

AFRL-IF-WP-TR-2002-1526

**CAMERON - OPTIMIZED
COMPILATION OF VISUAL
PROGRAMS FOR IMAGE
PROCESSING ON ADAPTIVE
COMPUTING SYSTEMS (ACS)**

**Wim Böhm
Bruce Draper
Ross Beveridge**

**Colorado State University
Computer Science Department
Fort Collins, CO 80523**



JANUARY 2002

FINAL REPORT FOR 08 MAY 1998 – 31 DECEMBER 2001

Approved for public release; distribution is unlimited.

**INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334**

NOTICE

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 US 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 IS RELEASABLE TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS). AT NTIS, IT WILL BE AVAILABLE TO THE GENERAL PUBLIC, INCLUDING FOREIGN NATIONS.

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.


AL SCARPELLI
Project Engineer/Team Leader
Embedded Info Sys Engineering Branch
Information Technology Division


JAMES S. WILLIAMSON, Chief
Embedded Info Sys Engineering Branch
Information Technology Division
Information Directorate

Do not return copies of this report unless contractual obligations or notice on a specific document requires its return.

REPORT DOCUMENTATION PAGE				<i>Form Approved</i> OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, searching existing 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 Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YY) January 2002		2. REPORT TYPE Final		3. DATES COVERED (From - To) 05/08/1998 – 12/31/2001	
4. TITLE AND SUBTITLE CAMERON - OPTIMIZED COMPILATION OF VISUAL PROGRAMS FOR IMAGE PROCESSING ON ADAPTIVE COMPUTING SYSTEMS (ACS)				5a. CONTRACT NUMBER F33615-98-C-1319	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER 69199F	
6. AUTHOR(S) Wim Böhm Bruce Draper Ross Beveridge				5d. PROJECT NUMBER ARPI	
				5e. TASK NUMBER FT	
				5f. WORK UNIT NUMBER 00	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Colorado State University Computer Science Department Fort Collins, CO 80523				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)				10. SPONSORING/MONITORING AGENCY ACRONYM(S)	
Information Directorate Air Force Research Laboratory Air Force Materiel Command Wright-Patterson AFB, OH 45433-7334		Defense Advanced Research Projects Agency (DARPA) Information Systems Office Arlington, VA 22203-1714		AFRL/IFTA	
11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S) AFRL-IF-WP-TR-2002-1526					
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.					
13. SUPPLEMENTARY NOTES Report contains color.					
14. ABSTRACT This report states the work performed by the Cameron project. The goal of the Cameron project is to make FPGAs and other adaptive computer systems available to more applications programmers, by raising the abstraction level from hardware circuits to software algorithms. To this end, we have developed a variant of the C programming language and an optimizing compiler that maps high-level programs directly onto FPGAs, and have tested the language and compiler on a variety of image processing (and other) applications. A high level language with its one step compiler for Adaptive Computing Systems has been designed and implemented, and the performance of DoD relevant applications has been analyzed on commercial off-the-shelf reconfigurable hardware.					
15. SUBJECT TERMS High Level Language, One step compilation, Adaptive Computing Systems					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT: SAR	18. NUMBER OF PAGES 330	19a. NAME OF RESPONSIBLE PERSON (Monitor) Al Scarpelli 19b. TELEPHONE NUMBER (Include Area Code) (937) 255-6548 x3603
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified			

Table of Contents

<u>Section</u>	<u>Page</u>
Executive Summary	v
1 Introduction	1
2 Deliverables	2
3 Technical Description	3
3.1 Compiler Structure	4
3.2 Compiler Optimizations	5
3.2.1 Optimizations Specific to SA-C	5
3.2.2 Code Generation and Low Level Optimizations	7
3.3 Applications	7
3.3.1 Simple Image Operators	8
3.3.2 Floating Point Precision	8
3.3.3 Complex Applications	10
4 Conclusion	11
References	12

Appendices

<u>Appendix</u>	<u>Page</u>
Appendix A: SA-C Manual	14
Appendix B: DDCF Manual	45
Appendix C: Compiler Manual	74
Appendix D: SA-C Compiler Dataflow Description	89
Appendix E: Abstract Hardware Architecture Description	115
Appendix F: DFG to VHDL Translator	125
Appendix G: An Automated Process for Compiling Dataflow Graphs into Reconfigurable Hardware	165
Appendix H: A High Level, Algorithmic Programming Language and Compiler for Reconfigurable Systems	176
Appendix I: Mapping a Single Assignment Programming Language to Reconfigurable Systems	184
Appendix J: Accelerated Image Processing on FPGAs	201
Appendix K: One-Step Compilation of Image Processing Applications to FPGAs	211
Appendix L: Precision vs. Error in JPEG Compression	222
Appendix M: Compiling ATR Probing Codes for Execution on FPGA Hardware	235
Appendix N: University of California Riverside Subcontract	244
Appendix O: KRI Subcontract - A Complete Development Environment for Image Processing Applications on Adaptive Computing Systems	316

EXECUTIVE SUMMARY

The Cameron Project has developed (1) a graphical programming environment to compose existing Image Processing modules into Adaptive Computing Applications, (2) a high-level programming language SA-C for writing new Image Processing routines. The central component of these tools is an optimizing compiler for mapping SA-C Image Processing programs to Adaptive Computing Processors. Our success lies in exploiting data flow and data dependence analysis techniques developed in the compiler and functional programming communities.

The approach taken by Cameron has two important advantages. First, IP developers can develop their applications in a high level, algorithmic language and may choose which optimizations they choose, but are not required to optimize circuits by hand. Second, the approach is ACS platform independent, which allows easy retargeting of applications.

To demonstrate the breadth and utility of our software, we have implemented the Image Processing portion of the VSIPL Library. Using this library and the Intel Image Processing Library as a base, we then composed solutions to several real-world IP kernels as well as two DoD relevant applications: the ARAGTAP pre-screener and the CSU/RSTA probing algorithm. Using these applications we have demonstrated the effectiveness of the high level programming language approach.

ACKNOWLEDGEMENT

The authors gratefully acknowledge the support of DARPA and the Air Force Research Laboratory under Air Force contract F33615-98-C-1319.

1 INTRODUCTION

Field-programmable gate arrays (FPGAs) and other reconfigurable systems offer a fundamentally new model of computation in which programs are mapped directly onto off the shelf (as opposed to ASIC) hardware, thereby promising a tremendous speed-up over traditional computers. Part of this speed-up comes from parallelism, while the rest comes from increased computational density. The potential for parallelism is obvious; FPGAs are composed of thousands of logic blocks, and with reprogrammable interconnections the degree of parallelism is limited only by the data dependencies within the program and I/O limitations. Computational density is equally important, however. Traditional processors implement a complex fetch-and-execute cycle that requires many special purpose units. Only a small fraction of the hardware is dedicated to the computation being performed. FPGA circuits, on the other hand, can be designed so that data flows through the circuit without repeated storage and retrieval, and without wasting resources on unused circuitry.

FPGAs are typically programmed using hardware description languages such as VHDL. Application programmers are typically not trained in these hardware description languages and usually prefer a higher level, algorithmic programming language to express their applications. The Cameron project has made a significant contribution in shifting the programming paradigm of adaptive and reconfigurable systems from hardware centered to software centered. Cameron's software paradigm is based on visual and high-level language programming in order to make adaptive computing accessible to software engineers and portable across platforms.

The Cameron software environment lets image processing experts compose programs for adaptive computing systems. We have designed and implemented (1) a graphical visual programming language based on Khoros, that allows programmers to compose existing image processing modules into ACS applications, and (2) a high-level programming language: SA-C (single assignment C) for writing new image processing routines. The central component of both of these tools is (3) an optimizing compiler for mapping image processing programs to ACS processors. Our key to success lies in exploiting data flow and data dependence analysis techniques originally developed in the compiler community and further refined by the parallel computing and functional programming communities. Data flow analysis has long been recognized as an extremely powerful tool for mapping computation onto parallel hardware. We have taken this observation one step further: data flow analysis is ideally suited to mapping high-level algorithms to statically and dynamically reconfigurable gate-level hardware.

The approach taken by the Cameron project has two important advantages. First, it is fully automatic; IP developers may choose which optimizations to use, but they are not required to optimize circuits by hand. Few IP experts are also experts in logic design. Second, the approach is ACS platform independent. It has been designed based on general data-flow principles that can be applied to any number of ACS systems. To demonstrate the breadth and utility of our software environment, we have implemented the image processing portion of the Vector, Signal, and Image Processing Library (VSIPL). Using this library and the Intel Image Processing Library as a base, we then composed solutions for several real-world IP kernels as well as for two DoD relevant applications: the ARAGTAP pre-screener and CSU Reconnaissance, Surveillance, and Target Acquisition (RSTA) probing algorithm. Using these applications we have demonstrated the

effectiveness of the compiler optimization and the success of the high level programming language approach.

2 Deliverables

The Cameron project has delivered all it promised, and more. This section contains each deliverable from the project proposal with a reference to the document(s) describing it. We also present extra work done.

- [1] SA-C language definition and extensions. *See Appendix A: SA-C Language.*
- [2] Definition, syntax and semantics of the Data Dependency and Control Flow Graph (DDCF graph). *See Appendix B: The SA-C Compiler Data Dependence and Control Flow (DDCF).*
- [3] SA-C to Khoros Pro (TM) compilation which extends the visual programming interface generating SA-C programs. *See Appendix O: Khoros Report.*
- [4] SA-C to DDCF graph compiler. *See Appendix C: The SA-C Compiler.*
- [5] SA-C to C compiler. *See Appendix C: The SA-C Compiler.*
- [6] Definition, syntax and semantics of the *extended* Data Flow Graph (DFG) and Abstract Hardware Architecture (AHA) graph forms. *See Appendix D: Dataflow Graph Description and Appendix E: The SA-C Compiler Abstract Hardware Description.*
- [7] DDCF graph to DFG and AHA translator. *See Appendix E: The SA-C Compiler Abstract Hardware Description.*
- [8] DDCF graph to DDCF graph machine independent program optimizations. *See Appendix G: A High-Level, Algorithmic Programming Language and Compiler for reconfigurable Systems.*
- [9] ACS platform specific program optimizations based on the extended DDCF graphs (low level optimizations). *See Appendix G: An Automated process for compiling dataflow graphs into reconfigurable hardware, and Appendix I: Mapping a Single Assignment Programming Language to Reconfigurable Systems.*
- [10] DDCF graph to VHDL compiler for routines that execute on the FPGA array. Includes complete documentation and extensive examples. *See Appendix F: Dataflow Graph to VHDL Translation.*
- [11] DDCF graph to ACS platform program loader. The loader generates a host-based sequencing order of pre-compiled and ACS platform optimized library routines. *See Appendix C: The SA-C Compiler.*

[12] VSIPL Image Processing libraries in SA-C integrated with the Khoros Pro (TM) based visual programming environment. SA-C source code for all VSIPL IP routines is available on our website. Due to the lack of a competitive implementation of VSIPL available during the course of this project, especially an implementation optimized for the Pentium MMX architecture, we shifted our focus to the highly optimized Intel Image Processing Library (IPL). *See Appendix J: Accelerated Image Processing on FPGAs and the Cameron webpage <http://www.cs.colostate.edu/cameron/>*

[13] Machine optimized ACS executables of the VSIPL IP libraries for at least one ACS platform. *See Appendix J: Accelerated Image Processing on FPGAs*

[14] Comprehensive system evaluation based on/ medium to large scale IP applications: Canny, Wavelet, Prewitt. *See Appendix J: Accelerated Image Processing on FPGAs, Appendix K: One-step Compilation of Image Processing Applications to FPGAs, Appendix L: Precision vs. Error in JPEG Image Compression.*

[15] Porting of an application already developed for adaptive computing to a large-scale ACS: Probing. *See Appendix M: Compiling ATR Probing Codes for Execution on FPGA hardware.*

Extra work:

[16] An elaborate website on Cameron with SA-C, Dataflow, Image-Processing applications and their FPGA performance is available [2]. Journal papers [23, 6, 9], conference papers [22, 17, 1, 20, 18, 19, 24, 11, 16, 3, 10, 15, 5, 4], technical reports [25, 7, 12, 13, 14, 21, 8] are also available on this site.

[17] SA-C to Morphosys compilation. *See Appendix N: UCR Final Report.*

[18] FPGA space estimation of SA-C operators. *See Appendix N: UCR Final Report.*

3 Technical Description

The design goals of SA-C are to have a language that can express image processing applications elegantly, and to allow seamless compilation to reconfigurable hardware. Variables in SA-C are associated with wires, not with memory locations. SA-C is a single-assignment side effect free language; each variable's declaration occurs together with the expression defining its value. This avoids pointer and von Neumann memory model complications and allows for better compiler analysis and translation to DFGs. The IP applications that have been coded in SA-C transform images to images and are easily expressed using single assignment. Data types in SA-C include signed and unsigned integers and fixed point numbers, with user-specified bit widths. SA-C has multidimensional rectangular arrays whose extents can be determined either dynamically or statically.

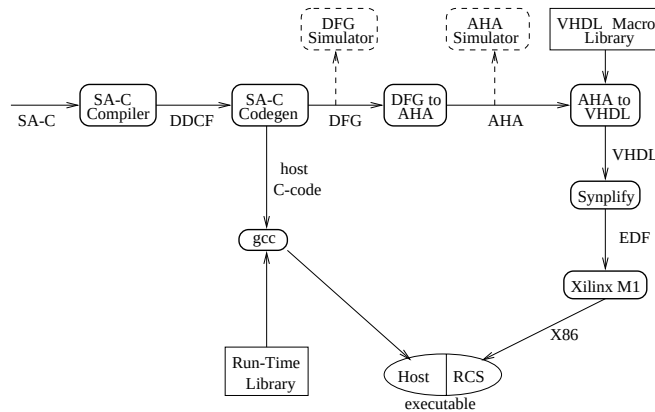
The most important aspect of SA-C is its treatment of **for** loops and their close interaction with arrays. SA-C is expression oriented, so every construct including a loop returns one or more values. A loop has three parts: one or more generators, a loop body and one or more return values. The

generators provide parallel array access operators that are concise and easy for the compiler to analyze. There are four kinds of loop generators: *scalar*, *array-element*, *array-slice* and *window*. The scalar generator produces a linear sequence of scalar values, similar to Fortran’s **do** loop. The array-element generator extracts scalar values from a source array, one per iteration. The array-slice generator extracts lower dimensional sub-arrays (e.g. vectors out of a matrix). Finally, window generators allow rectangular sub-arrays to be extracted from a source array. All possible sub-arrays of the specified size are produced, one per iteration. Generators can be combined through **dot** and **cross** products. The **dot** product runs the generators in lock step, whereas **cross** products produce all combinations of components from the generators.

Loop carried values are allowed in SA-C using the keyword **next** instead of a type specifier in a loop body. This indicates that an initial value is available outside the loop, and that each iteration can use the current value to compute a next value.

Concluding, SA-C loops provide a simple and concise way of processing arrays in regular patterns, often making it unnecessary to create nested loops to handle multi-dimensional arrays or to refer explicitly to the array’s extents or the loop’s index variables. They make compiler analysis of array access patterns significantly easier than in C or Fortran, where the compiler must analyze index expressions in loop nests and infer array access patterns from these expressions. In SA-C, the index generators and the array references have been unified; the compiler can reliably infer the patterns of array access.

3.1 Compiler structure



SA-C Compilation system.

The SA-C compiler provides one-step compilation to host executable and FPGA configurations. After parsing and type checking, the SA-C compiler converts the program to a hierarchical data dependence and control flow (DDCF) graph representation. DDCF graphs are used in many optimizations, some general and some specific to SA-C and its target platform. After optimization, the program is converted to a combination of dataflow graphs (DFGs) and host code. DFGs are then

compiled to VHDL code via Abstract Hardware Architecture (AHA) graphs. The VHDL code is synthesized and place-and-routed to FPGAs by commercial software tools.

To aid in program development, it is possible to view and simulate intermediate forms. For initial debugging the complete SA-C program can be executed on the host. All intermediate graph forms can be viewed, and DFG and AHA graphs can be simulated. The SA-C compiler can run in stand-alone mode, but it also has been integrated into the Khoros graphical software development environment.

3.2 Compiler Optimizations

Data dependence analysis is performed on the DDCF graph, and conventional optimizations are performed as DDCF-to-DDCF transformations, including Invariant Code Motion, Function Inlining, Switch Constant Elimination, Constant Propagation and Folding, Algebraic Identities, Dead Code Elimination and Common Subexpression Elimination. Another set of optimizations is more specific to the single assignment nature of SA-C and to the target hardware. Many of these are controlled by user pragmas.

3.2.1 Optimizations specific to SA-C

Many IP operators involve fixed size and often constant convolution masks. A *Size Inference* pass propagates information about constant size loops and arrays through the dependence graph. This pass is vitally important in this compiler, since it exploits the close association between arrays and loops in the SA-C language. Source array size information can propagate through a loop to its result arrays (downward flow), and result array size information can be used to infer the sizes of source arrays (upward flow). In addition, since the language requires that the generators in a dot product have identical shapes, size information from one generator can be used to infer sizes in the others (sideways flow).

Effective size inference allows other optimizations to take place. Examples are Full Loop Unrolling and array elimination. Full Unrolling of loops is discussed next. Array elimination can occur in two contexts described below. In Array Value Propagation, when the array is fixed size and all references to elements have fixed indices, the array is broken up into its elements, i.e. storage is avoided and elements are represented as scalar values. In Loop Carried Array Elimination the array is again replaced by individual elements, but here the values are carried from one iteration to the next.

Full Unrolling of loops with small, compile time assessable numbers of iterations can be important when generating code for FPGAs, because it spreads the iterations in code space rather than in time. Small loops occur frequently as inner loops in IP codes, for example in convolutions with fixed size masks.

Array Value Propagation searches for array references with constant indices, and replaces such references with the values of the array elements. When the value is a compile time constant, this enables constant propagation. As will be shown in the Prewitt example, this optimization may remove entire user defined (or compiler generated) arrays. When all references to the array have constant indices, the array is eliminated completely.

Loop Carried Array Elimination The most efficient representation of arrays in loop bodies is to have their values reside in registers. This eliminates the need for array allocation in memory and array references causing delays. This requires that the array size be statically known. The important case is that of a loop carried array that changes values but not size during each iteration. To allocate a fixed number of registers for these arrays two requirements need to be met. 1) The compiler must be able to infer the size of the initial array computed outside the loop. 2) Given this size, the compiler must be able to infer that the next array value inside the loop is of the same size.

To infer sizes, the compiler clones the DDCF graph representing the loop, and speculates that the size of the array is as given by its context. It then performs analysis and transformations based on this assumption. There are three possible outcomes of this process. If the size of the next array cannot be inferred, the optimization fails. If the size of the next array can be inferred, but is different from the initial size, it is shown that the array changes size dynamically, and the optimization fails again, but now for stronger reasons. If the size of the next array is equal to the initial size, the sizes of the arrays in all iterations have been proven, by induction, to be equal and the transformation is allowed, the important transformation being array elimination. This form of speculative optimization requires the suppression of global side effects, such as error messages.

N-dimensional Stripmining extends the more conventional stripmining optimization and creates an intermediate loop with fixed bounds. The inner loop can be fully unrolled with respect to the newly created intermediate loop. The performance of many systems today, both conventional and specialized, is often limited by the time required to move data to the processing units. **Fusion** of producer-consumer loops is often helpful, since it reduces data traffic and may eliminate intermediate data structures.

Common Subexpression Elimination (CSE) is an old and well known compiler optimization that eliminates redundancies by looking for identical subexpressions that compute the same value. The redundancies are removed by keeping just one of the subexpressions and using its result for all the computations that need it. This could be called “spatial CSE” since it looks for common subexpressions within a block of code. The SA-C compiler performs conventional spatial CSE, but it also performs **Temporal CSE**, looking for values computed in one loop iteration that were already computed in previous loop iterations. In such cases, the redundant computation can be eliminated by holding such values in registers so that they are available later and need not be recomputed.

A useful phenomenon often occurs with Temporal CSE: one or more columns in the left part of the window are unreferenced, making it possible to eliminate those columns. **Narrowing** the window lessens the FPGA space required to store the window’s values. Another optimization that sets the stage for window narrowing moves the computation of expressions fed by window elements into earlier iterations by moving the window references rightward, and uses a register delay chain to move the result to the correct iteration. This can remove references to left columns of the window, and allows window narrowing. This optimization trades window space for register delay chain space. Hence, whether this optimization actually provides a gain depends on the individual computation. We call this **Window Compaction**.

3.2.2 Code generation and low level optimizations

Some (combinations of) operators can be inefficient to implement directly in hardware. The evaluation of the whole expression can be replaced by an access to a **Lookup Table**, which the compiler creates by wrapping a loop around the expression, recursively compiling and executing the loop, and reading the results.

Dataflow Graphs (DFGs) are translated into a lower level form called Abstract Hardware Architecture (AHA). This is also a graph form, but with nodes that are more fine-grained than DFG nodes and that can be translated to simple VHDL components. AHA graphs have *clocked*, *semi-clocked* and *non-clocked* nodes. The clocked and semi-clocked nodes have internal state but only the clocked nodes participate in the *handshaking* needed to synchronize computations and memory accesses. Some clocked nodes communicate via an arbitrator with a local memory. An AHA graph is organized as a sequence of *sections*, each with a top and a bottom boundary. A section boundary consists of clocked nodes, whereas its internal nodes are non-clocked or semi-clocked. In the AHA model, a section fires when all clocked nodes at its top boundary can produce new values and all clocked nodes at its bottom boundary can consume new values. This contrasts with DFGs, where each node independently determines when it can fire.

Some low-level optimizations take place in this stage. A **Bitwidth Narrowing** phase is performed just before AHA graphs are generated. **Dead code elimination** and **graph simplification** sweeps are applied on the AHA graph. A **Pipelining** phase uses node propagation delay estimates to compute the delay for each AHA section, and adds layers of pipeline registers in sections that have large propagation delays. The maximum number of pipeline stages added to the AHA graph is user controlled. Reducing propagation delays is important because it increases clock frequency.

AHA graph simulation allows the user (or, more likely, the compiler or system developer) to verify program behavior. The AHA simulator strictly mimics the hardware behavior with respect to clock cycles and signals traveling over wires. This removes the need for time consuming VHDL simulation and hardware level debugging. AHA-to-VHDL translation is straightforward; AHA nodes translate to VHDL components, which are connected according to the AHA edges.

3.3 Applications

The SA-C language and compiler allow FPGAs to be programmed in the same way as other processors. Programs are written in a high-level language, and can be compiled, debugged, and executed from a local workstation. It so happens that for SA-C programs, the host executable off-loads the processing of loops onto an FPGA, but this is invisible. SA-C therefore makes reconfigurable processors accessible to applications programmers with no hardware expertise.

The empirical question from the applications perspective is whether image processing tasks run faster on FPGAs than on conventional general-purpose hardware, in particular Pentiums. The tests conducted in the context of this project suggest that in general, the answer is yes. Simple image operators are faster on reconfigurable processors because of their greater capabilities for I/O between the FPGA and local memory, although this speed-up is modest (a factor of ten or less). More complex tasks result in larger speed-ups, up to a factor of 800 in one test, by exploiting the parallelism within FPGAs and the strengths of an optimizing compiler. The reconfigurable

processor used in our tests is an Annapolis Microsystems WildStar with 3 Xilinx XV-2000E FPGAs and 12 local memories. Our conventional processor is a Pentium III running at 800 MHz. The chips in both processors are of a similar age and were the first of their respective classes fabricated at 0.18 microns.

3.3.1 Simple Image Operators

As part of our initial pledge to re-implement the VSIPL Image Processing Library in SA-C and our subsequent goal of re-implementing parts of the Intel IPL library for experimental comparison purposes, we implemented a large number of simple image operators in SA-C. Probably the simplest program we tested adds a scalar argument to every pixel in an image. (This matches a routine in the Intel IPL.) For the WildStar, we wrote the function in SA-C and compiled it to a single FPGA. We compared its performance to the matching routine from the Intel Image Processing Library (IPL) running on the Pentium. In so doing, we compare the performance of compiled SA-C code on an FPGA to hand-optimized (by Intel) Pentium assembly code. The WildStar outperformed the Pentium by a factor of 8. (See Appendix J for a detailed table.)

Why is the WildStar faster? The clock rate of an FPGA is a function of the latency of the programmed circuit, but in general FPGAs run at lower clock rates than Pentiums. For this program, the WildStar ran at 51.7 MHz, compared to 800 MHz for the Pentium. Unfortunately for the Pentium, however, memory response times have not kept up with processor speeds, and the 512x512 source image is too large to fit in its primary cache. As a result, the Pentium is not able to read or write data at anything close to 800MHz, so its primary advantage is squandered.

FPGAs, on the other hand, are capable of parallel I/O. The WildStar gives the FPGAs 32-bit I/O channels to each of four local memories, so the FPGA can both read and write 8 8-bit pixels per cycle. This is four times the I/O bandwidth of the Pentium. Also, the operator's pixel-wise access pattern is easily identified by the SA-C compiler, which is able to optimize the I/O so that the program reads and writes almost 8 pixels per cycle.

The FPGA outperforms the Pentium on the scalar addition task by slightly more than I/O considerations alone would predict. This is because the SA-C compiler exploits both data and pipeline parallelism. On every cycle, the FPGA (1) reads eight 8-bit pixels, (2) adds eight copies of the scalar to the eight pixels read on the previous cycle (in parallel), and (3) writes back to memory the sums of the scalar with the eight pixels read on the cycle before that.

This program represents one extreme in the FPGA vs. Pentium comparison. It is a simple, pixel-based operation that fails to fully exploit the FPGA, since only 9% of the lookup tables (and 9% of flip-flops) were used. However, it demonstrates that FPGAs will outperform Pentiums on simple image operators because of their parallel I/O capabilities. The exact amount of the speed-up will depend on the number of local memories the FPGA has access to and the speeds of the memories, but in general one expects a small speed-up of less than a factor of ten.

3.3.2 Floating Point Precision

The image processing applications mentioned above demonstrate the potential power of FPGAs, but may avoid some of their weaknesses. In particular, the applications above all rely on integer

mathematics. Current FPGAs are very adept at variable precision integer and fixed point mathematics, but are not well suited for floating point calculations. We therefore wanted to determine if some of the classical, floating point algorithms from image and signal processing could be well approximated on FPGAs using fixed point calculations.

More specifically, we investigated the relationship between error and bit-precision for the Discrete Cosine Transform (DCT). (We also tested JPEG, an image compression standard that used DCT; see Appendix L: Precision vs. Error in JPEG compression). We implemented DCT in fixed point, rather than floating point, arithmetic, and measured the increase in reconstruction error as the precision of the fixed point values is decreased. Because DCT depends on the frequency components of an image, we measure the precision/accuracy tradeoff for sets of real, artificial, and synthetic images created with different spectral components. Reconstruction error is measured in terms of total gray-level error, RMS gray-level error, RMS signal-to-noise ratio, and peak signal-to-noise ratio.

The data sets used to measure the relationship between precision and error in the DCT were composed of natural, artificial and synthetic images. The natural and artificial images were collected from the web. The artificial images are drawings extracted from Escher's work (five drawings). The natural ones are: six animal images, one color palette, one image of a seed, one aerial image of Fort Hood (TX), and one specular microscope image of cornea endothelial cells. All images were clipped to 256 x 256 and when necessary converted to black and white. Because the DCT maps between the spatial and frequency domains, we also tested synthetic images containing controlled frequencies. We tested three Gaussian filter images with different variances, three noise images (uniform, Gaussian and exponential noise), two impulse images with different spacing between the impulses, one sinusoid image, one image of a constant-intensity circle against a constant background, and one image of two concentric rings.

To evaluate the effect of precision on the reconstruction error, we converted each image into the frequency domain using the DCT, rounded the frequency values to the nearest integer, and then converted them back into the spatial domain using the inverse DCT. We then compared the original images to the reconstructed versions. We repeated this process for all 26 images and for all combinations of the multiplication and summation types (i.e. precisions). The multiplication types tested were: float, fix32.30, fix28.26, fix26.22, fix18.16, fix17.15, fix16.14, fix15.13, fix14.12, fix12.10, fix10.8, fix8.6, fix6.4, and fix4.2. (Remember that the cosine and α terms always require two bits to the left of the binary point to represent their sign and integer magnitude.) The summation types require 16 bits to the left of the binary point to represent their integer magnitude and sign, so the summation types tested were: float, fix32.16, fix28.12, fix24.8, fix22.6, fix20.4, fix18.2 and integer. All 112 of these combinations were executed.

Appendix L shows the errors resulting from the DCT/IDCT reconstruction process. Restricting the multiplication type to sixteen bits to the right of the binary point (with two bits to the left) introduces a negligible amount of error, implying that 18 bits of precision are enough for these terms. This is significant, since hardware multipliers require more resources than adders. Unfortunately, the DCT is more sensitive to the summation precision. Using fixed point precision for the summation terms creates a significant amount of error. This implies that in a strict implementation of the DCT, the tree of additions *must use floating point addition*.

Significantly, these results were consistent across the images tested. Appendix L shows the results of the total error for three more images. In general, the natural and artificial images had almost indistinguishable error curves. A small number of synthetic images presented slightly different results. The most significant difference was less error for some images where the background dominates. This is expected, since most of the 8×8 background windows are constant and equal to zero in this case. Nevertheless, the shape of the curves remain the same.

In applications where a slight increase in error is tolerable for the DCT, faster circuits can still be constructed. Using floating point computation as the best case, we computed the difference between floating point arithmetic and combinations of fixed point precisions. This measures how much error is added by each combination of precisions. Appendix L shows the average increase in error for each pixel for all artificial and natural images. Although there is some error, it is very small for most precision combinations. For example, if an average increased error of one gray level or less is acceptable, a fractional precision of 12 bits for the multiplication type and 6 for the summation type are enough. That means 14 bits ($2+12$) for the cosine and α variables and 22 ($16+6$) for the internal summation variables, a savings of 18 and 10 bits respectively over a 32 bit floating point representation. Moreover, fixed point arithmetic requires fewer logic blocks and less time than floating point arithmetic does. The synthetic images have similar results, but the increase in error was even smaller.

3.3.3 Complex Applications

In addition to simple image operators, we implemented a series of more complex applications in SA-C on FPGAs. Among these applications were detectors, Prewitt and Canny edge detectors, the Cohen-Daubechies-Feauveau wavelet (from the Honeywell benchmark suite), and the ARAGTAP pre-screener. The most impressive results, however, were achieved in ATR probing. The goal of probing is to find a target in a LADAR or IR image. A target (as seen from a particular viewpoint) is represented by a set of probes, where a probe is a pair of pixels that straddle the silhouette of the target. The idea is that the difference in values between the pixels in a probe should exceed a threshold if there is a boundary between them. The match between a template and an image location is measured by the percentage of probes that straddle image boundaries.

Probe sets must be evaluated at every window position in the image. What makes this application complex is the number of probes. In our example there are approximately 35 probes per view, 81 views per target, and three targets. In total, the application defines 7,573 probes per window position. Fortunately, many of these probes are redundant in either space or time, and the SA-C optimizing compiler is able to reduce the problem to computing 400 unique probes (although the summation trees remain complex). This is still too large to fit on one FPGA, so to avoid dynamic reconfiguration we distribute the task across all three FPGAs on the WildStar. When compiled using VisualC++ for the Pentium, probing takes 65 seconds; the SA-C implementation on the WildStar run in 0.08 seconds.

These times can be explained as follows. For the configuration generated by the SA-C compiler for the probing algorithm, the FPGAs run at 41.1 MHz. The program is completely memory IO bound: every clock cycle each FPGA reads one 32 bit word, containing two 12 bit pixels. As there are $(512 - 13 + 1) * (1024) * 13$ pixel columns to be read, the FPGAs perform $(512 - 13 + 1) * (1024) *$

$(13/2) = 3,328,000$ reads. At 41.1 MHz this takes 80.8 milliseconds.

The Pentium performs $(512 - 13 + 1) * (1024 - 34 + 1)$ window accesses. Each of these window accesses involves 7573 threshold operations. Hence the inner loop that performs one threshold operation is executed $(512 - 13 + 1) * (1024 - 34 + 1) * 7573 = 3,752,421,500$ times. The inner loop body in C is:

```
for(j=0; j<sizes[i]; j++){
    diff = ptr[set[i][j][2]*in_width+set[i][j][3]] -
           ptr[set[i][j][0]*in_width+set[i][j][1]];
    count += (diff>THRESH || diff<-THRESH);
}
```

where `in_width` and `THRESH` are constants. The VC++ compiler infers that ALL the accesses to the set array can be done by pointer increments, and generates an inner loop body of 16 instructions. (This is, by the way, much better than the 22 instructions that the gcc compiler produces at optimization setting -O6.) The total number of instructions executed in the inner loop is therefore $3,752,421,500 * 16 = 60,038,744,000$. If one instruction were executed per cycle, this would bring the execution time to about 75 seconds. As the execution time of the whole program is 65 seconds, the Pentium (a super scalar architecture) is actually executing more than one instruction per cycle!

4 Conclusion

The Cameron Project has created a language, called SA-C, for one-step compilation of image processing applications that target FPGAs. Various optimizations, both conventional and novel, have been implemented in the SA-C compiler.

The system has been used to implement routines from the Intel IPL, as well as more comprehensive applications, such as the ARAGTAP target acquisition prescreener. Compared to Pentium III/MMX technology built at roughly the same time, the SA-C system running on an Annapolis WildStar board with one Virtex 2000 FPGA has similar performance when it comes to small IPL type operations, but shows speedups up to 75 when it comes to more complex operators such as Prewitt, Canny, Wavelet, and Dilate and Erode sequences. Performance evaluation of the SA-C system has just begun. As performance issues become clearer, the system will be given greater ability to evaluate various metrics including code space, memory use and time performance, and to evaluate the tradeoffs between conventional functional code and lookup tables.

Currently, the VHDL generated from the AHA graphs ignores the structural information available in the AHA graph. We will soon be investigating the use of Relatively Placed Macros (RPM) as a method to make some of the structural information explicit to the synthesis tools. Providing constraints to specify the placement of nodes relative to each other may prove to decrease synthesis and place and route time.

Also, stream data structures are being added to the SA-C language, which will allow multiple cooperating processes to be mapped onto FPGAs. This allows expression of streaming video applications, and partitioning of a program over multiple chips.

References

- [1] J. Bins, B. Draper, W. Böhm, and W. Najjar. Precision vs. error in JPEG compression. In *SPIE Parallel and Distributed Methods for Image Processing III*, July 1999.
- [2] W. Böhm and B. Draper. The cameron project. Information about the Cameron Project, including several publications, is available at the project's web site, www.cs.colostate.edu/cameron.
- [3] W. Böhm, B. Draper, W. Najjar, J. Hammes, R. Rinker, M. Chawathe, and C. Ross. One-step compilation of image processing applications to FPGAs. In *IEEE Symposium on Field-programmable Custom Computing Machines*, April 2001.
- [4] W. Böhm, B. Draper, W. Najjar, J. Hammes, R. Rinker, M. Chawathe, and C. Ross. Compiling ATR probing codes for execution on FPGA hardware. April 2002.
- [5] W. Böhm, B. Draper, W. Najjar, J. Hammes, C. Ross, and M. Chawathe. Optimized compilation of embedded applications on FPGAs. In *Fifth Annual High Performance Computing Workshop*, Lincoln Labs., MIT, Nov. 2001.
- [6] W. Böhm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar. Mapping single assignment programming language to reconfigurable systems. *Journal of Supercomputing*, 21:117–130, 2002.
- [7] M. Chawathe. Data flow graph to VHDL translation, April 2000.
- [8] M. Chawathe, M. Carter, C. Ross, R. Rinker, A. Patel, and W. Najjar. Dataflow graph to VHDL translation. Technical report, Colorado State University, Dept. of Computer Science, 2000.
- [9] B. Draper, R. Beveridge, W. Böhm, M. Chawathe, and C. Ross. Accelerated image processing on FPGAs. *submitted to IEEE Transactions on Image Processing*, 2002.
- [10] B. Draper, W. Böhm, J. Hammes, W. Najjar, R. Beveridge, M. Ross, C. and Chawathe, M. Desai, and J. Bins. Compiling sa-c programs to FPGAs: Performance results. In *International Conference on Vision Systems*, pages 220–235, Oct. 2001.
- [11] B. Draper, W. Najjar, W. Böhm, J. Hammes, R. Rinker, C. Ross, and J. Bins. Compiling and optimizing image processing applications for fpgas. In *IEEE International Workshop on Computer Architecture for Machine Perception*, Sept. 2000.
- [12] J. Hammes. *Compiling SA-C to Reconfigurable Computing Systems*. PhD thesis, Colorado State University, 2000.
- [13] J. Hammes and W. Böhm. *The SA-C Language - Version 1.0*, 1999. Document available from www.cs.colostate.edu/cameron.
- [14] J. Hammes and W. Böhm. *The SA-C Compiler DDCF Graph Description*, 1999. Document available from www.cs.colostate.edu/cameron.
- [15] J. Hammes, W. Böhm, C. Ross, M. Chawathe, B. Draper, and W. Najjar. High performance image processing on FPGAs. In *Los Alamos Computer Science Institute Symposium*, Oct. 2001.
- [16] J. Hammes, W. Böhm, C. Ross, M. Chawathe, B. Draper, R. Rinker, and W. Najjar. Loop fusion and temporal common subexpression elimination in window-based loops. In *IPDPS 8th Reconfigurable Architecture Workshop*, April 2001.
- [17] J. Hammes, B. Draper, and W. Böhm. Sassy: A language and optimizing compiler for image processing on reconfigurable computing systems. In *International Conference on Vision Systems*, pages 522–537, Jan. 1999.
- [18] J. Hammes, R. Rinker, W. Böhm, and W. Najjar. Compiling a high level language compilation to reconfigurable systems. In *Compiler and Architecture Support for Embedded Systems*, pages 236–244, Oct. 1999.
- [19] J. Hammes, R. Rinker, W. Böhm, W. Najjar, and B. Draper. A high level, algorithmic programming language and compiler for reconfigurable systems. In *The 2nd International Workshop on the Engineering of Reconfigurable Hardware/Software Objects, Part of PDPTA*, pages 135–141, June 2000.

