

uvm_e_perm indicates that the calling thread does not have permission to pin the specified region.

uvm_e_badArg if the specified view/segment is not a mapped view/segment.

Described in

Section 4.2

uvm_unPinSegment

Tell the operating system that the application no longer depends on knowing the physical address of a segment.

Prototype

```
int uvm_unPinSegment(int viewId, int segmentNumber);
```

Arguments

viewId is the ID of the view available for the current processor.

segmentNumber of the region of memory.

Returns

An integer equal to 0 on success or a negative error code on failure.

Errors

No error codes are currently defined.

Described in

Section 4.2

uvm_getStartPA

Returns the physical base address for a segment by setting a pointer of type `PhyPtr*` passed in as an argument to the function.

Prototype

```
int uvm_getStartPA(int viewId, int segmentNumber, PhyPtr *resultPtr);
```

Arguments

viewId is the ID of the view available for the current processor.

segmentNumber of the region of memory.

`resultPtr` is set to the requested value on a successful call

Returns

An integer equal to 0 on success or a negative error code on failure.

Errors

`uvm_e_badArg` if the specified view/segment is not a mapped view/segment.

`uvm_e_unpinnedMem` if the requested segment is not pinned.

Described in

Section 4.2

7. References

1. The Polymorphous Computing Architectures (PCA) Program, Information Processing Technology Office (IPTO), Defense Advanced Research Projects Agency (DARPA), <www.darpa.mil/ipto/Programs/pca/index.htm>.
2. The PCA Morphware Forum. <www.morphware.org>.
3. “Introduction to Morphware: Software Architecture for Polymorphous Computing Architectures.” Version 1.0, February 23, 2004. Available at <www.morphware.org>.
4. L. Hammond. TVM-HAL specification, June 2003.
5. T. Anderson, B. Bershad, E. Lazowska, and H. Levy. ”scheduler activations: Effective kernel support for the user-level management of parallelism”. ACM Transactions on Computer Systems, 10(1):pp 53–79, February 1992.

Standards

6. “Portable Operating System Interface (POSIX)”, IEEE Standard 1003.1, 2004 Edition, Available at <www.ieee.org>.

APPENDIX G

Last Available Version of the Morphware Stable Interface Document

SVM ISSUES SUMMARY

February 2, 2007

SVM Issues Summary

February 2, 2007

There are three sections of this document:

- **SVM Open Issues**, which need a proposal, discussion, and/or decision by the Forum; and those that have been deferred, possibly indefinitely;
- **SVM Pending Issues**, which have been resolved but which still require editing of the SVM (and possibly other documents) to reflect the decision; and
- **SVM Closed Issues**, which are reflected in document, or have been dismissed as obsolete or no longer of concern.

Issues are colored by priority:

Red issue: high priority/urgent
Yellow issue: routine priority
Blue issue: low priority/not urgent/deferred
Green issue: closed

Some recent issues may not yet be assigned a priority.

SVM Open Issues

19	Request for the ability to express in-lane alignment of stream data and kernel computation. [from Merrimac users of SVM]
Proposal	Performance consequences already expressed in MM. Need to agree on convention for SIMD lane addressing, or add convention field to MM. Either: a. $\text{address \% num\_SIMD\_cores} = \text{SIMD\_lane}$ B. $\text{address / (local memory size / num\_SIMD\_cores)} = \text{SIMD\_lane}$
8/19/04	The Forum tentatively decided to adopt an MMX-like SIMD alignment, where words are striped across lanes, pending specific formalization of this proposal in the MM. See the MM specification and MM Issues document for the current implementation and issues related to SIMD data alignment.
6/21/05	Deferred.
Status	No immediate need to resolve this issue within PCA program.

20	Request for the ability to express scatter add operations. [from Merrimac users of SVM]
Proposal	Add versions of block-destination kernels that support an <code>updateOp</code> field. Add MM fields that indicate which <code>updateOps</code> (if any) supported. HLC may use this ability, but is not required to and should always be able to generate code for architectures without it.
8/19/04	This capability will not be added to the SVM at this time. Reservoir will work outside of the Morphware Forum with groups needing this capability.
Status	No immediate need to resolve this issue within PCA program.

39	A method is needed to turn off certain optimizations in the HLC, specifically including inlining.
Proposal	No specific proposal yet
8/25/05	Issue introduced by UT. UT suggested a command line switch as one option.
11/30/05	Reservoir states that R-Stream 2 cannot do this, but it could be considered for R-Stream 3.
Action	No action will be taken at this time.
Status	Deferred.

40	Interprocessor communication
Proposal	At 6/21/05 meeting, UT proposed implementing void* svm_ConfigGlobal(uint proc_id, size_t size, int id)
8/25/05	Issue introduced by UT, discussed by Forum.
11/30/05	Discussed by Forum; issue appears to be matching up multiple global memories with multiple processors.
Action	Resolution deferred to SVM 2.
Status	Deferred.

41	Allocation of global data structures, and dynamic vs. static data distribution.
Proposal	At 6/21/05 meeting, UT raised this issue and suggested developing a form of malloc with distribution directives, for example void* smalloc(uint _size, uint proc_id) UT also suggested static allocation might not be needed but, if it is, one possibility would be <variable-type> attribute(shared(<proc_num>))<variable name>;
8/25/05	Issue introduced by UT, discussed by Forum.
11/30/05	Discussed by Forum; issue appears to be similar to #40, except for global data structures instead of global memories. UT still believes LLC needs a way to distinguish private global and public global data structures.
Action	Resolution deferred to SVM 2.
Status	Deferred.

42	Machine allocation and configuration
Proposal	At 6/21/05 meeting, UT requested better specification of svm_Config*() calls, including specifically svm_setLevelConfig()
8/25/05	Issue introduced by UT, discussed by Forum.
11/30/05	Reservoir believes that this is a documentation issue only.
Action	Unresolved. Need clarification of specification improvement needed, comparison to SVM documentation.
Status	Open.

43	svm_kernelRun is nonblocking; it is possible for kernel data to be destroyed before the kernel is run
Proposal	Either required kernel data present until kernel execution ends, or add svm_kernelFinalize(). USC/ISI recommends the former.
11/28/05	Issue added at request of USC/ISI (East).
11/30/05	Discussed at length in meeting.
Action	Issue not resolved. Further offline discussion between Reservoir, Mercury, and ISI needed. Update status at next meeting.
Status	Open.

45	Need extensions to SVM memory model for GPUs and others (maybe FPCA?) (segmented, local memories); difficult for translator/library implementations
Proposal	No specific proposal yet
11/30/05	Issue introduced by GT+Reservoir
Action	Reservoir and/or GT submit a specific proposal.
Status	Deferred to SVM 2.

46	No SVM call to indicate when a block is done; difficult for translator/library implementations
Proposal	No specific proposal yet
11/30/05	Issue introduced by GT+Reservoir
Action	Reservoir and/or GT submit a specific proposal.
Status	Open.

47	SVM lacks n-D indexing. Some algorithms and target architectures (e.g. GPU “textures”) may benefit from n-D indexing in SVM
Proposal	No specific proposal yet
11/30/05	Issue introduced by GT+Reservoir
Action	Reservoir and/or GT submit a specific proposal.
Status	Deferred to SVM 2.

48	SVM “concept of operations” or “model of execution or computation” is unclear; conflates model of computation, mapping of app to hardware, and C-language API.
Proposal	Update the model of computation. A proposal by Charlie Garrett of Reservoir in the “SVM on Cell” document is a good starting point.
11/30/05	Issue introduced in Fall 2005, discussed at 11/30/05 meeting.
Action	Reservoir write concept of operations section for document.
Status	Open.

49	SVM “kernel” concept unclear; encompasses function, data structure, state, and initiation under same terminology.
Proposal	Change terminology for fundamental task unit, separate naming of different entities and concepts. See also SVM issues 55-59.
11/30/05	Issue introduced in Fall 2005, discussed at 11/30/05 meeting.
Action	Specific proposal needed.
Status	Open.

50	SVM limited to “big VLIW” temporal decomposition execution model, unworkable for some applications (MPEG, IRT).
Proposal	None.
11/30/05	Initial discussion.
Action	Specific proposal needed.
Status	Open.

51	Use of C as an “assembly language” is limiting, esp. in expression of dependencies.
Proposal	Consider “C--” (www.cminusminus.org) as a source of alternative ideas.
11/30/05	Initial discussion.
Action	Specific proposal needed.
Status	Open.

53	No coordination of this SVM Issues list and Reservoir SVM bug list.
Proposal	Discuss coordination of lists. One list, or separate SVM Design issues from SVM/HLC bug list?
11/30/05	Initial suggestion.
Action	GT+Reservoir decide and act accordingly.
Status	Open.

54	Multiple extensions needed to the execution model to accommodate component model “deployment”. These include const/volatile specs for properties in the interfaces, hierarchical node naming semantics, sub-domain execution managers, morphing capability.
Proposal	None.
11/30/05	Initial suggestion in context of “component model” discussion.
Action	Specific proposal needed.
Status	Open.