- [20] J. Hammes, R. Rinker, W. Böhm, W. Najjar, B. Draper, and R. Beveridge. Cameron: High level language compilation for reconfigurable systems. In *Conference on Parallel Architectures and Compilation Techniques*, Oct. 1999.
- [21] J. Hammes, R. Rinker, D. McClure, W. Böhm, and W. Najjar. *The SA-C Compiler Dataflow Description*, 1999. Document available from www.cs.colostate.edu/cameron.
- [22] W. Najjar, B. Draper, W. Böhm, and J. R. Beveridge. The Cameron Project: High-level programming of image processing applications on reconfigurable computing machines. In *PACT '98 - Workshop on Reconfigurable Computing*, pages 83–88, Oct. 1998.
- [23] R. Rinker, M. Carter, A. Patel, M. Chawathe, C. Ross, , J. Hammes, W. Najjar, and W. Böhm. An automated process for compiling data flow graphs into reconfigurable hardware. *IEEE Transactions on VLSI Systems*, 9(1):130–139, 2001.
- [24] R. Rinker, J. Hammes, W. Najjar, W. Böhm, and B. Draper. Compiling image processing applications to reconfigurable hardware. In *IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 56–65, July 2000.
- [25] C. Ross. A VHDL runtime system for dataflow execution on reconfigurable systems, April 2000.

CAMERON PROJECT: FINAL REPORT

Appendix A: SA-C Manual

The SA-C Language – Version 1.0

J. P. Hammes and A. P. W. Böhm
Colorado State University

SA-C is a single-assignment, expression-oriented language designed for applications that do computationally intensive work with large arrays, targeting reconfigurable computing modules. The language mixes ideas from C, Sisal and Fortran 90.

The appendix contains SA-C's grammar, as BNF rules that are output by the parser generator.

1 Tokens

Identifier names in SA-C are case sensitive. Identifiers must start with an alphabetic character, which can be followed by any number of alphabetic, numeric, and '_' characters. Figure 1 shows SA-C's reserved words. Also, the words **bits**, **int** and **uint** followed by one or more digits, and **fix** and **ufix** followed by one or more digits, a period, and one or more digits, are reserved.

Constants are denoted as in C, with the extension of boolean constants **true** and **false**, and bits constants **0bBBB** and **0xXXX** where BBB stands for any sequence of binary digits and XXX stands for any sequence of hexadecimal digits. Some example constants are:

```
true
5
5.5
0x8A
0b0100
```

There are two kinds of comments. The first is C's conventional `/*-*/` style, which do not nest. The second is signaled by `//` which extends to the end of the current line. Otherwise, SA-C is not line oriented; line breaks represent white space.

2 Type specifications

SA-C has scalar base types, complex types, and arrays composed of scalars or complex numbers.

accum	asin	exp	lookup	sinh
acos	asinh	expm1	matrix	sqrt
acosh	assert	export	max	step
and	at	extents	mean	stripmine
array	atan	fabs	median	st_dev
array_accum	atanh	false	min	sum
array_and	atan2	final	mode	switch
array_concat	bool	float	modf	tan
array_conperim	cbrt	floor	next	tile
array_histogram	ceil	fmod	no_dfg	true
array_max	complex	for	no_fuse	vals_at_first_max
array_max_indices	concat	frexp	no_inline	vals_at_first_min
array_mean	copysign	if	no_unroll	vals_at_last_max
array_median	cos	imag	or	vals_at_last_min
array_min	cosh	in	pow	vals_at_maxs
array_min_indices	cross	histogram	print	vals_at_mins
array_mode	cube	hypot	product	vector
array_or	dot	ldexp	real	while
array_product	double	log	return	window
array_st_dev	elif	log10	rint	
array_sum	else	log1p	sin	

Figure 1: SA-C’s reserved words.

2.1 Scalars

There are eight base types in SA-C: booleans, bits, signed integers, unsigned integers, signed fixed point numbers, unsigned fixed points, floats, and doubles. SA-C targets reconfigurable computing systems, hence some parts of a SA-C program will run on the host, and some parts will run on the FPGA array. The state of the art in FPGAs is such that code involving floats and doubles will currently run on the host. The SA-C compiler will issue a warning if floats or doubles are used in parts of the program running on the FPGA array.

The integer, bits, and fixed types require size or precision specifications. For ints the size specifies the total number of bits used in their representation. For fixed point numbers the first size specifier gives the total number of bits, and the second size specifier gives the number of bits used for the fraction part. An int of size one is disallowed, no size can be larger than 32, and no fraction can be larger than the total size. Some example base types are:

```
bool
bits5
uint6
fix16.4
ufix8.8
float
```

2.2 Complex Numbers

The complex types in SA-C have signed numeric subtypes int, fix, float, or double. Example complex types are:

```
complex int8
```

```
complex fix8.8
complex double
```

Complex numbers are written as “(“ **real part**, **imaginary part** “)” such as (1.0, 0.0). If x is a complex number, then **real**(x) is its real part, and **imag**(x) is its imaginary part.

2.3 Arrays

SA-C has multidimensional arrays with components limited to scalars and complex numbers. The language emphasizes array operations and includes mechanisms for taking array *slices* (or *sections*) and creating arrays through special loop constructs. All arrays are rectangular, and the array elements are stored at equal distances in memory, so that array element addresses can be calculated using simple linear expressions. The elements are not necessarily stored contiguously, as an array can be a (e.g. column) slice of another array.

The following array-related terms are defined, drawn from Fortran 90 terminology. The *rank* of an array is the number of dimensions it has. Thus, what we commonly call a 2D-array, or matrix, is an array of rank two. An array has an *extent* for each of its dimensions. Multiplying all the extents of an array together yields the number of elements, or the *size* of the array. An array’s rank and extents together comprise its *shape*. One or more of an array’s extents can be zero, which would imply that the array’s size is zero.

An array type in SA-C describes two characteristics of an array: its component type and its rank. The type does *not* define an array’s extents. Instead, during program execution each array carries its extents with it, and these extents can be accessed explicitly through the **extents** operator, and implicitly through the loop generators. As in C, SA-C array index ranges always start at zero, and multi-dimensional arrays have a storage sequence order that varies the right-most indices the fastest. The rank-two array examples in this document are shown with rows referenced by the left index and columns by the right index. An array type specifier consists of a base type followed by a comma-separated list of either colons or integer constants enclosed in square brackets. The colon indicates that the array extent is defined dynamically, an integer constant statically declares the extent. Static extents allow the compiler to generate more efficient code. Examples of array types are:

```
int8[:]           // 1D array of 8-bit signed integers
uint4[8,:]        // 2D array (8 rows) of 4-bit unsigned integers
bool[2,2,2]       // 3D ‘hypercube’ of booleans
```

Because an array’s extents are not part of its type, arrays of different sizes can have the same type.

3 Statements

SA-C has three statements: assignment, **print** and **assert**. The latter two are used primarily in debugging, whereas assignment statements form the bulk of statement blocks. Statements are terminated by semicolons.

3.1 Assignment statements

SA-C assignments differ from those in C in three ways. First, most assignments declare and define a variable simultaneously. Second, multiple values can be assigned concurrently. Third, the assignment is a statement, not an expression.

Except for variables with loop-carried dependencies (discussed later), each variable on the left-hand side of an '=' is declared and given a type. Here are examples of simple assignments:

```
uint8 a = b + 3;          // create and define scalar 'a'
int8 A[8] = f(c,8);       // create and define rank-1 array 'A'
bool M[:, :] = g(A,q);    // create and define rank-2 array 'M'
```

The left-hand side of an assignment can have a comma-separated list of identifiers, and the right-hand side must supply the appropriate number of values. For example,

```
int8 a, uint12 b, int2 c = 12, 44, 2;
```

assigns the three values to the variables. Assigning to multiple values is useful especially with functions, loops and conditionals that may return multiple values. All right-hand sides of an assignment are computed before the values are stored to the left-hand side variables, allowing the expressions to be computed concurrently. Since there are times when not all values returned from a multiple-value computation are needed, an underscore (don't care) can occur in place of an identifier on the left side of an assignment statement. Since arrays are created only monolithically (see section 4.2), it is not possible to assign to an individual array element, so the *only* entities that can occur on the left-hand side of an assignment are identifiers and underscores.

Unlike C, an assignment is not an expression, i.e. it does not return a value. This eliminates questions involving evaluation order that would arise if assignments were embedded within expressions.

3.2 Print and assert statements

To help in debugging, the **print** and **assert** statements allow printing values during program execution, and aborting under specified conditions.

The **print** statement needs two items: a boolean expression and a comma-separated sequence of strings and identifiers. During execution, the statement will print only if its boolean expression evaluates to **true**. If so, the strings and identifier values will be printed. Four examples of print statements are shown:

```
print (v>42, "x=", x, " z=", z);
print (true, "test point two", m);
print (w==3, qq);
print (v>42, "found a 'v' greater than 42");
```

These demonstrate that the constant **true** can be used if the statement is to print unconditionally. The **assert** statement differs from **print** in two ways: it acts only if the boolean expression evaluates to **false**, and it aborts execution after the information has been printed.

3.3 Scope of variables

Each variable that is declared and defined on the left-hand side of an assignment statement is in scope beginning with the semicolon that terminates that statement and ending with the closing parentheses that ends the statement block containing the statement, except that over parts of that lexical range it may be shadowed by newly declared variables with the same name. Such newly declared variables may be at the same level as the previously declared variable, or they may be in statement blocks that are nested within the current statement block. Figure 2 shows a variable *a* that is shadowed by two other variables of that name.

```
{
...
...
uint8 a = ...
uint8 b = ...
uint8 c = if (...) {
    uint8 d = ...
    uint8 a = ...
    uint8 k = ...
} --> (...)

    else {
        uint8 e = ...
        uint8 f = ...
    } --> (...);

uint8 g = ...
uint8 h = ...
uint8 a = ...
uint8 i = ...
uint8 j = ...
} --> (...);
```

Figure 2: Example of scoping and shadowing of variable *a*.

One consequence of the declaration and scoping rules is that, other than the printing and aborting that can take place through **print** and **assert** statements, a statement block has no external side-effects. In other words, the values of variables after exiting a statement block are the same as when entering the block. Additional variable scoping issues arise in the context of **for** loops, and will be dealt with after these loops have been described.

4 Expressions

Expressions play a dominant role in SA-C programs. The language’s integer scalar arithmetic and bit operators are the same as those in C, with the same associativities and precedences. The infix binary operators are shown in figure 3; all but the condition expression’s ‘:’ and ‘?’ are left-associative. The two unary operators, ‘!’ for boolean “not” and ‘-’ for integer “negate”, have higher precedence than the infix binary operators. Scalar expressions can be type cast in a manner similar to that of C: a scalar type in parentheses preceding the expression.

*	multiply
/	divide
%	mod
+	add
-	subtract
<<	bit-shift left
>>	bit-shift right
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
==	equal
!=	not equal
&	bit-wise and
^	bit-wise xor
	bit-wise or
&&	boolean and
	boolean or
? :	conditional operator

Figure 3: Infix binary operators, in precedence groupings, highest to lowest

The numeric types are: **uint**, **int**, **ufix**, **fix**, **float**, **double**. Signed integer and fixed point arithmetic is performed in two's complement. The arithmetic operators $*$ $/$ $+$ and $-$ apply to the numeric types only. The mod operator $\%$ applies to **uints** and **ints**. Integers consist of an optional sign and an integral part. Fixed point number consists of an optional sign, an integral part, and a fraction part. An integer can be considered a fixed point number with zero fraction size. The result type of an arithmetic expression is defined as follows.

- If either of the operands is signed, the result is signed. An operation on integer or fixed operands yields a result with the maximum integral and fraction operand sizes. Neither the integral, nor the fractional size can exceed 32. If the total size of the result exceeds 32, the size of the fraction will be reduced so as to make the total size 32. As an example, a **fix16.10** combined with a **fix16.14** will result in a **fix20.14**, whereas a **fix32.10** combined with a **fix32.20** will result in a **fix32.10**.

An operation on a float and an integer or fixed point results in a float. An operation on a double and a float or integer or fixed point results in a double.

- If both operands are unsigned integers, the result of an operation is an unsigned integer congruent modulo 2^n to the true mathematical result of the operation, where n is the maximum size of the operands (e.g., $(\text{uint8})0xFF + 1 = 0x00$).

The boolean operators apply to **boolean** operands only, the bit operators apply to **bits** operands only.

The shift operators have a left operand of type **bits** and a right operand of type **uint** or **int**. The result is of the same type as the left operand. A negative right operand gives rise to an unspecified result, and possibly (on the host, NOT on the FPGA) a run time warning.

The comparison operators `==` `!=` `<` `>` `<=` and `>=` apply to all scalar types, where **bits** are interpreted as **uints**, and *false* `<` *true*. The equality comparison operators `==` and `!=` also apply to complex numbers.

The cast operation is at the same precedence level as the other unary operators. Unless sub-expressions are explicitly cast, a cast expression defines the maximum precision of all its sub-expressions. An assignment **ltype** `lhs = rhs` has the same meaning as **ltype** `lhs = (ltype) rhs`, i.e. the right hand side is cast to the type of the left hand side.

Casting a value of some type to another type, either explicitly or through assignment, can have two effects: the value can be **converted**, i.e. its bit representation can be changed, or the value can be **interpreted** in terms of the new type without change in its bit representation (apart from truncating leftmost bits or adding zero bits on the left to adjust size.) Interpretation occurs when casting any type to and from **bits**. Conversion occurs when any **non bits** is cast to any **non bits**. Casting an unsigned number to a signed number first adds a sign bit to the representation, and then either pads or truncates according to size specifications. Casting a signed negative number to an unsigned number has an unspecified result and causes a runtime error on the host.

4.1 Statement blocks

A block of statements can occur as an expression. SA-C blocks differ from C blocks in that they always produce return values, signaled by the keyword **return** which is followed by a parenthesized, comma-separated list of expressions. For example,

```
{ uint4 a = 3;
  int6 b = 4;
  int8 c = a*a - b*b;
} return (c, a+b)
```

creates and assigns values to three scalar variables and returns two values computed with them. Since SA-C is a *single-assignment* language, no variable can be given more than one value.

When a statement block occurs on the right hand side of an expression, it must contain at least one statement. For example:

```
int8 x, int6 y = return (c, a+b);
```

is incorrect, and should be written as

```
int8 x, int6 y = c, a+b;
```

Statement blocks also occur as bodies of functions, conditionals and loops. In these cases, where the return value can be computed directly with no statements or declarations within the curly braces, the braces may be omitted.

4.2 Array operations

Arrays can be created in a number of ways. First, an array can be explicitly typed, sized by integer constants, and given an expression for each of its elements in an assignment. For instance:

```
uint8 A[3,4] = {{3,6,7,2}
                ,{4,3,2,1}
                ,{7,1,0,8}};
```

The size of the array must be specified on the left hand side of the assignment, and the patterns within the curly braces must match the size specifiers. Since array extents can be zero, it is possible for a set of curly braces to enclose no values. A zero extent makes an array's size zero, but the curly braces still must match the specified extents. For example, both of the following array expressions create size-zero arrays, but note how the arrays' extents require different patterns of curly braces:

```
int8 Z1[0,3] = {};
int8 Z2[3,0] = { {}, {}, {} };
```

The second way of creating an array is through a return value of a loop. These are dealt with separately in section 4.4.

The **extents** operator returns the extents of an array as multiple values. For example, assuming the above definition of *A*,

```
uint8 m, uint8 n = extents (A);
```

will read the extents of *A* and put them into the scalar variables *m* and *n*. Individual array elements can be referenced using a single set of square brackets, with comma-separated integer expressions. Array slices can be taken using colon notation, similar to that of Fortran 90. A lone colon in a given dimension signifies taking the entire index range of the dimension. A subrange can be specified with integer expressions on either or both sides of the colon, and an optional step as a third parameter. Here are some examples, assuming the above definition of array *A*:

```
A[1,2]           // returns scalar value '2'
A[:,1]           // returns vector {6,3,1}
A[0,:]           // returns vector {3,6,7,2}
A[2,1:3]         // returns vector {1,0,8}
A[0:1,0:1]       // returns matrix {{3,6},{4,3}}
A[0,2:]          // returns vector {7,2}
A[0,:2:2]        // returns vector {3,7}
```

The **array_concat(A,B)** operator creates an array by concatenating the arrays *A* and *B*, which have to be of equal type, and of equal extents in all but the rightmost dimension. For example, if *A* is as above and *B* is defined as

```
uint8 B[3,2] = {{3,2},{4,1},{7,8}}
```

then **array_concat(A,B)** will create the array:

$$\begin{bmatrix} 3 & 6 & 7 & 2 & 3 & 2 \\ 4 & 3 & 2 & 1 & 4 & 1 \\ 7 & 1 & 0 & 8 & 7 & 8 \end{bmatrix}$$

The **array_conperim(A,w,v)** operator creates an array by surrounding an existing array *A* with a **perimeter** of a certain width *w*. All perimeter elements get the value *v*. For example,


```
uint8[:,:] C = array_conperim(A,1,0);
```

creates the following array C:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 6 & 7 & 2 & 0 \\ 0 & 4 & 3 & 2 & 1 & 0 \\ 0 & 7 & 1 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

An array also can be referenced through the reduction operators described in figure 4. These have the appearance of function calls, but they are actually operators. All but the **array_histogram** operator take one mandatory argument: the array being reduced. The **array_histogram** operator requires two arguments: the source array and a range value for the histogram array being produced. An optional final argument can be given, specifying a boolean mask array whose shape must be identical to the value array being reduced.

Unless their operands are cast to a more appropriate type, the results of **array_sum**, **array_product**, **array_min**, **array_max**, **array_median**, **array_mean**, **array_st_dev**, **array_and** and **array_or** have the same type as the array components; **array_min_indices**, **array_max_indices** and **array_histogram** produce arrays of **uint32s**.

Most of these array reduction operators can be used with the **array_accum** operator, which allows separate reductions to be performed on regions of the source array, specified by an array of labels. (The **array_mode**, **array_min_indices** and **array_max_indices** cannot be used in an accumulation since they cannot guarantee that the resulting array would be rectangular. This is because different regions of the source array may have different numbers of modes, mins or maxes, which would lead to ragged arrays.) The **array_accum** operator always takes three arguments: a reduction, a range value for the resulting array, and a label array whose shape must be identical to the array being reduced. A run-time error will occur if a label in the label array exceeds the extent set by the range specification.

Figure 5 shows examples of the integer reduction operators and their return values, and figure 6 shows examples of the **array_accum** operator.

4.3 Conditional expressions

Conditionals in SA-C return values, so every **if** must have a matching **else**. Additional **elif** branches between the **if** and **else** branches are allowed. Each branch of a conditional has an optional block of statements, and must return the same number and types of values. Recall that for arrays, type matching means that they must have the same scalar types for their values, and must have the same rank. They are *not* required to have the same extents. Here is an example using a conditional expression:

```
int8 x =    if (m > 8) { int8 z = q*3 } return (z%4)
           elif (m < 5) { int8 s = q*q-p } return (m+s)
           else return(0);
```

Multiple value returns are very useful with conditionals when more than one value needs to be set based on a condition. For example:

array_sum	Sum the array elements. If the array is empty, the return value is zero.
array_product	Multiply the array elements. If the array is empty, the return value is one.
array_min	Find the minimum of the array elements. This reduction does not work on arrays of complex numbers. If the array is empty, a run-time error occurs.
array_max	Find the max of the array elements. This reduction does not work on arrays of complex numbers. If the array is empty, a run-time error occurs.
array_and	'And' the boolean or bits array elements. If the array is empty, a 'true' value or all ones value is returned.
array_or	'Or' the boolean or bits array elements. If the array is empty, a 'false' value or all zeroes value is returned.
array_median	Find the median of the array elements. This reduction does not work on arrays of complex numbers. If the array is empty, a run-time error occurs.
array_mean	Find the mean of the array elements. If the array is empty, a run-time error occurs.
array_st_dev	Find the standard deviation of the array elements, returning a scalar integer. If the array size is less than two, a run-time error occurs.
array_mode	Find the mode of the array elements, returning a rank-one integer array of the values which occur most. If the array is empty, an empty array is returned.
array_min_indices	Find the index locations of the minimal array elements, returning a rank-two integer array. This reduction does not work on arrays of complex numbers. If the array is empty, an empty array is returned.
array_max_indices	Find the index locations of the maximal array elements, returning a rank-two integer array. This reduction does not work on arrays of complex numbers. If the array is empty, an empty array is returned.
array_histogram	Create a histogram of the array elements, returning a rank-one integer array. This requires a range specifier as its second argument to set the size of the returned array.

Figure 4: Array reduction operators

$$A = \begin{bmatrix} 2 & 1 & 9 & 2 \\ 6 & 9 & 2 & 9 \\ 1 & 9 & 1 & 1 \end{bmatrix}, M = \begin{bmatrix} \text{true} & \text{false} & \text{true} & \text{true} \\ \text{true} & \text{false} & \text{false} & \text{true} \\ \text{false} & \text{true} & \text{false} & \text{true} \end{bmatrix}$$

array_sum (A)	52
array_sum (A, M)	38
array_product (A)	314928
array_product (A, M)	17496
array_max (A)	9
array_max (A, M)	9
array_min (A)	1
array_min (A, M)	1
array_max_indices (A)	$\begin{bmatrix} 0 & 2 \\ 1 & 1 \\ 1 & 3 \\ 2 & 1 \end{bmatrix}$
array_max_indices (A, M)	$\begin{bmatrix} 0 & 2 \\ 1 & 3 \\ 2 & 1 \end{bmatrix}$
array_min_indices (A)	$\begin{bmatrix} 0 & 1 \\ 2 & 0 \\ 2 & 2 \\ 2 & 3 \end{bmatrix}$
array_min_indices (A, M)	$\begin{bmatrix} 2 & 3 \end{bmatrix}$
array_histogram (A, 10)	$\begin{bmatrix} 0 & 4 & 3 & 0 & 0 & 0 & 1 & 0 & 0 & 4 \end{bmatrix}$
array_histogram (A, 10, M)	$\begin{bmatrix} 0 & 1 & 2 & 0 & 0 & 0 & 1 & 0 & 0 & 3 \end{bmatrix}$
array_mean (A)	4
array_mean (A, M)	5
array_mode (A)	$\begin{bmatrix} 1 & 9 \end{bmatrix}$
array_mode (A, M)	$\begin{bmatrix} 9 \end{bmatrix}$
array_st.dev (A)	4
array_st.dev (A, M)	4
array_median (A)	2
array_median (A, M)	6

Figure 5: Example of some array reduction operations.

$$A = \begin{bmatrix} 2 & 1 & 9 & 2 \\ 6 & 9 & 2 & 9 \\ 1 & 9 & 1 & 1 \end{bmatrix}, M = \begin{bmatrix} \text{true} & \text{false} & \text{true} & \text{true} \\ \text{true} & \text{false} & \text{false} & \text{true} \\ \text{false} & \text{true} & \text{false} & \text{true} \end{bmatrix}, L = \begin{bmatrix} 1 & 1 & 2 & 0 \\ 1 & 3 & 2 & 0 \\ 3 & 3 & 2 & 2 \end{bmatrix}$$

array_accum (array_sum (A), 4, L)		11	9	13	19	
array_accum (array_sum (A, M), 4, L)		11	8	10	9	
array_accum (array_product (A), 4, L)		18	12	18	81	
array_accum (array_product (A, M), 4, L)		18	12	9	9	
array_accum (array_max (A), 4, L)		9	6	9	9	
array_accum (array_max (A, M), 4, L)		9	6	9	9	
array_accum (array_min (A), 4, L)		2	1	1	1	
array_accum (array_min (A, M), 4, L)		2	2	1	9	
array_accum (array_histogram (A, 10), 4, L)	$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$					
array_accum (array_histogram (A, 10, M), 4, L)	$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$					
array_accum (array_mean (A), 4, L)		6	3	3	6	
array_accum (array_mean (A, M), 4, L)		6	4	5	9	
array_accum (array_st_dev (A), 4, L)		5	3	4	5	
array_accum (array_st_dev (A, M), 4, L)	run-time error (st_dev of size-one array)					
array_accum (array_median (A), 4, L)		9	2	2	9	
array_accum (array_median (A, M), 4, L)		9	6	9	9	

Figure 6: Examples of some **array_accum** operations.

```

uint8 new_xa, uint8 new_xb, uint8 val =
    if (A[xa] < B[xb])
        return (xa+1, xb, A[xa])
    else
        return (xa, xb+1, B[xb]);

```

embodies an operation that could occur during the merging of two arrays. If the current *A* value is less than the current *B* value, the *A* value is “taken” by returning an incremented index *xa*, an unchanged index *xb*, and the value from *A*. Otherwise *A*’s index is unchanged, *B*’s index is incremented, and the value from *B* is taken.

The **switch** expression takes an **int** or **uint** argument to determine which case branch is to be taken. The values in the cases are constants.

Here is an example:

```

uint8 x = switch(a-64) {
    case 0,1,2: { int8 z = q*3 }   return(z%4)
    case 3,4:   { int8 s = q*q-p } return(m+s)
    default:    return(0)
}

```

If the default branch is absent and none of the cases apply, the result of the switch expression is unspecified (and will give rise to a run time error in host implementations.) As in C, overlapping cases are not allowed.

4.4 Loops

Loops in SA-C have return values. There are two kinds of loops, denoted by the keywords **while** and **for**. The **while** loop is the more general, and is designed for situations where, upon encountering the loop, the executing program cannot determine how many iterations will take place. The **for** loop, in contrast, determines exactly how many times the loop will iterate. It is not possible to break out of a **for** loop early.

4.4.1 While loops

The **while** loop looks similar to its C counterpart, except for the return value(s) following its closing curly brace. Because the single-assignment principle is in direct conflict with a loop’s need to update an iteration variable, SA-C views a variable as if it has a separate instantiation for each iteration of the loop. The new value for an iteration can be assigned by using the keyword **next** in place of the type declaration that would otherwise occur. SA-C allows a **next** assignment to each variable at most once *within the body of a loop*, in addition to an assignment that takes place prior to the loop.

As an example, consider the following SA-C code:

```

uint8 n=5; uint8 ac=0;
uint8 s = while (n>0) {
    next ac = ac+n;
    next n = n-1;
} return(final(ac));

```

There are loop-carried dependencies in this example, in variables *ac* and *n*. Each of the variables assigned in the loop can be thought of as having a separate instantiation for each iteration, and the semantics of the loop can be seen as if it is fully unrolled:

```
n=5; ac=0;
ac1 = ac+n;
n1 = n-1;
ac2 = ac1+n1;
n2 = n1-1;
ac3 = ac2+n2;
n3 = n2-1;
ac4 = ac3+n3;
n4 = n3-1;
ac5 = ac4+n4;
n5 = n4-1;
s = ac5;
```

Note that any variable with a loop-carried dependency *must* have been defined before the loop is entered; otherwise the use of that variable in iteration one would not have a value. Simple static analysis by the compiler can detect the failure to give an initial value and report it as an error. Note also that the loop returns a value for *s* but leaves the initial values given to *n* and *ac* unchanged (because of the side-effect-free nature of statement blocks.) Thus any reads of either of these variables in statements following the loop will get a value of 5 for *n* and a value of 0 for *ac*.

As with all statement bodies in SA-C, the **while** loop's body can return multiple values, and if there are no statements the curly braces may be omitted (though this would cause nontermination if the loop is entered.) Return values of loops are fully dealt with in section 4.4.2.

4.4.2 For loops

A For loop is used when the number of iterations is known before the loop is executed. There are two types of for loops: for loops with loop carried dependencies, expressed using **next** assignments, and “forall” loops without loop carried dependencies.

For loops have three parts: a *generator*, a *body*, and a *return expression*.

Loop generators **For** loops use generators to create iterations, and to create values for those iterations. There are three *simple* forms: *scalar*, *array-component* and *window* generators. Simple generators can be combined into compound generators using **dot** and **cross** loop product operators, discussed later in this section.

Scalar generators SA-C uses a ‘~’ notation to generate scalar values. For example,

```
for uint8 i in [5~10]
```

will produce six iterations, with *i* having the values 5, 6, 7, 8, 9, and 10. Multiple values can be generated, using comma-separated lists. This example shows a SA-C generator of two target variables, and its C equivalent:

```

/* SA-C: */
for int8 i, int8 j in [5~10,2~4]

/* C: */
for (i=5; i<=10; i++)
    for (j=2; j<=4; j++)

```

Because a loop's iteration variables and ranges are often associated with array extents, a single expression (i.e., without a '~') automatically generates the appropriate index range for an array of that extent:

```

/* SA-C: */
for int8 i, int8 j in [m,n]

/* C: */
for (i=0; i<m; i++)
    for (j=0; j<n; j++)

```

If non-unit steps are needed, an optional **step** specification can be used:

```

/* SA-C: */
for int8 i, int8 j in [m,n] step (a,b)

/* C: */
for (i=0; i<m; i+=a)
    for (j=0; j<n; j+=b)

```

Occasionally scalar generators may be used to create a specific set of iterations without needing to reference an iteration variable. An underscore ('_') can take the place of such a variable:

```

for _ in [n]

```

will produce n iterations without having to declare an iteration variable.

Array-component generators An array component generator takes components (elements, vectors, planes, etc.) out of an array and sends these into the iterations of a loop. Here is an example of its simplest form: `for val in A`. Array A can be of any shape. The iterations will take element values from A , in storage sequence order, and assign them to *val*, one value per iteration. The variable *val* can be used inside the loop's body and in its return value expressions. Other array access patterns are possible. The general form of an array component generator is:

```

for val (access-pattern) in array-expression

```

For each dimension in the array to be accessed, the access pattern has either a '~' specifier to indicate that individual elements are to be accessed, or a ':' to indicate that a whole slice in the array is to be accessed. The default access pattern takes individual elements out of the array specified by the array-expression, so `for val in A` has the same meaning as `for val (~,~) in A` and `for vec (~,:) in A` accesses all rows in A , whereas `for vec (:,~) in A` accesses all columns in A . An access pattern with all colons is not allowed, as this does not generate a loop.

The **step** specification allows one to stride through an array: `for val in A step(2,2)` only accesses the even rows and columns of A . An underscore must occur in positions where a slice is accessed, and a value must occur in positions where individual elements are accessed.

The **at** specification is used in cases where one needs the array indices, as in `for val in A at (uint8 i, uint8 j)`. Here the variables i and j capture the array indices, and can be referenced both in the body of the loop and in the return expressions. Underscores are used in **at** specifications, both in positions where there is no iterating index, because a whole slice is accessed, and in positions where the user does not need to capture an index value. Example: `for vec (:,~) in A at (_,uint8 j)`.

Window generators Another method of generating sub-arrays is the **window** generator. This allows a window to “slide” over the source array, producing sub-arrays of the same rank as the source array. The **window** keyword signals such a generator, with a size specification. For example: `for window W[3,3] in A` will access all 3 by 3 sub-matrices W from A .

Windows work on arrays of any rank, but the ranks of the window size specification and the source array must be equal. A window generator can be given an optional step specifier, for example: `for window W[3,3] in A step (3,3)` will produce non-overlapping 3 by 3 sub-matrices W .

Dot and cross products Compound generators, made up of simple generators, can be formed as **dot** and **cross** products.

The simple generators that are combined in **dot** products must be of identical shapes, i.e. dimensionality and extents, so that corresponding elements from each component can be sent into a loop body. Here is an example of a **dot** product, where two arrays are added together on an element-by-element basis:

```
for a in A dot b in B {
    int10 s = a + b;
```

The **cross** product generates all combinations of values from a set of simple generators; each simple generator can be of any shape, e.g., `for int8 i in [1~3] cross int8 j in [2~3]` will produce the (i, j) value pairs (1,2), (1,3), (2,2), (2,3), (3,2), and (3,3).

Generator shapes The *shape* of an array has already been defined as its rank and extents. In SA-C a loop’s generator is considered to have a *shape* as well; it is used in defining how some of the loop’s return operators construct their values.

The rank of a scalar generator is equal to the number of comma-separated entries in the generator’s square brackets. The rank of an array-component generator is equal to the number of ‘~’ specifiers in its access pattern. The rank of a window generator is equal to the rank of the window itself. Each extent is equal to the number of produced iterations. Here are some examples:

```
uint4 A[6,8,9] = {{... ;
for v in A                                // generator shape [6,8,9]
for v in A[2:4,0,5:8]                    // generator shape [3,4]
for v (~,:,~) in A                        // generator shape [6]
for int8 i, int8 j in [5,6]              // generator shape [5,6]
for int8 i in [7~11]                     // generator shape [5]
```

In a valid **dot** product, the shapes of the components must be the same; the shape of a **dot** product is the same as the shape of each of its components. The shape of a **cross** product is obtained by concatenating the shapes of its components. For example:

```
uint4 A[6,8,9] = {{... ;
for v in A cross int8 i in [7]            // generator shape [6,8,9,7]
...
for v (:,~,~) in A cross w (~,:,~) in A  // generator shape [8,6,9]
...
```


Loop-indices can be assigned to local loop body variables, as in

```
for e in A step(2,2)
  uint8 i, uint8 j = loop_indices();
  ...
```

produces consecutive values for i and j starting from 0.

Loop return values There are three kinds of return values that can follow the closed curly brace of a loop: an expression, a reduction, and a structuring operation.

Final values A loop can return the final value of a nextified variable. For example, here is one way of calculating n -factorial:

```
uint8 ac = 1;
uint16 fact =
  for uint8 v in [2~n] {
    next ac = ac * v;
  } return(final(ac));
```

The variable ac is being updated in each iteration, and its final value is returned.

Reductions All of the array reduction operators have exact counterparts as loop reductions; the loop keywords are formed by removing the **array_** prefixes from the array reduction operator names. Loop reductions can be performed on scalar and complex values, unless specified otherwise. Unless they are cast to a more appropriate type, the result types of **sum**, **product**, **min**, **max**, **median**, **mean**, **st_dev**, **and** and **or** are the same as the operand types, and **histogram** produces an array of **uint32s**. The array element type of **vals_at_mins** and **vals_at_maxs** is determined in the same way as expression results (see section 4). Figure 7 defines the loop reductions.

Just like array reductions, the loop reduction forms can be given an optional mask value as a scalar boolean expression. The **accum** operator uses loop reduction operators in a manner similar to that of the **array_accum** operator; the label specification is a integer expression.

As an example, here is a loop that produces the sum and mean of the odd numbers from one to one hundred:

```
uint16 s, ufix16.4 m = for uint8 i in [1~100] step (2) return (sum(i), mean(i));
```

The logical reductions **and** and **or** are useful for determining whether some condition occurred during loop execution. For example, a loop to determine whether array B contains any values greater than 42 could look like this:

```
bool s = for b in B return (or (b>42));
```

Structure-building operators There are two kinds of structure-building operators that can be used in creating a loop return value: **array** and **concat**.

sum	Sum the values. If the loop zero trips, the return value is zero.
product	Multiply the values. If the loop zero trips, the return value is one.
min	Find the min of the values. This reduction does not work on complex numbers. If the loop zero trips, a run-time error occurs.
max	Find the max of the values. This reduction does not work on complex numbers. If the loop zero trips, a run-time error occurs.
and	'And' the boolean or bits values. If the loop zero trips, a 'true' value or all ones value is returned.
or	'Or' the boolean or bits values. If the loop zero trips, a 'false' value or all zeroes is returned.
median	Find the median of the values. This reduction does not work on complex numbers. If the loop zero trips, a run-time error occurs.
mean	Find the mean of the values. If the loop zero trips, a run-time error occurs.
st_dev	Find the standard deviation of the values. In the case of a zero or one trip loop, a run-time error occurs.
mode	Find the mode of the values, returning a rank-one array of the values which occur most. If the loop zero trips, an empty array is returned.
vals_at_mins(x,{p,q,r,..})	Find the specified values p,q,r,... in the loop instances where x is minimal. This returns a rank-two array, each row containing a set of values p,q,r,.. of the same type. This reduction does not work when x is a complex number. If the loop zero trips, an empty array is returned.
vals_at_maxs(x,{p,q,r,..})	Find the specified values p,q,r,... in the loop instances where x is maximal. This returns a rank-two array, each row containing a set of values p,q,r,.. of the same type. This reduction does not work when x is a complex number. If the loop zero trips, an empty array is returned.
vals_at_first_min(x,{p,q,r,..})	Find the specified values p,q,r,... in the first loop instance where x is minimal. This returns a rank-one array of values p,q,r,.. of the same type. This reduction does not work when x is a complex number. If the loop zero trips, an empty array is returned.
vals_at_first_max(x,{p,q,r,..})	Find the specified values p,q,r,... in the first loop instance where x is maximal. This returns a rank-one array of values p,q,r,.. of the same type. This reduction does not work when x is a complex number. If the loop zero trips, an empty array is returned.
vals_at_last_min(x,{p,q,r,..})	Find the specified values p,q,r,... in the last loop instance where x is minimal. This returns a rank-one array of values p,q,r,.. of the same type. This reduction does not work when x is a complex number. If the loop zero trips, an empty array is returned.
vals_at_last_max(x,{p,q,r,..})	Find the specified values p,q,r,... in the last loop instance where x is maximal. This returns a rank-one array of values p,q,r,.. of the same type. This reduction does not work when x is a complex number. If the loop zero trips, an empty array is returned.
histogram	Create a histogram of the array elements, returning a rank-one uint32 array. This requires a range specifier as its second argument to set the size of the returned array.

Figure 7: Loop reduction operators

The **array** operator builds an array out of the specified component in its parentheses. For example, the following loop returns an array of the even numbers from two to one hundred:

```
uint8 nums[:] = for uint8 v in [2*100] step (2) return (array (v));
```

The specified component of an **array** operator can itself be an array, and since SA-C does not have arrays-of-arrays, this returns a result array whose rank is the sum of the ranks of the generator and the component. Thus, if a loop's generator has shape [3,5] and the **array** operator is used on components of shape [8,2], the return value will be a rank-four array of shape [3,5,8,2]. A SA-C programmer can use the **vector**, **matrix** and **cube** keywords in place of **array** for return values of rank one, two and three respectively. As an example, assuming that the array A = {1,3,5}, then

```
for a in A {
  uint8 V[2] = for _ in [2] return(vector(a));
} return(matrix(V))
```

will create a two dimensional array with extents [3,2]:

$$\begin{bmatrix} 1 & 1 \\ 3 & 3 \\ 5 & 5 \end{bmatrix}$$

The **concat** and **tile** operators have meaning only where the components being combined are arrays. Concat works only in one dimensional loops and allows the component arrays to be concatenated to have different sizes in the last dimension, as in the **array_concat** operator. Tile works in loops of any dimension, and hence all component arrays must be of exactly the same shape. The components are tiled together in each dimension of the loop. The rank of the value from a **tile** operator is the max of the ranks of the generator and the component. The extents are obtained by right-aligning the two extents and multiplying them element-wise. For example, if the loop generator has shape [3,9,5] and the **tile** component has extents [7,11], the return value of the operator will be a rank-three array of extents [3,63,55].

For example, assuming again that A = {1,3,5}, then

```
for a in A {
  uint8 V[:] = for _ in [a] return(array(a));
} return(concat(V))
```

will create a one dimensional array with extents [9]:

$$[1 \ 3 \ 3 \ 3 \ 5 \ 5 \ 5 \ 5 \ 5]$$

As an other example, assuming that B is:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

then `for _ in [3] return(tile(B))` will create a matrix with extents [2,6]:

$$\begin{bmatrix} 1 & 2 & 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 & 3 & 4 \end{bmatrix}$$

and `for _,_ in [2,3] return(tile(B))` will create a matrix with extents [4,6]:

$$\begin{bmatrix} 1 & 2 & 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 & 3 & 4 \\ 1 & 2 & 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 & 3 & 4 \end{bmatrix}$$

The **accum** operator takes three arguments: a reduction, an integer range value for the resulting array, and a label value in the loop body. For each label value, the reduction is applied to each loop body with that label value. Label values should be in the range defined by the range specification, otherwise a run-time error will occur. As an example:

```
for v in V dot l in L
return(accum(sum(v),256,l));
```

produces an array of extents [256] with sums of values v for each label value l from 0 to 255.

Reductions **mode**, **vals_at_mins**, and **vals_at_maxs** cannot be used in an accumulation since they cannot guarantee that the resulting array would be rectangular.

Scope of variables in ‘for’ loops The scope rules for a variable created in a loop generator are similar to the rules for other variables. A generator target variable’s scope is the lexical range beginning with the variable’s declaration and ending with the closing parentheses of the loop’s return values. However, there is a semantic restriction on the use of generator target variables: they may not be referenced elsewhere in that loop’s generator. Also, the range expressions in a loop’s return specification may not reference any variable that has been declared in that loop’s generators or body.

5 Function definitions

SA-C has both user defined functions and intrinsic functions.

5.1 User Defined Functions

A SA-C function definition has a form similar to that used in C, but allowing for SA-C types and multiple return values. For example, the line

```
int8, uint4[:,:] f1 (uint8 A[:,:], int8 z, bool M[:]) {
```

is the beginning of function **f1**’s definition. It returns two values: a scalar 8-bit signed integer and a 2D-array of 4-bit unsigned integers. It takes three arguments. Here is a complete example of a SA-C transpose function:

```
uint8[:,:] transpose (uint8 A[:,:]) {
    uint8 res[:,:] = for V (:,~) in A return (matrix (V));
} return (res);
```

It can be written more simply:

```
uint8[:,:] transpose (uint8 A[:,:])
    return (for V (:,~) in A return (matrix (V)));
```

Function calls look the same as they do in C: the function name is followed by a parenthesized, comma-separated list of argument values.

A SA-C program consists of one or more function definitions. The function named “main” is considered the top-level function. All function names are in scope throughout the entire program. SA-C functions cannot be recursive, as a SA-C program is to be laid out on an FPGA.

Functions defined in other modules need to have a function prototype. For example,

```
int8 fi8 (int8);
```

declares **fi8** to have been defined elsewhere. Function prototypes of the same function can occur multiple times in the same file as long as they are consistent, and they can occur with the actual function definition.

5.2 Intrinsic Functions

SA-C recognizes the C math library functions in figure 8 as intrinsic functions. These functions will invoke the C math library on host implementations. In figure 8, **NUM** stands for any numerical non-complex type. Some functions return more than one result. The first result is defined as the result of the corresponding C math function, the second result is what the corresponding C function returns in a reference parameter.

Return Type(s)	Function	Parameter Type(s)
double	sin	NUM
double	cos	NUM
double	tan	NUM
double	asin	NUM
double	acos	NUM
double	atan	NUM
double	atan2	NUM, NUM
double	sinh	NUM
double	cosh	NUM
double	asinh	NUM
double	acosh	NUM
double	atanh	NUM
double	sqrt	NUM
double	cbrt	NUM
double	pow	NUM
double, double	modf	NUM
double	exp	NUM
double, int32	frexp	NUM
double	ldexp	NUM
double	log	NUM
double	log10	NUM
double	expm1	NUM
double	log1p	NUM
double	ceil	NUM
double	fabs	NUM
double	floor	NUM
double	fmod	NUM, NUM
double	copysign	NUM, NUM
double	hypot	NUM, NUM
double	rint	NUM

Figure 8: Return Type(s), Intrinsic Function, Parameter Type(s)

A Program examples

The following small programs may help to illustrate SA-C's language principles.

A.1 Matrix multiply

```
// matrix multiply of 'A' and 'B':
uint8[:,:] main (uint8 A[:,:], uint8 B[:,:]) {
    _, uint8 n = extents (A);
    uint8 m, _ = extents (B);
    assert (n==m, "matrices cannot be multiplied");
    uint8 res[:,:] =
        for VA (~,:) in A cross VB (:,~) in B {
            uint8 Ele = for a in VA dot b in VB return (sum (a * b)); // inproduct(VA,VB)
        } return (matrix (Ele));
    } return (res);
```

A.2 Flattening a matrix to a vector

```
int8[:] main (int8 A[:,:]) return(
    for V (~,:) in A return (tile(V))
);
```

A.3 Expanding an Image by doubling both extents

```
// Each pixel is replicated in both dimensions
uint8[:] main (uint8 A[:,:]) return(
    for e in A {
        uint8 E[2,2] = {{e,e},{e,e}};
    } return(tile(E))
);
```

A.4 Shrinking an Image

```
// Each 2*2 submatrix is shrunk to one value (the maximum)
uint8[:] main (uint8 A[:,:]) return(
    for window W[2,2] in A step (2,2) return(matrix(array_max(W)));
);
```

A.5 Restructuring a vector to a matrix

```
int8[:,:] main (int8 A[:], int8 n) {
    int8 s = extents (A);
    assert (s%n==0, "result array not rectangular", s, n);
    int8 res[:,:] = for window V[n] in A step (n) return (array (V));
    } return (res);
```

A.6 $S \times S$ Median filter

```
// s*s Median filter of an image, padded with an s/2 perimeter
uint8[:,:] main (int8 A[:,:], uint8 s) {
// s should be odd
    assert (( (bits8)(s) & 0b1 ) == 0b1, "s should be odd\n");
    uint8 res[:,:] = for window W[s,s] in array_conperim(A,s/2,0)
        return(array(array_median(W)));
} return(res);
```

A.7 Prewitt Horizontal Edge Detection

```
// Convolution with constant Mask
int16[:,:] main (uint8 A[:,:]) {
    int8 Mask[3,3] = { {-1,-1,-1} ,
                       { 0, 0, 0} ,
                       { 1, 1, 1} } ;

    int16 res[:,:] = for window W[3,3] in A {
        int16 ip = for w in W dot m in Mask
            return( sum (w*m) );
        } return( array(ip) );
} return( res );
```

A.8 Morphology: Conway's game of *Life*

```
uint1[:,:] main (uint1 A[:,:], uint16 n) {

    bool M[3,3] = { {true, true, true} ,
                    {true,false, true} ,
                    {true, true, true} } ;

    uint1 res[:,:] =
    for _ in [n] {
        next A = for window W[3,3] in array_conperim (A, 1, 0) {
            uint3 c = array_sum (W, M);
            uint1 v = (c==3 || c==2 && W[1,1]==1) ? 1 : 0;
        } return (array (v));
        print (true, A);
    } return (A);
} return (res);
```


A.9 Parallel Prefix

```
// Produce an array with the sum of input array elements 0 .. i in position i
uint8[:] main(uint8 vec[:]) {
    uint8 sz = extents(vec);
    uint8 dist = 1;
    uint16 cumvec[:] =
        while (dist < sz) {
            next vec = for e in vec at (uint8 i) {
                uint8 elem = ((i < dist) ? e : e + vec[i-dist]);
            } return(array(elem));
            next dist = 2 * dist;
        } return(vec);
    } return (cumvec);
```

A.10 Region Statistics

```
// For each region in A, indicated by a label in LabelPlane L
// with R different labels, produce the mean value.
uint8[:] main (uint8 A[:,:], uint8 L[:,:], uint8 R ) return(
    for pix in A dot lab in L return( accum(mean(pix),R,lab))
);
```

A.11 Region Statistics: array operators

```
// Same as above
uint8[:] main (uint8 A[:,:], uint8 L[:,:], uint8 R ) return(
    array_accum(array_mean(A),R,L)
);
```

B BNF rules

```
rule 1    program -> exports funcs
rule 2    exports -> /* empty */
rule 3    exports -> TOK_EXPORT export_list ';'
rule 4    export_list -> TOK_IDENT
rule 5    export_list -> export_list ',' TOK_IDENT
rule 6    funcs -> func
rule 7    funcs -> funcs func
rule 8    func -> opt_func_prag header '(' types_w_ids ')' body_w_return ';'
rule 9    func -> opt_func_prag header '(' ')' body_w_return ';'
rule 10   func -> opt_func_prag header '(' types_wo_ids ')' ';'
rule 11   func -> opt_func_prag header '(' types_w_ids ')' ';'
rule 12   func -> opt_func_prag header '(' ')' ';'
rule 13   opt_func_prag -> /* empty */
rule 14   opt_func_prag -> TOK_PRAGMA '(' func_pragmas ')'
rule 15   func_pragmas -> func_pragma
rule 16   func_pragmas -> func_pragmas ',' func_pragma
rule 17   func_pragma -> TOK_NO_INLINE
rule 18   func_pragma -> TOK_LOOKUP
rule 19   header -> types_wo_ids TOK_IDENT
rule 20   types_wo_ids -> type_wo_id
rule 21   types_wo_ids -> types_wo_ids ',' type_wo_id
rule 22   type_wo_id -> type optbrackets
rule 23   types_w_ids -> type_w_id
rule 24   types_w_ids -> types_w_ids ',' type_w_id
rule 25   type_w_id -> type TOK_IDENT optbrackets
rule 26   base_type -> TOK_INT
rule 27   base_type -> TOK_FIX
rule 28   base_type -> TOK_BITS
rule 29   base_type -> TOK_BOOL
rule 30   base_type -> TOK_FLOAT
rule 31   base_type -> TOK_DOUBLE
rule 32   type -> base_type
rule 33   type -> TOK_COMPLEX base_type
rule 34   optbrackets -> /* empty */
rule 35   optbrackets -> '[' optsizes ']'
rule 36   optsizes -> optsize
rule 37   optsizes -> optsizes ',' optsize
rule 38   optsize -> ':'
rule 39   optsize -> TOK_INTNUM
rule 40   for_loop -> opt_loop_prag TOK_FOR generators '{' stmts '}' TOK_RETURN '(' ret_exprs ')'
rule 41   for_loop -> opt_loop_prag TOK_FOR generators TOK_RETURN '(' ret_exprs ')'
rule 42   opt_loop_prag -> /* empty */
rule 43   opt_loop_prag -> TOK_PRAGMA '(' loop_pragmas ')'
rule 44   loop_pragmas -> loop_pragma
rule 45   loop_pragmas -> loop_pragmas ',' loop_pragma
rule 46   loop_pragma -> TOK_NO_UNROLL
rule 47   loop_pragma -> TOK_NO_DFG
rule 48   loop_pragma -> TOK_STRIPMINE '(' const_list ')'
rule 49   loop_pragma -> TOK_NO_FUSE
rule 50   generators -> simple_gen opt_cmpnds
rule 51   opt_cmpnds -> /* empty */
```

```

rule 52  opt_cmpnds -> more_dots
rule 53  opt_cmpnds -> more_crosses
rule 54  more_dots -> TOK_DOT simple_gen
rule 55  more_dots -> more_dots TOK_DOT simple_gen
rule 56  more_crosses -> TOK_CROSS simple_gen
rule 57  more_crosses -> more_crosses TOK_CROSS simple_gen
rule 58  simple_gen -> TOK_IDENT opt_extract TOK_GEN expr opt_elegen_step opt_at
rule 59  simple_gen -> c14 TOK_GEN_SC '[' c12 ']' opt_expr_step
rule 60  simple_gen -> TOK_WINDOW TOK_IDENT '[' c17 ']' TOK_GEN expr opt_expr_step opt_at
rule 61  opt_extract -> /* empty */
rule 62  opt_extract -> '(' colon_twids ')'
rule 63  colon_twids -> colon_twid
rule 64  colon_twids -> colon_twids ',' colon_twid
rule 65  colon_twid -> ':'
rule 66  colon_twid -> '~'
rule 67  opt_elegen_step -> /* empty */
rule 68  opt_elegen_step -> TOK_STEP '(' opt_exprs ')'
rule 69  opt_exprs -> emp_expr
rule 70  opt_exprs -> opt_exprs ',' emp_expr
rule 71  emp_expr -> '_'
rule 72  emp_expr -> expr
rule 73  opt_expr_step -> /* empty */
rule 74  opt_expr_step -> TOK_STEP '(' exprs ')'
rule 75  opt_at -> /* empty */
rule 76  opt_at -> at_spec
rule 77  at_spec -> TOK_AT '(' c14 ')'
rule 78  c12 -> ele2
rule 79  c12 -> c12 ',' ele2
rule 80  ele2 -> expr '~' expr
rule 81  ele2 -> expr
rule 82  c19 -> ele9
rule 83  c19 -> c19 ',' ele9
rule 84  ele9 -> type_w_id
rule 85  ele9 -> TOK_NEXT TOK_IDENT
rule 86  ele9 -> '_'
rule 87  c14 -> ele4
rule 88  c14 -> c14 ',' ele4
rule 89  ele4 -> type_w_id
rule 90  ele4 -> '_'
rule 91  c17 -> expr
rule 92  c17 -> c17 ',' expr
rule 93  ret_exprs -> ret_expr
rule 94  ret_exprs -> ret_exprs ',' ret_expr
rule 95  ret_expr -> TOK_FINAL '(' TOK_IDENT ')'
rule 96  ret_expr -> loop_reduction
rule 97  ret_expr -> struct_op '(' expr ')'
rule 98  ret_expr -> TOK_ACCUM '(' loop_reduction ',' expr ',' expr ')'
rule 99  loop_reduction -> reduce_op '(' expr ')'
rule 100 loop_reduction -> reduce_op '(' expr ',' expr ')'
rule 101 loop_reduction -> reduce_op '(' expr ',' expr ',' expr ')'
rule 102 loop_reduction -> TOK_VAL_AT_MINS '(' expr ',' value_cluster ',' expr ')'
rule 103 loop_reduction -> TOK_VAL_AT_MINS '(' expr ',' value_cluster ')'
rule 104 loop_reduction -> TOK_VAL_AT_MAXS '(' expr ',' value_cluster ',' expr ')'

```

```

rule 105 loop_reduction -> TOK_VAL_AT_MAXS '(' expr ',' value_cluster ')'
rule 106 value_cluster -> '{' exprs '}'
rule 107 arr_reduce_op -> opt_loop_prag TOK_ARR_SUM
rule 108 arr_reduce_op -> opt_loop_prag TOK_ARR_MIN
rule 109 arr_reduce_op -> opt_loop_prag TOK_ARR_MAX
rule 110 arr_reduce_op -> opt_loop_prag TOK_ARR_RED_AND
rule 111 arr_reduce_op -> opt_loop_prag TOK_ARR_RED_OR
rule 112 arr_reduce_op -> opt_loop_prag TOK_ARR_MEDIAN
rule 113 arr_reduce_op -> opt_loop_prag TOK_ARR_MAX_INDICES
rule 114 arr_reduce_op -> opt_loop_prag TOK_ARR_MIN_INDICES
rule 115 arr_reduce_op -> opt_loop_prag TOK_ARR_PRODUCT
rule 116 arr_reduce_op -> opt_loop_prag TOK_ARR_MEAN
rule 117 arr_reduce_op -> opt_loop_prag TOK_ARR_ST_DEV
rule 118 arr_reduce_op -> opt_loop_prag TOK_ARR_MODE
rule 119 arr_reduce_op -> opt_loop_prag TOK_ARR_HIST
rule 120 arr_reduce_op -> opt_loop_prag TOK_ARR_CONCAT
rule 121 reduce_op -> TOK_SUM
rule 122 reduce_op -> TOK_MIN
rule 123 reduce_op -> TOK_MAX
rule 124 reduce_op -> TOK_RED_AND
rule 125 reduce_op -> TOK_RED_OR
rule 126 reduce_op -> TOK_MEDIAN
rule 127 reduce_op -> TOK_PRODUCT
rule 128 reduce_op -> TOK_MEAN
rule 129 reduce_op -> TOK_ST_DEV
rule 130 reduce_op -> TOK_MODE
rule 131 reduce_op -> TOK_HIST
rule 132 struct_op -> TOK_ARRAY
rule 133 struct_op -> TOK_VECTOR
rule 134 struct_op -> TOK_MATRIX
rule 135 struct_op -> TOK_CUBE
rule 136 struct_op -> TOK_CONCAT
rule 137 struct_op -> TOK_TILE
rule 138 stmts -> /* empty */
rule 139 stmts -> stmts stmt
rule 140 stmt -> cl9 '=' exprs ';'
rule 141 stmt -> cl9 '=' TOK_LOOP_INDICES '(' ')' ';'
rule 142 stmt -> cl9 '=' arr_slice ';'
rule 143 stmt -> TOK_PRINT '(' expr ',' pr_entities ')' ';'
rule 144 stmt -> TOK_ASSERT '(' expr ',' pr_entities ')' ';'
rule 145 stmt -> error ';'
rule 146 pr_entities -> pr_entity
rule 147 pr_entities -> pr_entities ',' pr_entity
rule 148 pr_entity -> TOK_IDENT
rule 149 pr_entity -> TOK_STRING
rule 150 exprs -> expr
rule 151 exprs -> exprs ',' expr
rule 152 expr -> TOK_IDENT '(' exprs ')'
rule 153 expr -> TOK_INTRINSIC '(' exprs ')'
rule 154 expr -> TOK_ARR_CONPERIM '(' expr ',' expr ',' expr ')'
rule 155 expr -> TOK_EXTENTS '(' TOK_IDENT ')'
rule 156 expr -> '{' stmt stmts '}' TOK_RETURN '(' exprs ')'
rule 157 expr -> TOK_IDENT '(' ')'

```

```

rule 158  expr -> TOK_IDENT
rule 159  expr -> TOK_IDENT '[' array_indices ']'
rule 160  expr -> TOK_INTNUM
rule 161  expr -> TOK_FLOATNUM
rule 162  expr -> TOK_TRUE
rule 163  expr -> TOK_FALSE
rule 164  expr -> arr_reduce_expr
rule 165  expr -> switch
rule 166  expr -> TOK_ARR_ACCUM '(' arr_reduce_expr ',' expr ',' expr ')'
rule 167  expr -> '(' expr ',' expr ')'
rule 168  expr -> TOK_REAL '(' expr ')'
rule 169  expr -> TOK_IMAG '(' expr ')'
rule 170  expr -> expr '+' expr
rule 171  expr -> expr '-' expr
rule 172  expr -> expr '*' expr
rule 173  expr -> expr '/' expr
rule 174  expr -> expr '%' expr
rule 175  expr -> expr '<' expr
rule 176  expr -> expr '>' expr
rule 177  expr -> expr TOK_LE expr
rule 178  expr -> expr TOK_GE expr
rule 179  expr -> expr TOK_NE expr
rule 180  expr -> expr TOK_EQUAL expr
rule 181  expr -> expr TOK_AND expr
rule 182  expr -> expr TOK_OR expr
rule 183  expr -> expr '&' expr
rule 184  expr -> expr '|' expr
rule 185  expr -> expr '^' expr
rule 186  expr -> expr TOK_LEFT_SHIFT expr
rule 187  expr -> expr TOK_RIGHT_SHIFT expr
rule 188  expr -> '!' expr
rule 189  expr -> '--' expr
rule 190  expr -> '(' type_wo_id ')' expr
rule 191  expr -> '(' expr ')'
rule 192  expr -> for_loop
rule 193  expr -> while_loop
rule 194  expr -> conditional
rule 195  switch -> TOK_SWITCH '(' expr ')' '{' cases opt_default '}'
rule 196  cases -> /* empty */
rule 197  cases -> cases case
rule 198  case -> TOK_CASE const_list ':' body_w_return
rule 199  const_list -> constval
rule 200  const_list -> const_list ',' constval
rule 201  constval -> TOK_INTNUM
rule 202  constval -> '--' TOK_INTNUM
rule 203  opt_default -> /* empty */
rule 204  opt_default -> TOK_DEFAULT ':' body_w_return
rule 205  arr_reduce_expr -> arr_reduce_op '(' expr ')'
rule 206  arr_reduce_expr -> arr_reduce_op '(' expr ',' expr ')'
rule 207  arr_reduce_expr -> arr_reduce_op '(' expr ',' expr ',' expr ')'
rule 208  arr_slice -> '{' '}'
rule 209  arr_slice -> '{' exprs '}'
rule 210  arr_slice -> '{' slices '}'

```

```

rule 211 slices -> arr_slice
rule 212 slices -> slices ',' arr_slice
rule 213 while_loop -> TOK_WHILE '(' expr ')' '{' stmts '}' TOK_RETURN '(' ret_exprs ')',
rule 214 conditional -> TOK_IF '(' expr ')' body_w_return elifs TOK_ELSE body_w_return
rule 215 conditional -> expr '?' expr ':' expr
rule 216 elifs -> /* empty */
rule 217 elifs -> elifs elif
rule 218 elif -> TOK_ELIF '(' expr ')' body_w_return
rule 219 body_w_return -> '{' stmts '}' TOK_RETURN '(' exprs ')',
rule 220 body_w_return -> TOK_RETURN '(' exprs ')',
rule 221 array_indices -> array_index
rule 222 array_indices -> array_indices ',' array_index
rule 223 array_index -> expr
rule 224 array_index -> triple
rule 225 triple -> colon_bounds
rule 226 triple -> colon_bounds ':' expr
rule 227 colon_bounds -> opt_expr ':' opt_expr
rule 228 opt_expr -> /* empty */
rule 229 opt_expr -> expr

```

CAMERON PROJECT: FINAL REPORT

Appendix B: DDCF Manual

The SA-C Compiler Data-Dependence-Control-Flow (DDCF)

J. P. Hammes and A. P. W. Böhm
Colorado State University

Data-Dependence-Control-Flow (DDCF) graphs are used as an intermediate representation in the SA-C compiler, suitable for performing a variety of optimizations. The graphs are acyclic and hierarchical, i.e. some nodes contain subgraphs within them. The entire SA-C language can be represented by DDCF graphs.

1 DDCF Nodes

There are two kinds of DDCF nodes: *simple* nodes are bottom-level, whereas *compound* nodes contain subgraphs. All nodes have input and output *ports* that interface the node to the rest of the graph by means of *edges*. Each input port may have either an incoming edge or a literal value. Each output port has zero or more outgoing edges.

Every port has a SA-C type tag that specifies the data type of the values that travel through the port. When an input port and an output port are connected by an edge, their type tags must match. Some nodes have node-specific information associated with them. For example, a ND_FCALL node will contain the name of the function being called. See section 10 for a complete description of the node-specific information that applies to each node type.

Compound nodes contain ND_G_INPUT, ND_G_INPUT_NEXT and ND_G_OUTPUT nodes (called I/O nodes) that serve as the interface between the node's internals and its exterior. Each of these I/O node types has node-specific information that tells which external port the node is associated with, as well as which compound node it lives in. A compound node “knows” its input and output nodes, via arrays of node identifying numbers, as well as its other internal nodes via a linked list of node identifiers.

Figure 1 shows the conventions used when drawing DDCF nodes. For all nodes, input ports occur along the top of the node, and output ports occur along the bottom. There is an implicit left-to-right ordering of the ports. A *simple* node's interior is shown with its node type and any node-specific information that may apply. A *compound* node contains a subgraph, with I/O nodes represented as small black rectangles along the top and bottom of the compound node, to reduce clutter. A ND_G_INPUT_NEXT is distinguished from a ND_G_INPUT by the fact that there is a dashed implicit edge from a ND_NEXT node back to its associated ND_G_INPUT_NEXT node. An input port can be targeted by an edge or by a constant value. All values pass through the boundaries of a compound node via I/O nodes.

Every port has a SA-C *type* tag (see the TypeInfo structure in appendix D. The tag tells the kind of data, whether it is scalar or array, the rank, and optionally a size for each dimension if it is known. A ‘-1’ value indicates that a dimension's size is not known; this is represented in a SA-C program by a ‘.’ within the declaration's square brackets. When an edge connects an output port to an input port, the type tags of the ports must match, except for the dimension sizes. (This mismatch can occur only when edges from ND_CASE nodes target the same output port. There may be different

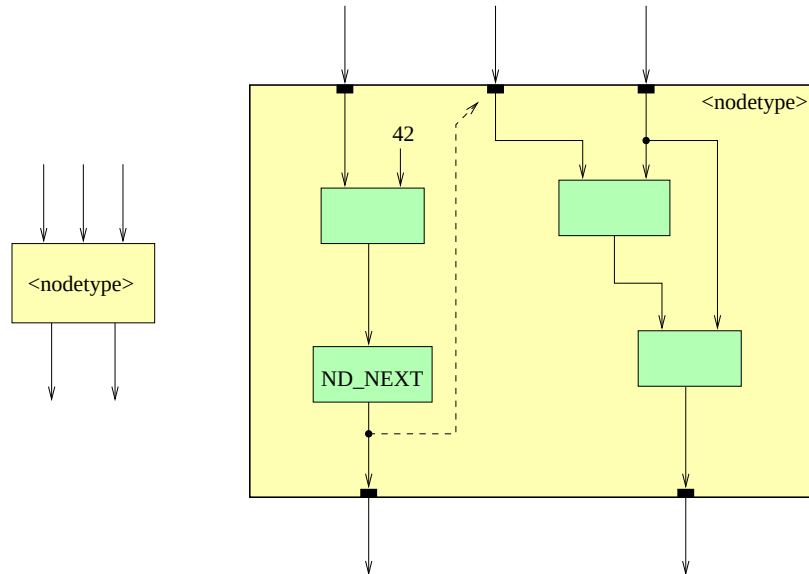


Figure 1: An example of a simple node (left) and compound node (right.) Both have three input ports and two output ports. The compound node contains nine internal nodes (including its I/O nodes), and its middle input is “nextified”.

sizes on the different output ports, and the input port they target will show a ‘-1’ since the size can vary at run time.) Since an edge connects ports with matching types, it is reasonable to show the edge with that type.

2 DDCF Graphs of SA-C Functions

All functions in a SA-C program, both those that are defined and those that are only prototyped, have an entry in the top-level DDCF data structure represented as a linked list of `FuncGraph`s. See appendix D for detailed information.

The `nodes` array is expandable. The `nodes_allocated` field indicates the size of the array, and the `nodes_used` field indicates the number of nodes that are currently used. Node allocations take place through a call to `alloc_ddcf_node`, which receives the function’s `FuncGraph` pointer as its parameter. If all the nodes are used, the routine allocates a larger array, copies the old information into the new array, and deallocates the old array. If the `nodes_used` field equals zero, the function is only a prototype. Otherwise, `nodes[0]` is a `ND_FUNC` node representing the compound node for the function definition.

3 Numeric and Bit Operator Nodes

Figure 2 shows the names and descriptions of the DDCF nodes that perform numeric and bit-manipulation operations. The nodes are simple, with straightforward meanings.

name	inputs	outputs	cmpnd	description
ND_ADD	2	1	N	add
ND_SUB	2	1	N	subtract
ND_MUL	2	1	N	multiply
ND_DIV	2	1	N	divide
ND_MOD	2	1	N	mod
ND_LT	2	1	N	less than
ND_GT	2	1	N	greater than
ND_LE	2	1	N	less than or equal
ND_GE	2	1	N	greater than or equal
ND_EQ	2	1	N	equal
ND_NEQ	2	1	N	not equal
ND_AND	2	1	N	boolean and
ND_OR	2	1	N	boolean or
ND_BIT_AND	2	1	N	bit and
ND_BIT_OR	2	1	N	bit or
ND_BIT_EOR	2	1	N	bit exclusive or
ND_LEFT_SHIFT	2	1	N	bit left shift
ND_RIGHT_SHIFT	2	1	N	bit right shift
ND_COMPLEX	2	1	N	create complex value from two scalars
ND_REAL	1	1	N	take real value from complex
ND_IMAG	1	1	N	take imaginary value from complex
ND_NOT	1	1	N	boolean not
ND_NEG	1	1	N	numeric negate
ND_MUL_MACH	2	1	N	A multiply node whose output width is the sum of its input widths.
ND_LEFT_SHIFT_MACH	2	1	N	A left shift node whose output width is wider than its left input width.
ND_SQRT_MACH	2	1	N	A square root node whose output width is half of its input width.

Figure 2: Numeric and bit-manipulation nodes. The last three are variations that don't correspond to the traditional SA-C rules for determining the width of the output.

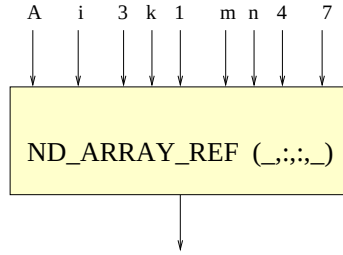
4 Array Nodes

Figure 3 shows the various node types that create and use array values.

name	ins	outs	cmpnd	description
ND_ARRAYREF	var	1	N	This node takes an element or a slice from an array. Node-specific information indicates which dimensions are sliced (via a ':' spec) and which have a specified index. The first input is the source array. In addition, each slice dimension creates three inputs (low, high and step), whereas each specified index creates one input. See text for an example.
ND_ARR_DEF	var	1	N	This node creates a constant array from a collection of scalar values. The number of inputs equals the size of the array. Node-specific information holds the array's shape information.
ND_ARR_CONPERIM	3	1	N	This node creates an array with a constant-value perimeter. The first input is the source array. The second is the perimeter width. The third is the perimeter value.
ND_ARR_CONCAT	2	1	N	This node concatenates two arrays.
ND_ARR_VERT_CONCAT	2	1	N	This node concatenates two 2D arrays in the vertical dimension.
ND_EXTENTS	1	var	N	This returns the extents of the input array. The number of outputs equals the rank of the array.

Figure 3: Array-related nodes

The ND_ARRAYREF node can extract both array elements (scalars) or slices. It has node-specific information (see section 10) that indicates the pattern of extraction, in this example $(\_ , : , : , \_)$. The meaning of the inputs is determined by the extraction pattern. For example, the SA-C expression $A[i, 3:k, m:n, 4, 7]$ will produce the following node:



5 Miscellaneous Nodes

Figure 4 shows various other node types. The ND_FUNC node represents an entire function definition. Its input ports represent the function's parameters, and its output ports represent the function's return values. The ND_FUNC node is *always* the first node in that function's array of nodes.

The ND_PRINT and ND_ASSERT nodes may have string constants as inputs. Since the type tags for input ports represent SA-C types, and SA-C has no strings, type information for these inputs

does not exist.

The ND_VOIDED node occurs where a previous node in the array has been deleted, and it should simply be skipped over whenever nodes are being processed.

name	ins	outs	cmpnd	description
ND_FUNC	var	var	Y	This is the top-level function node. It has an input for each parameter, and an output for each return value.
ND_FCALL	var	var	N	This node calls a user-defined function. It has an input for each argument, and an output for each return value. Node-specific information indicates which function is called.
ND_INTRINCALL	var	var	N	This node calls an intrinsic function. Node-specific information indicates which function is called.
ND_PRINT	var	0	N	This node prints values and strings. The first input is a boolean, determining whether or not the node will print. The rest are SA-C values and strings.
ND_ASSERT	var	0	N	This node tests a SA-C assertion (the first input.) The other inputs are SA-C values and strings.
ND_CAST	1	1	N	This node casts a value from one type to another.
ND_G_INPUT	0	1	N	This is an input node for a compound parent node. Node-specific information indicates the parent node id, and the port number it associates with.
ND_G_OUTPUT	1	0	N	This is an output node for a compound parent node. Node-specific information indicates the parent node id, and the port number it associates with.
ND_G_INPUT_NEXT	0	1	N	This is an input node for a loop parent node. Node-specific information indicates the parent node id, and the port number it associates with. The input is a “nextified” variable of the loop.
ND_VHDL_CALL	var	var	N	This is a call to an external VHDL routine.
ND_VOIDED	0	0	N	This is an empty spot in the nodes array, and should be ignored.

Figure 4: Miscellaneous nodes

6 Switches

The ND_SWITCH node is a compound node corresponding to switch and conditional expressions in SA-C. Figure 5 describes the nodes associated with switches.

Every ND_SWITCH node contains exactly one ND_SWITCH_KEY node that sinks an expression whose value is used to select among the various ND_CASE graphs. One ND_CASE graph may lack a ND_SELECTORS node, and corresponds to a default in the switch expression. Every other ND_CASE graph has exactly one ND_SELECTORS node that sinks the case values, which are always constants. SA-C conditional expressions (if-else) are represented by ND_SWITCH graphs since the conditional is a special case of the more general switch. The ND_G_OUTPUT nodes of a ND_SWITCH graph are the only DDCF nodes in which an input port can be targeted by more than one edge.

As an example, figure 6 shows the DDCF graph produced by the following SA-C expression:

```
switch (n+2) {
```

name	ins	outs	cmpnd	description
ND_SWITCH	var	var	Y	This is the top-level switch node. It has an input for each value used in the switch, and an output for each return value.
ND_SWITCH_KEY	1	0	N	This node is a sink for the switch select expression.
ND_SELECTORS	var	0	N	This node is a sink node for the keys of a ND_CASE node.
ND_CASE	var	var	Y	This node corresponds to a case or default of a switch. The number of inputs is the number of values that are needed in the case. The number of outputs is the number of values returned by the switch expression.

Figure 5: Nodes associated with switch expressions

```

case 3, 4 : return (m*4)
case 5   : return (42)
default  : { uint8 p = n*2+m; } return (p) }

```

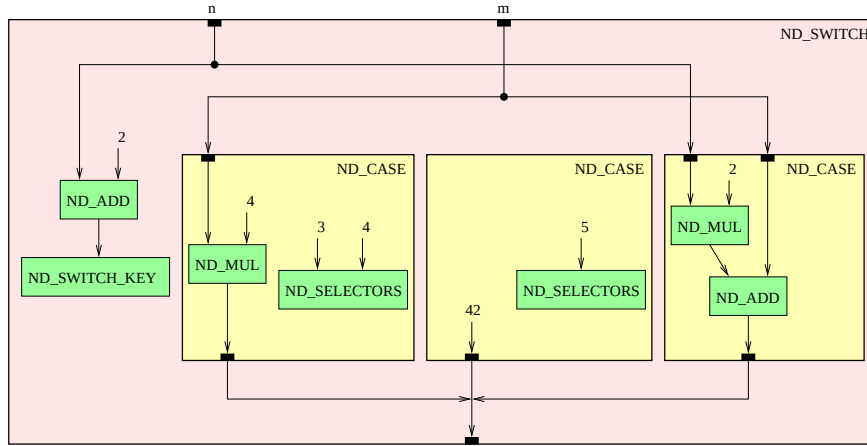


Figure 6: Example of DDCF graph of a switch expression.

7 Loops

Figure 7 shows the top-level loop nodes. There are three loop graph nodes: ND_FORALL, corresponding to a SA-C `for` loop without nextified variables; ND_FORNXT, from a `for` loop with nextified variables; and ND_WHILE, corresponding to a `while` loop. The two `for` loop forms use the loop generator graphs ND_CROSS_PROD and ND_DOT_PROD to produce internal loop values, whereas the ND_WHILE graph uses a ND_WHILE_PRED to enclose the predicate expression that controls the loop. All three loop forms use the same set of loop-return nodes, which are discussed first.

name	ins	outs	cmpnd	description
ND_FORALL	var	var	Y	Parallel for loop.
ND_FORNXT	var	var	Y	A for loop with loop-carried dependencies.
ND_WHILE	var	var	Y	A while loop.
ND_WHILE_PRED	var	0	Y	Graph containing the while loop predicate expression.

Figure 7: Loop nodes

7.1 Loop-return nodes

The various loop-return nodes are all simple nodes. The optional mask that is available in many of the SA-C reductions is mandatory in the DDCF graphs; if the mask was not explicitly specified, a **true** input is placed on the mask input of the reduction node. Figure 8 defines the various loop-return nodes.

7.2 Parallel For Loops

Each SA-C **for** loop has exactly one generator graph, either a ND_CROSS_PROD or a ND_DOT_PROD compound node. (When a loop has only one simple generator, i.e. no explicit dot or cross product, it will be enclosed in either a ND_CROSS_PROD or a ND_DOT_PROD graph.) There are three simple generator nodes that can occur within the compound node: ND_SCALAR_GEN, ND_WINDOW_GEN and ND_SLICE_GEN. In the case of a ND_CROSS_PROD graph, the order of the simple generators is determined by the order in which they occur in the graph’s list of internal nodes. A ND_LOOP_INDICES node will occur in a ND_CROSS_PROD or ND_DOT_PROD graph if the SA-C loop contains a **loop_indices** call. Figure 9 describes the loop generator nodes.