55	Multiple issues with kernel state access per presentation by Jim Kulp on 11/30/05: - o Kernel state access should be via library call - o Kernel state elements should be characterized as initialization only, changeable from outside only, changeable from inside only, readable only in “finished” state - o Provision is needed for batch access to multiple state values - o Standard generic state elements should be defined (e.g. current kernel state, execution statistics)
Proposal	None.
11/30/05	Initial suggestion in context of “component model” discussion.
Action	Specific proposal needed.
Status	Open.

56	Kernel interface, implementation, and state elements should be defined using macros
Proposal	SVM_KERNEL_INTERFACE, SVM_KERNEL_IMPLEMENTATION, and SVM_KERNEL_STATE macros proposed; see Jim Kulp charts of 11/30/05 for additional detail.
11/30/05	Initial proposal in context of “component model” discussion. See also SVM issue #49.
Action	Awaiting discussion/resolution.
Status	Open.

57	Kernel initialization process needs to be better defined, supporting SVM constructs implemented. Need to address obtaining initial state values, a reset capability, loading of work function on target. See Jim Kulp charts of 11/30/05 for addl. discussion.
Proposal	svm_KernelInit should carry initial state and I/O bindings, svm_KernelReset should be defined; see Jim Kulp charts of 11/30/05 for additional detail.
11/30/05	Initial proposal in context of “component model” discussion.
Action	Awaiting discussion/resolution.
Status	Open.

58	Parameterize and unify streams, FIFOs, and blocks
Proposal	No specific proposal yet.
11/30/05	Initial proposal in context of “component model” discussion.
Action	Specific proposal needed.
Status	Open.

59	Provide “connectivity kernels” for barrier synchronization and similar functions.
Proposal	No specific proposal yet.
11/30/05	Initial proposal in context of “component model” discussion.
Action	Specific proposal needed.
Status	Open.



SVM Pending Issues

There are no pending issues at this time.

SVM Closed Issues

1	Aliasing is not necessary with the new set of built-in kernels and could be removed. Its use is mentioned in several sections for both streams and blocks.
Solution	Leave aliasing in for now
8/19/04	The Forum accepted the recommendation.
Action	No action required
Status	Closed

2	mm_Mem is not a C type; it's an object in the PCA Machine Model. This needs to be changed. It may be appropriate to change it to a <code>void*</code> or a <code>char*</code> . Furthermore, the relationship between the namespace of the machine model and the SVM API should be defined.
Proposal	Use <code>char*</code> , argument must be string literal matching MM name.
8/19/04	The Forum accepted the recommendation.
Action	Change the document to use <code>char*</code> arguments that are string literals matching the machine model name
Status	HW IDs are now implementation-specific types.

3	The function of the <code>maxBuffering</code> field of <code>svm_streamInitRam</code> and <code>svm_streamInitFIFO</code> needs to be defined better and its functionality needs to be proven if it really does prevent deadlock. As deadlock is a correctness issue and not a performance issue, it is not appropriate to call this a "performance hint." Conversely, if the issue is only performance, deadlock should not be mentioned.
Solution	Omit <code>maxBuffering</code> for now. It needs more thought.
8/19/04	The Forum accepted the recommendation.
Action	Omit the <code>maxBuffering</code> field from the document. That is, remove all mention of <code>maxBuffering</code> from parameter lists and API descriptions.
Status	Closed

4	Does <code>streamInitWithDataFIFO</code> require <code>initLength</code> as an input?
Action	Mike Vahey
8/19/04	The Forum tabled this item pending a recommendation by the MONARCH team, to be presented before the next Forum meeting.
Action	The <code>streamInitWithDataFIFO</code> call has since been removed from the SVM.
Status	Closed

5	Because the capacity of a block “might be dynamically computed at runtime,” some provisions should be made for ensuring that the calculated capacity does not exceed the resource as defined in the machine model.
Proposal	Leave as undefined behavior. High performance systems should not be burdened w/ lots of run-time checks.
8/19/04	The Forum accepted the recommendation.
Action	Ensure the document reflects that runtime checking of dynamically allocated blocks is not defined or required.
Status	Closed

6	Space for spilling local variables is “optional” and not supported by some PCA architectures. This indicates that some provision for HLC-LLC feedback should be made. This may or may not be contained in the SVM API.
Proposal	Remove "optional". Spilling is allowed if a RAM is directly connected to stream processor. Spilling is not allowed if no RAM is directly connected. If spilling is allowed, then the HLC should allocate a block. The HLC may allocate a block of zero size if it believes spilling is not required. A built-in kernel never spills. Black-box kernels obey this convention.
8/19/04	The Forum rejected the recommendation. Reservoir should propose an additional field(s) to the MM that indicates whether a processor can, or can not, spill.
6/22/05	SupportSpilling field added to the MM.
Status	Closed

7	The “optional data type that represents kernel data” mentioned in Section 5 is poorly defined.
Explanation	A <code>struct</code> that contains fields used to pass arguments to, and return results from, the kernel. Henceforth called the kernel argument <code>struct</code> .
8/19/04	Not discussed. Adopt explanation above
Resolution	In Section 4 define the kernel argument <code>struct</code> in the document as “A <code>struct</code> that contains fields used to pass arguments to, and return results from, the kernel.”
Status	Fixed in document. Issue closed.

8	The examples of the use of <code>kernelAddDependence</code> in Section 5.2 should illustrate the functionality of the kernels in code. Currently, their functionality is stated in the comments and the examples do not make their intended points.
Explanation	Code is correct, but could be clarified by: - Adding stream argument “s” to <code>myKernelInit</code> calls in Section 5.2, to show that communication is via stream. - For the loop example, re-declare the block “b” to show that communication is via a block. Expand explanation paragraph (above 5.3) to make clear that k1 is also dependent on k2.
8/19/04	Not discussed. Adopt explanation above.
Action	Incorporate the Explanation in the document (if not already there)
Status	Fixed code, clarified text. Issue closed.

9	Figure 3 and <code>kernelInit</code> refer to “state variables,” which are never clearly defined.
Explanation	Should refer to kernel argument <code>struct</code> . (Also refer to it elsewhere as appropriate.)
8/19/04	Not discussed. Adopt explanation above
Action	Incorporate the Explanation into the document (if not already there)
Status	Fixed in document. Issue closed.

10	The multiplexing of kernels is not well defined. “Iteration” and “yielding” need to be explicitly defined. Also, the implementation of the built-in kernels such that they may be multiplexed needs to be explained.
Proposal	Remove multiplexing from the SVM and MM. Most real DMA engines can't even do simple streaming DMA. If multiplexing is supported, then represent it "multiple" DMA engines sharing a common link to limit total bandwidth. The HLC can always achieve multiplexing manually.
8/19/04	The Forum postponed a decision on this issue, pending further discussion.
6/22/05	All mention of multiplexing removed from document.
Status	Closed. May be re-opened in future for SVM 2.

11	It is unclear if <code>kernelPause</code> applies to black-box and built-in kernels.
Proposal	<code>kernelPause</code> applies to black-box kernels but not to built-in kernels
8/19/04	The Forum tabled the <code>kernelPause</code> issue pending a recommendation by Jim Kulp regarding the “accessor” alternative, to be presented before the next Forum meeting.
Action	This was resolved at the March 2005 meeting: There is a new machine model parameter to indicate whether built-in kernels can be paused.
Status	Closed

12	The logic behind why pausing a kernel in a set of kernels causes <code>kernelWaitMultiple</code> to return is not clear. It should be explained.
Explanation	<code>kernelWaitMultiple</code> is usually used to handle two or more kernels which may pause unexpectedly (to respond to an error condition, for instance). Since it is not known which kernel will pause first, two sequential <code>kernelWait</code> calls cannot be used.
8/19/04	Adopt explanation above
Action	Incorporate Explanation into document
Status	Closed.

13	Upon termination of a kernel, Section 6.2 states that the results from that kernel may be used by the control thread. When can the control thread release the memory used for the kernel? Suggest adding a <code>DONE</code> state.
Proposal Agreed	Memory can be freed when kernel variable goes out of scope (meant to be stack allocated.) The HLC guarantees that <code>kernelWait</code> is called on running kernels before they go out of scope (see issue 21).
Action	Incorporate into document.
Status	Closed.

14	Section 6.3 states that FIFOs must be drained before mapping new streams to them. However, it is possible to specify that a FIFO be buffered at the source using the <code>maxBuffering</code> “performance hint.” How can we drain a FIFO if it has been buffered?
Solution	Remove <code>maxBuffering</code> for now (as in issue 3).
8/18/04	The Forum accepted the recommendation.
Action	Incorporate solution into document
Status	Document updated. Issue closed.