Figure 10 shows an example of each of the three simple generator nodes, as well as cross- and dot-product examples. Figure 11 shows an example of a parallel **for** loop graph.

7.3 For loops with loop-carried dependencies

A **for** loop with at least one “nextified” variable is designated by a ND_FORNXT node, but is otherwise like a parallel **for** loop. Three simple nodes relate to nextified variables: ND_NEXT, ND_FEED_NEXT and ND_G_INPUT_NEXT. The incoming value for a nextified variable flows to the graph through an external ND_FEED_NEXT node, and into the graph through a ND_G_INPUT_NEXT node rather than through a ND_G_INPUT node. A value flowing into a ND_NEXT node is the value that is carried into the next iteration of a loop. It has an implied back edge to the ND_G_INPUT_NEXT node associated with the nextified variable; this back edge is represented as node-specific information for the ND_NEXT node. Figure 12 shows a ND_FORNXT example.

7.4 While Loops

SA-C **while** loops use the same return reductions and array constructors as used by **for** loops. **While** loops have no generators, but rather use a predicate to determine whether to continue iterating. The ND_WHILE_PRED graph encloses the predicate expression. Its result goes to its output port but does not connect to anything externally. Figure 13 shows an example of a **while** loop.

name	ins	outs	cmpnd	description
ND_CONSTRUCT_ARRAY	1	1	N	Array return.
ND_CONSTRUCT_CONCAT	1	1	N	Concat return.
ND_CONSTRUCT_TILE	1	1	N	Tile return.
ND_REDUCE_SUM	2	1	N	Sum reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_PRODUCT	2	1	N	Product reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_MIN	2	1	N	Min reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_MAX	2	1	N	Max reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_AND	2	1	N	And reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_OR	2	1	N	Or reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_MEAN	2	1	N	Mean reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_ST_DEV	2	1	N	Standard deviation reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_MODE	2	1	N	Mode reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_MEDIAN	2	1	N	Median reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_VALS_AT_XXXS	2	1	N	Values-at-mins and Values-at-maxs reductions; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_VAL_AT_FIRST_XXXS	2	1	N	Values-at-first-max and Values-at-first-min reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_VAL_AT_LAST_XXXS	2	1	N	Values-at-last-max and Values-at-last-min reduction; the first input is the reduced value, and the second input is the mask.
ND_REDUCE_HIST	3	1	N	Histogram reduction; the first input is the reduced value, the second input is the mask, and the third input is the range.
ND_ACCUM_SUM	4	1	N	Accumulated sum reduction; the first input is the reduced value, the second is the label, the third is the range, and the fourth is the mask.
ND_ACCUM_PRODUCT	4	1	N	Accumulated product reduction; the first input is the reduced value, the second is the label, the third is the range, and the fourth is the mask.
ND_ACCUM_MIN	4	1	N	Accumulated min reduction; the first input is the reduced value, the second is the label, the third is the range, and the fourth is the mask.
ND_ACCUM_MAX	4	1	N	Accumulated max reduction; the first input is the reduced value, the second is the label, the third is the range, and the fourth is the mask.
ND_ACCUM_AND	4	1	N	Accumulated and reduction; the first input is the reduced value, the second is the label, the third is the range, and the fourth is the mask.
ND_ACCUM_OR	4	1	N	Accumulated or reduction; the first input is the reduced value, the second is the label, the third is the range, and the fourth is the mask.
ND_ACCUM_MEAN	4	1	N	Accumulated mean reduction; the first input is the reduced value, the second is the label, the third is the range, and the fourth is the mask.
ND_ACCUM_ST_DEV	4	1	N	Accumulated standard deviation reduction; the first input is the reduced value, the second is the label, the third is the range, and the fourth is the mask.
ND_ACCUM_MEDIAN	4	1	N	Accumulated median reduction; the first input is the reduced value, the second is the label, the third is the range, and the fourth is the mask.
ND_ACCUM_HIST	5	1	N	Accumulated histogram reduction; the first input is the reduced value, the second is the label, the third is the accum range, the fourth is the hist range, and the fifth is the mask.

Figure 8: Loop-return nodes

name	ins	outs	cmpnd	description
ND_CROSS_PROD	var	var	Y	Array cross product graph.
ND_DOT_PROD	var	var	Y	Array dot product graph.
ND_WINDOW_GEN	var	var	N	Array window generator node. The first input is the source array. Two additional inputs exist for each dimension of the source array, one for the window size and one for the step. The first output is the array window that is generated. One additional output exists for each dimension, representing the array index being produced.
ND_SCALAR_GEN	var	var	N	Array scalar generator node. There are three inputs for each value being generated, representing start value, end value and step. There is one output for each value being produced.
ND_SLICE_GEN	var	var	N	Array slice generator node. The first input is the source array. One additional input exists for each iterative dimension, representing the step size. It is mandatory and set to '1' if the SA-C source did not specify a step size. The first output is the array component that is generated. One additional output exists for each iterative dimension, representing the array index being produced.
ND_LOOP_INDICES	0	var	N	Loop indices node. There is one output for each dimension of the loop.

Figure 9: Loop-generator nodes

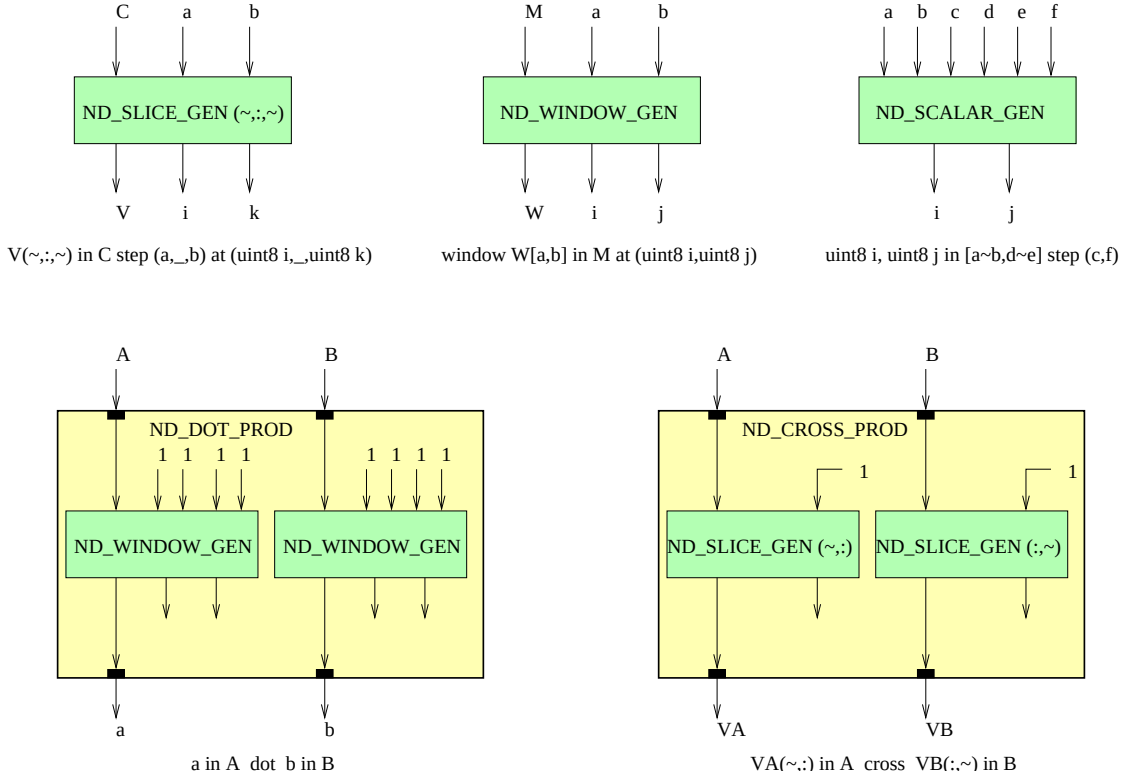


Figure 10: Examples of generator nodes.

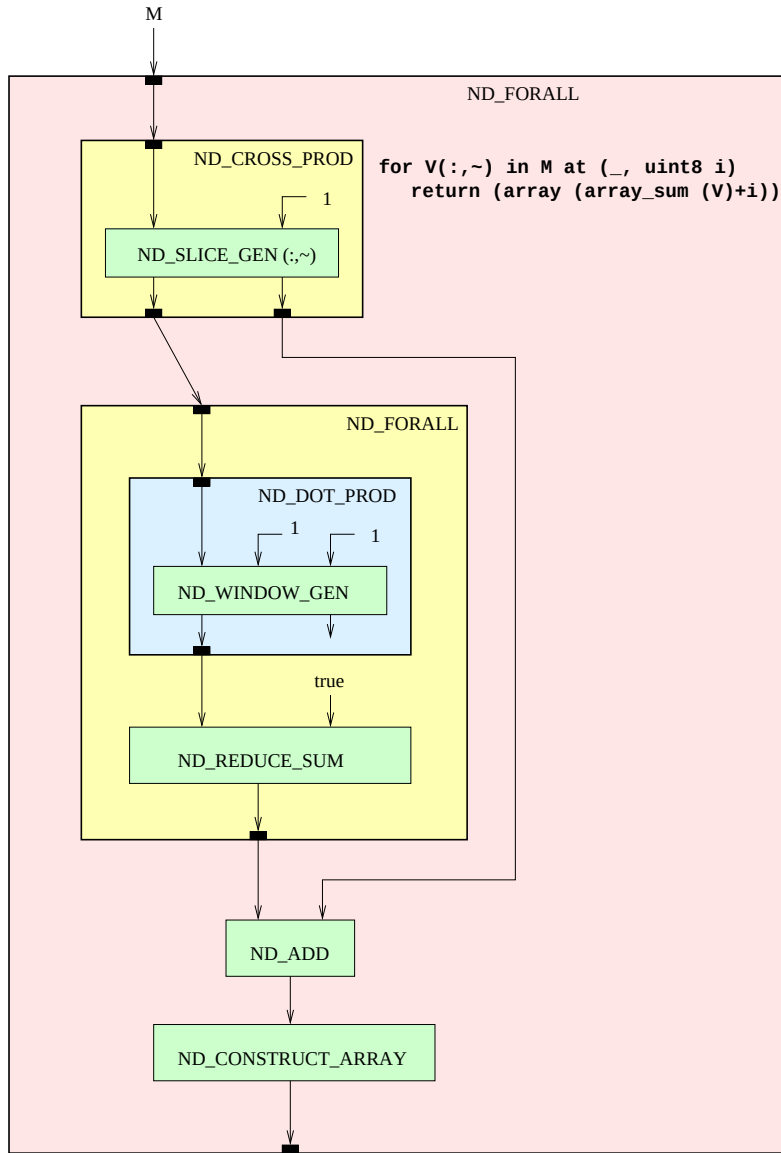
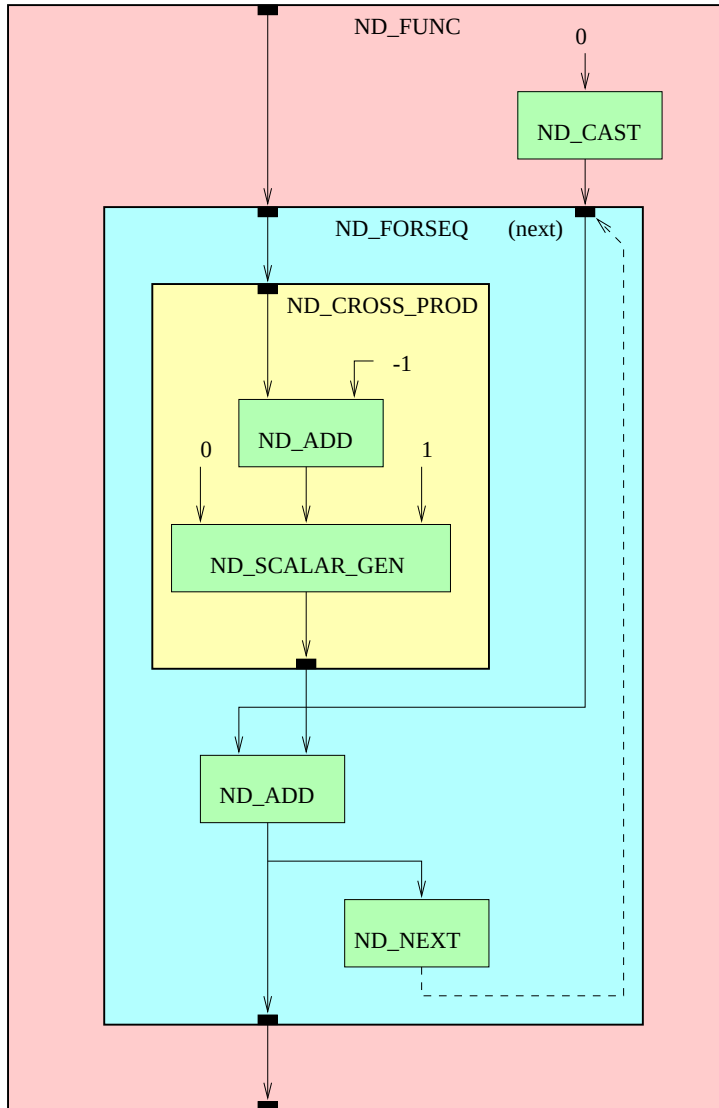


Figure 11: Examples of generator nodes.



```
uint8 main (uint8 n) {
    uint8 ac = 0;
    uint8 res =
        for uint8 i in [n] {
            next ac = ac + i;
        } return (final (ac));
    } return (res);
```

Figure 12: Example of a for loop with a nextified variable.

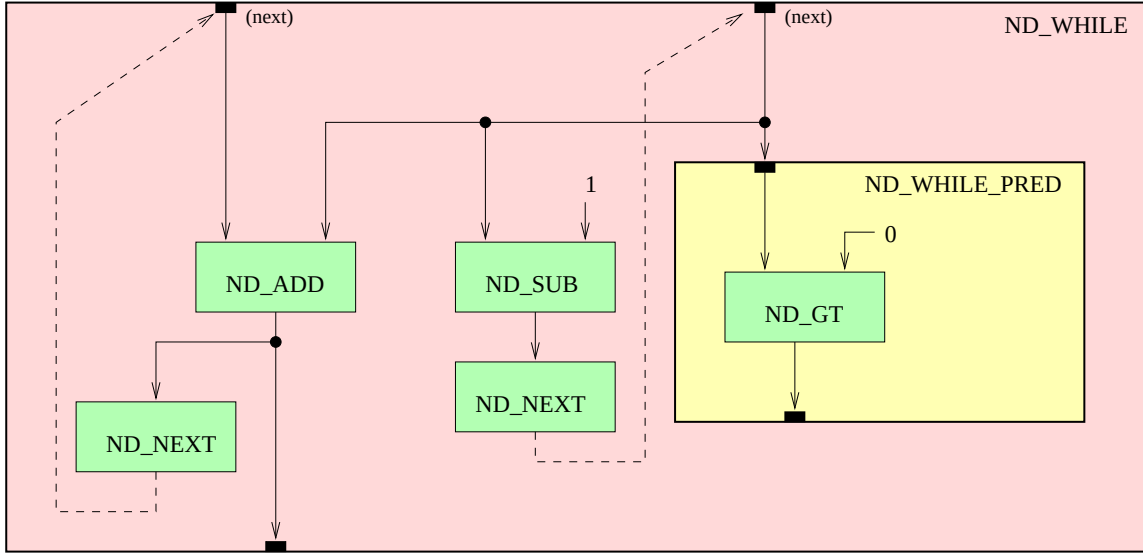


Figure 13: Example of a while loop.

8 MACRO nodes

When the SA-C compiler unrolls a loop, it uses a set of nodes related to the various REDUCE nodes to form the result. These nodes have a variable number of inputs, depending on the number of iterations the unrolled loop had. The following nodes have pairs of inputs, the first conveying a value and the second a boolean indicating whether the value should be included in the reduction:

- ND_REDUCE_SUM_MACRO
- ND_REDUCE_PRODUCT_MACRO
- ND_REDUCE_AND_MACRO
- ND_REDUCE_OR_MACRO
- ND_REDUCE_MIN_MACRO
- ND_REDUCE_MAX_MACRO
- ND_REDUCE_MEDIAN_MACRO
- ND_REDUCE_UMEDIAN_MACRO
- ND_REDUCE_IMEDIAN_MACRO

The six nodes of the VAL_AT_XXX family have input counts based on the number of values being captured, referred to here as v . There will be an input cluster for each iteration of the unrolled loop, and each input cluster will have $v + 2$ inputs, one for the compared value, one for the boolean value, and v for the capture values associated with that iteration. The outputs differ between 1D and 2D return cases. The ND_REDUCE_VAL_AT_MAXS and ND_REDUCE_VAL_AT_MINS each returns 2D a array as a single output. The other four return 1D arrays with known extent, so they have v outputs, one for each of the captured values.

The `ND_REDUCE_HIST_MACRO` node takes a pair of inputs for each iteration of the unrolled loop, and one more input to convey the extent of the returned array. There is one output, which is the result array.

9 DFG-related nodes

When a DDCF loop is being converted to a DFG, a variety of new nodes are introduced.

name	ins	outs	cmpnd	description
<code>ND_RC_COMPUTE</code>	var	var	Y	This graph encloses a DFG loop and its interface code.
<code>ND_MALL_XFER</code>	1	1	N	This node's input takes an input array, and mallocs space for it. The output is the address of the allocated memory.
<code>ND_SIZE</code>	1	1	N	This node takes an input array, and returns the size (i.e. number of elements).
<code>ND_W_LOOP_EXTENT</code>	3	1	N	This node takes extent, size and step as inputs, and returns the number of iterations that result from those values.
<code>ND_MALLOC_TGT_<n>_BIT_<m>D</code>	var	1	N	This node has an input for each dimension of the array it is allocating space for. The rank is <code><m></code> . The inputs are the extents, and the output is the address of the allocated memory. The bit-width of the values is <code><n></code> .
<code>ND_TRANSFER_TO_HOST_RET_ARRAY</code>	var	1	N	This node transfers an array back to the host. Its first input is a trigger from the DFG, the second is the address of the source array, and the rest are the array's extents.
<code>ND_TRANSFER_TO_HOST_SCALAR</code>	2	1	N	This node transfers a scalar back to the host. Its first input is a trigger from the DFG, the second is the address of the scalar.
<code>ND_RC_FORALL</code>	var	var	Y	This is a FOR loop graph that is being turned into a DFG.
<code>ND_RC_WINDOW_GEN_1D</code>	4	var	N	This is a 1D window generator. Its four inputs are an array address, a window size, a window step, and the array's extent. The number of outputs is equal to the window size. The outputs are the scalar values of the window, followed by the index of the window.
<code>ND_RC_WINDOW_GEN_2D</code>	7	var	N	This is a 2D window generator. Its first input is an array address. This is followed by two sets of three inputs, conveying the window size, a window step, and array's extent for each of its two dimensions. The number of outputs is equal to the window size. The outputs are the scalar values of the window, followed by two outputs for the indices of the window.
<code>ND_RC_SLICE_GEN_2D_COL</code>	4	var	N	This is a column slicing generator. Its first input is an array address. The second is the step size. The last two are the input array's extents. The number of outputs is equal to the slice size. The outputs are the scalar values of the slice, followed by an output for the index.
<code>ND_RC_SLICE_GEN_2D_ROW</code>	4	var	N	This is a row slicing generator. Its first input is an array address. The second is the step size. The last two are the input array's extents. The number of outputs is equal to the slice size. The outputs are the scalar values of the slice, followed by an output for the index.
<code>ND_WRITE_TILE_<n>D_<m>D</code>	var	1	N	This node writes a tile to memory, where <code>n</code> is the dimension of the tile and <code>m</code> is the dimension of the loop. The first <code>v</code> inputs are the scalar values of the tile. After that there is one input for the target address of the result array, followed by <code>m</code> inputs for the loop extents, followed by <code>n</code> inputs for the tile extents. The output is a termination trigger.

Figure 14: DFG-related nodes

10 Implementation

Refer to appendix D for the various data structure definitions used to store DDCF graphs. There is a `FuncGraph` for each SA-C function, and each has an array of DDCF nodes. (If the function is just a prototype, the nodes array is `NULL`.) The `DdcfNode` structure represents a node in a DDCF graph, and has the following fields:

nodetype The kind of node it is.

loc File, function, and line number information relating back to the SA-C source.

num_inputs The number of input ports the node has.

inputs Pointer to an array of `InputPort` structures.

num_outputs The number of output ports the node has.

outputs Pointer to an array of `OutputPort` structures.

specific A union containing information that varies with node type. This is discussed in more detail in the following section.

dim_sizes Used only for FOR loop nodes. It contains loop shape information.

10.1 Node-specific information

Node-specific information applies to the following node types:

ND_NEXT An integer tells the node id of the `ND_G_INPUT_NEXT` node that is associated with this node. This is an implicit back edge for this value in the next iteration.

ND_INPUT An integer tells what port number this node associates with in the parent graph. Another integer tells the id of the parent graph.

ND_INPUT_NEXT Same as `ND_INPUT`.

ND_OUTPUT Same as `ND_INPUT`.

ND_FCALL A string tells the name of the called function.

ND_INTRINCALL An `Intrinsic` value specifies the intrinsic function being called.

ND_ARRAYREF A character string specifies a pattern of ‘:’ and ‘_’ that specify sliced and non-sliced dimensions.

ND_SLICE_GEN A character string specifies a pattern of ‘:’ and ‘~’ that specify sliced and iterated dimensions.

ND_SCALAR_GEN An integer tells the rank of the generator.

ND_ARR_DEF An integer tells the rank of the array. An array of integers tells the extent of each dimension.

ND_REDUCE_VAL_AT_MAXS An integer tells the number of values being collected.

ND_REDUCE_VAL_AT_MINS Same as `ND_REDUCE_VAL_AT_MAXS`.

any compound node An integer array tells the node ids of this graph's ND_G.INPUT and ND_G.INPUT_NEXT nodes. Another integer array tells the node ids of this graph's ND_G.OUTPUT nodes. An integer list tells what nodes are within this graph. The order of the nodes in this list is relevant in two ways. First, it represents a valid topological sort of the nodes in the graph, meaning that code generation can be performed simply by stepping through the list rather than following graph edges. Second, for a ND_CROSS_PROD node, the simple generators within the cross product appear in the list in left-to-right order with regard to the SA-C source that produced the loop.

10.2 Text representation

The SA-C compiler's '-ddcf' option will output a text representation of the internal DDCF structures. Appendix E shows a BNF syntax for the text file. Figures 16 and 15 show the text form and its graphic representation for the following SA-C function:

```
uint8, float f1 (uint8 A[:,:])
    return (array_min (A), array_mean ((float[:,:])A));
```

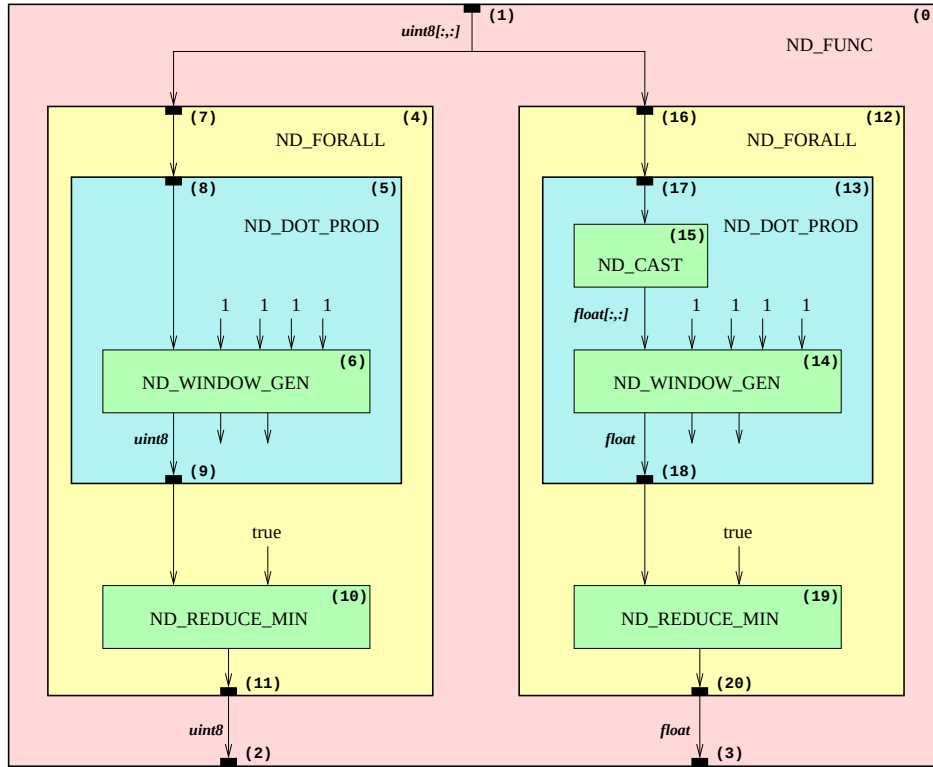


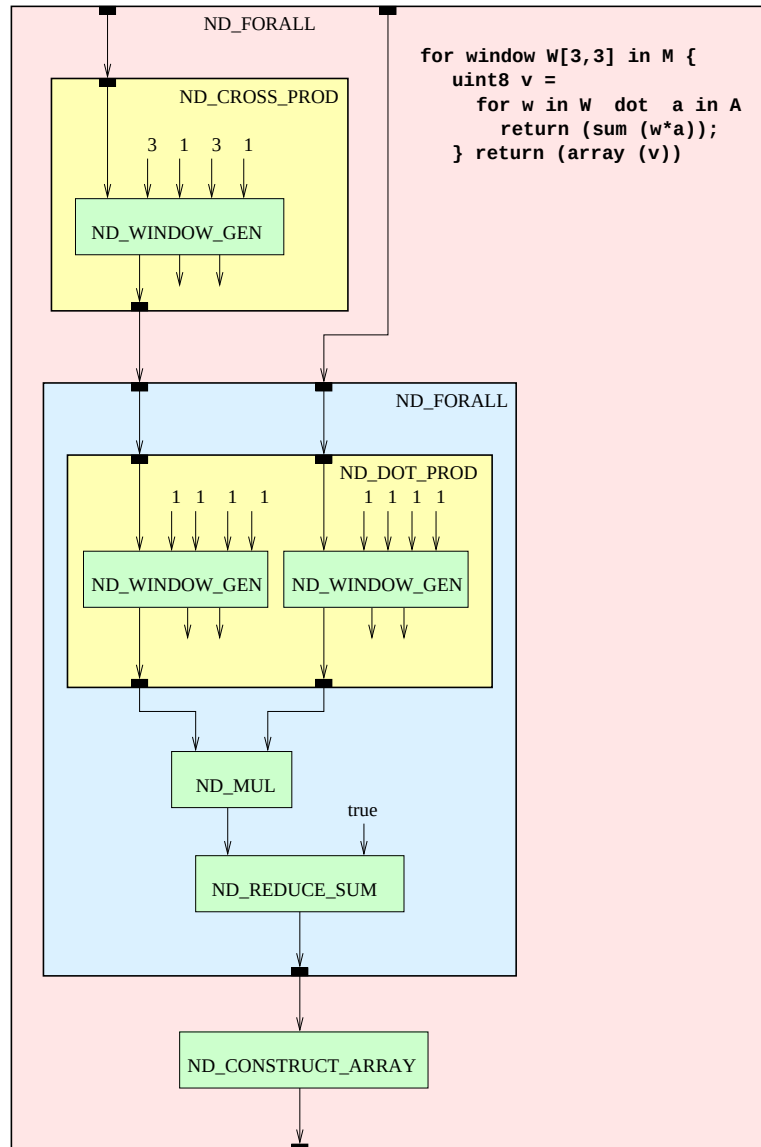
Figure 15: Graphical representation of function example in text.

```

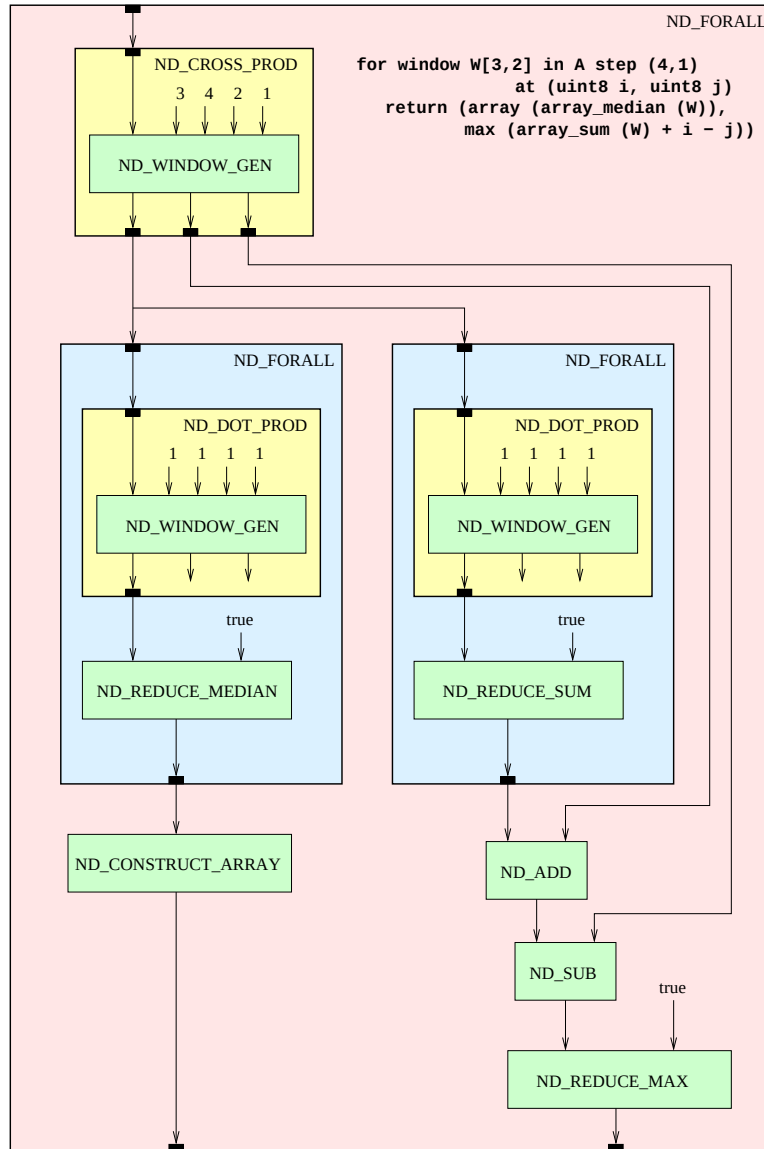
function "f1"
  param types : uint8[:,:]
  return types: uint8, float
  0 ND_FUNC (my nodes: 1 4 12) <"tst.sc", "f1", 1>
    1 inputs: (nodes 1)
      port 0 <uint8[:,>
    2 outputs: (nodes 2 3)
      port 0 <uint8>
      port 1 <float>
    ;
  1 ND_G_INPUT (input 0 for graph node 0) <"tst.sc", "f1", 1>
    0 inputs:
    1 outputs:
      port 0 <uint8[:,> 12.0 4.0
    ;
  2 ND_G_OUTPUT (output 0 for graph node 0) <"tst.sc", "f1", 1>
    1 inputs:
      port 0 <uint8>
    0 outputs:
    ;
  3 ND_G_OUTPUT (output 1 for graph node 0) <"tst.sc", "f1", 1>
    1 inputs:
      port 0 <float>
    0 outputs:
    ;
  4 ND_FORALL (my nodes: 7 5 20 10) extents [:,:,::,:,::,:] dummy_iters 0 <"tst.sc", "f1", 2>
    1 inputs: (nodes 7)
      port 0 <uint8[:,>
    1 outputs: (nodes 11)
      port 0 <uint8> 2.0
    ;
  5 ND_DOT_PROD (my nodes: 8 6) <"tst.sc", "f1", 2>
    1 inputs: (nodes 8)
      port 0 <uint8[:,>
    1 outputs: (nodes 9)
      port 0 <uint8[1,1> 20.0
    ;
  6 ND_WINDOW_GEN left_pad_col 0 <"tst.sc", "f1", 2>
    5 inputs:
      port 0 <uint8[:,>
      port 1 <uint1> value "1"
      port 2 <uint1> value "1"
      port 3 <uint1> value "1"
      port 4 <uint1> value "1"
    3 outputs:
      port 0 <uint8[1,1> 9.0
      port 1 <uint32>
      port 2 <uint32>
    ;
  7 ND_G_INPUT (input 0 for graph node 4) <"tst.sc", "f1", 2>
    0 inputs:
    1 outputs:
      port 0 <uint8[:,> 5.0
    ;
  8 ND_G_INPUT (input 0 for graph node 5) <"tst.sc", "f1", 2>
    0 inputs:
    1 outputs:
      port 0 <uint8[:,> 6.0
    ;
  9 ND_G_OUTPUT (output 0 for graph node 5) <"tst.sc", "f1", 2>
    1 inputs:
      port 0 <uint8[1,1>
    0 outputs:
    ;
  10 ND_REDUCE_MIN <"tst.sc", "f1", 2>
    2 inputs:
      port 0 <uint8>
      port 1 <bool> value "true"
    1 outputs:
      port 0 <uint8> 11.0
    ;
  11 ND_G_OUTPUT (output 0 for graph node 4) <"tst.sc", "f1", 2>
    1 inputs:
      port 0 <uint8>
    0 outputs:
    ;
  12 ND_FORALL (my nodes: 15 13 22 21 18) extents [:,:,::,:,::,:] dummy_iters 0 <"tst.sc", "f1", 2>
    1 inputs: (nodes 15)
      port 0 <uint8[:,>
    1 outputs: (nodes 19)
      port 0 <float> 3.0
    ;
  13 ND_DOT_PROD (my nodes: 16 14) <"tst.sc", "f1", 2>
    1 inputs: (nodes 16)
      port 0 <uint8[:,>
    1 outputs: (nodes 17)
      port 0 <uint8[1,1> 22.0
    ;
  14 ND_WINDOW_GEN left_pad_col 0 <"tst.sc", "f1", 2>
    5 inputs:
      port 0 <uint8[:,>
      port 1 <uint1> value "1"
      port 2 <uint1> value "1"
      port 3 <uint1> value "1"

```

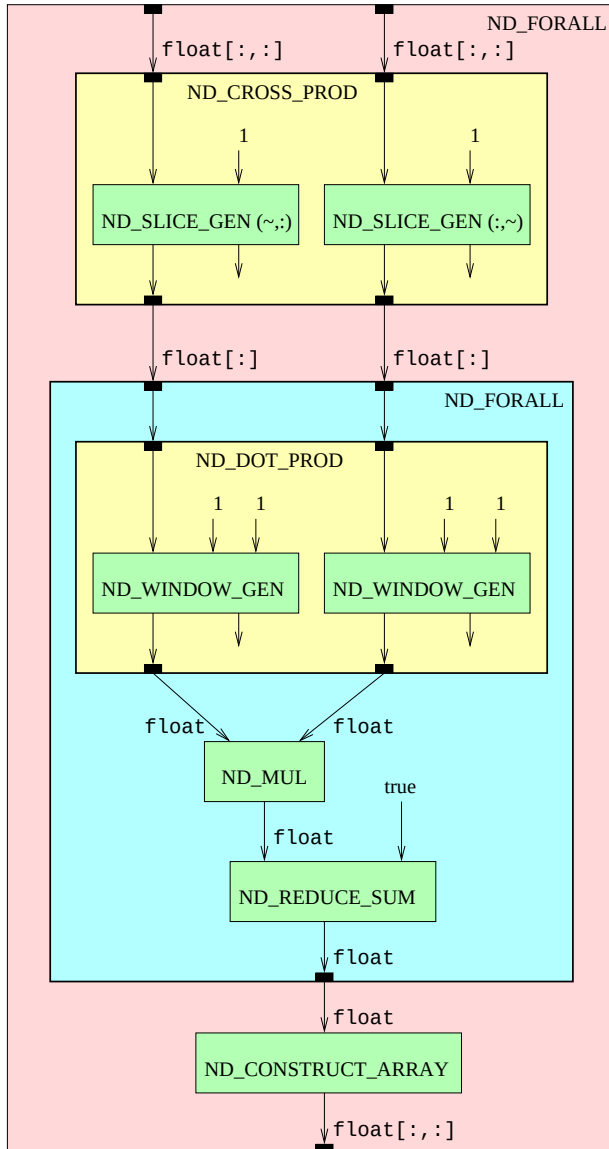
A Example of loop with window generator



B Example of loop with stepped window generator



C Matrix multiply



```

for VA(~,:) in A cross
    VB(:,~) in B {
        float Ele =
            for a in VA dot b in VB
                return (sum (a*b));
            } return (matrix (Ele))