15	Section 6.4 should explain why each of the restrictions to SVM API code is necessary to allow static analysis in the LLC. (This refers to the second set of restrictions.)
Explanation	Overall philosophy: the LLC should be able to do simple translation of each LLC construct without extensive analysis. The LLC should be able to translate each stream, block, or kernel instance in a largely static context. In particular, hardware resources and synchronization involved should always be the same for a given instance. (Different contexts may be handled by different instances and control flow). Hence: [restriction 1]. The LLC should not need to do global analysis to determine which SVM API calls refer to the same streams, blocks, or kernels. Aliasing of streams, blocks, and kernels is not allowed. Hence: [restrictions 2, 3]. The LLC should be able to treat each control thread as a separate compilation problem. Hence: [restriction 5]. This discussion led to some proposed adjustments to the restrictions, described below.
Proposal (Part 1)	Strengthen restriction 1. It current reads as follows: “If there are re-initializations of a given kernel instance, then the same stream and block arguments must be passed to each invocation of the initialization function.” Change it to the following: There may be only one initialization call for a stream, block, or kernel instance. All calls to kernelAddDependence for a given kernel instance must follow the initialization call within the same basic block.”
Proposal (Part 2)	Strengthen restriction 3 to account for arrays: Stream, block, and kernel variables may not be part of structures or arrays except for extensions of the kernel base data type.
Proposal (Part 3)	Remove restriction 4: Any function that contains a stream, block, or kernel variable must not be called recursively or through a function pointer, even indirectly (e.g., it cannot be called from a recursive function or called from a function called from a recursive function). This restriction was intended to allow inference of access to global stream and kernel variables, but such global variables are now prohibited (see issue 30). Note that new restrictions are proposed in issue 21 and 30.
8/19/04	Not discussed. Adopt above proposal
Action	Incorporate Explanation and proposal solutions into document
Status	Above proposals made obsolete by resolution of issue 28: kernels are just threads with swapping restrictions. Issue closed.

16	The FIFO cleanup explanation and example in Appendix A is problematic. Could we have an <code>svm_drainFIFO</code> command instead of placing this responsibility on the HLC? The requirement for the HLC to create custom draining kernels may not be feasible.
Proposal	Modify semantics of <code>kernelEnd</code>: <code>kernelEnd</code> will interrupt a kernel even if it is blocked in a push or pop operation. This permits the following simple protocol for draining FIFOs that connect communicating kernels: - control thread calls <code>kernelEnd</code> up on upstream, downstream kernels - control thread calls <code>kernelWait</code> on upstream, downstream kernels - new kernel upstream pushes capacity items, then pushes EOS - new kernel downstream pops capacity items, then pops until EOS Note that this does not fully resolve FIFO draining for streams accessed from the control thread; this issue preserved as part of issue 28.
8/19/04	The Forum tabled this item pending a recommendation by the MONARCH team, to be presented before the next Forum meeting.
Action	MONARCH team makes a recommendation before the December 04 meeting.
6/21/05	<code>svm_drainFIFO</code> command added.
Status	Closed.

17	Appendix B uses dynamic memory allocation within kernel work functions, which is prohibited by the SVM API. Why does the SVM API specification document include code that does not adhere to the rules that it sets forth?
Proposal	Remove reference code from specification. It over-specifies the intended behavior; is redundant with SVM text; and is hard to maintain.
8/19/04	Not discussed. Adopt above proposal
Resolution	Remove reference code from specification (if not already removed). Closed.

18	Some of Appendix B is incorrect, not reflecting changes that have been made in the main document to calling arguments of the functions. For example, in several places the deleted constant <code>svm_Stream_Length_All</code> is still used instead of the <code>untilEOS</code> flag. Implementations for some functions don't appear. There may be other problems.
Proposal	Remove reference code from specification (rationale in issue 17).
8/19/04	Not discussed. Adopt above proposal
Resolution	Remove reference code from specification (if not already removed).
Status	Closed.

21	It is not clear what the semantics are when running kernels or non-empty FIFO streams go out of scope in the code. Do they continue to persist, or are they automatically terminated by the LLC?
Proposal Agreed to	Add this restriction (as #5 in the first set): when a kernel variable goes out of scope, the kernel must be UNSTARTED or FINISHED. When a FIFO-based stream goes out of scope, the stream must be empty.
8/19/04	The Forum accepted this recommendation pending further review and possible additional or different recommendations by the MONARCH team, to be presented before the next Forum meeting.
Resolution	Incorporated in document, confirmed March 2005 Morphware meeting.
Status	Closed. See also issue 13.

22	How is memory layout for streams handled w/EOS in RAM? - Presumably EOS takes up room. Is the EOSOverhead (bits/record) field in the MM? If the layout of streams is defined where do these bits reside? - Since you can't push EOS unless you have an item to attach to it how do we handle EOS for empty streams?
Proposal	Introduce a new model of EOS. Instead of expressing EOS as tags on other data items, an EOS is an item itself. The API is: pushEOS() – pushes an EOS item pop(), peek(i) – as before. Returns undefined data for EOS items. checkEOS() – returns true if last item popped or peeked was EOS. peekEOS(i) – equivalent to peek(i) followed by checkEOS(). In this model, a producer looks like: <pre>while (...) { push(); } pushEOS();</pre> A consumer looks like: <pre>pop(); while (!checkEOS()) { ... pop(); }</pre> There are several advantages to this model of EOS: - supports empty streams (EOS with no other items) - allows EOS on streams that do not support peek - better accounts for space of EOS items in FIFOs - might simplify kernel code
8/19/04	The Forum tabled this item pending a recommendation by the MONARCH team, to be presented before the next Forum meeting.
6/22/05	The proposal was rejected in favor of the existing approach to EOS. While the existing EOS approach is clean in many ways, it still does not explicitly address the EOS Overhead for memory streams and this remains an open issue. However, a discussion of the issue should be added to the “execution model” section of the SVM document.
Action	Reservoir to discuss in “execution model” section of SVM document.
Status	EOS functionality is covered in detail in section 3.0. Added LLC/run-time responsibility for allocating EOS to text in that section. Issue closed.

23	Should the HLC explicitly allocate space for the kernel argument struct, and use DMA to transfer the struct to a memory accessible by the kernel?
Action	This is better left to the LLC, but feedback is needed from LLC writers to determine if they want the HLC to do this.
8/19/04	The Forum accepted the recommendation.
Action	Leave allocation of space to the LLC. Reflect this decision in the document
Status	Noted in document, section 5.0. Issue closed.

24	Is SVM based on C89 or C99
Action	Not sure. What are the tradeoffs? Discuss at meeting.
8/19/04	The SVM should be based on C89.
Action	Reflect the decision in the SVM document.
Status	Added C89 note to document. Issue closed.

25	Should there be a representation for a multidimensional array access into a block to aid the LLC with recognizing the access pattern.
Action	Not Sure. What information do LLC compiler writers want? Discuss at meeting.
8/19/04	No such representation should be implemented at this time, pending further study.
6/22/05	Closed for SVM 1.x. May be re-opened for SVM 2.0.
Status	Closed

26	There are restrictions on kernels, which are driven by the assumption that kernels are executed by processors that only have local memories. This assumption leads to, essentially: 1. No access to global state 2. No pointers? Can you have a pointer into local memory? 3. Bounded size stack But what about kernels on processors with caches, or control code on processors with local memories?
Proposal	Place no restrictions on what happens inside a kernel. Make the current restrictions dependent on caches/memories the processor can access: if you have a local memory, you have these constraints.
8/19/04	The Forum accepted the recommendation.
6/22/05	Local memory definition should be covered in the "execution model" section of the SVM document.
Action	Reservoir to discuss in "execution model" section of SVM document.
Status	Fixed in document. Moved to machine model section. Issue closed.

27	Does the HLC need to tag anything volatile?
8/19/04	The SVM probably should have something for accessing kernel state. There is both a user syntax question and an implementation question to resolve. Jim Kulp will propose an approach as part of his state accessor proposal.
3/30/05	Jim Kulp proposed a parameter be added to <code>svm_kernelInit</code> to indicate the ability to read external data. The Forum accepted the recommendation.
Action	<code>bool ctrlReadsExtKernelData</code> added to <code>kernelInit</code>
Status	Closed

28	(related to issue 16) How is the HLC complicated by compiling to a set of control threads instead of a set of kernels?
Proposal	Proposed at 8/19/04 meeting: A single “primary master” processor calls kernels on “secondary master” processors. Thus, kernels are like threads, except that they cannot be swapped and one can call a kernel on a processor while the processor is busy without interrupting it.”
8/19/04	The Forum accepted the recommendation.
Action	Incorporate into the document, should be done by somebody who understands the issue.
Status	Clarified document: kernels are just threads swapping restrictions. Closed.

29	Since you can’t peek an unpeekable stream we need to add a constraint on <code>streamPeek</code> and <code>streamPeekEOS</code> .
Proposal	Add the following constraint on <code>streamPeek</code> and <code>streamPeekEOS</code> : If the stream is allocated to a FIFO, then <code>FIFOPeek > 0</code> for that FIFO in the machine model
8/19/04	Adopt above proposal.
Action	Incorporate the proposal into the document.
Status	Closed.

30	Global variables are difficult to analyze with local reasoning. Should we add a restriction that prevents kernels, stream, and block variables from being global or dynamically allocated?
Proposal	Add the restriction: All kernel, stream, and block variables must be declared as local variables within a function. They cannot be global variables.
8/19/04	The Forum accepted the recommendation.
Action	Incorporate the proposal into the document.
Status	Closed.

31	Should there be <code>kill</code> functionality to allow a kernel to be asynchronously stopped?
Proposal	Clarify SVM document to indicate that <code>kernelEnd</code> and <code>kernelPause</code> are asynchronous already.
6/21/05	The Forum accepted the recommendation.
Action	Reservoir updated the SVM document.
Status	Closed.

32	Replace string HW IDs with integer hardware IDs
6/21/05	HAL replaced char * names with integer IDs.
Action	Reflect the change in the SVM document.
Status	Subsequent debate concluded that HW IDs should be implementation specific types. Changed accordingly. Issue closed.

33a	Need a means to derive low-overhead resource IDs from the string-based IDs specified by SVM 1.0.
Proposal	Three options suggested by Reservoir at 6/21/05 meeting: 1. Use a header file to specify integer IDs at compile time <pre>#define TProc_0_0_0_1_0 36</pre> 2. Use a function call to get an integer ID at run time ... <pre>int TProc_0_0_0_1_0 = svm_getResourceID("TProc_0_0_0_1_0")</pre> 3. ... or pointer to architecture specific struct <pre>mm_Proc* TProc_0_0_0_1_0 = svm_getResourceID("TProc_0_0_0_1_0")</pre> Of these, Reservoir proposed using #3.
6/21/05	Forum accepted the proposal with some modifications.
Action	Update the MM to reflect new naming convention.
Status	Naming convention added as section 4 of MM document. Closed.

33b	Need a means to specify which morphs are configured.
Proposal	Suggested by Reservoir at 6/21/05 meeting: Add new fields to root of machine model: • <DefaultMorphs> a list of morphs configured by default • <InitialProcessor> a processor to execute main() on Add functions: • svm_setLevelConfiguration(char* levelName, char* morphNames); • svm_getLevelConfiguration(char* levelName, char* morphNames); Must specify all morphs for level (and child levels) even if configured before call.
6/21/05	The Forum accepted the proposal.
Action	Added DefaultMorph and InitProcessor to Root object in machine model. Added new Machine Model Interaction section to SVM. Replaced all instances of mm_Proc and mm_Mem with svm_Processor and svm_Memory.
Status	Closed.

34	Blocks are currently passed by reference; should be passed by value
Proposal	Pass by value
6/21/05	The Forum accepted the recommendation
Action	Reflect the decision in the SVM document.
Status	Fixed in document. Issue closed.

35	Overhead for block and stream I/O implemented as function calls is far too high
Proposal	Implement as macros instead: <pre>#define SVM_BLOCKREAD(block, offset, value, type) #define SVM_BLOCKWRITE(block, offset, value, type) #define SVM_STREAMPEEK(stream, depth, value, type) #define SVM_STREAMPOP(stream, value, type) #define SVM_STREAMPUSH(stream, value, type)</pre>
6/21/05	The Forum accepted the recommendation.
8/25/05	Issue re-opened by Reservoir due to experiments showing that performance of macro approach is still inadequate.
11/30/05	Reservoir implemented new version of macros that reduces to array accesses.
Action	Forum accepted the recommendation to implement I/O as macros, using the new implementation.
Status	Fixed in document. Issue closed.

36	Inadequate MM information passed to the LLC by the SVM
Proposal	At 8/25/05 meeting, UT suggested the following be passed: - o # of stream processors and global memory - o Size of SRF banks and global memory - o Distinction between primary and slave processors - o Maybe a de-allocation routine at end of program
Revised Proposal	At 11/30/05 meeting, proposal was altered to - o Provide a global memory flag to indicate from which memory the HLC will load data, with an unspecified change in SVM 2 - o SVM code will designate how many processors will be utilized, including both processors and DMAs; and - o Add a finalization call at the end of SVM routines.
8/25/05	Issue introduced by UT.
11/30/05	Discussed at Forum meeting.
Action	The Forum accepted the revised proposal
Status	Added additional parameters to the <code>setConfiguration</code> call to convey this information. Also added <code>finalizeConfiguration</code> call. Issue closed.

37	Initialization routine should be invoked only for resources actually used
Proposal	Create or rename a call to provide a clearly-identified capability for enumerating resources.
8/25/05	Issue introduced by UT.
11/30/05	Proposal introduced by Reservoir in response to discussion at meeting.
Action	Proposal accepted by the Forum.
Status	Subsequent debate concluded that no initialization was needed. Instead, there should be calls to obtain efficient implementation-specific hardware IDs and calls to configure morphs. Changed document accordingly; calls are described in machine model interaction section.

38	Need a better method for notifying LLC of the end of a work function
Proposal	No specific proposal yet
8/25/05	Issue introduced by UT. UT indicated preference for updating a shared data structure over sending of a message.
11/30/05	Reservoir indicates that the issue was caused by an HLC bug that has been fixed.
Action	No further action required.
Status	Closed.

44	Cache coherency and boundaries
Proposal	HLC allocate a cache line in a block to a stream processor if the block is not read-only, and the hardware does not support cache coherency..
11/28/05	Issue added at request of USC/ISI (East).
11/30/05	Proposal introduced by USC/ISI (Jinwoo Suh). See also coordinated proposal for MM issue #4.
Action	Accepted by the Forum; however, it is not planned to add this capability the R-Stream HLC at this time.
Status	Noted restriction in document. Issue closed.

60	Provide a means for a processor to copy back a kernel's arguments
Proposal	Add a field to <code>kernelWait</code> to indicate whether a structure will be returned.
11/30/05	Issue raised, proposal offered during discussion of SVM issue 38.
Action	The Forum accepted the proposal.
Status	Added <code>returnKernelArgStruct</code> argument. Issue closed.

52	No facility for SIMD memory alignment, crucial for performance.
Proposal	None. See also SVM issue #19.
11/30/05	Initial discussion.
Action	Incorporated into issue #19.
Status	Closed.

APPENDIX H

Last Available Version of the Morphware Stable Interface Document

MACHINE MODEL (MM) ISSUES SUMMARY

February 7, 2007

Machine Model (MM) Issues Summary

February 7, 2007

There are three sections of this document:

- **MM Open Issues**, which need a proposal, discussion, and/or decision by the Forum; and those that have been deferred, possibly indefinitely;
- **MM Pending Issues**, which have been resolved but which still require editing of the MM (and possibly other documents) to reflect the decision; and
- **MM Closed Issues**, which are reflected in document, or have been dismissed as obsolete or no longer of concern.

Issues are colored by priority:

Red issue: high priority/urgent

Yellow issue: routine priority

Blue issue: low priority/not urgent/deferred

Green issue: closed

Some recent issues may not yet be assigned a priority.

MM Open Issues

21	Data alignment models.
Proposal	Raised at Dec. 2006 meeting in discussing SIMD processor parameters. Generally, the SIMD alignment requirements may need a more sophisticated model than the current inlane/crosslane parameters provide; Cell is an example.
12/05/06	Issue introduced by Reservoir
Action	Deferred to SVM 2.
Status	Deferred.

22	Network performance model.
Proposal	Raised at Dec. 2006 meeting in discussing network parameters, and in subsequent discussions . The current network model is <i>ad hoc</i> and should be upgraded to better model the range of targets.
12/05/06	Issue introduced by Reservoir
Action	Deferred to SVM 2.
Status	Deferred.

23	Cache parameters.
Proposal	Raised at Dec. 2006 meeting. Is a constant <code>CacheMissRate</code> useful? Is there a difference between <code>CacheMissSize</code> and <code>CacheLineSize</code> ? <code>CacheCohereny</code> likely needs amore detailed model than a Boolean yes/no.
12/05/06	Issue introduced by GT+Reservoir
Action	Deferred to SVM 2.
Status	Deferred.

24	FIFO parameters.
Proposal	Raised prior to Dec. 2006 meeting. There is no rate or bandwidth information on FIFO memories in current parameter list. Similarly, an indicator is needed for how much data a read can pop, and how that is related to the size of a FIFO.
12/05/06	Issue introduced by GT+Reservoir
Action	Deferred to SVM 2.
Status	Deferred.

25	Connection bandwidth model.
Proposal	Raised prior to Dec. 2006 meeting in discussing connection bandwidth parameters. Current memory parameters (<code>RAMLinearBandwidth</code> , <i>etc.</i>) ascribe bandwidth to memory at end-points of a connection, rather than to the connection itself.
12/05/06	Issue introduced by Reservoir
Action	Deferred to SVM 2.
Status	Deferred.

2	Operations supported by a given processor.
Proposal	At August 2005 Morphware meeting, suggested creating a mechanism to indicate which C operations are supported by a given processor. Example: MONARCH FPCA does not support <code>divide</code> .
6/21/05	Requested by MONARCH team.
11/30/05	MONARCH team (Matt Kramer) to make a specific proposal.
3/28/06	Add <code>OperationNotSupported</code> parameter to current list of processor parameters; type is list of <code>mm_Keyword</code> , <code>mm_Keyword</code> is <code>t_div</code> , <i>etc.</i> , and <code>t</code> is a type indicator (<code>char</code> , <code>short</code> , <i>etc.</i>) (Schweitz)
Action	Fill in remaining details (complete list of parameter options).
Status	Discuss at July 06 meeting.

5	No way to specify kernel fusion constraints for GPUs & others
Proposal	No specific proposal yet
11/30/05	Issue introduced by GT+Reservoir
Action	Reservoir and/or GT submit a specific proposal.
Status	Open.