```

D Internal typedefs

```
#include <stdio.h>

extern FILE *yyin;

#define MAXRANK 8
#define FALSE 0
#define TRUE (!FALSE)

typedef struct {
    int line;
    char *file;
    char *func;
} Location;

typedef struct intcell
{
    int val;
    struct intcell *link;
} IntList;

typedef enum {
    Tnone,
    Unknown,      /* type not yet known */
    Bits,         /* bit vector */
    Bool,         /* boolean */
    Uint,         /* unsigned integer */
    Int,          /* signed integer */
    Ufix,         /* unsigned fixed point */
    Fix,          /* signed fixed point */
    Float,        /* 32-bit floating point */
    Double,       /* 64-bit floating point */
    CxInt,        /* complex with signed integer */
    CxFix,        /* complex with signed fixed point */
    CxFloat,      /* complex with 32-bit floats */
    CxDouble      /* complex with 64-bit floats */
} Type;

typedef enum {
    Knone,
    Array,        /* array */
    Scalar,       /* scalar */
} Kind;

typedef struct tinfo
{
    Type type;
    int totsize;
    int fracsize;
    Kind kind;
    int dims;
```

```

    int dim_sizes[MAXRANK];      /* -1 indicates no size available */
    struct tinfo *link;
} TypeInfo;

typedef enum {
    /* compound nodes */
    ND_SWITCH,
    ND_CASE,
    ND_WHILE,
    ND_FORALL,
    ND_FORNXT,
    ND_FUNC,
    ND_CROSS_PROD,
    ND_DOT_PROD,
    ND_WHILE_PRED,

    /* 2-input, 1-output operator nodes */
    ND_ADD,
    ND_SUB,
    ND_MUL,
    ND_DIV,
    ND_MOD,
    ND_COMPLEX,
    ND_LT,
    ND_GT,
    ND_LE,
    ND_GE,
    ND_NEQ,
    ND_EQ,
    ND_AND,
    ND_OR,
    ND_BIT_AND,
    ND_BIT_OR,
    ND_BIT_EOR,
    ND_LEFT_SHIFT,
    ND_RIGHT_SHIFT,

    /* loop-related nodes */
    ND_SCALAR_GEN,
    ND_WINDOW_GEN,
    ND_LOOP_INDICES,
    ND_REDUCE_SUM,
    ND_REDUCE_MIN,
    ND_REDUCE_MAX,
    ND_REDUCE_AND,
    ND_REDUCE_OR,
    ND_REDUCE_VAL_AT_MAXS,
    ND_REDUCE_VAL_AT_MINS,
    ND_REDUCE_PRODUCT,
    ND_REDUCE_MEAN,
    ND_REDUCE_ST_DEV,
    ND_REDUCE_MODE,
    ND_REDUCE_MEDIAN,

```

```

ND_REDUCE_HIST,
ND_CONSTRUCT_ARRAY,
ND_CONSTRUCT_CONCAT,
ND_CONSTRUCT_TILE,

ND_ACCUM_SUM,
ND_ACCUM_MIN,
ND_ACCUM_MAX,
ND_ACCUM_AND,
ND_ACCUM_OR,
ND_ACCUM_PRODUCT,
ND_ACCUM_MEAN,
ND_ACCUM_ST_DEV,
ND_ACCUM_MEDIAN,
ND_ACCUM_HIST,

/* new macro nodes for unrolled loops */
ND_REDUCE_SUM_MACRO,
ND_REDUCE_PRODUCT_MACRO,
ND_REDUCE_AND_MACRO,
ND_REDUCE_OR_MACRO,
ND_REDUCE_MIN_MACRO,
ND_REDUCE_MAX_MACRO,
ND_REDUCE_MEDIAN_MACRO,

/* input and output nodes for compounds */
ND_G_INPUT,
ND_G_INPUT_NEXT,
ND_G_OUTPUT,

/* 1-input, 1-output operator nodes */
ND_REAL,
ND_IMAG,
ND_NOT,
ND_NEG,

/* other various kinds... */
ND_ARR_CONCAT,
ND_ARR_VERT_CONCAT,
ND_CAST,
ND_FCALL,
ND_EXTENTS,
ND_INTRINCALL,
ND_ARR_CONPERIM,
ND_ARRAYREF,
ND_ARR_DEF,
ND_SWITCH_KEY,
ND_SELECTORS,
ND_NEXT,
ND_PRINT,
ND_ASSERT,
ND_VOIDED,
ND_FEED_NEXT,

```

ND_GRAPH,

ND_RC_COMPUTE,
ND_MALL_XFER,
ND_RANK,
ND_SIZE,
ND_W_LOOP_EXTENT,
ND_MALLOC_TGT_8_BIT,
ND_MALLOC_TGT_16_BIT,
ND_MALLOC_TGT_32_BIT,
ND_TRANSFER_TO_HOST_RET_ARRAY,
ND_TRANSFER_TO_HOST_SCALAR,
ND_RC_FORALL,
ND_RC_WINDOW_GEN_1D,
ND_RC_WINDOW_GEN_2D,
ND_WRITE_AND_ADVANCE,
ND_WRITE_TILE_1D_1D,
ND_WRITE_TILE_1D_2D,
ND_WRITE_TILE_1D_3D,
ND_WRITE_TILE_2D_1D,
ND_WRITE_TILE_2D_2D,
ND_WRITE_TILE_2D_3D,
ND_WRITE_TILE_3D_1D,
ND_WRITE_TILE_3D_2D,
ND_WRITE_TILE_3D_3D,
ND_REDUCE_ISUM_MACRO,
ND_REDUCE_USUM_MACRO,
ND_REDUCE_IMIN_MACRO,
ND_REDUCE_UMIN_MACRO,
ND_REDUCE_IMAX_MACRO,
ND_REDUCE_UMAX_MACRO,
ND_USUM_VALUES,
ND_ISUM_VALUES,
ND_UMIN_VALUES,
ND_IMIN_VALUES,
ND_UMAX_VALUES,
ND_IMAX_VALUES,
ND_AND_VALUES,
ND_OR_VALUES,
ND_IADD,
ND_UADD,
ND_ISUB,
ND_USUB,
ND_ULT,
ND_ILT,
ND_ULE,
ND_ILE,
ND_UGT,
ND_IGT,
ND_UGE,
ND_IGE,
ND_UEQ,
ND_IEQ,

```

    ND_UNEQ,
    ND_INEQ,
    ND_BIT_COMPL,
    ND_CHANGE_WIDTH,
    ND_CHANGE_WIDTH_SE,
    ND_RC_SELECTOR
} DdcfType;

typedef enum {
    IntrinSin,
    IntrinCos,
    IntrinTan,
    IntrinAsin,
    IntrinAcos,
    IntrinAtan,
    IntrinAtan2,
    IntrinSinh,
    IntrinCosh,
    IntrinTanh,
    IntrinAsinh,
    IntrinAcosh,
    IntrinAtanh,
    IntrinSqrt,
    IntrinCbrt,
    IntrinPow,
    IntrinModf,
    IntrinExp,
    IntrinFrexp,
    IntrinLdexp,
    IntrinLog,
    IntrinLog10,
    IntrinExpm1,
    IntrinLog1p,
    IntrinCeil,
    IntrinFabs,
    IntrinFloor,
    IntrinFmod,
    IntrinCopysign,
    IntrinHypot,
    IntrinRint
} Intrinsic;

typedef struct edge {
    int node;
    int port;
    struct edge *link;
} Edge;

typedef struct {
    int id;
    TypeInfo ty;
    int is_const;
    char constval[128];

```

```

    Edge *back_edges;
} InputPort;

typedef struct {
    int id;                /* port number for its node */
    int unique_id;         /* unique id within its function */
    TypeInfo ty;
    Edge *targets;
} OutputPort;

#define In_next_id specific.in_next_id
#define Io_num specific.io_info.io_num
#define My_graph specific.io_info.my_graph
#define My_inputs specific.g_info.my_inputs
#define My_outputs specific.g_info.my_outputs
#define My_nodes specific.g_info.my_nodes
#define Sym specific.sym
#define Intrin_func specific.intrin_func
#define Reftypes specific.reftypes
#define DefDims specific.arr_def_info.dims
#define DefRank specific.arr_def_info.rank
#define Rank specific.rank
#define VecSize specific.vecsize

typedef struct {
    DdcfType nodetype;
    Location loc;
    int num_inputs;
    InputPort *inputs;
    int num_outputs;
    OutputPort *outputs;
    union {
        int in_next_id;          /* for ND_NEXT nodes */
        struct {
            int io_num;
            int my_graph;
        } io_info;              /* for ND_G_INPUT and ND_G_OUTPUT nodes */
        char *sym;                /* for ND_FCALL nodes */
        Intrinsic intrin_func;    /* for ND_INTRINCALL nodes */
        char *reftypes;           /* for ND_ARRAYREF and ND_SLICE_GEN nodes */
        int rank;                 /* for ND_SCALAR_GEN nodes */
        struct {
            int rank;
            int *dims;
        } arr_def_info;          /* for ND_ARR_DEF nodes */
        struct {
            int *my_inputs;
            int *my_outputs;
            IntList *my_nodes;
        } g_info;                /* for any compound graph node */
        int vecsize;              /* for ND_REDUCE_VAL_AT_MxxS */
    } specific;
    int dim_sizes[8];             /* to hold extents of loop nodes */

```



```

    } DdcfNode;

typedef struct funcgraph {
    char *name;
    TypeInfo *params;
    TypeInfo *rets;
    DdcfNode *nodes;
    int nodes_used;
    struct funcgraph *link;
} FuncGraph;

typedef struct loop_strings {
    int id;
    DdcfType ty;
    int n;
    int var[8];
    char init[8][256];
    char incr[8][256];
    char terminate[8][256];
    char size[8][256];
    char tosize[256];
    struct loop_strings *link;
} GenStrings;

```

E BNF of DDCF file format

```
rule 1    program -> functions
rule 2    functions -> function
rule 3    functions -> function functions
rule 4    function -> TOK_FUNCTION TOK_STRING params returns nodes
rule 5    params -> TOK_PARAM TOK_TYPES ':' types
rule 6    params -> TOK_PARAM TOK_TYPES ':'
rule 7    returns -> TOK_RETURN TOK_TYPES ':' types
rule 8    types -> type
rule 9    types -> type ',' types
rule 10   type -> scalar_type dims
rule 11   scalar_type -> TOK_UINT
rule 12   scalar_type -> TOK_INT
rule 13   scalar_type -> TOK_UFIX
rule 14   scalar_type -> TOK_FIX
rule 15   scalar_type -> TOK_FLOAT
rule 16   scalar_type -> TOK_DOUBLE
rule 17   scalar_type -> TOK_BITS
rule 18   scalar_type -> TOK_BOOL
rule 19   scalar_type -> TOK_COMPLEX TOK_INT
rule 20   scalar_type -> TOK_COMPLEX TOK_FIX
rule 21   scalar_type -> TOK_COMPLEX TOK_FLOAT
rule 22   scalar_type -> TOK_COMPLEX TOK_DOUBLE
rule 23   dims -> /* empty */
rule 24   dims -> '[' places ']'
rule 25   places -> place
rule 26   places -> place ',' places
rule 27   place -> ':'
rule 28   place -> TOK_UINTNUM
rule 29   nodes -> /* empty */
rule 30   nodes -> nodes node
rule 31   node -> TOK_UINTNUM TOK_NODETYPE special_info loop_extents '<'
rule 32   node -> TOK_UINTNUM TOK_NODETYPE ';'
rule 33   opt_pragms -> /* empty */
rule 34   opt_pragms -> TOK_PRAGMAS '(' prag_list ')'
rule 35   prag_list -> prag
rule 36   prag_list -> prag_list ',' prag
rule 37   prag -> TOK_NO_INLINE
rule 38   prag -> TOK_NO_UNROLL
rule 39   prag -> TOK_NO_FUSE
rule 40   prag -> TOK_LOOKUP
rule 41   prag -> TOK_NO_DFG
rule 42   prag -> TOK_STRIPMINE '(' numlist ')'
rule 43   numlist -> TOK_UINTNUM
rule 44   numlist -> numlist ',' TOK_UINTNUM
rule 45   loop_extents -> /* empty */
rule 46   loop_extents -> TOK_EXTENTS '[' lval ',' lval ',' lval ',' lval
rule 47   lval -> ':'
rule 48   lval -> TOK_UINTNUM
```

```

rule 49  special_info -> /* empty */
rule 50  special_info -> '(' TOK_INPUT TOK_UINTNUM TOK_FOR TOK_GRAPH
        TOK_NODE TOK_UINTNUM ')'
rule 51  special_info -> '(' TOK_OUTPUT TOK_UINTNUM TOK_FOR TOK_GRAPH
        TOK_NODE TOK_UINTNUM ')'
rule 52  special_info -> '(' TOK_MY TOK_NODES ':' numbers ')'
rule 53  special_info -> '[' gen_pattern ']'
rule 54  special_info -> '(' TOK_BACK TOK_TO TOK_NODE TOK_UINTNUM ')'
rule 55  special_info -> TOK_STRING
rule 56  special_info -> TOK_RANK TOK_UINTNUM
rule 57  special_info -> TOK_VEC_SIZE TOK_UINTNUM
rule 58  numbers -> /* empty */
rule 59  numbers -> numbers TOK_UINTNUM
rule 60  gen_pattern -> gen_place
rule 61  gen_pattern -> gen_place ',' gen_pattern
rule 62  gen_place -> '~'
rule 63  gen_place -> ':'
rule 64  gen_place -> '_'
rule 65  gen_place -> TOK_UINTNUM
rule 66  inputs -> TOK_UINTNUM TOK_INPUTS ':' opt_identify input_ports
rule 67  input_ports -> /* empty */
rule 68  input_ports -> input_ports input_port
rule 69  input_port -> port_spec opt_value
rule 70  port_spec -> TOK_PORT TOK_UINTNUM '<' type '>'
rule 71  port_spec -> TOK_PORT TOK_UINTNUM '<' '>'
rule 72  opt_value -> /* empty */
rule 73  opt_value -> TOK_VALUE TOK_UINTNUM
rule 74  opt_value -> TOK_VALUE TOK_STRING
rule 75  opt_value -> TOK_VALUE TOK_TRUE
rule 76  opt_value -> TOK_VALUE TOK_FALSE
rule 77  outputs -> TOK_UINTNUM TOK_OUTPUTS ':' opt_identify output_ports
rule 78  output_ports -> /* empty */
rule 79  output_ports -> output_ports output_port
rule 80  output_port -> port_spec targets
rule 81  targets -> /* empty */
rule 82  targets -> targets target
rule 83  target -> TOK_UINTNUM '.' TOK_UINTNUM
rule 84  opt_identify -> /* empty */
rule 85  opt_identify -> '(' TOK_NODES numbers ')'

```

CAMERON PROJECT: FINAL REPORT

Appendix C: Compiler Manual

The SA-C Compiler – Version June, 2001

J. P. Hammes, Monica Chawathe and A. P. W. Böhm
Colorado State University

This document describes the use of the SA-C compiler and its related software. The language, SA-C, is described separately.

1 Overview of SA-C compilation system

A pure SA-C compiler lives at the heart of the compilation system. It is supplemented by a generic transformer, macro preprocessor, dataflow simulator and run-time system. All of these are coordinated by a Perl script called **scc** that handles the various user-controlled options, calls the various components in the proper order, and manipulates temporary intermediate files.

2 Using the SA-C compilation system

To use the SA-C compiler and dataflow simulator, the environment variable **SASSYHOME** must be set with the path to the root directory of the SA-C compiler. To run the compiler, execute

`$SASSYHOME/scc`

Most users will set the environment variable in a `.cshrc` file, and create an alias for **scc**. The compiler flags can be viewed by executing **scc** with no arguments, and consist of the following:

-G	Compile from generic *.sg files to *.sc files (Not present at all)
-C	Compile to *.c files.
-c	Compile to *.o files.
-ddcf	Compile to *.ddcf files.
-dfg	Where possible, compile loops to *.dfg files. The rest of the code compiles to *.c files. This option also invokes optimizations. If any of the hardware flags (-aha, -vhdl, -edf, -x86, -b86, -bo or -bout) are also present, it will override the -dfg flag and produce an executable for running on FPGAs or AHA simulator. Otherwise, dataflow graphs will be generated and the executable will invoke the dataflow graph simulator.

Table 1: Compiler Goal Flags

-aha	Compile the dataflow graphs into abstract hardware architecture (AHA) graphs. If a -vhdl, -edf, -x86, -b86, -bo, -bout flag is also present, it will override the -aha flag and produce an executable for running on FPGAs. Otherwise, AHA graphs will be generated and the executable will invoke the AHA simulator.
-vhdl	Produce *.vhd files for AHA graphs that were generated. The host code that is produced will be compiled so as to invoke FPGA execution of converted loops.
-edf	Produce *.xnf files from the VHDL files that were generated. The host code that is produced will be compiled so as to invoke FPGA execution of converted loops.
-x86	Produce *.x86 files from the *.edf files that were generated. The host code that is produced will be compiled so as to invoke FPGA execution of converted loops.
-b86	Produce *.b86 files (*.x86 files bundled with tag information)
-bo	Produce *.bo file (*.o file bundled with corresponding *.b86 files)
-bout	Produce *.bout file (a.out file bundled with corresponding *.b86 files)
-ws	Compile hardware-related files for the WildStar board (The default is to compile for the StarFire board).
-vsim	Generate files used in ModelSim simulation.
-LAD <freq>	Set the LAD frequency to <freq>. (Only 33MHz and 66MHz allowed, with the default being 33MHz).
-mf	Generate and use a makefile for generating *.x86 files instead of the 'scc' script.

Table 2: Hardware Related Flags

-O	Turn on DDCF graph level optimizations.
-fs	Turn on final-srunching.
-pipe <num>	Pipeline the AHA graph with a maximum of <num> pipeline stages.
-ndv	Turn off use of dope vectors.
-nbw	Turn off bitwidth narrowing.
-extv	Print generator and loop information after each loop related optimization.

Table 3: Optimization Flags

-nbc	Turn off array bounds checking.
-nchk	Turn off DDCF consistency checks.
-dfgr	Print why loops did not get converted into DFGs.

Table 4: Check Flags

-D <xx>[=<yy>]	Pass a #define to the 'cpp' preprocessor.
-U <xx>	Pass a #undef to the 'cpp' preprocessor.
-I <path>	Add <path> to the list of directories searched by 'cpp' for #include files.
-L <path>	Add <path> to the list of directories searched by 'cpp' for library files.
-l <lib>	Add <lib> when linking.
-gmp	Turn on support of 32 bits and link the gmp library.
-heap	Produce code to generate a heap trace.
-gddb	Produce code to generate a 'gdb' debug trace.

Table 5: C, cpp, Linker Flags

-o <filename>	Use the specified <filename> to name the target file.
-host	Do not compile any hardware related files (.aha, .edf, .x86, etc.), only recompile the C code generated.
-buf <sz>	Set the buffer size for 'buffered concats' to <sz>.
-m	The main function name is equal to the SA-C file name, without the .sc extension. This is useful in the Khoros workspace compiler environment.
-q	Suppress error and optimization reports from the compiler. This is useful for automated test scripts. The compiler's system return value indicates whether the compile produced any errors.
-k	Keep the intermediate files produced by 'scc' script.
-v	Print the commands called by the scc script before executing them.
-n	Print the commands called by the scc script but do not execute them. (You just print some stuff, not the actual commands)

Table 6: Miscellaneous Flags

The compiler, see figure 1, takes one or more source files as arguments; each can be generic SA-C (.sg), SA-C (.sc), DDCF graph (.ddcf), C code (.c), or object code (.o). The file suffix determines the entrance point in the compile process. The -c, -G, -ddcf, -C, -dfg, -aha, -vhdl, -edf, -x86, -b86, -bo and -bout options, described below, determine the stopping points of the compile process. Some examples show how the process works:

- “scc -ddcf file1.sg file2.sc” compiles the files to DDCF graphs.
- “scc -C file1.sc file2.ddcf” compiles a SA-C file and a DDCF file to C files.
- “scc file1.c file2.c file3.ddcf file4.o -o run” compiles the four files to an executable called run.
- “scc -dfg file.sc” compiles the SA-C file to an executable with dataflow simulation of the *.dfg files that are produced.
- “scc -x86 file.sc” compiles the SA-C file to an executable with reconfigurable hardware execution of the loops that were translated to *.x86 files.

Note that all arguments following the -cflags: argument are passed to the C compiler, so these must be specified at the end of the command line.

3 Running a SA-C Program

When a SA-C program is compiled to an executable and run, its input data become the arguments to the “main” function, and that function's return values become the program's output. This I/O can be handled in two ways: via `stdin` and `stdout`, or by specified data files. Both approaches can use ASCII formats but the data file specification can also use binary file formats. The ASCII file format is intended to be used for test purposes only, and the input will be read sequentially and completely before program execution starts.

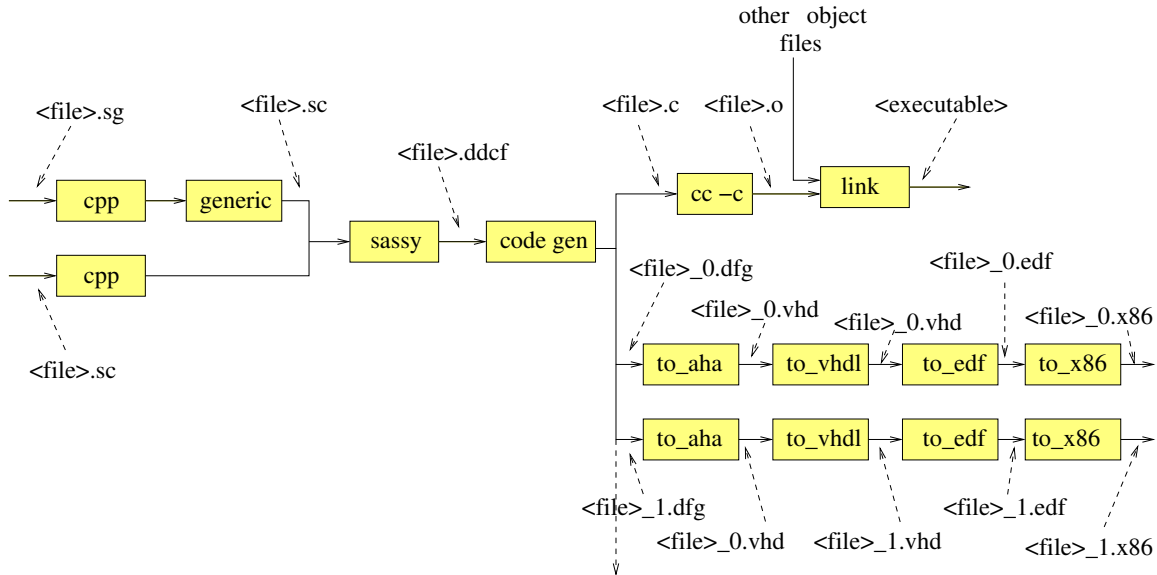


Figure 1: Compilation routes of “scc”.

The ASCII (text) representations of program input are similar, but not identical, to the representations of values within SA-C source code. The differences occur with boolean values and array specifications. First, boolean values in input files may use the letters ‘T’, ‘t’, ‘F’, and ‘f’ to signify “true” and “false” values. Second, array expressions in program input must be preceded by their extents. For example:

```
[2,5] {{3, 7, 9, 8, 4},{1, 3, 2, 4, 4}}
```

3.1 I/O using stdin and stdout

The default approach to I/O is to use `stdin` and `stdout` for input and output. Of course, redirection of either or both of these allows the use of source and target files through the user’s shell. For example, let the file containing a matrix multiply executable be called “mm”, and let the file `mm.dt` contain:

```
[3,2] { {2,1}, {3,4}, {6,2}}
[2,5] { {2,5,1,9,3}, {5,3,7,4,2} }
```

Then, `mm <mm.dt` will produce

```
[3,5]
{ { 9, 13, 9, 22, 8}
  , { 26, 27, 31, 43, 17}
  , { 22, 36, 20, 62, 22}
}
```

on standard output.

3.2 I/O via specified files

A second approach to I/O specifies source and target information as arguments to the executable; each input parameter and each output value gets its own specification, typically a file name.

Input parameters on the command line have the following format:

```
-i<specifier><number> operand
```

The number (zero-based as in C) specifies a main function input parameter. If the specifier is 'f', the operand specifies a file name. If the specifier is 'v', the operand is a value, denoted as it would be in an input file. (More complex input values may need to be surrounded by quotes in certain shells or operating systems.) Each input file can be either ASCII or binary.

Output parameters on the command line have the following format:

```
-of<kind><number> filename
```

The "kind" specifier is either 'a' for ASCII or 'b' for binary.

Continuing the matrix multiply example above:

```
mm -if0 mmA.dt -if1 mmB.dt -ofa0 mm.res
```

would take its inputs from the files `mmA.dt` and `mmB.dt` and put its result, in ASCII format, in the file `mm.res`.

3.3 Binary file format

The SA-C binary file format is designed to be compatible with *pgm* and *ppm* image files. The additional information needed by the SA-C run-time system is put in a comment line. Figure 2 shows an example of a binary file with its text preamble. The SA-C system pays attention only to its special commented line and the block of binary data. The necessary comment line contains:

- the string "SA-C"
- a format indicator, currently "format1"
- a SA-C type, with all extents specified; it will be one or two tokens, depending on whether or not the type is complex
- an endian indicator, 'E' for big-endian and 'e' for little-endian

This comment line can appear anywhere among the other comment lines of the file. It is important to note that SA-C's extents occur in reverse order as compared to the two dimensions that are specified to *pgm*.

In the case of a P5 or P6 file, the SA-C information line may be omitted; data type `uint8` will be assumed. This makes it possible to read and write *pgm* and *ppm* files without modification.

```

P5
# this is a comment line
# SA-C format1 uint8[8,15] E
# this is another comment line
15 8
255
< binary data >

```

Figure 2: An example showing binary file format. Note that the SA-C dimensions are in opposite order to those that occur outside of the comment area.

3.4 Run-Time Options

The run-time options consist of the following flags:

-time	To print number of clock-cycles required for execution.
-v	To print the function name at start and end of each host-interface functions.
-view	To run the DFG simulator with a viewer.
-debug	To print host to simulator/ hardware communication in 'hosttrace' file.
-trace	To print data generated in the simulation of graphs at every clock cycle (useful in hardware debugging).
-vsim	To print data-files used in ModelSim Simulation.
-oc	To overclock the board (at 25 MHz) in case the frequency from place and route is under 25MHz.

Table 7: Run-Time Options

4 Tools

The tag information can be grabbed from the bundled files (*.b86, *.bo, *.bout) using the utility tool `$SASSYHOME/bundler/bndlinfo`. The tag contains hardware occupancy and frequency given by both Synplify and Place and Route tools along with time required by each phase during compilation. It also contains the scc command line used to generate the bundled file.

5 Generic SA-C Code

Given SA-C's size- and precision-parameterized type system and the need to create many versions of programs with related types (e.g. an Image Processing routine manipulating matrices of **uint4**, **uint8**, **uint16**, or **uint24** data) it is useful to be able to generate these various SA-C programs from one source. The *cpr* macro facility could be used for this, but it has drawbacks: A C macro is one line, perhaps with backslashes used for layout. Error messages refer to invocations of the one line macro, not to the macro's source text. These problems make the use of the C preprocessor unattractive for producing multiple versions of a SA-C function, so a generic text expansion mechanism has been provided in the SA-C compilation system. Generic SA-C code has the following form:

```

$generic
  $param1='text11' , $param2'text12' ...$paramN='text1N' ;
  $param1='text21' , $param2'text22' ...$paramN='text2N' ;
  ....
  $param1='textM1' , $param2'textM2' ...$paramN='textMN' ;

$in

  sassy code with $param1 ... $paramN

$end_generic

```

The keyword **\$generic** starts a generic function definition, **\$end_generic** ends it. Each line between **\$generic** and **\$in** declares the values of a set of macro parameters. For each line the SA-C text between **\$in** and **\$end_generic** will be expanded. These generic constructs cannot be nested. Error messages refer to the file containing the generic SA-C code, the function that causes the error, and the line number in the generic SA-C file where the error occurs. Generic SA-C files have a suffix “.sg”.

Here are some examples of generic SA-C code. The generic code appears on the left, and its SA-C equivalent on the right. First, a straightforward definition of one generic function:

<pre> /* Generic Code */ \$generic \$saxpy='saxpy8' , \$ti='uint4',\$tr='uint8'; \$saxpy='saxpy12' , \$ti='uint6',\$tr='uint12'; \$saxpy='saxpy16' , \$ti='uint8',\$tr='uint16'; \$in \$tr \$saxpy (\$ti x, \$ti Y[:]) { \$tr r = for y in Y return(sum(x*y)); } return(r); \$end_generic </pre>	<pre> /* SA-C Code */ uint8 saxpy8 (uint4 x, uint4 Y[:]) { uint8 r = for y in Y return(sum(x*y)); } return(r); uint12 saxpy12 (uint6 x, uint6 Y[:]) { uint12 r = for y in Y return(sum(x*y)); } return(r); uint16 saxpy16 (uint8 x, uint8 Y[:]) { uint16 r = for y in Y return(sum(x*y)); } return(r); </pre>

Not only types can be generic, any piece of SA-C text can be. For example,

<pre> /* Generic Code */ \$generic \$or= 'boolor' , \$t='bool' , \$op=' ' ; \$or= 'uint1or' , \$t='uint1' , \$op=' ' ; \$in \$t \$or (\$t x, \$t y) return(x \$op y); \$end_generic </pre>	<pre> /* SA-C Code */ bool boolor (bool x, bool y) return(x y); uint1 uint1or (uint1 x, uint1 y) return(x y); </pre>

6 SA-C Parallel Construct

In SA-C, there exists an implicit parallelism in the loop-body. To make full use of multiple chips and boards, one may want to run multiple SA-C loops in parallel. One needs to explicitly specify the loops that need to run in parallel. This can be expressed using the parallel construct.

```
uint8 R1[:], uint8 R2[:], uint8 R3[:] = parallel {  
    // PRAGMA (hardware ... )  
    for .....  
    // PRAGMA (hardware ... )  
    for .....  
};
```

One can specify which hardware to use for each loop using the hardware pragma (explained in detail in the Section 7.2). Each parallel loop can return multiple values. The ordering of the return values of the construct is implicitly obtained from the ordering of the loops present within the construct.

7 Optimizations, Pragas and Internal graphs

In the SA-C compiler, optimizations are triggered by any of the following compiler options: **-O**, **-dfg**, **-vhdl**, **-edf**, **-x86**, **-b86**, **-bo** and **-bout**. The optimizations can be further controlled by pragmas in the SA-C source code. Certain loops can be compiled to dataflow graphs (DFGs) for execution by a simulator or for compilation to VHDL (via AHA graphs), as described below.

7.1 Optimizations

A variety of optimizations are performed at DDCF graph level. (Compilation with **-O -ddcf** options will write DDCF files of the optimized graphs.) Optimization passes consist of the following:

- *Invariant Code Motion* hoists invariant code out of loop bodies.
- *Push Array Casts* can, under certain conditions, push array casts to the places in which the array is being referenced, removing the need to do an array copy.
- *Constant Switch* takes a switch graph with a constant key value and replaces the graph with the case graph corresponding to the key.
- *Size Propagation* analyzes loops and arrays, propagating extents information through the graph in order to enable full loop unrolling.
- *Array Value Propagation* replaces an array reference with its corresponding expression when possible.
- *Constant Folding* replaces an operator with a value if the operator's inputs are constants.
- *Identities* applies various algebraic identities and other rules. For example, $a * 1$ is replaced with a , and $a * 0$ is replaced by zero.
- *Strength Reduction* replaces certain operators with simpler equivalents. For example, multiplying an unsigned int by a power of two is replaced by a left shift.

- *Dead Code Elimination* removes code if its values are not used in producing the output results.
- *Full Loop Unrolling* replaces a loop with multiple explicit loop bodies if the number of loop iterations is statically known.
- *Canonify Macros* orders the input edges in increasing order of the source nodes and source ports (to allow CSE of part of the macro).
- *Common Subexpression Elimination (CSE)* eliminates redundant nodes if they compute the same value. This is done by looking for DDCF nodes with the same operation and the same inputs.
- *Extent Elision* replaces extents node (being fed by an array reference) by extent calculation logic, if possible.
- *Duped Inputs* removes the inputs being fed into a loop but not used in it.
- *Concat Masked* converts a concat or concat-masked node being fed by array definitions with a concat-masked-many node and removes the array definitions.
- *Window Narrow* decreases size of the window (and also image) in the innermost dimension if the pixels are not being used and increases the step accordingly. This lessens the FPGA space needed to hold the window values.
- *Function Inlining* replaces a function call with the function's body.
- *Lookup Tables* replaces function calls with references to a lookup table that holds precomputed values.
- *Nextified Array Size Propagation* analyzes nextified arrays, propagating extents information through the graph in order to enable converting the array into nextified scalars.
- *Temporal CSE* looks for values computed in one loop iteration that were computed in earlier iterations (only in the innermost dimension) and replaces the redundant computation by a chain of registers (of compile-time fixed length). This leaves some window elements (in the innermost dimension) unreferenced thus allowing window narrowing.
- *Scrunch* pushes computation of expressions fed by window elements into earlier iterations by moving the window references (in the innermost dimension) and using a register chain to move the result. This also creates a window that can be narrowed.
- *Array Reference Elision* replaces two array references, if the first takes a slice and it feeds only the second, with a single reference.
- *Convert Nextified Array* convert a nextified array of known size into nextified scalars.
- *Loop Fusion* can, under certain circumstances, fuse two consecutive loops into one loop.
- *N-Dimensional Blocking* stripmines a loop by enclosing it in another loop that reads chunks of data with which to feed the inner loop.
- *N-Dimensional Stripmining* stripmines a loop by enclosing it in another loop that reads chunks of data with which to feed the inner loop. The inner loop will subsequently be unrolled, yielding the effect of partial unrolling in multiple dimensions. It is parameterized by the outer-loop window size.

- *N-Dimensional Partial Unroll* stripmines a loop by enclosing it in another loop that reads chunks of data with which to feed the inner loop. The inner loop will subsequently be unrolled, yielding the effect of partial unrolling in multiple dimensions. It is parameterized by the inner-loop iterations.
- *CSE Macro* identifies macro nodes being fed by some identical inputs and replaces eliminates partial redundant computations.
- *Final Scrunch* is similar to scrunch except that it scrunches all computations into a window of size one in the innermost dimension.
- *Pipeline* inserts registers across AHA graphs to minimize the critical path delay.

Optimizations interact with each other. A sequence of a subset of optimizations forms *cyclic opts*, with the sequence repeated until stability is reached (i.e. a pass occurs in which the DDCF is not changed). The cyclic sequence is

1. Code Motion
2. Push Array Casts
3. Constant Switch
4. Size Propagation
5. Array Value Propagation
6. Constant Folding
7. Identities
8. Strength Reduction
9. Dead Code Elimination
10. Canonify Macros
11. CSE
12. Extent Elision
13. Duped Inputs
14. Concat Mask
15. Window Narrow

The full optimization path is as follows:

1. Function Inlining (skipping lookup functions)
2. Lookup Tables
3. Function Inlining (inlining loopup functions)
4. Nextified Array Size Propagation
5. Temporal CSE

6. Scrunch
7. Array Reference Elision
8. *cyclic opts*
9. Convert Nextified Array
10. *cyclic opts*
11. Loop Fusion
12. Temporal CSE
13. Scrunch
14. *cyclic opts*
15. Array Reference Elision
16. *cyclic opts*
17. N-Dimensional Blocking
18. N-Dimensional Stripmining
19. N-Dimensional Partial Unroll
20. Array Reference Elision
21. *cyclic opts*
22. CSE Macro
23. Final Scrunch
24. *cyclic opts*

This path is arranged as it is because of certain interactions among optimizations. The first of the Function Inlining passes (step 1) skips the inlining of functions designated as lookup tables, but it does inline function calls that occur in the function body of the lookup table. The next pass (step 2) converts the designated functions into lookup tables, followed by inlining of these functions (step 3) to their call locations. The loop fusion (step 11), temporal CSE (step 12), scrunch (step 13), array reference elision (step 15) and the cyclic optimizations (steps 14 and 16) are performed repeatedly until no more loops can fuse.

7.2 Pragmas

The compiler flags **-O**, **-dfg**, **-aha**, **-vhdl**, **-edf**, **-x86**, **-b86**, **-bo** and **-boutenable** optimizations; pragmas in SA-C source code allow control of individual optimizations (except that final scrunch is controlled by **-fs** flag). Some pragmas are associated with loops, others with functions. In either case the pragma appears immediately before the loop or function it is controlling. For example,

```
// PRAGMA (no_unroll, no_dfg)
for ...
```

will prevent the **for** loop from being unrolled or converted to a dataflow graph. As this example shows, multiple pragmas may be listed, and their order does not matter. For a parameterized pragma, the n-dimensional parameter is specified in paranthesis (with each dimension parameter separated by comma). For example,

```
// PRAGMA (part_unroll(2,3))
for ...
```

will create an unrolled inner-loop corresponding to 2*3 iterations.

The pragmas are as follows:

1. **no_inline** indicates that a function should not be inlined. The default action is to inline all functions.
2. **no_unroll** indicates that a loop should not be unrolled. The default action is to fully unroll a loop where possible.
3. **nextify_cse** indicates that temporal CSE optimization must be done over the loop
4. **scrunch** indicates that scrunch optimization must be done over the loop;
5. **no_dfg** indicates that a loop should not be turned into a dataflow graph (DFG) for execution by the simulator. The default action is to turn a loop into a DFG if possible.
6. **lookup** indicates that a function should be converted to a lookup table which uses the physical memory of the hardware for implementation.
7. **rom** indicates that a function should be converted to a lookup table which uses the ROM on the hardware for its implementation.
8. **no_fuse** indicates that the loop should not be fused with another loop that is being fed by its output. The default action is to fuse loops wherever possible.
9. **stripmine** is a parameterized pragma that causes a loop to undergo N-Dimensional Stripmining. The inner loop subsequently will be fully unrolled. The parameters indicate the size of window used in the new outer loop.
10. **block** is a parameterized pragma that causes a loop to undergo N-Dimensional Stripmining for array blocking. The inner loop will not be unrolled. The parameters indicate the size of window used in the new outer loop.
11. **part_unroll** is a parameterized pragma that causes a loop to undergo N-Dimensional Stripmining. The inner loop subsequently will be fully unrolled. The parameters indicate the iterations of the inner-loop.
12. **hardware** is a parameterized loop pragma that is used to specify hardware related details (board, chip, memory use) for the loop. For example, the pragma for a loop with two inputs A and B and a single return can be :

```
// PRAGMA (hardware(board : wildstar chip : 1 memin : A 1 B 2 memout : 3))
for ...
```

The input memory is specified as a list of variable name and memory number pair while the output memories are specified as a list of memory numbers. The output memories are associated implicitly with the return values of the loop (by virtue of ordering).

13. **vhdl** is attached to a SA-C function prototype, and indicates that the function call will be passed through the compiler and into the dataflow graph, to be filled in externally with an outside VHDL routine.

The optimization order described previously will determine exactly what is done. For example, if a loop can be fully unrolled and it is given a **stripmine** pragma, the loop will be unrolled (since unrolling occurs earlier than stripmining) and the pragma will be lost since the loop no longer exists.

The Loop Fusion optimization fuses loops that have a producer-consumer relationship, i.e. the output of the “upper loop” feeds the input of the “lower loop”. In the transformation, a new outer loop is created, enclosing the two loops. The lower loop is then melted away, leaving only its body. The new outer loop is given the pragmas of the upper loop, and the upper loop loses all of its pragmas. The lower loop’s pragmas are lost, since it no longer exists as a loop.

When N-Dimensional Stripmining takes place, a new outer loop is created, enclosing the original loop. Loops are also created to handle the “fringes” that may be left over since the window generator of the new outer loop has a non-unit step. The fringe loops are given **no_dfg** pragmas to assure that their computations, probably small, do not produce DFGs. The new outer loop is given copies of the original loop’s pragmas, but with the **stripmine** pragma removed. (The **stripmine** pragma is also removed from the original loop.)

N-Dimensional Blocking is similar to stripmining, but the inner loop is given a **no_unroll** pragma by the compiler. Thus the outer loop will run on the host, and the inner loop will become a dataflow graph that computes with chunks of the source array.

7.3 Producing Dataflow Graphs (DFGs)

The **-dfg** compiler option causes the compiler to convert inner loops to dataflow graphs wherever it can. (The option also causes optimizations to occur, the same as if **-dfg -O** had been specified.) These DFGs will go into files with a **.dfg** suffix. When executing the SA-C program, these DFGs will be executed by a simulator whose behavior somewhat matches the execution on reconfigurable hardware.

7.4 Producing Abstract Hardware Architecture Graphs (AHAs)

The **-aha** compiler option causes the compiler to convert inner loops to dataflow graphs wherever it can and then convert the DFGs into abstract hardware architecture graphs. (The option also causes optimizations to occur, the same as if **-aha -O** had been specified.) These AHAs will go into files with a **.aha** suffix. When executing the SA-C program, these AHAs will be executed by a simulator whose behavior matches the execution on reconfigurable hardware (except for memory contention).

8 Installation

The SA-C compiler sources are contained in a tarred and compressed file called **sassy.june_xx_01.tar.Z**. When uncompressed and untarred, this will produce a SA-C home directory with subdirectories for various things such as the compiler, the run-time system, and test codes. The first installation step is to create the SA-C home directory and move the **sassy.june_xx_01.tar.Z** file into it.

Once the source file is in place, the following command can be used to build the directories of sources:

```
> zcat sassy.june_xx_01.tar.Z | tar xvf -
```

The next step is to compile the sources. While in the SA-C home directory, type:

```
> make
```

This will descend into the various subdirectories and compile their contents.

Since the SA-C compilation system is controlled by Perl scripts, two items must be dealt with. First, the `scc` script in the SA-C home directory must be told the location of the `cpp` macro preprocessor on the host system. This is done by editing the line:

```
$cpp = "/usr/local/cpp";
```

Second, the location of `perl` can vary from system to system. The first line of all the Perl scripts must hold the correct path to `perl`, and will have to be altered if `perl` does not live at `/usr/local/bin/perl`. The Perl scripts in the SA-C compiler are:

- `<SA-C home>/scc`
- `<SA-C home>/test/sassy_test`
- `<SA-C home>/test/aha_test/sassy_test`
- `<SA-C home>/test/examples/sassy_test`
- `<SA-C home>/test/lib_test/sassy_test`
- `<SA-C home>/test/mix_test/sassy_test`
- `<SA-C home>/test/separate_comp/sassy_test`
- `<SA-C home>/test/sim_test/sassy_test`
- `<SA-C home>/test/test_IO/sassy_test`

Once the system has been compiled and the Perl scripts altered, the compiler can be tested. First, while still in the SA-C home directory, execute:

```
> setenv SASSYHOME $PWD
```

Then move into the `test` directory and run the test script:

```
> cd test
> ./sassy_test
```

The script will descend into the subdirectories and run the compiler tests they contain.

Once the compiler has been shown to work correctly, refer to the material in section 2 to see how a user should use the system.

CAMERON PROJECT: FINAL REPORT

Appendix D: SA-C Compiler Dataflow Description

The SA-C Compiler Dataflow Description

J. P. Hammes, R. E. Rinker, D. M. McClure, A. P. W. Böhm, W. A. Najjar
Colorado State University

1 Dataflow Graphs

Dataflow graphs are used internally by the SA-C compiler as a program representation that precedes the VHDL form and can be executed using a token-driven semantics. They represent a low-level, non-hierarchical and asynchronous program. The graphs take into account sequencing but not timing; however, the node functions have been created in such a way as to allow reasonably straightforward translations into the synchronous circuits that finally will be mapped onto reconfigurable hardware.

The functional elements of dataflow graphs are *nodes*, each of which has a node-type, one or more input ports, and one or more output ports. The nodes of a dataflow graph are connected by directed edges, each of which connects an output port to an input port. An output port can connect to any number (including zero) of input ports, but an input port can be fed by only one output port.

When a dataflow graph executes, nodes are “fired” and data are communicated via *tokens*. Every token is created by the output port of a dataflow node. (Data are brought in from outside the dataflow graph through special INPUT nodes.) Each token represents a single k -bit untyped entity, where k is specified by the output port of its source. Each node-type has a firing rule that specifies its behavior when it fires. When a node produces a value at one of its output ports, a token of that value is sent to every input port that is fed by that output. Every input port has a queue of unbounded size. The order of the tokens at an input is the same as the order in which they were produced – i.e., edges behave as queues.

A node may fire only when its firing rule is met. If multiple nodes are ready to fire, they may fire in any order or concurrently. The behavior of most nodes is very simple: it fires only if each of its inputs sees a token. The act of firing consumes one token from each input, and the value of its output is a function of only those inputs. If an input needs a constant value, it is fed by the constant rather than by an output port. The constant input may be viewed as producing a token of that value whenever such a token is needed to fire its node. Tokens have no type, but each has a size as a number-of-bits. The bit-size of an output must match the sizes of each input it feeds. However, the size of a node’s input does *not* have to match the other inputs or the outputs of its node. The exact behavior of a node whose inputs and outputs are of various bit-sizes depends on its node-type.

2 Dataflow node descriptions

SA-C dataflow node functions can be classified into six kinds: Arithmetic, Bit, Selector, Generator, Reduction and I/O nodes. Arithmetic nodes perform common functions such as addition, comparison, and logical operations. Bit nodes perform shifts, sub-word selections, and width changing operations. Selector nodes involve choosing one of a number of inputs to pass to the output. Generator nodes take inputs and use them to specify sequences of output tokens. Reduction nodes take token sequences and reduce or store them. I/O nodes handle the interface between the dataflow graph and the outside world. Bit positions for an n -bit value are specified as b_0 (least significant bit) through b_{n-1} (most significant bit). A node has input ports designated I_0 through I_{p-1} , and output ports O_0 through O_{q-1} , where p is its number of inputs and q is its number of outputs. Some node types have a fixed number of inputs, but for some node types the number of inputs and/or outputs can vary.

2.1 Arithmetic nodes

Table 1 shows the names and descriptions of the dataflow nodes that perform arithmetic and bit-manipulation operations. While the tokens themselves have no type, the Arithmetic nodes implicitly treat their inputs as either signed or unsigned integers. The signed values are represented in twos-complement form.

The Arithmetic nodes have straightforward characteristics and semantics:

1. A node can fire only when a token is present on every one of its inputs.
2. The act of firing consumes exactly one token from each input.
3. Every node has exactly one output.
4. The sizes of a node's inputs are unified by extending the precisions of its inputs to match the largest input size. The new bits are placed to the left of the incoming bits. For unsigned values, the new bits are zeroes. For the signed operators, the value is *sign-extended*, that is the new bits are the same as the most-significant bit of the incoming token.
5. After the result value has been computed, the value's size is adjusted to match the specified size of the output port. If the size is reduced, the appropriate number of left-most bits are truncated. If the size is increased, the new bits are zeroes for operators with unsigned outputs, and sign-extended for operators with signed outputs.

The comparison nodes produce a zero if false, and a one if true; the output can be specified to be any bit-size, but only the least significant bit can be non-zero.

2.2 Multi-input Arithmetic nodes

Multi-input arithmetic nodes exist for some operators. They allow an arbitrary number of input values, each with an associated boolean mask value. Various implementations may

name	inputs	description
UADD	2	unsigned addition of values from I_0 and I_1
IADD	2	signed addition of values from I_0 and I_1
USUB	2	unsigned subtraction of values from I_0 and I_1
ISUB	2	signed subtraction of values from I_0 and I_1
NEGATE	1	negate the signed value from I_0
ULT	2	unsigned $<$ comparison of values from I_0 and I_1
ILT	2	signed $<$ comparison of values from I_0 and I_1
ULE	2	unsigned $\leq$ comparison of values from I_0 and I_1
ILE	2	signed $\leq$ comparison of values from I_0 and I_1
UGT	2	unsigned $>$ comparison of values from I_0 and I_1
IGT	2	signed $>$ comparison of values from I_0 and I_1
UGE	2	unsigned $\geq$ comparison of values from I_0 and I_1
IGE	2	signed $\geq$ comparison of values from I_0 and I_1
UEQ	2	unsigned equality comparison
IEQ	2	signed equality comparison
UNE	2	unsigned inequality comparison
INE	2	signed inequality comparison
BIT-AND	2	bit-wise AND of values from I_0 and I_1
BIT-OR	2	bit-wise OR of values from I_0 and I_1
BIT-EOR	2	bit-wise exclusive OR of values from I_0 and I_1
BIT-COMPL	1	bit-wise complement of value from I_0

Table 1: Arithmetic nodes

choose the order in which the values are reduced. The Multi-input arithmetic nodes are shown in Table 2.

2.3 Multi-input Comparison Nodes

Table 3 lists the multi-input comparison nodes. These nodes have input ports organized in clusters of $n + 2$ where n is the number of output values/ports. The first input port in each cluster is a comparison value, the next n ports are *captured* values, and the last input port in each cluster is a boolean mask. For each cluster with a boolean mask of true, it's comparison value is compared with all other comparison values to find the [*first* or *last*] [*max* or *min*] of all comparison values (depending on type of node). The output of the node is the n captured values associated with the selected comparison value.

An example may help illustrate the functionality of these nodes. Figure 1 shows an example VAL-AT-FIRST-UMAX-MANY node. For this example there are $n = 2$ captured values per comparison value, so the input ports appear in clusters of $n + 2 = 4$. Since this is a VAL-AT-FIRST-UMAX-MANY node, all comparison values with a boolean mask of true will be compared to find the leftmost maximum among them. The output of the node is the two captured values associated with the selected comparison value.

name	inputs	description
USUM-MANY	var	sum the unsigned input values; each input pair represents a value and a boolean mask; the arithmetic is performed at a bit width equal to the output port's bit width
ISUM-MANY	var	sum the signed input values; each input pair represents a value and a boolean mask; the arithmetic is performed at a bit width equal to the output port's bit width
UMIN-MANY	var	'min' the unsigned input values; each input pair represents a value and a boolean mask; the arithmetic is performed at a bit width equal to the output port's bit width
IMIN-MANY	var	'min' the signed input values; each input pair represents a value and a boolean mask; the arithmetic is performed at a bit width equal to the output port's bit width
UMAX-MANY	var	'max' the unsigned input values; each input pair represents a value and a boolean mask; the arithmetic is performed at a bit width equal to the output port's bit width
IMAX-MANY	var	'max' the signed input values; each input pair represents a value and a boolean mask; the arithmetic is performed at a bit width equal to the output port's bit width
UMEDIAN-MANY	var	find the median of the unsigned input values; each input pair represents a value and a boolean mask
IMEDIAN-MANY	var	find the median of the signed input values; each input pair represents a value and a boolean mask
AND-MANY	var	'and' the input values; each input pair represents a value and a boolean mask
OR-MANY	var	'or' the input values; each input pair represents a value and a boolean mask

Table 2: Multi-input Arithmetic nodes

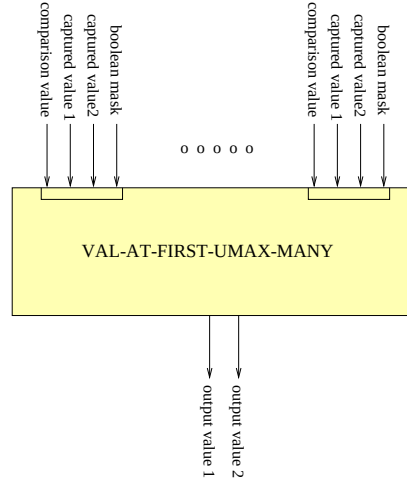


Figure 1: Example Multi-Input Comparison node (a VAL-AT-FIRST-UMAX-MANY node with n=2)

2.4 Bit nodes

The Bit nodes perform shifts, concatenates, sub-word selects and size-changing operations. Table 4 describes the Bit nodes, which have the following characteristics and semantics:

name	inputs	description
VAL-AT-FIRST-IMAX-MANY	var	return the signed <i>captured</i> value or values at the first max among many <i>comparison</i> values (see Figure 1). Input ports are organized in clusters of $n + 2$ where n is the number of <i>captured</i> values per cluster (equal to the number of output ports/values). Input port I_0 in each cluster is the comparison value; ports I_1 to I_n in each cluster are the <i>captured</i> values; port I_{n+1} in each cluster is a boolean mask. Output ports O_0 to O_{n-1} carry the captured values at the first max comparison value with a boolean mask of true.
VAL-AT-FIRST-UMAX-MANY	var	the same as VAL-AT-FIRST-IMAX-MANY but with un-signed comparison and captured values.
VAL-AT-FIRST-IMIN-MANY	var	the same as VAL-AT-FIRST-IMAX-MANY, but the output ports carry the captured values at the first min comparison value with a boolean mask of true.
VAL-AT-FIRST-UMIN-MANY	var	the same as VAL-AT-FIRST-IMIN-MANY, but with unsigned comparison and captured values.
VAL-AT-LAST-IMAX-MANY	var	the same as VAL-AT-FIRST-IMAX-MANY, but the output ports carry the captured values at the last max comparison value with a boolean mask of true.
VAL-AT-LAST-UMAX-MANY	var	the same as VAL-AT-LAST-IMAX-MANY, but with unsigned comparison and captured values.
VAL-AT-LAST-IMIN-MANY	var	the same as VAL-AT-FIRST-IMIN-MANY, but the output ports carry the captured values at the last min comparison value with a boolean mask of true.
VAL-AT-LAST-UMIN-MANY	var	the same as VAL-AT-LAST-IMIN-MANY but with unsigned comparison and captured values.

Table 3: Multi-input Comparison nodes

1. A node can fire only when a token is present on every one of its inputs.
2. The act of firing consumes exactly one token from each input.
3. Every node has exactly one output.
4. After the result value has been computed, the value's size is adjusted to match the specified size of the output port. If the size is reduced, the appropriate number of leftmost bits are truncated. For the CHANGE-WIDTH-SE node, if the size is increased, the most significant (leftmost) bit is sign-extended. For all other nodes, the size is increased by padding zeroes to the left of the value.

2.5 Selector node

Currently, there is only one selector node, called SELECTOR, that corresponds to the switch of a high-level language. Its first input, I_0 , takes the key value that is used to choose among the input values. Each case value uses two consecutive inputs, the first as the match value and the second as the chosen expression. The default expression is the last input, and it must be present. The bit width of every key compare input is guaranteed to equal the width of key input (I_0), so an absolute compare of the bit strings can be made without the

name	inputs	description
L-SHIFT	2	left-shift the value from I_0 by the distance from I_1 ; zeroes enter from the right
R-SHIFT	2	right-shift the value from I_0 by the distance from I_1 ; zeroes enter from the left
BIT-SELECT	3	I_0 and I_1 specify a range of bits to be taken from the value on I_2
BIT-CONCAT	2	I_0 and I_1 are concatenated to form a result whose width is the sum of the two input widths
CHANGE-WIDTH	2	change the bit-width of the value on I_0 to the <i>constant</i> size on I_1
CHANGE-WIDTH-SE	2	change the bit-width of the value on I_0 to the <i>constant</i> size on I_1 by sign-extending it if the size is increased

Table 4: Bit nodes

need for considering sign extension. Also, the bit width of every value input is guaranteed to equal the bit width of the output. The following node corresponds to the SA-C switch expression (assume that “key” is a `int7`):

```
switch (key) {
    case -1 : return (E0)
    case 3,-7 : return (E1)
    case 2 : return (E2)
    default : return (E3) }
```

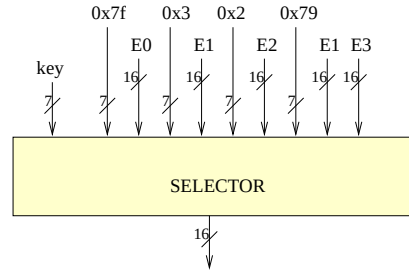


Figure 2: Selector node example.

Except for the default expression, which must be the last input, the order of the `<value,expression>` pairs does not matter. Duplicate case values are not allowed.

2.6 Generator nodes

Generator nodes produce streams of scalar tokens. There are four basic types of generator nodes:

1. Scalar generators produce sequences of integers.

2. Window generators produce streams of values drawn from specified windows of a source array read from memory.
3. Slice generators produce streams of values drawn from specified slices of a source array read from memory.

Slice generators use only 2D arrays as source arrays; the other generator nodes can use 1-, 2- or 3-D source arrays, with the potential to use arrays of up to 8-D if that functionality should be needed. The various generator nodes are listed in Table 5.¹

An example 2-D window generator node is shown in Figure 3. Input port 0 carries the start address of the source array from which windows of values will be read. The input ports then come in clusters of three for each dimension of the window – with one port each carrying the window extent, step size, and source array extent in that dimension. This is followed by $(dims - 1)$ ports carrying the image offsets, where $dims$ is the number of dimensions of the window (2-D for the example in Figure 3). The last input port carries the the number of dummy iterations. There are $(num\_els + dims + 2)$ output ports, where num_els is the number of elements in a single window. Output ports 0 to $(num\_els - 1)$ carry the values read from the current window in the source array memory, the next two ports carry the indices of that window in the source array, while the last two ports carry the *dummy* and *done* signals, indicating whether the current iteration is a dummy and when all iteration have been executed respectively.

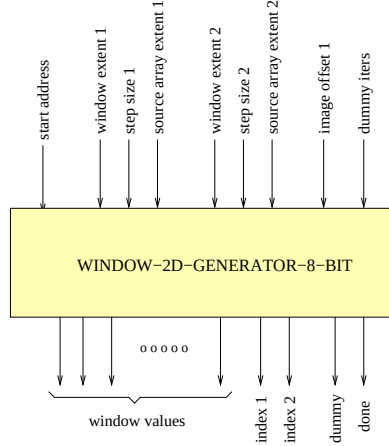


Figure 3: Generator node example.

¹In several of the tables that follow, entire families of nodes are represented by using node names with one or more parameter placeholders. For the WINDOW GENERATOR nodes in Table 5, for example, β represents bit-size (which can take on values of 1, 2, 4, 8, 16, or 32), and δ represents dimensionality of the source array (which can take on values of 1, 2 or 3). Consequently, WINDOW- δ D-GENERATOR- β -BIT represents a family of $3 \times 6 = 18$ related nodes. As this node-naming convention is used in the tables that follow, each parameter's meaning and its possible values will be specified.

name	ins	outs	description
SCALAR-GENERATOR	$3\delta+1$	$\delta+2$	generate δ streams of integer tokens for a generator with <i>rank</i> = δ ($\delta = 1, 2$ or 3); the δ input triples carry the start value, end value, and step size used to generate the output stream, and the last input port carries the dummy iterations; output ports O_0 to $O_{\delta-1}$ carry the output stream, and the last two output ports have the dummy and done signal.
WINDOW- δ D-GENERATOR- β -BIT	$4\delta+2$	$n+\delta+2$	generate n streams of scalar tokens, for the n elements in that part of a source array currently under a moving δ -dimensional window ($\delta = 1$ or 2 or 3), those elements to be read from memory in β -bit units ($\beta = 1, 2, 4, 8, 16$ or 32); input port I_0 carries the starting address of the source array, the next δ input triples carry the window size, step size, and array extent in one of the window's δ dimensions, the next $(\delta - 1)$ input ports carry the image offsets, last two input ports carry image size in words and dummy iterations; output ports O_0 to O_{n-1} carry the scalar elements of that part of the source array currently under the moving window, the next δ output ports carry indices of the top-left element of the current window in the δ -dimensional array, the last two output ports have the dummy and done signal.
SLICE-2D-GENERATOR-ROW- β -BIT	7	$n + 3$	generates n streams of scalar tokens for the n β -bit elements in a specified row slice of a 2D array read from memory. Input I_0 carries the start address of the source array, I_1 carries the step size between slices to be generated, I_2 and I_3 carry the extents of the 2D source array, I_4 carries the image offset, while I_5 and I_6 carry image size in words and dummy iterations; outputs O_0 to O_{n-1} carry the values of the array elements in the current slice, O_n carries the index of the current row slice in the 2D source array, and the last two output ports have the dummy and done signal.
SLICE-2D-GENERATOR-COL- β -BIT	7	$n + 3$	same as SLICE-2D-GENERATOR-ROW- β -BIT but the slice elements are from a column (rather than a row);

Table 5: Generator nodes

2.7 Array Reference nodes

Array reference nodes return the value of a single array element. The input ports of an ARRAYREF node carry the information needed to locate an array element in memory (start address, array extents, and indices within the array of the element being referenced); while the single output port carries the value of the array element thus referenced. ARRAYREF nodes are currently implemented for 1-, 2- and 3-D arrays, but could be implemented for arrays of up to 8-D should that functionality be needed. The currently implemented ARRAYREF nodes are listed in Table 6, the single entry of which represents a family of $3 \times 6 = 18$ nodes (see footnote 1, above).

2.8 Reduction nodes

Reduction nodes handle values produced by a loop body, reducing those values by a corresponding arithmetic function. Currently implemented reduction nodes are listed in Tables

name	ins	outs	description
ARRAYREF- δ D- β -BIT	$3\delta+1$	1	generate a token with the value of a β -bit element ($\beta = 1, 2, 4, 8, 16$ or 32) of a δ -D array ($\delta = 1, 2$ or 3); input port I_0 carries the start address of the source array, ports I_1 to I_δ carry the indices of the referenced element in the source array, ports $I_{\delta+1}$ to $I_{2\delta}$ carry the extents of the source array in its δ dimensions, ports $I_{2\delta+1}$ to $I_{3\delta-1}$ carry the image offsets, while port $I_{3\delta+1}$ carries the image size in words; output port O_0 carries the value of the array element being referenced;

Table 6: Array Reference nodes.

7 and 8. An example reduction node is shown in Figure 4. The functionality of this node is as follows: a stream of *values* from a source array enters the node (on input port I_0), along with a *label* (integers beginning from 0, on port I_1) identifying from which region of the source array the current value is drawn, and a boolean *mask* (on port I_3) indicating whether the current value should be considered when determining the MIN for its particular region. The Accum-Umin-Values node determines the MIN of all non-dummy iteration values for each region of the source array, and writes to memory a 1-D array holding those minimum values (extent of output array == number of labeled regions in the input array).

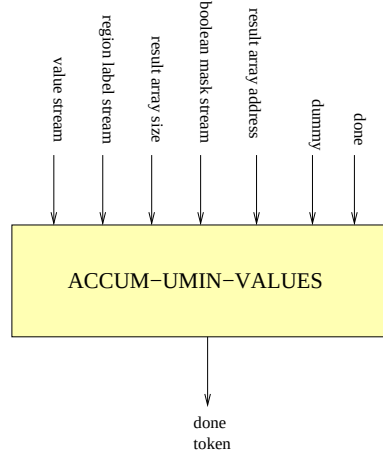


Figure 4: Reduction node example.

2.9 Circulate nodes

CIRCULATE nodes are associated with loops in the source code. Since these loops are unrolled in a dataflow graph, communicating altered values from one iteration of a loop to the next could be problematic. CIRCULATE nodes solve this problem by delivering such accumulation or *nextified* variables from the exit of a loop, back to the point within the loop where the variable is to be used in the next iteration. In while-loops, the circulate nodes are broken down as two nodes: MERGE and STEER nodes. CIRCULATE nodes

name	ins	outs	description
USUM-VALUES	5	1	sum a stream of unsigned tokens; I_0 is the value token stream, I_1 is the mask value stream, I_2 is the address to store the result, I_3 and I_4 are the dummy and done signals; O_0 emits a TRUE token when done;
ISUM-VALUES	5	1	sum a stream of signed tokens; I_0 is the value token stream, I_1 is the mask value stream, I_2 is the address to store the result, I_3 and I_4 are the dummy and done signals; O_0 emits a TRUE token when done;
UMIN-VALUES	5	1	min a stream of unsigned tokens; I_0 is the value token stream, I_1 is the mask value stream, I_2 is the address to store the result, I_3 and I_4 are the dummy and done signals; O_0 emits a TRUE token done;
IMIN-VALUES	5	1	min a stream of signed tokens; I_0 is the value token stream, I_1 is the mask value stream, I_2 is the address to store the result, I_3 and I_4 are the dummy and done signals; O_0 emits a TRUE token when done;
UMAX-VALUES	5	1	max a stream of unsigned tokens; I_0 is the value token stream, I_1 is the mask value stream, I_2 is the address to store the result, I_3 and I_4 are the dummy and done signals; O_0 emits a TRUE token when done;
IMAX-VALUES	5	1	max a stream of signed tokens; I_0 is the value token stream, I_1 is the mask value stream, I_2 is the address to store the result, I_3 and I_4 are the dummy and done signals; O_0 emits a TRUE token when done;
AND-VALUES	5	1	'and' a stream of tokens; I_0 is the value token stream, I_1 is the mask value stream, I_2 is the address to store the result, I_3 and I_4 are the dummy and done signals; O_0 emits a TRUE token when done;
OR-VALUES	5	1	'or' a stream of tokens; I_0 is the value token stream, I_1 is the mask value stream, I_2 is the address to store the result, I_3 and I_4 are the dummy and done signals; O_0 emits a TRUE token when done;
HIST-VALUES	6	1	create a histogram from a stream of tokens; I_0 is the value token stream, I_1 is the mask value stream, I_2 is the result array size, I_3 is the result array start address, I_4 and I_5 are the dummy and done signals; O_0 emits a TRUE token when done;
ACCUM-UMIN-VALUES	7	1	write to a result array in memory the minimum unsigned value in each of several labeled regions of a source array streamed into port 0 (result array size = number of labeled regions in source array); input I_0 is the stream of values from the source array, I_1 is the region number for the current value on I_0 , I_2 is the result array size (number of labeled regions in source array), I_3 is a boolean mask stream, I_4 is the address to which the result array should be written, I_5 and I_6 are the dummy and done signal; output O_0 emits a TRUE token when the node's processing is complete;
ACCUM-IMIN-VALUES	7	1	same as ACCUM-UMIN-VALUES but with signed values;
ACCUM-UMAX-VALUES	7	1	same as ACCUM-UMIN-VALUES but with maximum values;
ACCUM-IMAX-VALUES	7	1	same as ACCUM-UMAX-VALUES but with signed values;
ACCUM-USUM-VALUES	7	1	same as ACCUM-UMIN-VALUES but sum the values in each region of the source array rather than finding their min;

Table 7: Reduction nodes

name	ins	outs	description
ACCUM-ISUM-VALUES	6	1	same as ACCUM-USUM-VALUES but with signed values;
ACCUM-AND-VALUES	6	1	same as ACCUM-UMIN-VALUES but AND the values in each region of the source array rather than finding their min;
ACCUM-OR-VALUES	6	1	same as ACCUM-AND-VALUES but OR the values in each region of the source array;
ACCUM-HIST-VALUES	7	1	same as ACCUM-UMIN-VALUES, but create a histogram for each labeled region of the source array; input I_0 is the stream of values; I_1 is the stream of region labels in the source array; I_2 and I_3 are the 2-D result array's extents; I_4 is a stream of boolean mask values; I_5 is the result array start address; I_6 is the number of values in the incoming stream; output port O_0 emits a TRUE token when the node's processing is complete;

Table 8: Reduction nodes (continued)

and STEER nodes are the only node types that generate backward-directed edges in SA-C dataflow graphs.

CIRCULATE node specifics are given in table 9 while MERGE node and STEER node specifics are given in table 10. A data flow graph containing a CIRCULATE node, and the SA-C code that generated it, are given in Section 4 below.

name	ins	outs	description
CIRCULATE	4	2	circulate an accumulation (or <i>nextified</i>) variable from the exit point of an unrolled loop back to the point within the loop where the variable is next used; input port I_0 receives the current value of the nextified variable after each iteration, I_1 holds the initial value of the nextified variable, I_2 and I_3 hold the dummy and done signals; the final value of the nextified variable is delivered on output port O_0 upon completion of the loop, a back edge from port O_1 delivers the current value of the nextified variable back to the point within the loop where it is next used every iteration;

Table 9: Circulate node.

2.10 Write nodes

There are four basic types of write nodes – WRITE-SCALAR nodes, WRITE-TILE nodes, WRITE-ARRAY nodes and WRITE-VALS-TO-BUFFER.

2.10.1 Write-Scalar nodes

Currently, WRITE-SCALAR nodes are always associated with CIRCULATE nodes (though this could change in the future). In the current implementaion, when the final value of a *nextified* variable (see previous section) is to be returned by a dataflow graph, a CIRCU-

name	ins	outs	description
MERGE	2	1	select between the current iteration value and the initialization value; input port I_0 receives the current value of the nextified variable after each iteration, I_1 holds the initial value of the nextified variable; O_0 delivers the current value of the nextified variable to the STEER node and also for the calculation of the termination condition of the while-loop (if required).
STEER	2	2	steers the current value of the nextified variable as the final value or on the backedge to be used by the next iteration; input port I_0 receives the current iteration value from a MERGE node, I_1 receives the done signal; the final value of the nextified variable is delivered on output port O_0 upon completion of the loop, a back edge from port O_1 delivers the current value of the nextified variable back to the point within the loop where it is next used every iteration;

Table 10: Merge and Steer nodes.

LATE node delivers that value to a WRITE-SCALAR node upon completion of the loop, and the WRITE-SCALAR node writes the value to the appropriate location in memory.

WRITE-SCALAR node specifics are given in Table 11.

name	ins	outs	description
WRITE-SCALAR	2	1	write to memory the final value of a <i>nextified</i> variable upon completion of a loop; input port I_0 carries the value to be written; port I_1 carries the address to be written to; output port O_0 delivers a boolean TRUE token when writing is complete;

Table 11: Write-Scalar node.

2.10.2 Write-Tile nodes

WRITE-TILE nodes write results of m -dimensional loops to target memory at each non-dummy iteration. The result per iteration is an n dimensional subarray that is tiled. (Thus, the rank of the output array written to memory is equal to the greater of m and n .) To fully specify a WRITE-TILE node, both the dimensionality of the loop and the rank of the tile produced must be specified. Currently implemented WRITE-TILE nodes are listed in Table 12, each entry of which represents a family of $3 \times 6 = 18$ related nodes.

Figure 5 shows an example WRITE-TILE node for which the original loop is 2-dimensional and the tile to be written for each element in that array is 3-dimensional. Since the loop is two-dimensional, two input ports carry the loop iterations (ports 9-10). Since the tile to be written for each element has 3 dimensions, three input ports carry the tile's extents (ports 11-13) and two input ports carry the image offsets (ports 14-15). For this example the tile has an extent of 2 in each of the three dimensions, thus requiring that each tile have eight elements. Thus eight input ports carry the incoming stream of tile values (ports 0-7). Port 8 carries the start address of the output array being written while the last two ports carry

the dummy and done signal. Output port 0 emits a TRUE token when all the tiles have been written to memory (that is, it receives the done token).

name	ins	outs	description
WRITE-TILE- δ D-1D- β -BIT	$n + 2\delta + 2 + l$	1	for each iteration of a δ -dimensional loop ($\delta = 1, 2$ or 3), write a 1-dimensional tile of β -bit tokens ($\beta = 1, 2, 4, 8, 16$ or 32) to memory; I_0 to I_{n-1} carry the n values in the current tile, I_n is the start address, I_{n+1} to $I_{n+\delta}$ carry the iteration counts of the δ -dimensional loop, $I_{n+\delta+1}$ carries the extents of the 1D tile, the next ports carry $l(= \max(\delta, 1) = \delta) - 1$ image offsets, the dummy and done signals; output port O_0 emits a TRUE token when writing is complete;
WRITE-TILE- δ D-2D- β -BIT	$n + \delta + 4 + l$	1	for each iteration of a δ -dimensional loop ($\delta = 1, 2$ or 3), write a 2-dimensional tile of β -bit tokens ($\beta = 1, 2, 4, 8, 16$ or 32) to memory; inputs I_0 to I_{n-1} (where $n = x \times y$, with x and y the dimensions of the tile) carry the values in the current tile, I_n is the start address, I_{n+1} to $I_{n+\delta}$ carry the iteration counts, $I_{n+\delta+1}$ and $I_{n+\delta+2}$ are the extents of the 2D tile, the next ports carry $l(= \max(\delta, 2)) - 1$ image offsets, the dummy and done signals; output port O_0 emits a TRUE token when writing is complete;
WRITE-TILE- δ D-3D- β -BIT	$n + \delta + 5 + l$	1	for each iteration of a δ -dimensional loop ($\delta = 1, 2$ or 3), write a 3-dimensional tile of β -bit tokens ($\beta = 1, 2, 4, 8, 16$ or 32) to memory; inputs I_0 to I_{n-1} (where $n = x \times y \times z$, with x, y and z the dimensions of the tile) carry the values in the current tile, I_n is the start address, I_{n+1} to $I_{n+\delta}$ carry the iteration counts, $I_{n+\delta+1}$ to $I_{n+\delta+3}$ are the extents of the 3D tile, the next ports carry $l(= \max(\delta, 3)) - 1$ image offsets, the dummy and done signals; output port O_0 emits a TRUE token when writing is complete;

Table 12: Write-Tile nodes.

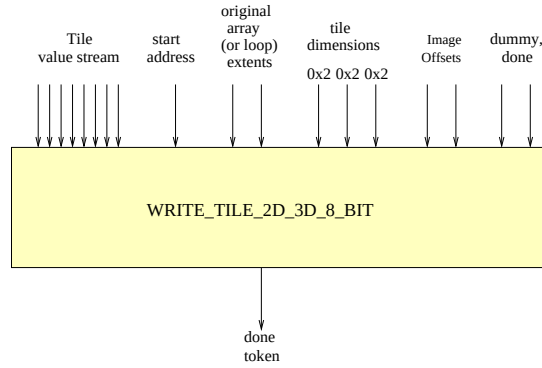


Figure 5: Write-Tile node example.

2.10.3 Write-Array nodes

WRITE-ARRAY nodes write results of m -dimensional loops to target memory at each non-dummy iteration. The result per iteration is an n dimensional subarray that is not tiled. (Thus, the rank of the output array written to memory is equal to the sum of m and n .) The WRITE-ARRAY node structure is identical to that of a WRITE-TILE. They differ in the way they produce the output array. Currently, WRITE-ARRAY-1D-1D, WRITE-ARRAY-1D-2D and WRITE-ARRAY-2D-1D exists at the DFG level. Note that for a Write-Array node $l = m + n$.

2.10.4 Write-Vals-To-Buffer nodes

WRITE-VALS-TO-BUFFER nodes write results of one-dimensional loops when the output array size is unknown at run-time. The non-masked values are written into a buffer (in the memory). When the buffer is full, the host is signalled to empty the buffer. A counter stores the number of values currently written in the buffer. (This counter is reset when the buffer is emptied).

WRITE-VALS-TO-BUFFER node specifics are given in Table 13.

name	ins	outs	description
WRITE-VALS-TO-BUFFER	$2 * n + 5$	1	write to buffer the non-masked values every iteration of a 1D loop; n pairs of inputs contain values and their masks, input port I_{2n} carries the buffer size, I_{2n+1} carries the start address of the buffer, I_{2n+2} has the counter address in the memory while I_{2n+3} and I_{2n+4} are the dummy and done signals; output port O_0 delivers a boolean TRUE token when writing is complete;

Table 13: Write-Vals-To-Buffer node.

2.11 I/O nodes

I/O nodes handle the interface with the outside world. Each node specifies a *channel* (input or output channel, as appropriate), designated by an input constant. I/O nodes are listed in Table 14.

name	ins	outs	description
INPUT	0	1	get a value from the input channel <i>constant</i> specified by the corresponding input port of the DFG wrapping node.
OUTPUT	1	0	take a value token from I_0 ; A TRUE token indicates completion of execution of the DFG.

Table 14: I/O nodes

2.12 LoopInfo node

LoopInfo nodes provide the iteration counts for each dimension of the loop. They also provide the product of iteration count of the current dimension to the innermost dimension. For an n -dimensional loop have $2n$ input ports: I_0 to I_{n-1} carry the iterations per dimension of the loop while I_n to I_{2n-1} carry the multiplied iterations. ($I_{n+i} = \prod_{k=i}^{n-1} I_k$). LoopInfo node is listed in Table 15.

name	ins	outs	description
LOOPINFO	$2n$	0	Provide iteration counts for an n -dimensional loop; I_0 to I_{n-1} carry the iterations per dimension of the loop while I_n to I_{2n-1} carry the multiplied iterations.

Table 15: LoopInfo node

3 Dataflow file format

To allow easy inspection and interfacing of dataflow graphs with the tools and programs that use them, a text representation is used. This representation is identical to the DDCF format.

The zero-th node of the DFG is a DFGWRAP node. It is a compound node that contains the actual DFG graph to be executed. The inputs to this wrapper node are compile-time constants that correspond to the addresses in the source memory where the run-time constants are stored. The DFGWRAP node contains a single output node. An arrival of a TRUE token on the input port of this node (during simulation) indicates completion of execution of the DFG.