6	MM lacks model of GPU hardware features such as interpolation, z-sort
Proposal	No specific proposal yet
11/30/05	Issue introduced by GT+Reservoir
Action	Reservoir and/or GT submit a specific proposal.
Status	Open.

7	Insufficient detail in SIMD models, <i>e.g.</i> startup cost and branch cost
Proposal	No specific proposal yet
11/30/05	Issue introduced by GT+Reservoir
Action	Reservoir and/or GT submit a specific proposal.
Status	Open.

8	MM cannot model no-scatter constraint of GPU and others
Proposal	No specific proposal yet
11/30/05	Issue introduced by GT+Reservoir
Action	Reservoir and/or GT submit a specific proposal.
Status	Open.

9	Insufficient detail in modeling GPU resource constraints, <i>e.g.</i> number of registers, cost of instruction combinations, cost of register usage vs. # of threads
Proposal	No specific proposal yet
11/30/05	Issue introduced by GT+Reservoir
Action	Reservoir and/or GT submit a specific proposal.
Status	Open.

10	Analytical/procedural machine model needed; XML representation not adequate to support HLC $\leftrightarrow$ LLC feedback.
Proposal	MM should be written in the HLC's intermediate representation.
11/30/05	Issue discussed, resolution proposed by Reservoir.
Action	Additional discussion suggested. Specific proposal needed from Reservoir.
Status	Open.

11	Improved memory modeling needed: clarification of kernel data structure locations, transfer of state, clarification of data structure aliasing, dependency representation, memory consistency, cache modeling.
Proposal	None.
11/30/05	Initial discussion.
Action	Additional discussion suggested. Specific proposal needed from Reservoir and/or Mercury.
Status	Open.

12	Need to express finite queues for efficiency.
Proposal	None.
11/30/05	Initial discussion.
Action	Specific proposal needed from Reservoir.
Status	Open.

13	No coordination of this MM Issues list and Reservoir MM bug list.
Proposal	Discuss coordination of lists. One list, or separate MM Design issues from MM/HLC bug lists?
11/30/05	Initial suggestion.
Action	GT+Reservoir decide and act accordingly.
Status	Open.

14	Support needed for hierarchy.
Proposal	None.
11/30/05	Initial suggestion in context of “component model” discussion.
Action	Mercury clarify issue, offer specific proposal.
Status	I thought we had this; what does Jim K. mean?

15	Support needed for scaling, and alternative configurations.
Proposal	None.
11/30/05	Initial suggestion in context of “component model” discussion.
Action	Mercury offer specific proposal.
Status	Open.

16	Support needed for interconnects that span the hierarchy.
Proposal	None.
11/30/05	Initial suggestion in context of “component model” discussion.
Action	Mercury offer specific proposal.
Status	Open.

17	Support needed for modeling inter-morph latency.
Proposal	None.
11/30/05	Initial suggestion in context of “component model” discussion.
Action	Mercury offer specific proposal.
Status	Open.

18 (see also RT issue #2)	Support needed to observe layout constraints when resources are re-allocated.
Proposal	Create “coordinated machine models” by assigning same ID to ingredients that should be shared across multiple MMs. (Thies)
3/28/06	Initial suggestion in context of run-time system discussion.
Action	MIT offer specific proposal(s) to implement.
Status	Discuss at July 06 meeting.

19 (see also RT issue #3)	Need to expose context switching cost to resource manager, to support multithreading.
Proposal	Modify machine model to expose cost. (Thies)
3/28/06	Initial suggestion in context of run-time system discussion.
Action	MIT offer specific proposal(s) to implement at July 06 meeting.
Status	Open

20	Need means to deal with “intrinsic”
Proposal	Extend MM with primary object mm_Intrinsic, with parameters Name and RequiredIngredient (Schweitz)
3/28/06	Initial proposal.
Action	Discuss at July 06 meeting.
Status	Open

MM Pending Issues

There are no current pending issues.

MM Closed Issues

1	Create a Machine Model issues list.
Proposal	At August 2005 Morphware meeting, Reservoir requested a Machine Model issues list be started.
Action	GT start list.
Status	Closed.

3	MM document numbering.
Proposal	At August 2005 Morphware meeting, Reservoir suggested changing numbering of MM document to synchronize to SVM numbering. Thus the current MM 2 draft would become MM 1.2 when released.
8/25/05	The Forum accepted the recommendation.
Status	Corrected version number to 1.2.

4	Cache coherency information
Proposal	Add capability to indicate whether hardware supports cache coherency. If not supported, provide information on cache line boundary.
11/28/05	Issue added by request of USC/ISI (East).
11/30/05	Proposal introduced by USC/ISI (Jinwoo Suh). See also coordinated proposal for SVM issue #44.
Action	Accepted by the Forum; however, it is not planned to add this capability the R-Stream HLC at this time.
Status	Added SupportsCoherency and CacheLineSize fields.

APPENDIX I

Last Available Version of the Morphware Stable Interface Document

RUN TIME (RT) ISSUES SUMMARY

May 22, 2006

Run Time (RT) Issues Summary

May 22, 2006

There are three sections of this document:

- **RT Open Issues**, which need a proposal, discussion, and/or decision by the Forum; and those that have been deferred, possibly indefinitely;
- **RT Pending Issues**, which have been resolved but which still require editing of the MM (and possibly other documents) to reflect the decision; and
- **RT Closed Issues**, which are reflected in document, or have been dismissed as obsolete or no longer of concern.

Issues are colored by priority:

Red issue: high priority/urgent

Yellow issue: routine priority

Blue issue: low priority/not urgent/deferred

Green issue: closed

Some recent issues may not yet be assigned a priority.

RT Open Issues

1	Portability of resource manager is problematic is resrouce untis are target architecture-specific.
Proposal	Make the unit of resource allocation be a PCA Machine Model. (Thies)
3/28/06	Initial suggestion in context of run-time system discussion.
Action	MIT offer specific proposal(s) to implement.
Status	Discuss at July 06 meeting.

2 (see also MM issue #18)	Support needed to observe layout constraints when resources are re-allocated.
Proposal	Create “coordinated machine models” by assigning same ID to ingredients that should be shared across multiple MMs. (Thies)
3/28/06	Initial suggestion in context of run-time system discussion.
Action	MIT offer specific proposal(s) to implement.
Status	Discuss at July 06 meeting.

3 (see also MM issue #19)	Need to expose context switching cost to resource manager, to support multithreading.
Proposal	Modify machine model to expose cost. (Thies)
3/28/06	Initial suggestion in context of run-time system discussion.
Action	MIT offer specific proposal(s) to implement.
Status	Discuss at July 06 meeting.

4	Difficulty in migrating stream programs to a different amount of resources.
Proposal	Accept “lossy” transitions, introduce programmer annotations to indicate tolerances. (Thies)
3/28/06	Initial suggestion in context of run-time system discussion.
Action	MIT offer specific proposal(s) to implement.
Status	Discuss at July 06 meeting.



RT Pending Issues

There are no current pending issues.

RT Closed Issues

There are no current closed issues. This is the first version of this document.

APPENDIX J

Final Consultant's Report

Mr. Randall Judd

SIGNAL PROCESSING LIBRARIES ON TILED ARCHITECTURES

Draft 0p4

June 24, 2007

Signal Processing Libraries on Tiled Architectures

Randall Judd

Draft 0p4

June 24, 2007

Introduction

The Problem and the Goal

Microprocessor technology is continuously evolving driven by requirements of the marketplace and competition between chip manufacturers. The goals are many but in general higher performance, lower power, and better usability in diverse [compute spaces](#) are desirable features.

In addition the military (DOD) also drives the technology since the requirements of DOD systems are frequently specialized with specifications for security, mechanical, power, and cooling that are not mirrored in the commercial sector. The SWEPT constraints (I think originating with Bob Graybill; Size, Weight, Energy, Performance, Time) demonstrates the problem. DOD applications frequently have diverse requirements for the size and weight of the hardware, and of course energy may be a problem for some platforms since the military frequently supplies its own and the platform supplying the power has limitations (of varying severity) on how much it can produce. Energy is a double edged sword since energy goes hand in hand with cooling the hardware. Finally data throughput and time (latency) to solution are important. These SWEPT requirements the military has are not as big a concern to most civilian applications.

Even though DOD has special needs they don't supply enough demand to drive the commercial market. Fiscal constraints requires buying off the shelf hardware in most situations.

Even though general purpose microprocessor ([GPMP](#)) architectures are rapidly advancing, the needs of the user community have not been fully met. This has lead to the use of graphic processor units ([GPU](#)) for special purpose processing for which GPUs were not originally designed. This highlights the need for special purpose processors ([SPP](#)). However one of the attractions of the GPU is the commodity nature of the chip which makes them available and inexpensive.

So the *problem* is producing portable software for tiled architectures which best meet the SWEPT requirements. One way to meet some of these requirements is the use of signal processing libraries. A good standard library will improve performance, reduce program complexity, and time to develop; and act as an abstraction layer providing portable code.

The *goal* of this paper is to focus on the signal processing library and discuss methods to improve performance of signal processing libraries on tiled architectures.