For simulation, the DFG (in the DDCF format) is converted into internal DFG representation. The 0th DFGWRAP node is removed and all the node numbers are shifted by one. Also the input nodes get one input port carrying the compile-time constant address location corresponding to the run-time constant they must provide.

4 Example Dataflow Graphs

4.1 DFG with Unrolled Loop

Consider the following SA-C program:

```
int16[:,:] main (uint8 Image[:,:]) {  
  
    int16 H[3,3] = { {-1, 0, 1},  
                    {-1, 0, 1},  
                    {-1, 0, 1}} ;  
  
    int16 R[:,:] = for window W[3,3] in Image {  
                    int16 iph = for h in H dot w in W  
                                return( sum(h*w) );  
    } return( array(iph) );  
}
```

This code performs the convolution of a 3×3 mask over over a large input `Image` array, as one might see in an edge detection routine – e.g. the Prewitt edge detection algorithm.

As shown in Figure 6, both loops of this nested structure are converted to a single dataflow graph by the SA-C compiler. The Window-Generator node near the top of this graph reads elements from a 3×3 window of the `Image` array at each iteration, and as this data flows through the graph, the necessary convolution is performed. Notice the multiplies explicit in the code have been removed by the compiler, replaced with either *no-ops* (for multiplication by 0) or *negate* operations (for multiplication by -1). Thus, the nine multiplies and eight additions explicit in the code at each iteration of the for-loop, have been replaced with three negation operations and five additions at each iteration.

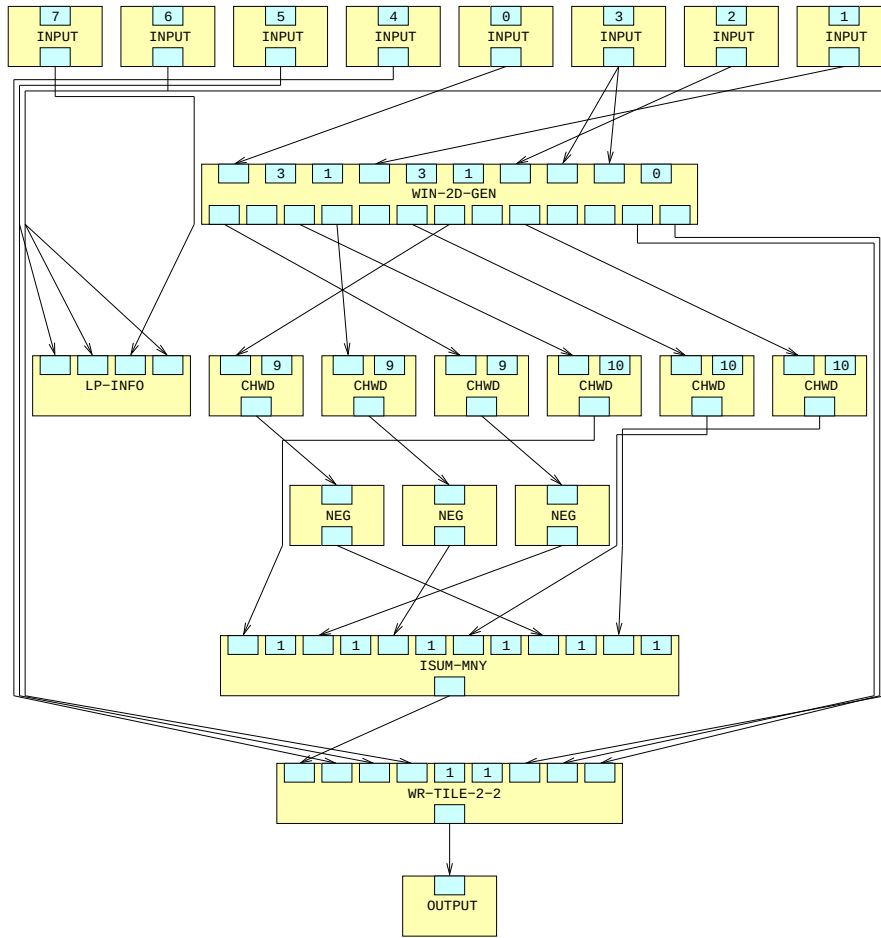


Figure 6: Dataflow Graph with unrolled loop.

4.2 DFG with Circulate Nodes and Back-edges

The following SA-C routine uses a *nextified* variable `count`, and thus generates a dataflow graph containing a CIRCULATE node – the only node type in SA-C dataflow graphs, recall, which generate back-edges. Nextified variables allow for the accumulation of some value from one iteration of an unrolled loop to the next.

```
uint16 main (uint8 Image[:,:], uint8 val) {
    uint16 count = 0;

    uint16 occurrence =
```

```

        for e1 in Image {
            next count = ( e1 == val ? count + 1 : count );
        } return( final(count) );
    } return(occurrence);

```

This routine simply counts the number of occurrences of the value `val` in the input array `Image`.

Figure 7 shows the dataflow graph generated from this code. The current value of the nextified variable `count` is circulated back up the graph at each iteration of the loop via back-edges from the CIRCULATE node's output port O_1 . When the loop terminates, the final value of this variable leaves output port O_0 and is written to memory by the WRITE-SCALAR node. The SELECTOR node serves as a conditional statement: if the comparison performed by the UEQ node is false (not equal to the 1 on the SELECTOR node's input I_1), the current value of `count` is selected and passed to the CIRCULATE node; if the comparison is true, an incremented version of `count`, entering input I_2 from the UADD node above, is selected and passed to the CIRCULATE node.

4.3 DFG with VAL-AT-FIRST-UMAX-MANY node

The following SA-C code implements a part of the *dilation* edge smoothing algorithm. A small `kernel` array is passed over a larger `src` image array, overlapping elements of the two arrays are summed, and the max sum within the current boundary of the kernel is selected. A 2×2 kernel was used for this example to keep the DFG generated to a manageable size; a more typical kernel size is 5×5 .

```

uint8[:,:] main(uint8 src[:,:], uint8 kernel[2,2])
{
    uint16 r, uint16 c = extents(kernel);
    uint8 result[:,:] =
        for window win[r,c] in src
        {
            uint8 maxvals[:] =
                for elem1 in win dot elem2 in kernel
                {
                    bool b = (elem2 != 0);
                } return(vals_at_first_max(elem1+elem2,{elem1},b));
            uint8 maxval = maxvals[0];
        }return(array(maxval));
    }return(result);

```

Figure 8 shows the dataflow graph generated from this code. Notice the four additions required for each position of the kernel are done in parallel by the four UADD nodes on level 3 of the DFG (for a 5×5 kernel, of course, twenty-five additions would be done in parallel). The first (leftmost) maximum among these sums is selected by the VAL-AT-FIRST-UMAX-MANY node on level 4, and written to memory by the WRITE-TILE node below.

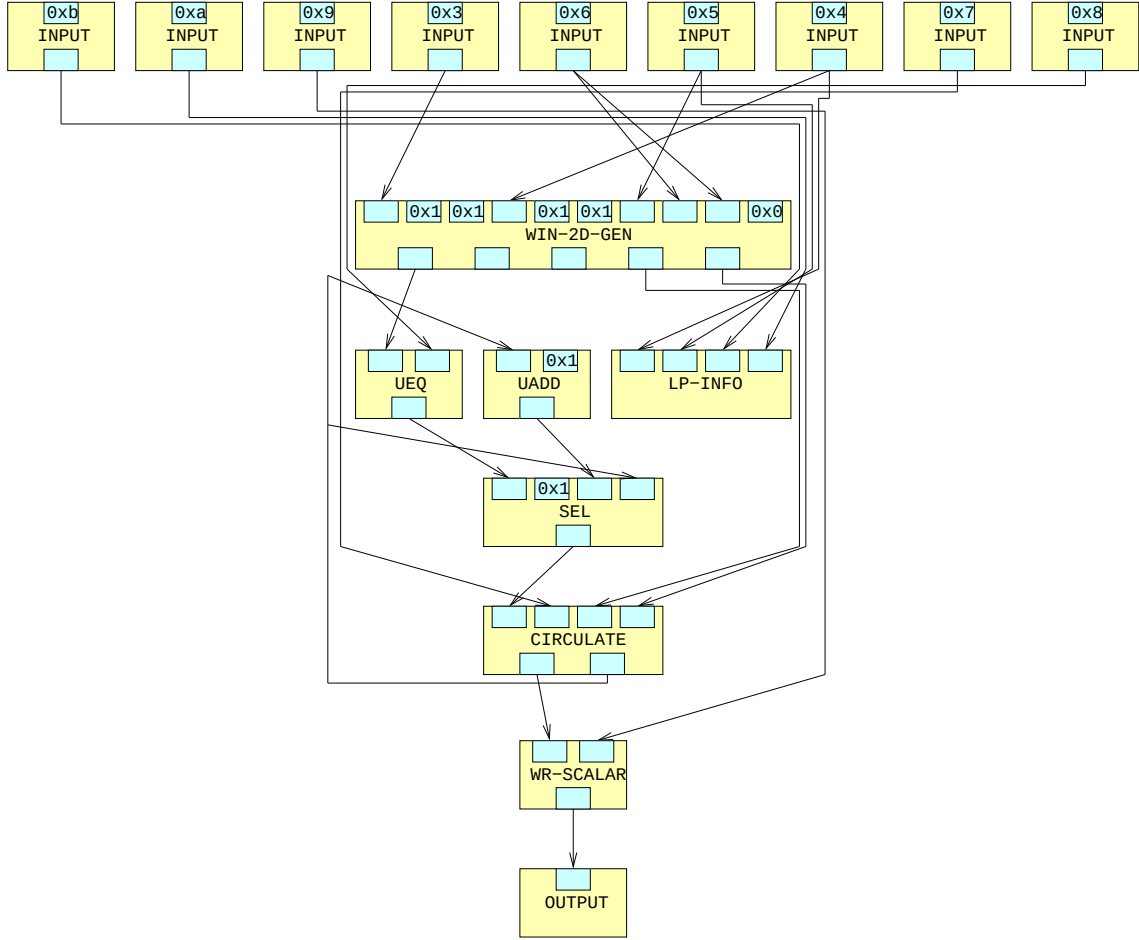


Figure 7: Dataflow Graph with CIRCULATE node and back-edges.

5 A Complete Example

Our final example demonstrates the transformation of a SA-C program from source code to data dependency and control flow (ddcf) graph, and the RC_COMPUTE node within that ddcf graph which contains the dataflow graph (DFG). Consider the following SA-C program:

```
uint8[:] main (uint12 A[:], uint12 x) {
    uint12 R[:] =
        for window W[4] in A
```

```
        return (array (x + (uint12) array_sum (W)));  
    } return (R);
```

The compiler will unroll the implicit loop created by “array_sum”, and enclose it in an RC_COMPUTE node that contains the loop and the host-RCS interface. This is shown in Figure 9. The RC_FORALL graph is then converted to a DFG, as shown in Figure 10. The text representation of this DFG is shown in Figure 11 and 12.

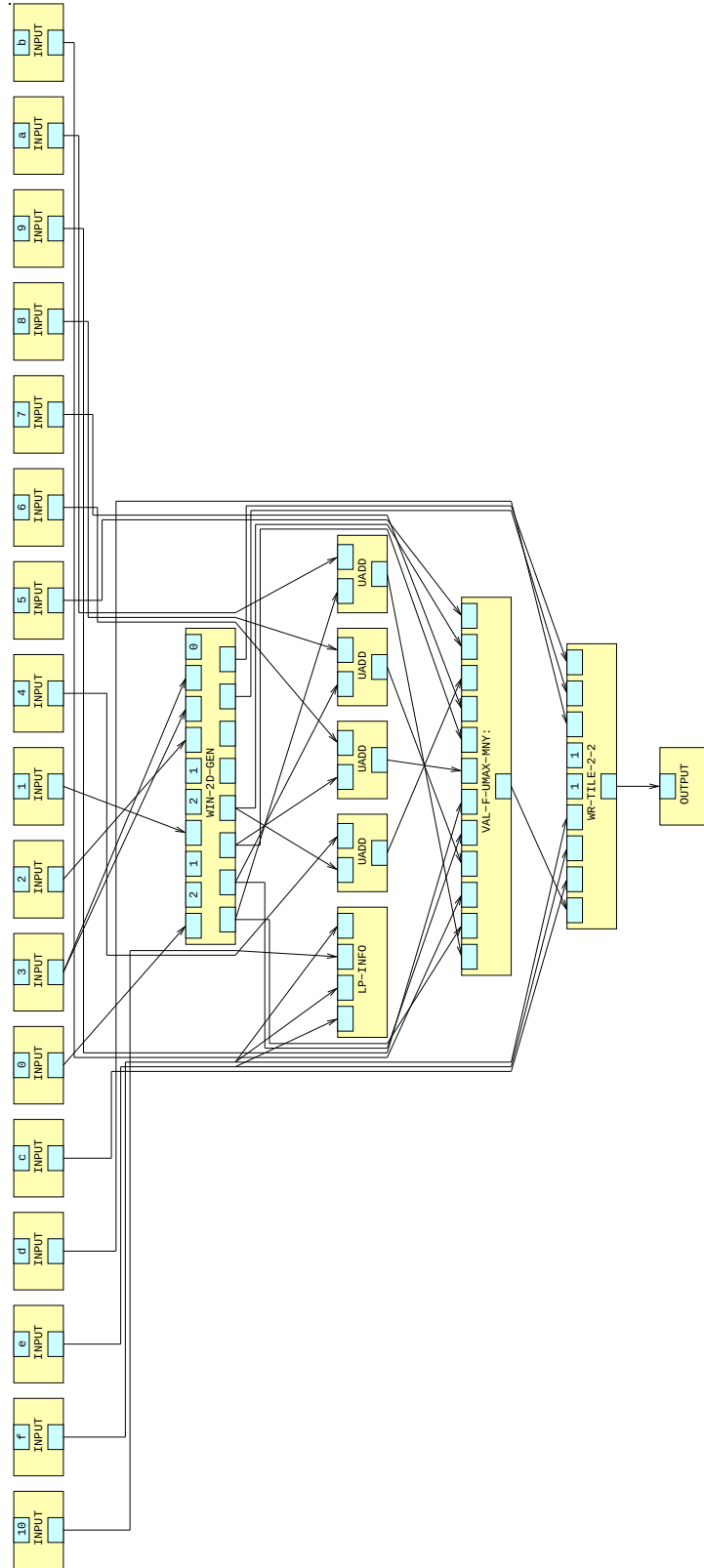


Figure 8: Dataflow Graph with VAL-AT-FIRST-MAX node.

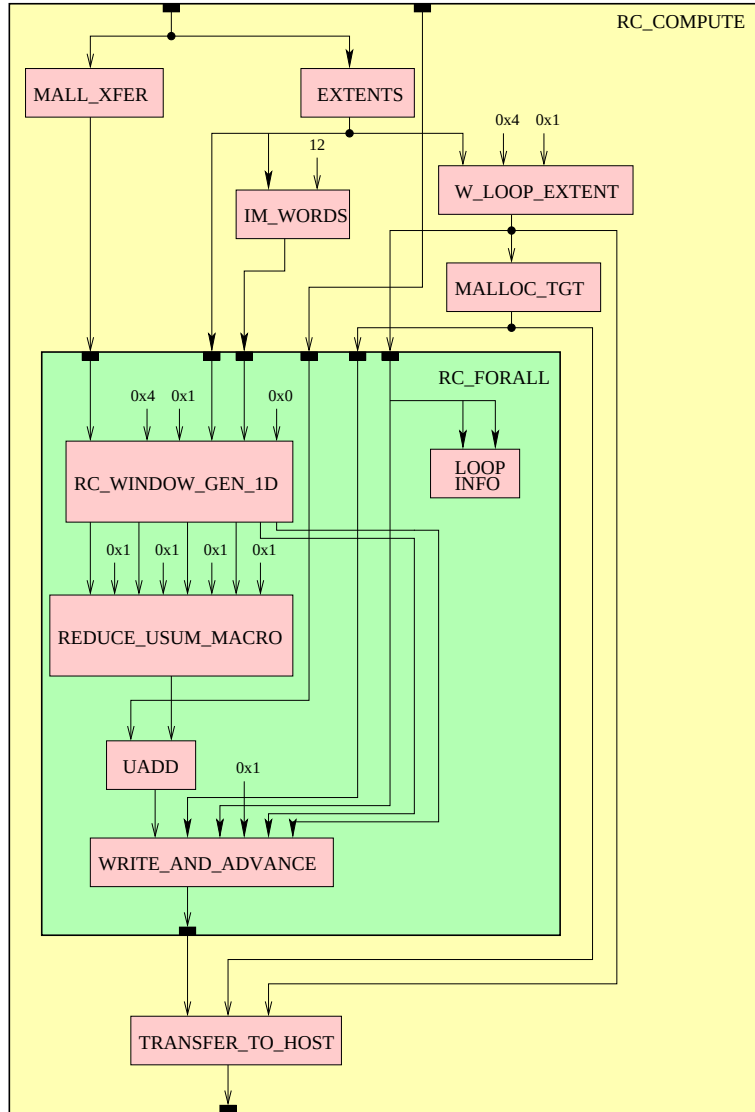


Figure 9: DDCF graph of converted loop with interface.

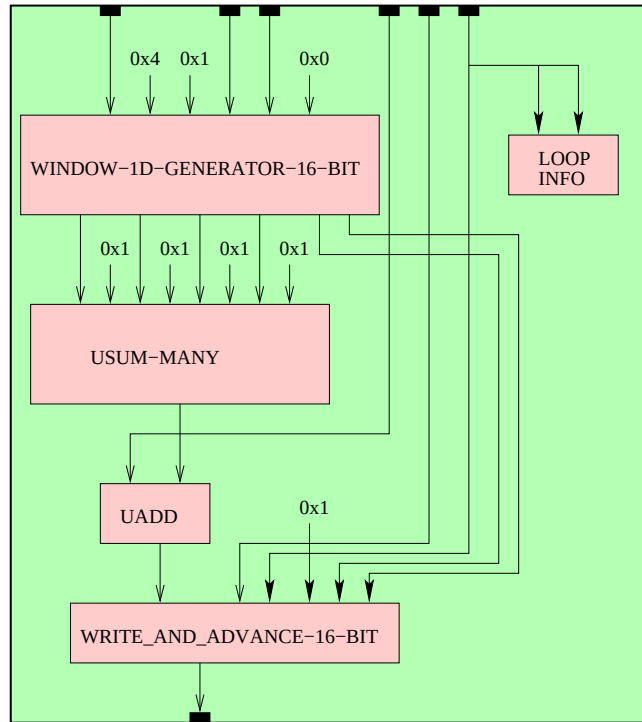


Figure 10: DFG converted loop.

```

dfg "tst_0"
0 ND_DFG_WRAP (my nodes: 1 2 3 4 5 6 7 8 9 10 11) <"tst.sc", "main", 3>
  6 inputs: (nodes 4 6 5 9 3 2)
    port 0 <bits32> value "0x0"
    port 1 <bits32> value "0x1"
    port 2 <bits32> value "0x2"
    port 3 <bits32> value "0x3"
    port 4 <bits32> value "0x4"
    port 5 <bits32> value "0x5"
  1 outputs: (nodes 12)
    port 0 <bits32>
;
1 ND_LOOPINFO <"tst.sc", "main", 3>
  2 inputs:
    port 0 <bits32>
    port 1 <bits32>
  0 outputs:
;
2 ND_G_INPUT (input 5 for graph node 0) <"tst.sc", "main", 3>
  0 inputs:
  1 outputs:
    port 0 <bits32> 11.2 1.0 1.1
;
3 ND_G_INPUT (input 4 for graph node 0) <"tst.sc", "main", 3>
  0 inputs:
  1 outputs:
    port 0 <bits32> 11.1
;
4 ND_G_INPUT (input 0 for graph node 0) <"tst.sc", "main", 3>
  0 inputs:
  1 outputs:
    port 0 <bits32> 7.0
;
5 ND_G_INPUT (input 2 for graph node 0) <"tst.sc", "main", 3>
  0 inputs:
  1 outputs:
    port 0 <bits32> 7.4
;
6 ND_G_INPUT (input 1 for graph node 0) <"tst.sc", "main", 3>
  0 inputs:
  1 outputs:
    port 0 <bits32> 7.3
;
7 ND_RC_WINDOW_GEN_1D <"tst.sc", "main", 3>
  6 inputs:
    port 0 <bits32>
    port 1 <bits3> value "0x4"
    port 2 <bits1> value "0x1"
    port 3 <bits32>
    port 4 <bits32>
    port 5 <bits1> value "0x0"

```

Figure 11: Text representation of DFG.

```

7 outputs:
  port 0 <bits12> 8.0
  port 1 <bits12> 8.2
  port 2 <bits12> 8.4
  port 3 <bits12> 8.6
  port 4 <bits32>
  port 5 <bits1> 11.4
  port 6 <bits1> 11.5
;
8 ND_REDUCE_USUM_MACRO <"tst.sc", "main", 4>
8 inputs:
  port 0 <bits12>
  port 1 <bits1> value "0x1"
  port 2 <bits12>
  port 3 <bits1> value "0x1"
  port 4 <bits12>
  port 5 <bits1> value "0x1"
  port 6 <bits12>
  port 7 <bits1> value "0x1"
1 outputs:
  port 0 <bits12> 10.1
;
9 ND_G_INPUT (input 3 for graph node 0) <"tst.sc", "main", 4>
0 inputs:
1 outputs:
  port 0 <bits12> 10.0
;
10 ND_UADD <"tst.sc", "main", 4>
2 inputs:
  port 0 <bits12>
  port 1 <bits12>
1 outputs:
  port 0 <bits12> 11.0
;
11 ND_WRITE_TILE_1D_1D <"tst.sc", "main", 4>
6 inputs:
  port 0 <bits12>
  port 1 <bits32>
  port 2 <bits32>
  port 3 <bits32> value "0x1"
  port 4 <bits1>
  port 5 <bits1>
1 outputs:
  port 0 <bits1> 12.0
;
12 ND_G_OUTPUT (output 0 for graph node 0) <"tst.sc", "main", 3>
1 inputs:
  port 0 <bits1>
0 outputs:
;

```

Figure 12: Text representation of DFG (contd).

CAMERON PROJECT: FINAL REPORT

Appendix E: Abstract Hardware Architecture Description

SA-C Abstract Hardware Architecture Description

Charles A. Ross
Computer Science Department
Colorado State University
Fort Collins, CO, USA
E-mail: rossc@cs.colostate.edu

1 Introduction

An AHA graph is a complex graph structure, created from many nodes. Each node provides some small unit of functionality. It is not until these nodes are combined to create a larger structure, that more complex behaviors arise. The following is an intuitive description of AHA. The AHA simulator provides executable AHA semantics.

2 Detailed Description of AHA

The AHA was added as an intermediate stage between the DFG and VHDL. The translation from DFG to VHDL was proving to be more difficult than first envisioned. This is due to the nature of DFG graphs and VHDL, and the large incongruity between them. Both the DFG and VHDL have executable semantics, however the DFG is un-timed. VHDL on the other hand is heavily time dependent; Every VHDL component utilizes a hardware clock. DFG nodes can be large conceptual blocks (Such as “Window Generator”). In practice, components with such a complex behavior are difficult to build in VHDL. DFG nodes execute according to standard DFG semantics; A node fires whenever all its inputs are available. In VHDL, logic must be synthesized which will signal each node when it is to execute. Because debugging VHDL is very difficult, it was decided that the translation from AHA to VHDL should be as trivial as possible, consisting of nothing more than macro-expansion.

2.1 DFG to AHA

The AHA was introduced to bridge the majority of the gap to VHDL. The DFG to AHA translation introduces timing, breaks large monolithic nodes into smaller clusters of nodes, and results in a graph that is easily translateable into VHDL. The AHA to VHDL translation is nothing more than a special type of code generation. All of the nodes in the AHA have simple VHDL components. AHA nodes are assembled like Lego blocks to create complex behaviors. It is designed to mimic data-flow semantics in hardware.

2.2 Finite Buffers = Clock Level Balancing

Because infinite buffers (Which are implicit in the DFG) cannot be synthesized in hardware, and not all nodes consume and produce data at the same rate, each node needs to be able refuse tokens from the nodes feeding it. In a condition that a node is blocking, waiting for some particular event, the nodes leading up to this node may need to pause as well. This introduces the need for “Clock Level Balancing”. Clock Level Balancing is a process which ensures that all of the tokens entering a node are at the same level of latency. If this is *not* the case, the graph may deadlock or run slowly. In order to balance the clock levels, each node must be localized to a level. To begin, the input nodes of the AHA are labeled as Level-0. All of the nodes that are fed by the input nodes, have their inputs at level 0. The maximum level of all the inputs of a node is added to the latency of the node to compute the level of the outputs. Many nodes have no latency, and so the inputs and outputs are in the same level. After all of the inputs and outputs of every node have been computed, Buffers are added to inputs of nodes which have less latency than the other inputs. The size of the buffer is computed using the difference in the clock levels of the inputs.

2.3 Token Driven = Handshaking

The goal of the AHA is to maintain the token driven semantics of the DFG as much as possible, while making it easier to synthesize. As the tokens flow through the graph, it needs to guarantee that no tokens will be lost or created while they travel over the edges of the graph. In the DFG this is done with the unsynthesizable infinite buffers on each edge. In the AHA, handshaking is used to ensure that no nodes produces tokens that cannot be accepted, and no nodes accept tokens that are not being produced. This handshaking is vital to the operation of the AHA, and is not explicitly laid out in the graph, but is rather implicit logic between connected nodes.

There are three types of nodes used in the AHA graph: **Clocked**, **Semi-Clocked**, and **Un-Clocked**. Clocked nodes have a non-zero latency between when they accept a token, and the resulting token(s) are produced. Semi-Clocked Nodes have no latency, but still utilize a clock signal in their internals. Un-Clocked nodes have no latency, nor do they use a clock. Only the Clocked nodes participate in handshaking.

2.4 Handshaking = Sections

In order to make the handshaking more easily understandable, the graph is divided into *sections*. Sections contain the nodes which need to fire at the same time. Three types of nodes make up a section, **Producers**, **Consumers**, and **Internals**. The Producers are arranged conceptually at the “top” of the section, and the Consumers are positioned at the bottom. The Internal nodes consist of Semi-Clocked and Un-Clocked nodes and lie between the Producers and Consumers. Each Clocked node exists in two sections; In one of the sections it is a Consumer, in the other it is a Producer. For this reason, it is useful to think of Clocked nodes as having a top and bottom half. These halves fire independently; The top half fires when a value is accepted, while the bottom fires when it produces. Clocked nodes exist on the boundary between sections. Sections are built using a connect-components algorithm. All of the outputs of a clocked node are “connected” as well as all of the inputs, but inputs and outputs are not connected together. All of the inputs and outputs of Un-Clocked and Semi-Clocked nodes are connected. Input and Output ports that have an edge

between them are also connected. Connected components are grown until all of the “connected” nodes have been placed into sections.

2.5 Memory Contention & Arbitration

Some nodes in the AHA graph require access to off-chip memory storage. There can be an arbitrary number of memory clients in the graph. However, on any given architecture, there is a finite number of memory pathways. This gives rise to memory contention between the memory clients. In fact, this problem existed even in the OnePE and TwoPE systems. In the TwoPE system, the window generators were extremely complex, in order to avoid this issue entirely. In the OnePE system, this was addressed by adding a Memory Arbitrator. This arbitrator simply decides which node is allowed to access memory in each clock cycle. In the OnePE system, this was designed by Cameron, and was a priority based arbitration. Each node was assigned an ID. The node with the lowest ID, which requested memory, is given it for that clock cycle. In the AHA, we use a memory arbitrator which is provided by Annapolis Micro-Systems. The provided memory arbitrator, is in fact, *identical* to the one we implemented in the OnePE system. The existence of the memory arbitrator allows the memory contention issues to be addressed at run-time, with a minimum analysis. It also allows the system to run as fast as the memory bandwidth will allow. There are no wasted cycles, if the application requires more bandwidth than is available.

3 Descriptions of arithmetic nodes

Each of the nodes available in the AHA have been tailored to be as efficient as possible, while maintaining a behavior that is intuitive. This allows them to be assembled in a Lego-like fashion, without concern as to how they interact.

There are a large number of arithmetic nodes in the AHA. They are all **Un-Clocked** combinational nodes; they simply perform computations. Table 1 lists all of the combinational nodes and the function they perform.

By assembling these nodes together appropriately, we construct all of the mathematical functions needed in our programs. The other nodes (**Clocked** and **Semi-clocked**) perform other functions, such as providing the loop structure, reorganizing data, accessing memory and other off-chip functionality, etc.

4 Descriptions of other nodes

All of the **Clocked** nodes have several signals in common. These signals allow the AHA to function very similarly to a token driven data-flow machine. All of them are a single bit, and control the behavior of the communication from node to node, as well as communication off-chip. Table 2 Lists all the nodes in the AHA model, and lists whether they are **Clocked**, **Un-Clocked**, or **Semi-Clocked**.

- **Clock Input** - provides the clock signal to the node. This allows the nodes to synchronize data transfer, and off-chip communications.
- **Reset Input** - provides a signal to the node which tells it when to reset all of its internal state. This is also how the nodes are placed into a known state to begin computations.

name	inputs	description
UADD	2	unsigned addition
IADD	2	signed addition
USUB	2	unsigned subtraction
ISUB	2	signed subtraction
UMUL	2	unsigned multiplication
IMUL	2	signed multiplication
NEGATE	1	negate signed value
ULT	2	unsigned < comparison
ILT	2	signed < comparison
ULE	2	unsigned $\leq$ comparison
ILE	2	signed $\leq$ comparison
UGT	2	unsigned > comparison
IGT	2	signed > comparison
UGE	2	unsigned $\geq$ comparison
IGE	2	signed $\geq$ comparison
UEQ	2	unsigned equality comparison
IEQ	2	signed equality comparison
UNEQ	2	unsigned inequality comparison
INEQ	2	signed inequality comparison
LEFT_SHIFT	2	left shift binary value, padding zeros
RIGHT_SHIFT	2	right shift binary value, padding zeros
CHANGE_WIDTH	1	truncate or zero extend a binary value
CHANGE_WIDTH_SE	1	truncate or sign extend a binary value
BIT_AND	2	bit-wise AND
BIT_OR	2	bit-wise OR
BIT_EOR	2	bit-wise exclusive OR
BIT_COMPL	1	bit-wise complement

Table 1: Arithmetic nodes

- **Ready_In** *Output* - signals the nodes feeding this one that it is ready to accept incoming data.
- **Ready_Out** *Output* - signals the nodes which are consuming this nodes values that it is ready to produce outgoing data.
- **AllReady_In** *Input* - indicates that all of the nodes producing values for this node (and all of the other nodes consuming their values etc..) are ready to produce and consume. This is a common signal for an entire section. All of the producer's "Ready_Out" values, and all the consumers "Ready_In" values are and-ed together, and fed back into the nodes. Consumers receive the and-ed value in their "AllReady_In" port. Producers receive it in the "AllReady_Out"

name	Clocking	name	Clocking
UADD	Un-Clocked	IADD	Un-Clocked
USUB	Un-Clocked	ISUB	Un-Clocked
UMUL	Un-Clocked	IMUL	Un-Clocked
NEGATE	Un-Clocked	ULT	Un-Clocked
ILT	Un-Clocked	ULE	Un-Clocked
ILE	Un-Clocked	UGT	Un-Clocked
IGT	Un-Clocked	UGE	Un-Clocked
IGE	Un-Clocked	UEQ	Un-Clocked
IEQ	Un-Clocked	UNEQ	Un-Clocked
INEQ	Un-Clocked	LEFT_SHIFT	Un-Clocked
RIGHT_SHIFT	Un-Clocked	CHANGE_WIDTH	Un-Clocked
CHANGE_WIDTH_SE	Un-Clocked	BIT_AND	Un-Clocked
BIT_OR	Un-Clocked	BIT_EOR	Un-Clocked
BIT_COMPL	Un-Clocked	TOK_GEN	Clocked
COUNTER	Semi-clocked	READ_WORD	Clocked
WRITE_WORD	Clocked	PACK	Unlocked
UNPACK	Unlocked	SHIFT_REGISTER	Clocked
FIFO_PACK	Clocked	FIFO_UNPACK	Clocked
FIFO_PACK_FULL	Clocked	BUFFERX	Clocked
DELAY_REGISTER	Semi-clocked	REPLICATOR	Clocked
REPLICATOR_EXTRA	Clocked	MASK	Clocked
CIRCULATE	Clocked	ROMDEF	N/A
ROMREF	Un-Clocked	DONE	Clocked
ADDR_CALC	Clocked		

Table 2: All Nodes

- **AllReady_Out** *Input* - indicates that all of the nodes consuming values from this node are ready to consume.
- **Last_{In/Out}** - Most nodes make use of a “Last” Signal. The last signal is misnamed. Historically, it was used to tag the last element of a sequence, to allow the nodes to reset, and flush buffers etc. Now, the Last signal comes after the last element of the sequence. Its use is the same, but instead of accompanying valid data, it accompanies garbage tokens. Nodes that accept a “Last_In” use it to reset their internal state, and to ready themselves for a new sequence. They also do not act upon the data received when the Last_In signal is high. Most also generate a “Last_Out” when when they receive a the last token in the sequence, to inform following nodes of the end of sequence information.

The Semi-clocked nodes use some of the same signals. They use **Clock**, **Reset**, and an **AllReady** signal. The “AllReady” signal is very similar to the “AllReady_In” and “AllReady_Out” used in the Clocked nodes, except that there does not need to be two. The Semi-Clocked nodes do not

span a section boundary, so they always consume and produce at the same time. They are also always ready, so they do not have a **Ready** signal.

4.1 TOK_GEN

The token generator node is the node which drives the loop. It takes a single unsigned integer n , as input. It then produces n consecutive 0's on its one bit output, followed by a single 1. The bit is interpreted as a “last” signal. The 0's represent that the loop is not finished. The 1 represents that the loop has completed. The bit generated from the token generator can be thought of as a “Last” signal, without any accompanying data.

4.2 COUNTER

The counter is one of the few nodes that is Semi-Clocked. It has three inputs: **Init**, **Increment**, and **Data**. Init and Increment are both integer values. Data is a single bit. When the node is reset, the output takes on the value of the Init signal. The counter then adds the increment to the output for every clock cycle in which Data is 0. When Data becomes 1, it resets back to Init. It is used to compute linear functions incrementally. An excellent example is the beginning address for an inner loop. The address of the beginning of each row in an image can be computed in this manner. Like the Token Generator, the Data input is really just a “Last” signal, without any data.

4.3 READ_WORD

The Read Word node reads a stream of values from the off-chip memories. They provide data to the loops in the chip, and are vital to the operation of any program. There are really 4 versions of the Read Word node. Two are built to access a 32-bit memory, and the other two are built to access a 64-bit memory. Of the 32-bit versions, one uses a single memory and accesses 32-bit words. The other attaches to two memories simultaneously to simulate a 64-bit memory. Likewise, of the 64-bit memories, one treats it as a 64-bit memory. The other reads half-words to emulate a 32-bit memory. These nodes simply take an address, fetch the value at that address, and produce it. Because of memory latency, these nodes have a good deal of internal buffering. The buffers allow them to produce a steady stream of values given a steady stream of addresses. Currently, only codes that read 32-bit words are synthesized, although 64-bit memories may be utilized at half rate. When the Last signal is high, the read words do not initiate a memory transaction, instead they simply add a dummy value to the queue with the Last flag set. When it reaches the end of the queue, it generates the Last_Out signal. This is one of the few nodes that does not completely reset when it gets a Last signal. It simply passes it through.

4.4 WRITE_WORD

Like the inverse of the Read Word, the Write Word takes a stream of Address-Data pairs and stores the data in the off-chip memory at the desired location. There are also 4 versions of the Write Word node, which correspond to the 4 Read Word nodes. Like the Read Word nodes, currently only 32-bit write words are synthesized, but they may be attached to 64-bit memories. The Last_In signal is used to suppress the memory transaction. Write Word has no outputs.

4.5 PACK

Pack is a very simple Un-Clocked node that takes an arbitrary number of variable bit inputs and concatenates them into a single value. This is mostly used to pack pixel values (typically 8-bits) into words (32-bits) suitable for writing into memory. They are used often throughout the code to reorganize data. Because they are only wiring, they dissolve during synthesis.

4.6 UNPACK

Unpack nodes are the logical inverse to the Pack nodes. They take a single value, and break it into an arbitrary number of variable bit objects. They are primarily used to unpack data coming from memory into smaller pieces. Unpack nodes can be used in conjunction with Pack nodes to reorganize data, or to recombine data. This combination is used heavily when synthesizing many of the tree style nodes in the DFG. Unpack nodes also dissolve during synthesis.

4.7 SHIFT_REGISTER

Shift registers are seldom used. They are mostly legacy code for generating sliding windows of data. They accept a stream of values, and provide a window of consecutive values in the stream simultaneously. With recent optimizations in the the DDCF and DFG layers, these are no longer needed. They are replaced by more efficient Delay Registers. They were inefficient because of the logic needed to handle Last_In signals: They flush their internal buffers, and delete any partial windows, and generate a Last_Out signal, only if they generated any data in the current stream.

4.8 FIFO_PACK

A FIFO Pack is a serialized, clocked version of a pack node. It accepts a series of values, concatenates them, and produces them at once. For example, a FIFO Pack might accept four 8-bit integers, concatenate them, and produce a single 32-bit value every fourth cycle. These are used in a similar way to Pack nodes, except they are used when the computational body cannot (or does not) support the data parallelism that would be needed to use pack nodes. It uses the Last_In signal to flush its internal state, producing any partial packed data.

4.9 FIFO_UNPACK

Similarly, FIFO Unpacks take a single value, and break it into a serial stream of smaller values. Again, a good example is reading 32-bit words from memory, every fourth cycle, and producing a series of four 8-bit values from each. Like FIFO Pack, they are used when the computational body cannot make use of the data parallelism of an Unpack. It sends a single “Last_Out” signal when it receives a “Last_In”.