Tiled architectures

In the [PCA](#) program diagrams show chip design with a tiled look which leads to the term *tiled architecture*. In general the term can be misleading since there are multi-core chips which also look tiled when one looks at a picture. Things get even more confusing since a TRIPS chip has two cores where each core has sixteen tiles. Examination of PCA architectures developed by the ISI West/Raytheon team (Monarch) shows multi-core with a high-speed ring communication and some thing that looks like tiles in the middle of the

ring which is termed a field programable gate array. The MIT teams chip (RAW) shows a very tiled looking architecture with a 4 by 4 array of tiles; however the RAW tile is more capable than the TRIPS tile so might be considered a 4 by 4 array of cores.

So the question is “what’s multi-core and what’s tiled. I will not attempt to answer that question by defining the differences between a tile and a core. I don’t know what the differences are; and in some instances there may be no difference. Instead we will examine the characteristics necessary for a successful signal processing library on a PCA type chip where success is defined as lower latency for a particular algorithm.

Note that lower latency does not necessarily mean higher chip utilization. With this type architecture there may be many FLOPS to spare, and those spare FLOPS may be hard to utilize. I am interested in running algorithms faster since in general the throughput problem is solved by adding more computers.

Characteristics of a tiled architecture

- Lots of arithmetic logic units (ALU)
 - An ALU is not necessarily synonymous with a tile.
 - My definition of an ALU may not be the same as that of a more hardware oriented reader. I mean something that will
 - Do arithmetic.
 - Support a thread of execution. The thread may not be independent.
 - Support getting data from, or storing data to “memory”.
- Low latency communication between ALUs
 - ALUs have a neighborhood. Communication latency for nearest neighbors will generally be better
- Memory local to ALUs may be limited.
 - Data access patterns may be important.
- The chip is programable in the normal sense.
 - No FPGAs here

Compilers and Abstraction Layers

In order to achieve maximum performance commercial signal processing libraries are written at a very low level generally using some assembly level programming and a great deal of specialized hardware knowledge. Generally high performance libraries are written by vendors who also produce the hardware the library is designed to work on. When a major hardware revision takes place then a great deal of work is required to update the library source to produce performance on the new hardware. This is a barrier to portable high performance libraries as well as a barrier to frequent hardware upgrades.

It is more cost effective both in time and in money to write libraries using a higher level language. Much of the onus for performance is then placed upon the compiler to do a good job of optimizing the signal processing code. The compiler acts as an abstraction point for performance so that the library is written in a high level language and the language is compiled down to high performance code.

However compilers are written by people and some are better than others. A compiler depends upon the underlying programming language to communicate enough information

so the compiler can do a good job. The ability of a compiler to distribute computations across tiles depends upon its ability to determine optimal barrier points in the code.

In addition pointers which reference data may be created at run time. This makes it difficult for the compiler to tell when (in the program flow) to dereference the data associated with the pointer since the pointer information is not known at compile time; so the compiler has to follow a safe path and can not assume any concurrency unless there is some mechanism in the language to pass concurrency information to the compiler.

One approach discussed in the Morphware forum is for a high level language which has constructs enforcing a programming style which allows the application programmer to expose concurrency uses and implementation dependent library which acts as an abstraction point and calls to this library allow the high level compiler to start multiple threads which are compiled in the low level library.

Signal processing libraries

An application rich in signal processing will use the least memory and be the most efficient if the application is written using no library at all. However an application written without the use of a library will be costly to write and optimize; difficult to maintain; and, unless one remains within the confines of an abstraction layer such as a high level language, not very portable. Even using a high level language as a high level language will generally not produce portable performance. So signal processing libraries are important.

Nobody debates much what a signal processing library is. Generally the type of library is obvious from the context of the discussion. However, there are multiple algorithms for data processing and the best algorithm depends upon both data and hardware characteristics as well as processing requirements. I don't want to delve into every possibility but I will try to place boundaries on the discussion.

In general I will be discussing (what I consider to be) normal signal processing type libraries. Image processing is of interest but we won't discuss it here. Data will be blocked in fairly large chunks generally considered to be a vector or matrix. We might also have a data cube or speak of a tensor; however, data objects of a higher order than a matrix are generally used only as a mechanism for manipulating the data. Most math, linear algebra, and signal processing routines are defined for vectors and matrices.

The signal processing routines in this document work on chunks of data at a time and frequently do a fairly limited operation. The disadvantage is extra memory may be required to store the data between operations. The advantage is that doing a fairly limited operation over a large piece of data allows the process to be optimized for that operation.

The size of the data chunks is generally defined by the application. For instance if great frequency resolution is desired one does a very long FFT. Tradeoffs ensue however because, for instance, the better the frequency resolution the worse the time resolution. I don't want to wander too far into this side trip on data analysis requirements but the point is more than algorithms, hardware, and performance drive signal processing re-

quirements. The entire goal of the application is to analyze data given some set of requirements defined by the physical problem and the library needs to support this.

The signal processing API used here will process data in chunks. The goal is to examine if improved performance can be achieved using multiple threads to the chunk in parallel. This will involve looking at the algorithms used for the signal processing function and examining if it might be parallelized at the function level. This is not a new concept but in general purpose processors are so fast that trying to do this on a cluster will result in poor performance because of communication times. However with tiled architectures assuming a low latency communication between threads algorithms that don't work on a network of workstations might make more sense for a network of tiles.

Chaining

Chaining data calculations together is one method to improve performance. The coding associated with the chaining step helps the compiler to make good decisions. Chaining is difficult across a function boundary. For instance a vector-vector add followed by a scalar-vector multiply are generally more efficient if the compiler can do the add and multiply in one step. Most signal processing libraries handle this problem by adding single function calls to do some of the more common. In this document I will not look at chaining except when done as part of a function definition.

Streaming

One point I would like to discuss is the method of data processing delineated by streaming processing. . It is easy to think of streaming as a thread or operations which acts as a stream sync at the input; operates on the data enclosed in the stream; and then sources a new stream on the output.

However when one thinks of the details of implementing this type operation then things start to become less clear. The atomicity of data and methodology of streams come into question when streams enter in. Questions quickly occur such as "what does a data stream look like?", "how much state is carried with the stream?", "how many discrete data points are in the stream?", "what if the length of the stream is shorter (longer) on the input than on the output?"; and many more questions. For streaming one becomes enmeshed in fairly lengthy discussions of "what is a stream?"

So the concept of streaming is not difficult and I don't believe even the instantiation of streaming on a particular system for a particular problem is difficult since the answers to the questions above are generally driven by necessity and not hard to solve. The problem with discussing streaming in any sort of concrete way on systems in general is there are many ways streams might be implemented and no standards are available with enough usage to act as a common base.

For PCA and tiled architectures in general the streaming concept has become important because one method to keep ALUs busy is to stream data directly from one ALU to the next and avoid accessing bulk memory. This fine grain streaming might be used to write faster signal processing algorithms however the method for doing this is not well understood or portable.

For the purpose of this document I will assume that any streaming type calculation is done as a function of the compiler and the high level language. The compiler and the high level language would act as the abstraction layers allowing portability. Ideally the compiler could do it all reducing the library writers effort; however it is more likely that the library writer will use an HLC which allows streaming to be exposed to the compiler directly.

Analysis

In this section I look at writing and designing signal processing libraries for tiled architectures. Two facets are of interest, these being first the effects of the algorithm chosen and second the ability to expose parallelism to the underlying hardware using a language and its associated compiler.

There exist many more examples of signal processing algorithms than I have time to look at. For this paper I have considered element-wise operations (such as a vector or matrix add), simple linear algebra (such as dot and matrix product), and the discrete fourier transform.

This is mostly a paper study, since I don't have access to resources (or time) for development, testing, and evaluation of many algorithms. I do have access to the TRIPS tool chain which includes a C compiler for the TRIPS chip, a simulator, and some other evaluation tools. Results below that have any qualitative or quantitative flavor are derived from testing on the TRIPS tool chain.

Code below is in C. The C language may not be the best language for exposing concurrency to a compiler however C is the compiler of choice for most hardware development at a research level; and most low level code at any level not written in assembly is written in C. It is a good choice for examining details of concurrency.

For the signal processing API we will use VSIPL.

Element-Wise Operations

Operations in this class are simple, numerous, and important. Things like scalar-vector add/subtract/multiply, vector-vector add/subtract/multiply, are in this class. Also even simpler functions like copy and negate reside in this class. These type operations are rich in SIMD opportunities and finding concurrence is simple.

In general most signal processing functions have a very similar structure which involve three parts; *setup*, *calculate*, and *finalize*.

In the *setup* data pointers, strides, offsets, lengths, etc. are calculated and stored in registers. Calculations may also be done to examine data locality and other run-time state. Based upon the run-time state a particular algorithm for the *calculate* phase may be selected or modified.

In the *calculate* phase generally a loop or nested loops are executed and step through the data. A calculation is done on each data set and the result is placed in the output data. Since every calculation is independent this meets the definition of embarrassingly parallel.

For element-wise *finalize* generally just requires the return statement; however if the function maintains some state information or if input objects or data need to have state restored to initial input values that can also happen here.

We will look at the scalar vector add function. Although many element-wise functions exist, there are enough similarities between the algorithms that doing more than one is probably excessive.

Scalar Vector Add

The *default* code for a standard scalar vector add obtained from the VSIPL Reference version is below.

```

1 void (vsip_svadd_d)(-
2     vsip_scalar_d alpha,-
3     const vsip_vview_d *b,-
4     const vsip_vview_d *r) {-
5     vsip_length n = r->length;-
6     vsip_stride bst = b->stride * b->block->rstride,-
7     rst = r->stride * r->block->rstride;-
8     vsip_scalar_d *bp = (b->block->array) + b->offset * b->block->rstride,-
9     *rp = (r->block->array) + r->offset * r->block->rstride;-
10    while(n-- > 0){-
11        *rp = alpha + *bp;-
12        rp += rst; bp += bst;-
13    }-
14 }-
```

Note that VSIPL passes in objects that contain information for the data location in memory, the stride through the data, and the length of data the object references. Here we have an input vector *a* , an output vector *r* and a constant *alpha*. We want to add the constant to each element of the input vector and place the result in the output vector. Note lines 6&7 provide the stride calculation and line 8&9 provide a pointer to the first data element. Line 5 provides the length so we know how long the loop is. None of this information is known at compile time (at least not by this function). this portion is then the setup. Also note that there is not requirement for other libraries, not even VSIPL libraries, to follow this exact data layout. This is the setup phase

Lines 10-13 are the calculate phase and the finalize stage is automatic since only a return is needed.

We note first that in the calculate phase there are three additions for each loop iteration; one to do the scalar-vector add and one each for the input and output data pointer calculation. So one would hope the compiler would set up the addition for the add and the pointer calculation as a single operation. I am not as familiar as I would like to be with the actual workings of a chip in doing these pointer calculations but it is obvious that if the registers are available parallel data access and data pointer (pre)calculations with work function(line 11) is important for maximum performance.

The inner loop of the default code can be done entirely in parallel except for the need to calculate the location of the next data point. Another way to do this inner loop is to unroll the loop so that several calculations are done for each loop call. Preferably the compiler

would do this type optimization but in practice C compilers have trouble doing this when the limits of the loop are determined at run time. Using the TRIPS tool chain the inner loop is expanded (manual in-lining) so that lines 10-13 look as follows:

```
while(n > 3){-
    *rp = alpha + *bp;-
    rp += rst; bp += bst;-
    *rp = alpha + *bp;-
    rp += rst; bp += bst;-
    *rp = alpha + *bp;-
    rp += rst; bp += bst;-
    *rp = alpha + *bp;-
    rp += rst; bp += bst;-
    n -= 4;-
}-
while(n-- > 0){-
    *rp = alpha + *bp;-
    rp += rst; bp += bst;-
}
```

This type code is in some ways nonsense; however it does produce more efficient code using the TRIPS Tool chain than the default version. We note that the constant “3” contained in the first while loop could be considered metadata associated with the number of available tiles for doing the calculation.

The following code was also tried with the TRIPS tool chain in an attempt to get better performance.

```
while(n > 3){-
    *rp = alpha0 + *bp;-
    rp += rst; bp += bst;-
    *rp = alpha1 + *bp;-
    rp += rst; bp += bst;-
    *rp = alpha2 + *bp;-
    rp += rst; bp += bst;-
    *rp = alpha3 + *bp;-
    rp += rst; bp += bst;-
    n -= 4;-
}-
while(n-- > 0){-
    *rp = alpha + *bp;-
    rp += rst; bp += bst;-
}
```

The difference between this code and the one above is the constant has been spread out among four register constants.

So what am I trying to do here? This is a compiler based approach to improving performance on tiled architectures. I am trying to signal to the compiler where calculations can be distributed. In the first example I am trying to signal to the compiler that it can do

4 adds as one step. In the second I created four register constants of the same number to try even harder to signal to the compiler that it should save this constant in a register on 4 separate tiles.

So why do I call this nonsense? This seems to work to improve performance in particular on the TRIPS tool chain C compiler; however I don't see any real improvement over the default code from the library writers point of view. Writing code like this is looking to get lucky. There is no rhyme or reason so the programmer can not know what the outcome will be. I would like this to be more a science and less a shot in the dark.

However writing this code does help one examine and think about the various mechanisms one might use to distribute the calculations. For instance one could (at runtime) during the setup phase create entirely independent loops, one for each available tile, to do this calculation. On an SMP type architecture where each tile has access to the same memory this would probably be the proper way to do it. However to do this we need a library of calls and some sort of support to launch the independent threads for each loop. There is no way with just C to do this. This would be an opportunity for a high-level compiler; however the mechanisms and/or low-level libraries to allow this on tiled architecture are not yet common enough for any standard to exist. I am not sure any libraries exist to allow this type parallelism at a low enough level for low-latency communications sufficient for improved performance in any but the most simple operations.

I also am not sure the current chip designs allow enough independent data access by tiles to the memory where the data is stored. In current architectures data access patterns and cache performance tend to be very important for improved performance. If every tile has a choke point going to memory where data is stored then it is much more difficult to create algorithms with improved performance.

Linear Algebra

For this section I will look at the dot product and the matrix product. Many of the observations made in the element-wise section above are still valid here; however more communications is necessary to achieve the final answer.

Dot Product

Below find the default dot product from the VSIPL reference version.

```
1 vsip_scalar_d (vsip_vdot_d)(-
2   const vsip_vview_d* a,-
3   const vsip_vview_d* b) {-
4     vsip_length n = a->length;-
5     vsip_stride ast = a->stride * a->block->rstride,-
6                       bst = b->stride * b->block->rstride;-
7     vsip_scalar_d *ap = (a->block->array) + a->offset * a->block->rstride,-
8                       *bp = (b->block->array) + b->offset * b->block->rstride;-
9     register vsip_scalar_d t = 0;-
10    /* do sum */-
11    while(n-- > 0){-
12      t += (*ap * *bp);-
13      ap += ast; bp += bst;-
14    }-
15    /* return dot sum */-
16    return t;-
17 }
```

This has many similarities to the element-wise operation above with lines 4-9 containing setup; lines 11-14 containing the calculation loop and line 16 contains the finalize section where the answer from the dot product is returned. Given that a single number is calculated from all the data this is not an element-wise operation; however it contains many of the same opportunities for concurrency.

To expose this concurrency to the compiler the following code was tried on the TRIPS tool chain. Showing just the conversion of lines 11-17 we have

```
while(n > 3){-
  t0 += *ap * *bp;-
  t1 += *(ap + ast) * *(bp + bst);-
  t2 += *(ap + ast2) * *(bp + bst2);-
  t3 += *(ap + ast3) * *(bp + bst3);-
  ap += ast + ast3; bp += bst + bst3;-
  n -= 4;-
}-
while(n-- > 0){-
  t0 += (*ap * *bp);-
  ap += ast; bp += bst;-
}-
return t0 + t1 + t2 + t3;-
}
```

So basically we have tried to signal to the compiler 4 independent sums which are combined as a final step.

Note this type code unrolling improves the performance of the TRIPS simulator by a factor of about 4. This code also improved the accuracy of the dot product for very long vectors.

Matrix product

It is easy to think of the matrix product as a bunch of dot products; but life is a little more complicated than that. An example of the calculation phase of the matrix product of an M by N A matrix times an N by P B matrix producing an M by P C matrix is given by

```

1 for(i=0; i<M; i++){
2     for(j = 0; j<P; j++){
3         for(k=0; k<N; k++){
4             C[i][j] = A[i][k] * B[k][j];
5         }
6     }
7 }

```

So it is easy to see that lines 3-5 is just the dot product of the i 'th row of A with the k 'th column of B . We examine this more closely though and we see that there is really no required order to the loops. So we could actually do this as an i,j,k loop (as shown) or as an i,k,j loop or as an j,i,k or j,k,i or k,i,j or k,j,i . It turns out these various options can be important. Although the algorithm above uses row major indexing there is no requirement that the matrices of a signal processing library would necessarily be accessed as row major, or that all the matrices even have the same major.

Because of the differences in data access patterns the VSIPL reference version performs checks during the setup phase and picks one of three possible algorithms to use. The algorithm chosen is selected by testing for performance given various major selections of the input/output matrices.

I won't use the VSIPL reference code here since it is a bit long and the pointer math is a little confusing to the uninitiated. Instead I will just expand out the i,j,k matrix calculation above to examine concurrency for this particular algorithm. A note of caution; this code shown has not been compiled and run, so there is a high probability an error exists.

First we note it is generally more efficient to accumulate a sum into a register. In addition we would like to do some unrolling similar to what was done above. We note that i,j is constant in the innermost loop. One way to unroll this on the outer loop is (using an unroll factor of 3)

```

1 int M_init = M/3;-
2 int M_final = M%3;-
3 register float t0,t1,t3;-
4 register float bj;-
5 for(i=0; i < M_init; i++){-
6     for(j = 0; j<P; j++){-
7         t0 = 0; t1 = 0; t2 = 0;-
8         for(k=0; k<N; k++){-
9             bj = B[k][j]; -
10            t0 += A[i*3][k] * bj;-
11            t1 += A[i*3 + 1][k] * bj;-
12            t2 += A[i*3 + 2][k] * bj;-
13        }-
14        C[i*3][j] = t0;-
15        C[i*3 + 1][j] = t1;-
16        C[i*3 + 2][j] = t2;-
17    }-
18 }-
19 for(i=M-M_final; i<M; i++){-
20     for(j=0; j<P; j++){-
21         t0 = 0;-
22         for(k=0; k<N; k++){-
23             t0 += A[i,k]*B[k,i];-
24         }-
25         C[i][j] = t0;-
26     }-
27 }-
--

```

So the procedures here are pretty much the same ones used above; but the code is much more complicated and the number of options for in-lining are numerous. In addition, depending on the loop order (of i,j,k), the algorithm will change. For instance if k were not in the outer loop you could not accumulate into a register; however more pre-fetching of input values (line 9) would be possible. This makes it difficult to tell if you are using the best method or not. All that can be done is write, test, and measure to determine what works.

The fact that the best algorithm depends on the data order of the input matrices indicates to me that no compiler will ever be able to sort this out given just a single simple matrix multiply for input even if the data order were known at compile time and was not a run-time parameter.

Note on line 9 the pre-fetching of a value reused in the next three calculations. As mentioned above, for other loop ordering of the basic matrix multiply this type pre-fetching may be more important. On a standard architecture holding this variable local to the calculation is a clear win when compared to fetching it from memory each time. However for tiled where you might expect each of the accumulate operations to happen on independent tiles then this constant would need to be distributed. My point here is that this is a difference and the affect may not be as straightforward as for the serial version.

Block Matrix Product

Although I have spent time thinking about how to expose concurrency to tiled architectures for signal processing algorithms most of my experience has been with serial code. However there is very little new under the sun and other people (who have spent much more time thinking about this than I) have developed algorithms for clusters and SMPs. Given the pace of hardware development the original lifetime of some of these algorithms was probably short since the cost of communication has become a fatal liability in the face of ever faster logic. However with the advent of multi-core and tiled architectures some of these algorithms should be revisited. I have not done a search on these type algorithms however my main text for [matrix algorithms](#) has an entire (although short) chapter on parallel matrix computations.

The algorithm of choice for tiled architecture should probably be block algorithms. Although I am discussing matrix products here, there are [block methods](#) available for many linear algebra algorithms. The advantage of block algorithms is it is easier to divide up significant sections of the work among multiple tiles, and for large matrices block algorithms have better cache performance.

Fourier Transform

For the Fourier transform I have not been able to do concrete parallel code for tiled architecture similar to what I have done above. The simplest algorithms are recursive in nature and not very suitable for tiled in their most natural form. There are opportunities for pre-calculating a *plan* and as part of that calculation use a recursion type formula but this is future work.

I will start this discussion with a very simple and strait-forward [discrete fourier transform](#) algorithm (dft below). Normally an FFT algorithm would be used; however this algorithm is useful if other algorithms will not work; for instance in the case of a prime number length.

Note that this [algorithm](#) is really a matrix vector product of a coefficient matrix and the input data vector. Note this is a complex product I don't want to go into the details of a discrete fourier transform however there are many texts which cover the basics. Since the elements of the coefficient matrix are redundant they may be passed in as a single vector which I have called (in the code) the *wt* vector. Note the *re* stands for the real part and the *im* stands for the imaginary part.

In line [9 and 13](#) we set the value of an index which selects the proper element from the weight matrix. The step in line 13 can give compilers trouble when automatically determining concurrency; yet there is a great deal of concurrency in this code. For starters the real and imaginary calculations can be done independently. In addition every iteration of the outer *j* loop is independent of every other iteration; so all of the work between line 7 and line 21 could be placed on a new tile for each iteration of *j*.

DFT

```
1 void dft(double *in_re, double *in_im, -
2         double *wt_re, double *wt_im, -
3         double *out_re, double *out_im, -
4         unsigned long N){-
5     unsigned long j,i,k;-
6     for(j=0; j < N; j++){-
7         register double t_r = in_re[0];-
8         register double t_i = in_im[0];-
9         k=0;-
10        for(i=1; i<N; i++){-
11            register double re = in_re[i], wre;-
12            register double im = in_im[i], wim;-
13            k += j;-
14            if(k > N-1) k -= N;-
15            wre=wt_re[k]; wim=wt_im[k];-
16            t_r += re * wre - im * wim;-
17            t_i += im * wre + re * wim;-
18        }-
19        out_re[j] = t_r; out_im[j] = t_i;-
20    }-
21    return;-
22 }
```

Of course most DFTs are done with the fast fourier transform algorithm (FFT). There is no one FFT algorithm. Much work has been done and there are many algorithms to choose from. The basic idea however is pretty straightforward if we ignore all the details.

Generally the number of operations (complex operations here) to do a matrix vector multiply where the matrix is square and the vector is of length N is N multiplies plus $(N-1)$ adds times the number of rows; or in this case $(2*N^2-N)$ for a default DFT.

The basic idea of the DFT is the redundancy of the coefficient matrix means it can be factored. If we assume that the length N is factorable into n_0 and n_1 such that $N = n_0 * n_1$ then these FFT algorithm can be factored into n_1 smaller DFTs of size n_0 plus n_0 smaller DFTs of size n_1 .

So, using the formula, if N is 91 it is easy to see the default DFT algorithm would require 16,471 operations. If we factor this into $(7)(13)$ then we can reduce our operations to $(13 * (49-7) + 7 * (169-13))$ or 1,638 operations. So even with this very small DFT (with somewhat contrived numbers) we see a big reduction in FLOPS. Even further reductions are possible when one starts writing specialized small building blocks. The most famous building block is of size 2.

I am not going to attempt any more details of the FFT here, I consider my understanding of the algorithm to be still somewhat naive. However since each of the building block steps may be done independently at each stage of an FFT calculation (you must sync up between stages) then opportunities for concurrency are clear.

Some work needs to be done to determine the proper size for synchronization between tiles. For instance if one has a basic DFT size of 4096 running only 2 point DFT building blocks then synching 2048 small DFTs at every stage might not be as efficient as just doing 64 DFT's of length 64 and then with only one data scatter and synchronization doing 64 more DFT's to finish the problem. Note that internal to each process an FFT could be done without the need for communication. There is no need to use the default DFT for a stage if the FFT works better.

As above in the linear algebra (matrix multiply) section there is nothing new here. Much work has been done by clever researchers to examine algorithms for distributed FFTs and some of these algorithms might be of use in tiled architectures. In particular my primary FFT text (or at least my current primary text) [*Computational Frameworks for the FFT*](#) has goes into detail on high-performance FFT algorithms including for distributed and shared memory systems.

Conclusions

Although, being a generalist who has not specialized in hardware or software, I am not particularly well qualified in many of the disciplines needed to solve the programming problems of extracting performance from the new tiled and multi-core architecture it is clear that a few statements can be made.

First Standard signal processing algorithms contain a great deal of concurrency at many levels. Most signal processing algorithms contain loop structures which can be mined for parallel operations.

Although concurrency is important it is also important to examine the algorithm to be sure one that works best for the architecture is used. Many algorithms developed for older parallel hardware which have fallen victim to excessive communication latency may be of use in the low latency highly parallel world of tiled chips.

It is clear that much work still needs to be done. At the top of my list are high-level language features which allow the programmer to make explicit concurrency in the code. Just below that is the need for low level compilers which work well with the high-level languages. Finally work needs to be done to examine various algorithms for use, perhaps with some modification, on tiled and multi-core architectures.

Additional Information

Compute Space

Here a *compute space* means a set of applications (or algorithms) with similar computing requirements. Frequently one speaks of signal, image, knowledge, or display processing. So one might speak of a signal processing compute space to indicate the family of applications which would use a signal processing library. One might also speak of an element-wise compute space to speak of a family of functions operating on data sets an element at a time where each operation on a data element is independent of every other operation. The point here is that *compute space* is not strongly defined except to indicate a set of algorithm requirements that are similar.

GPMP

General purpose multiprocessor are used for normal computers and workstations.

GPU

A *Graphics processor unit* is a specialized piece of hardware either integrated into a chip or standalone as a graphics processor chip which is specialized to handle calculations needed to display images on a display.

SPP

A *special purpose processor* is not strongly defined here. It is generally designed to work with a GPMP as a coprocessor.

PCA

A *polymorphous computing architecture* is a chip designed to allow fundamental architecture characteristics to be changed dynamically to match the compute needs of the application. For instance the [TRIPS](#) chip will allow chip states for a default morph which allows a single thread to run on all the ALUs of a core; or a threaded morph which will allow 4 independent threads to run on a quarter of the ALUs of a core.

TRIPS

The *terra-op, reliable, intelligently adaptive processing system* is a chip designed at the University of Texas at Austin as part of the DARPA [PCA](#) program.

Block Methods

See [Matrix Computations](#) chapter one for instance.

Bibliography

Golub, G. H. & Van Loan C. F (1996). *Matrix Computations*. Baltimore and London: The Johns Hopkins University Press.

Van Loan, C. F. (1992). *Computational Frameworks for the fast Fourier transform*. Philadelphia, PA: SIAM