4.10 FIFO_PACK_FULL

This is simply an optimized version of a FIFO Pack, that is guaranteed to have exactly enough data to complete a word. By making this assumption, the node can be optimized. Therefore, some stages of AHA generation/analysis identify this condition and replace the FIFO Pack with a FIFO Pack Full. The general FIFO Pack flushes its buffers when the stream stops short, leaving some of

the bits in the packed array uninitialized. The FIFO Pack Full is simply guaranteed only to receive a Last_In signal with the first piece of data to pack.

4.11 BUFFERX

BufferX nodes are simply X -Deep buffers of data. They have a latency of X , but can reach a steady state. They are used to balance the delay of various paths of data through the program. This allows the program to run full speed without having to wait for data to be consumed. The Last signals are simply buffered up and carried through the queue like normal data.

4.12 DELAY_REGISTER

The Delay Register node is a Semi-Clocked node. It simply delays a value by a single clock cycle. On its first clock cycle after a reset, it produces a single garbage value, followed by its stream of input, offset by one. It is used to store a value into the next iteration. They are used to simulate a sliding window of data, by chaining them together.

4.13 REPLICATOR

Replicators are used to feed constants into a loop. They take an integer **Count** and a small piece of **Data**, and simply outputs the Data, Count times. When it receives a last signal it generates a single dummy value. It does not have a Last_Out signal. The dummy value it generates will be accompanied by a Last_Out signal generated from another node.

4.14 REPLICATOR_EXTRA

This node operates exactly like the Replicator node except that it outputs the value one additional time. This is because a Replicator Extra node is more optimized than putting an incrementor before the Count line. AHA Optimizations identify situations where an incrementor is positioned before the count input of a replicator and replaces it with a replicator extra. This condition happens frequently during DFG to AHA conversion.

4.15 MASK

Mask nodes are used to remove values from a stream of data; “masking them out”. It takes a stream of data - bit pairs as input. The bit is used to remove the data from the stream. When the mask is 1, the data is removed from the stream. Otherwise it is copied to the output. It copies the Last signal with the rest of the data. Values tagged with the Last signal cannot be masked.

4.16 CIRCULATE

Circulate nodes are the AHA equivalent of nextified variables in SA-C. They begin with an initial value, and then are fed a new value via a back-edge. This introduces the ability to have data loops. It simply ignores values with Last_In tags. It does not have any Last_Out signal.

4.17 ROMDEF

RomDef nodes do not generate any behavioral VHDL. They simply cause a static array to be built representing a ROM table. Their only output needs to be attached to a RomRef node in order for it to be useful.

4.18 ROMREF

The RomRef node takes the ROM defined by the RomDef node and synthesizes it. This is handled by the synthesis software. The VHDL itself is simply an index into the static array from the RomDef. The result is a combinational circuit implementing the ROM.

4.19 DONE

The Done node reads a stream of single bits, which are nothing more than Last signals. It ignores 0s, and copies 1s to its output. The 1 represents a “finished” signal streaming out of an inner loop. The 1 that the Done node produces from its output indicates that the loop has completed. It is used to consume the last signals generated in inner loops. It reduces the number of tokens flowing into the outer loop.

4.20 ADDR_CALC

The Addr Calc node is used to efficiently compute the beginning of several evenly spaced address of memory. It accepts a **base** and an **offset**, and produces n values in serial: ($base$, $base + offset$, $base + 2 \cdot offset$, $base + 3 \cdot offset$, $\dots$, $base + n \cdot offset$). It generates a single Dummy value, accompanied by a “Last-Out” when it receives a value with the “Last-In” high.

CAMERON PROJECT: FINAL REPORT

Appendix F: DFG to VHDL Translator

THESIS

DATAFLOW GRAPH TO VHDL TRANSLATION

Submitted by
Monica Chawathe
Department of Computer Science

In partial fulfillment of the requirements
for the Degree of Master of Science
Colorado State University
Fort Collins, Colorado
Summer 2000

Chapter 1

Dataflow Graph to VHDL Translation

Since dataflow graphs represent a non-hierarchical, asynchronous program, it is simple to translate them into electrical circuits (written in VHDL). Each node of the dataflow graph corresponds to an operation to be written in VHDL while the edges correspond to the input and output signals required for these operations. The operations can be either synchronous or asynchronous (purely combinational) internally. Complex operations like generator nodes, reduction nodes, etc. which require access to memory and/or state machines are usually synchronous while simple mathematical or logical operations are asynchronous.

1.1 Abstract Target System

Some of the DFG nodes are part of the loop body while others form the logic for controlling the execution of loop iterations. The translator targets an abstract system that partitions the DFG nodes, as shown in Figure 1.1, into two categories for ease of translation.

- Inner-loop Body (ILB) Nodes

These nodes form the core of the loop-body. They are executed once every loop-iteration. All the arithmetic and logical operators fall under this category. The translator forms one VHDL entity for the inner-loop body by grouping all these nodes together. Since these nodes are purely combinational, they require only one cycle for execution.

- Wrapper Nodes

These nodes form the glue-logic required to handle the loop-iterations. They consist of input nodes, generator nodes, reduction nodes and circulate nodes of the DFG. Since the wrapper nodes are synchronous internally, they may need more than one cycle per loop iteration for execution on the reconfigurable hardware. Handshaking must exist between them to deal with synchronization. The translator handles both the data and handshaking connections between the wrapper nodes and the inner-loop body component.

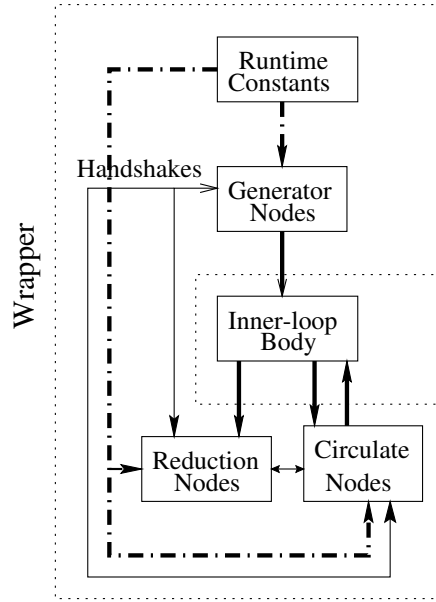


Figure 1.1: System Structure

1.2 Overview of Tasks

The first step performed by the translator is on the DFG as a whole. It identifies VHDL signal names for every port of the DFG nodes. It also determines the format for mapping every DFG node into VHDL. This is followed by partitioning of the DFG based on the abstract target system. The translator then maps the inner loop body into a single ILB component. Since ILB nodes are purely combinational, they are associated only with data connections which can be obtained directly from the DFG. The mapping of the wrapper nodes is more complex as there exists handshaking between various wrapper nodes. For the wrapper nodes, the translator not only handles the data connections

but also handles the loop synchronization, memory accesses and handshake signals. The following sections describe these tasks in detail.

1.3 Signal Mapping

Each edge of the dataflow graph is driven by a unique output port but each output port can drive multiple edges. The translator maps these output ports into unique VHDL signal names and then inserts their names into corresponding target input ports. The identification of unique signal names can be explained with an example. Consider the DFG shown in Figure 2.2. The signal name for the zeroth output port of the UADD node (node number 10) is UADD10OUT0. Similarly the signal name for the zeroth port of the USUM_MANY node (node number 8) is USUM_MANY8OUT0. This mapping is slightly different for the INPUT nodes. Since they always have a single output port, the translator uses only the node number for the signal mapping. The signal name corresponding to the output port of INPUT node (node number 9) is I9. This mapping is followed by their insertion into target input ports. Figure 1.2 shows the UADD node after signal mapping and insertion is complete. The inputs to some ports can be compile time constants. The translator converts these constant, integer values into their binary representation as standard logic vectors.

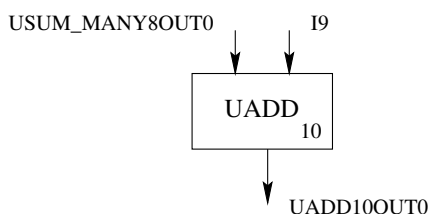


Figure 1.2: Signal Mapping

1.4 Node Mapping

The translator identifies the mapping format for every DFG node. The simple DFG nodes (arithmetic operators) like unsigned add(UADD), signed subtract(ISUM) have a direct mapping to a VHDL statement. The translator simply typecasts the input signals/ standard logic vectors and then performs the operation. Eg. The UADD node shown Figure 1.2 is mapped into

```
UADD100UT0 <= unsigned(USUM_MANY80UT0) + unsigned(I9);
```

The complex nodes are implemented in VHDL as library routines with their component prototype being available to the translator. The prototype has parameters corresponding to the actual input and output ports of the DFG node and also the handshake signals required by that node. The translator uses these available prototypes to map the complex nodes.

1.5 Inner-loop Body (ILB)

After the basic mapping, the translator generates a computation VHDL file corresponding to the inner-loop body. This file describes the electrical circuit for the loop body in VHDL and is independent of the targeted RCS board. It is made up of three parts: the library and package inclusions, the entity declaration and architecture.

1.5.1 Library and Package Inclusions

The translator includes the libraries and packages required for handling the ILB nodes and their signals. These packages can be divided into three categories.

1. IEEE Library Packages (`std_logic_1164`, `std_logic_signed`, `std_logic_unsigned`, `std_logic_arith`)

These standard packages provide routines to handle basic arithmetic and logical operations. The first package handles standard logic vectors (bit vectors). It provides routines that help declaration of signals as standard logic vectors. It also provides functions for bit-level manipulations like width change/ and/ or operations, etc. The second and the third packages provide routines for type-casting of bit vectors into signed or unsigned numbers. The last package provides routines for arithmetic operations.

2. Project Defined Packages (`cammacro`, `camtypes`)

These packages have been designed to simplify the process of generating code for complex ILB nodes like square-root, multiple value add, etc. The first package contains the entity and architecture for the complex ILB nodes. The latter deals with types defined for handling multiple values. Eg. `byte`(8 bits), array of words (multiple 32 bit values), etc.

3. Compile Time Package (parameters)

The translator creates a *parameters* package per DFG. It contains the compile time constants corresponding to that DFG. Eg. Size of a tile to be written

1.5.2 Entity Declaration

The entity declaration deals with the identification of input and output signals for the ILB. This helps in defining the prototype required for connection of the ILB component with the wrapper nodes. The signals of the entity can be split into three categories: run time inputs to the ILB, loop iteration variant inputs and loop outputs. A run-time input is the output of an *input* node of the DFG, acting as an input to an ILB node. The loop iteration variant inputs are the outputs of the generator nodes and circulate nodes. They are called variant inputs because they change every loop iteration. The final category of signals are the outputs of the ILB. These signals act as inputs to the write-tile nodes, value nodes and circulate nodes. The translator maps each run time input to its corresponding edge signal. It maps the multiple input pixels of a generator node into one signal corresponding to the whole window of that node. The translator performs the grouping of the pixels to match the prototype of the generator node which outputs a single signal for the complete window. Similarly, it groups the output data for a write-tile node into one signal corresponding to the tile. This methodology of grouping signals simplifies the wrapper connection as there exists a direct mapping to the prototype of the wrapper nodes.

1.5.3 Architecture

The architecture section of the VHDL file defines the intermediate signals and the actual circuit required for the computation. The translator identifies the intermediate signals as the edges originating from an ILB node and terminating in another ILB node. For the actual circuit design, VHDL supports different styles. The translator selects the style depending upon the node to be mapped.

- **Structural Design:** It uses building blocks to build the circuit. The translator uses this style to map complex nodes because the complex nodes can be implemented independently as building blocks.

- **Dataflow Design:** This refers to a series of equations (VHDL statements) that represent flow of data. These equations are not placed in any particular order as they are all executed concurrently. This style is used in the overall mapping of the DFG by the translator. The order in which the nodes are translated is immaterial because of concurrent execution of the statements.
- **Behavioral Design:** This style is used to describe the behavior of a building block. It usually contains a series of sequential statements. The implementation of the complex nodes may use this design style but the translator does not use it because the DFG representation does not contain information about timing between nodes.

The translator categorizes the ILB nodes into the following categories: arithmetic (monadic and dyadic operators), bit nodes, selector node, multi-input arithmetic nodes and multi-input comparison nodes. The translator also performs special tasks for the generator and write-tile nodes. It splits the concatenated window signal arriving from the generator node into corresponding edge signals (pixel values). It also groups the appropriate edge signals (pixels) to obtain a tile signal for the write-tile node.

1.5.3.1 Arithmetic Nodes

These nodes have one or two inputs and a single output. Since most of the nodes are simple nodes, they are directly mapped into a single VHDL statement. The exception to this rule is the square-root node. Since it is a complex node, it is transformed by using a black-box with prototype shown in Table 1.1.

	Signal Name	Signal Type	Signal Size
Input	Val	Bit vector	2y
Output	Result	Bit vector	y

Table 1.1: Prototype for Square-Root of a Bit Vector of Size $2y$

1.5.3.2 Bit Nodes

The bit-concat, change-width, change-width-se DFG nodes have a direct mapping to the concat operator (&) and the change width operators like EXT, SXT respectively. The translator performs a simple node mapping to obtain the resultant VHDL statement. During the mapping of the change width nodes, the translator needs to handle exception cases where the input and/or the output size is one bit. The bit-select and the shift operators are a bit more complex but can still be transformed into a single VHDL statement as shown in Table 1.2.

Node Type	Inputs	VHDL Description
BIT-SELECT	lb, ub, val	result <= val(ub downto lb);
L-SHIFT	val, shift	This operation occurs by performing bit-select operation followed by concatenating zeros on the right side.
R-SHIFT	val, shift	This operation occurs by performing bit-select operation followed by concatenating zeros on the left side.

Table 1.2: Mapping of Complex Bit Nodes

1.5.3.3 Selector Node

The selector node is transformed into a multiplexer circuit. The translator uses the *when-use* statement in VHDL for this mapping. The translation is shown in Figure 1.3. The result is assigned one of the input values depending upon the matching key. There exists a default value in case of no match.

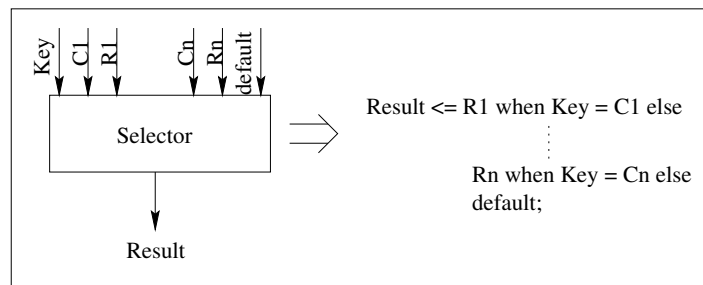


Figure 1.3: Mapping of Selector Nodes

1.5.3.4 Multi-Input Arithmetic Nodes

These nodes have a varying number of input values. Each value is associated with a boolean mask. These nodes are complex and the translator has a prototype available for them. The prototype shown in Table 1.3 is the same for every operation but the black-box mapped is different (depending on the operation: sum, max, min, etc). Since the DFG node has multiple input values, the translator maps them into a single input signal (corresponding to the prototype) by concatenating them. It also performs concatenation of the mask bits. The translator also checks whether the mask for each input is a true. If so, it performs an optimization by calling a routine which performs the same operation without a mask. This decreases the logic and space required on the board for the operation.

	Signal Name	Signal Type	Signal Size	Description
Generic	nVal	Natural number	-	Number of input values
	insize	Natural number	-	Size of each input
	outsize	Natural number	-	Size of the output
Input	Val	Bit vector	nVal*insize	Concatenated Values
	mask	Bit vector	nvals	Concatenated Mask
Output	Result	Bit vector	outsize	Output Value

Table 1.3: Prototype for Multi-Input Arithmetic Nodes

1.5.3.5 Multi-Input Comparison Nodes

These nodes have a varying number of values to be compared with each other. Each value is associated with a boolean mask and a set of values to be captured. Thus the node is made up of multiple input clusters, each cluster having a comparison value, a mask and a set of captured values. The output of this node is the set of captured values corresponding to the comparison value selected by the operation (min/max at first/last). The translator has a prototype corresponding to the operations as shown in Table 1.3. The translator concatenates the comparison value and mask of each cluster into respective bit vectors. It also concatenates the set of captured values of all the clusters into one bit vector. Currently the implementation of the black-box works only if the size of the captured values is the same as the comparison values.

	Signal Name	Signal Type	Signal Size	Description
Generic	nVal	Natural number	-	Number of clusters
	nCapVal	Natural number	-	Number of captured values per cluster
	insize	Natural number	-	Size of input values
Input	CompareVal	Bit vector	nVal*insize	Concatenated comparison values
	CapturedVal	Bit vector	nVal*nCapVal*insize	Concatenated sets of captured values
	mask	Bit vector	nVal	Concatenated mask
Output	ResultOut	Bit Vector	nCapVal*insize	Concatenated captured value

Table 1.4: Prototype for Multi-Input Comparison Nodes

1.6 Wrapper

The translator also generates VHDL code for handling the loop iterations and run-time constants. This glue-logic is connected to the ILB entity to complete the translation of the DFG. The connections among the wrapper nodes are partially dependent on the targeted reconfigurable board. This is because some boards have multiple FPGA chips and/ or multiple memories per chip. For boards with multiple FPGAs, the translator can partition the wrapper system and place the subparts on different FPGAs. The position of the partition can give rise to different target models.

1.6.1 Target System

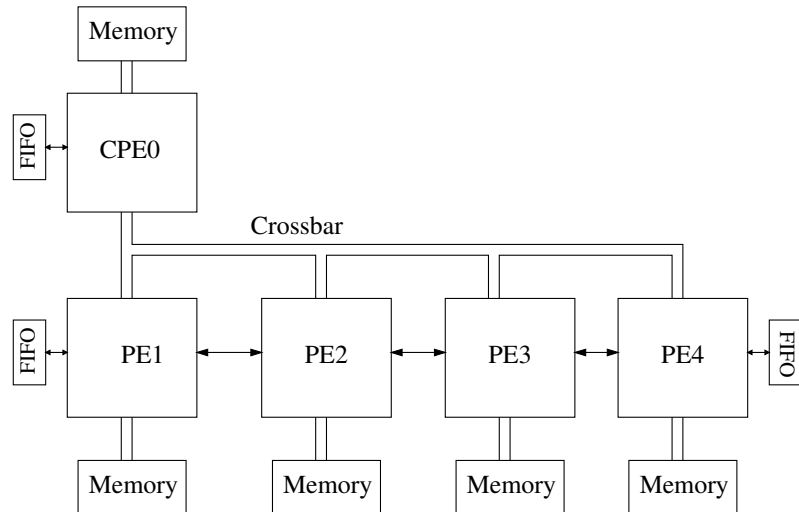


Figure 1.4: Wildforce Board

The translator currently produces code for the Wildforce board. This board, as shown in Figure 1.4, consists of five Xilinx 4036XL FPGAs, such that CPE0 can broadcast data to the other four FPGAs via a 36-bit crossbar. Each processing element (PE) has its own memory organised as 32-bit words. CPE0, PE1 and PE4 also have 36-bit FIFOs available for their use. The host machine can communicate with the Wildforce board through memory, FIFOs or communication signals like reset and interrupt lines. The DFG can be mapped onto the Wildforce board such that it uses one or more PEs with their memories and/or FIFOs. This gives rise to multiple models for the target system.

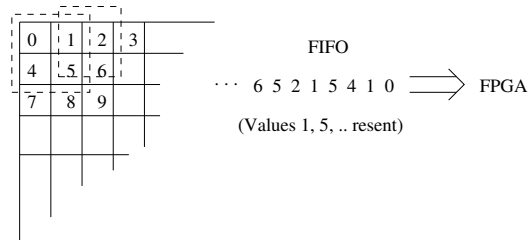


Figure 1.5: 2x2 Sliding Window Data Stream

The first option uses FIFOs to stream the data in and out of the board. This model requires only the ILB to be present on the FPGA because the host streams the input windows through the FIFOs and also collects the output of the ILB. This model was discarded because the host needed to send the same data pixels multiple times through the FIFO (as shown in Figure 1.5). Also, the transfer rate through the FIFO was lower than DMA transfer into memory. Hence the option of using memory for transfer of input and output images was explored. Currently the translator generates code for two target models. The models are shown in Figure 1.6.

- OnePE Model

The model uses only one FPGA (CPE0) and its local memory. The other four PEs and their memories are unused. This model is simple and moving it to another board involves no changes in the translator. Since the model uses only one memory, this shared resource can act as a bottleneck. In order to utilize all the memories, one can map the same configuration on each PE with different chunks of images in their memories. This can increase the total throughput.

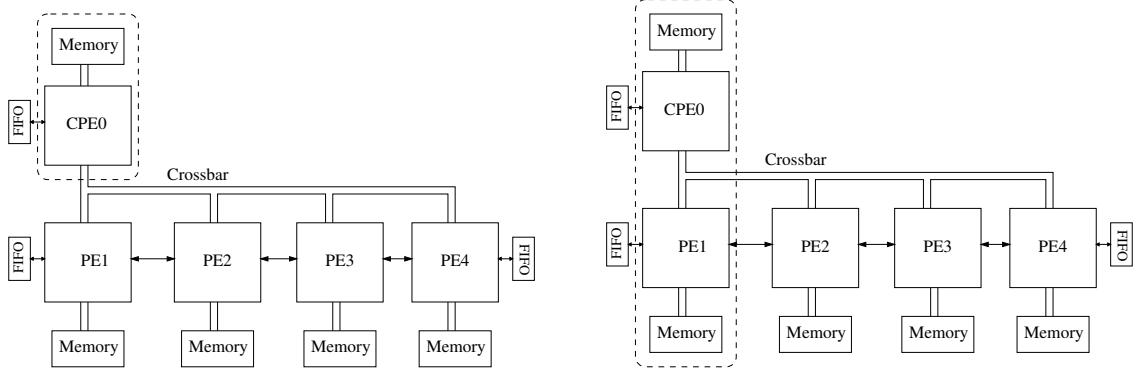


Figure 1.6: (a) OnePE Model (b) TwoPE Model

- TwoPE Model

The model uses two FPGAs (CPE0 and PE1) connected via a crossbar. The memory of CPE0 is used as the input memory while the PE1 memory is used as the output memory. This gives more memory bandwidth as there is no contention between inputs and outputs for access to the memory.

1.7 Wrapper Mapping for OnePE Model

Since a single FPGA is used in this model, the translator produces one VHDL wrapper file. Each node is mapped to its corresponding VHDL entity and the glue logic for loop synchronization and handshaking is generated. The translator includes all the libraries and packages included for the ILB ((`std_logic_1164`, `std_logic_signed`, `std_logic_unsigned`, `std_logic_arith`, `cammacro`, `camtypes`, `parameters`). It also includes another project defined library package (*onepelib*). This package contains the entity and architecture for the wrapper nodes implemented for the OnePE model [2]. The library inclusions are followed by an entity declaration for the whole system. This consists of the FPGA chip signals like clock, reset, interrupt request and acknowledge lines and memory signals. The generation of the architecture is more complex and is split into subsections for ease of translation: loop iteration synchronization, handshaking between the wrapper nodes, memory access arbitration, parameter mapping and node mapping.

1.7.1 Loop Iteration Synchronization

The loop-variant input of the ILB arrives from the generator nodes and circulate nodes. For different loop iterations, generator nodes may take different number of clock cycles before their data is available for ILB use. This is because each generator node may require different number of memory accesses to read all the pixels corresponding to the current iteration. Every generator and circular node must wait for all the input data to be available before transferring it into the ILB to prevent computation of garbage values in the intermediate cycles. Also since the ILB is purely combinational, all the outputs of the ILB are available in the next clock cycle for consumption by the write-tile nodes, value nodes and/or circulate nodes. These nodes must consume the values in the cycle they are produced because the ILB (being purely combinational) has no buffers to hold them. Since they internally hold the ILB output in buffers before transferring it to memory, they are not ready to consume the output of the ILB if their buffers are full. This implies that the transfer of the input data into the ILB must happen only if all the consumer nodes have buffer space to hold the ILB output.

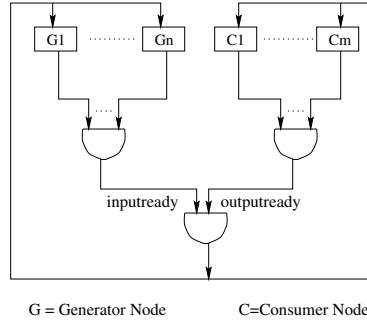


Figure 1.7: Loop Synchronization for OnePE Model

The translator uses two bit vectors: one to handle the synchronization of nodes generating loop variant input for the ILB and one to handle the synchronization of nodes consuming output of the ILB. In the first bit vector, a set bit implies that the node corresponding to that bit position has its data available for the ILB. By ANDing all the bits, one obtains an *inputready* signal which tells when all the inputs for the loop iteration are available. Similarly the latter bit vector provides the *outputready* signal which tells when all the consumers nodes are ready to handle the output of the

ILB. When both *inputready* and *outputready* signals are set, the loop iteration gets executed and all nodes prepare for the next iteration. The loop synchronization is shown in Figure 1.7.

1.7.2 Handshake Signals

In order to handle the execution of the loop, some nodes need information that can be provided by other nodes. This information is obtained via handshake signals. Eg. end of row (EOR), end of file (EOF), *done*, *final*, etc. The generator nodes know when the end of row (EOR) and/or end of file (EOF) conditions occur for their image. The reduction nodes need to know when these conditions occur in order to increment their row counters. The translator allocates one bit for EOR and one bit for EOF signal per image/ generator node. The translator ANDs these bits to obtain the end of row and end of file signals used by the reduction nodes. Similarly, the memory arbitrator needs to know when all the loop iterations are complete. This is handled by having one bit per reduction node. This bit is set when the node empties its buffers after receiving the EOF signal. These bits are anded to obtain the *done* signal for the memory arbitrator. Also, the memory arbitrator is responsible for giving a *ready* signal to the wrapper nodes indicating the start of the loop iterations. The write-scalar node needs to know when the final value to be written is available. This signal is given by the circulate node when all its iterations are completed.

1.7.3 Run-Time Constants and Memory Access

Since multiple nodes may require access to memory in the same cycle, the OnePE model uses a memory arbitrator entity which dynamically determines which node gets access to memory in the current cycle. The nodes requiring access to memory are *input* DFG nodes, generator nodes and reduction nodes. The *input* DFG nodes require access to memory only once to obtain the value of the corresponding run-time parameter. Hence the memory arbitrator obtains all these values once, sets a *ready* line and then handles the requests from the generator and reduction nodes. It arbitrates memory until it receives a *done* signal informing it that all iterations are complete.

To handle the run-time constants, the translator provides the memory arbitrator with an array of memory locations (one for each run-time constant). The memory arbitrator outputs an array of

the run-time constants read from the memory. The translator maps these values read to the VHDL signal (*Id*) of the corresponding *input* DFG node.

The generator and reduction nodes communicate with memory via the memory arbitrator using a set of signals. The set of signals associated with each node contain the corresponding image *base address*, number of *rows* and *words* of that image. This mapping of image signals is explained in the Image Parameter Mapping section. The set also contains a *request* line (to request access to memory), a *write* line (to specify read or write access), the *row* and *word* to be accessed from the corresponding image and an *acknowledge* line (to tell the node when the request has been processed). It also contains lines for *DataIn* (data read) and *DataOut* (data to be written).

The prototype available to the translator for the memory arbitrator is shown in Table 1.5. It contains signals to handle requests from the generators and reduction nodes. It also contains signals to obtain the run-time constants from memory. Finally, it contains signals that connect directly to the top-level wrapper entity chip signals and two signals used for handshaking.

	Signal Name	Signal Type	Signal Size	Description
Generic	NumConstants	Natural Number	-	Number of runtime constants
	NumIOPorts	Natural Number	-	Number of nodes requiring memory access
Chip Signals	-	-	-	Clock, Reset, Interrupt and memory lines
Run-time Constants	MA_AddrConst	Address Array	NumConstants	Memory locations
	MA_Constants	Array of Words	NumConstants	Corresponding Values
Generator and Consumers	MA_Base	Array of Words	NumIOPorts	Image Base Address
	MA_NumRows	Array of Words	NumIOPorts	Image Rows
	MA_NumWords	Array of Words	NumIOPorts	Image Words
	MA_Req	Array of Bits	NumIOPorts	Request lines
	MA_Ack	Array of Bits	NumIOPorts	Acknowledge lines
	MA_Write	Array of Bits	NumIOPorts	Read/Write lines
	MA_Row	Array of Bits	NumIOPorts	Requested Row
	MA_Word	Array of Bits	NumIOPorts	Requested Word
	MA_DataIn	Array of Words	NumIOPorts	Data Read
	MA_DataOut	Array of Words	NumIOPorts	Data to be written
HandShake	MA_Ready	Bit	1	An output signal set when run-time constants read
	Done	Bit	1	Input signal informing when all iterations complete

Table 1.5: Prototype for Memory Arbitrator in OnePE Model

1.7.4 Compile Time Constants

In order to simplify the generator and write-tile nodes, the translator maps compile-time constants related to these nodes into a library package called *parameters*. For each generator node, the translator defines constants for its window size (rows and columns), step size (in row and column dimension) and data size. These constants are mapped into signal names as WinRows, WinCols, WinRowStep, WinColStep, DataSize. To make the names unique, the node number is appended to each signal name. For each write-tile node, the translator defines constants for the tile size (rows and columns) and data size. It also declares constants for total number of generator nodes, total number of reduction nodes, total number of nodes providing inputs to the ILB and total number of nodes consuming the output of the ILB.

1.7.5 Image Parameter Mapping

The translator needs to map the parameters for each image corresponding to a generator or a reduction node. The base address for every image maps directly from a signal on its corresponding input port. The mapping of rows and columns (in number of pixels) is also direct for two-dimensional generator nodes. For one-dimensional generator nodes, the row maps to a constant value one while the column maps directly from an input port signal. For the value nodes and write scalar nodes, only a single output value exist. This implies that the rows and columns are mapped to constant value one. This mapping is summarised in Table 1.6.

Node Type	Base Address	Rows	Columns
1D Element Generators	Port 0	Constant 1	Port 2
2D Element Generators	Port 0	Port 2	Port 4
1D Window Generators	Port 0	Constant 1	Port 3
2D Window Generators	Port 0	Port 3	Port 6
Values	Port 2	Constant 1	Constant 1
Write-Scalar	Port 1	Constant 1	Constant 1

Table 1.6: Image Parameter Mapping for Generator, Write-Scalar and Value Nodes

The mapping for the write-tile nodes is more complicated for the rows and columns of images. This is because the run-time constants give the number of tiles to be written in each dimension and not the number of rows and columns. The number of tiles to be written can be obtained from the signal

corresponding to a fixed input port. The number of rows and columns (in number of pixels) is obtained by direct multiplication of tile size and the number of tiles to be written. This calculation is summarised in Table 1.7. In the table, a write-tile node of the format is xD_yD where x is the for-loop rank and y is the tile dimension.

Write Tile Type	Base Address	Rows	Columns
1D_1D	Port $(n - 3)$	Constant 1	Port $(n - 2)*ColTileSize$
1D_2D	Port $(n - 4)$	RowTileSize	Port $(n - 3)*ColTileSize$
2D_1D	Port $(n - 4)$	Port $(n - 3)$	Port $(n - 2)*ColTileSize$
2D_2D	Port $(n - 5)$	Port $(n - 4)*RowTileSize$	Port $(n - 3)*ColTileSize$

Table 1.7: Image Parameter Mapping for Write-Tile Nodes (n being total number of input ports)

One thing to be noted is that the image is stored in memory as words and not as pixels. Hence one needs to map the number of pixels (in column dimension) into corresponding number of words. For write-scalar and value nodes the number of words is constant value one. For the generator and write-tile nodes, the translator calculates the number of words by using the data size of each pixel and number of columns. For 32-bit pixels, the number of words is equal to the number of columns. For all other data sizes, the translator splits the bit-vector corresponding to number of columns into two parts: most significant part (MSP) corresponding to number of words and least significant part (LSP) corresponding to the pixel number inside the word. The size (x) of the LSP is obtained from by solving $2^x * datasize = 32$ (size of a word). It can be noted that the number of words is equal to the MSP when LSP is equal to zero and is equal to MSP+1 otherwise. The translator generates the VHDL code to perform this selection.

1.7.6 Node Mapping

The wrapper node mapping involves data mapping and handshake signal mapping for every node. Data mapping is straight-forward because there is a direct mapping to the run-time constants and/or ILB entity inputs and outputs. This mapping can be directly obtained from the DFG. The mapping of handshake signals involves selection of the entry (corresponding to the node) from an array of signals and then mapping it. Table 1.8 shows the prototype and its mapping for a generator node while Table 1.9 shows the prototype and its mapping for a write-tile node.

Signal Type	Signal Names	Mapping
Generics	DataSize, Win_Width, Wind_Height, Stride_X, Stride_Y	Constants defined in the <i>parameters</i> library
MemArb Signals	Row, Word, DataOut, DataIn Req, Ack, Write	Entry corresponding to current node in respective MemArb signal arrays
Image Signals	Rows, Columns, Words	Corresponding entries in the image parameter arrays
Handshake Signals	Ready	<i>Ready</i> signal from MemArb (input)
	Store	Corresponding entry in <i>inputready</i> array (output)
	ILB_Ack	Anded signal of <i>inputready</i> and <i>outputready</i> (input)
	EOR, EOF	Corresponding entry in EOR and EOF array (output)
ILB input	ILB_Data	Input to the ILB from this node
Chip Signals	Clock	Wrapper entity clock

Table 1.8: Prototype and Mapping for Generators in OnePE Model

Signal Type	Signal Names	Mapping
Generics	DataSize, Tile Size(row+column)	Constants defined in the <i>parameters</i> library
MemArb Signals	Row, Word, DataOut, DataIn Req, Ack, Write	Entry corresponding to current node in respective MemArb signal arrays
Image Signals	Rows, Columns, Words	Corresponding entries in the image parameter arrays
Handshakes	Ready	<i>Ready</i> signal from MemArb (input)
	Store	Anded signal of <i>inputready</i> and <i>outputready</i> (input)
	ILB_Ready	Corresponding entry in <i>outputready</i> array (output)
	EOR, EOF	Anded EOR and EOF signal(input)
	Done	Corresponding entry in <i>Done</i> array (output)
ILB output	ILB_Data	Output from the ILB for this node
Chip Signals	Clock	Wrapper entity clock

Table 1.9: Prototype and Mapping for Write-Tile Nodes in OnePE Model

Table 1.10 shows the prototype and its mapping for a circulate node. The circulate node provides input for the ILB and also consumes the output of the ILB. Hence it outputs two handshake signals: one for the *inputready* array and the other for the *outputready* array. It also receives the two *take* signals: *TOP_Take* and *BOT_Take*. The former signal indicates when the input to the ILB is to be generated for the current iteration while the latter signal indicates when the output produced by the ILB is to be consumed. Currently the signals are identical (and equal to the ANDed value of *inputready* and *outputready*). This is because the ILB is purely combinational, making all wrapper nodes to execute in the same clock cycle. When the ILB becomes multiple clock cycle, the consumer

nodes will execute a few cycles after the nodes generating input for the ILB, thus making TOP_Take to differ from BOT_Take. The circulate node may feed its final value into a write-scalar node. The write-scalar node thus writes a single value back into memory. Table 1.11 shows the prototype and its mapping for this node.

Signal Type	Signal Names	Mapping
Handshakes	Ready	Ready signal from MemArb
	TOP_Ready	Corresponding entry in <i>inputready</i> array (output)
	TOP_Take	Anded signal of <i>inputready</i> and <i>outputready</i> (input)
	BOT_Ready	Corresponding entry in <i>outputready</i> array (output)
	BOT_Take	Anded signal of <i>inputready</i> and <i>outputready</i> (input)
	Final	Output for the write-scalar node
Data	Data_Init	Port 1 signal
	Iters	Port 2 signal
	Data_In, Data_Out	Input and Output from ILB
Chip Signals	Clock	Wrapper entity clock

Table 1.10: Prototype and Mapping for Circulate Node in OnePE Model

Signal Type	Signal Names	Mapping
MemArb Signals	Row, Word, DataOut, DataIn Req, Ack, Write	Entry corresponding to current node in respective MemArb signal arrays
Image Signals	Rows, Columns, Words	Corresponding entries in respective arrays
Handshakes	Ready	<i>Ready</i> signal from MemArb
	Store	<i>Final</i> signal from a circulate node
	Done	Corresponding entry in Done array (output)
Data	ILB_Data	Signal on input port0
Chip Signals	Clock	Wrapper entity clock

Table 1.11: Prototype and Mapping for Write-Scalar Node in OnePE Model

The value nodes also write a single value into memory at the end of all iterations. The value node gets an ILB data every loop iteration. It performs a given operation on this stream of data when the corresponding mask is true. On completion of all loop iterations the final value is written into memory. The mapping on this node is shown in Figure 1.8. It gets split into four parts: the OP node that perform the actual arithmetic operation, the MUX node which selects whether the newly computed value is to be selected or not (depending on the current mask value), the register node which initializes the intermediate result and holds it for *iter* iterations and the write-scalar node which write the final value into memory. The OP node which performs the arithmetic operation

(like signed/ unsigned dyadic add, minimum, maximum) is mapped into a single VHDL command. Also, this updated result is selected only when the incoming mask is true. This selection is mapped into a MUX by using the *when-else* statement of VHDL. The intermediate result is stored in a register. For loop synchronization, the register sets the bit corresponding to the value node in the *outputready* array when it is ready for the next iteration. It receives the ANDed value of *inputready* and *outputready* which tells when the next iteration starts. Also, the initialization of the intermediate result occurs in this register node. Nodes like *add* and *or* require initialization to value zero while *and* nodes get initialized to one. The register node updates its value for *iter* iterations. It then sets a *final* signal which is used by the write-scalar node to write the current DataOut value. The write-scalar node uses the memory arbitrator and image signals corresponding to the value node to get memory access. It also sets the bit corresponding to the value node in the *done* array after it has completed writing the value into memory. The ILB data for the OP node, the mask, *iter* is obtained directly from the DFG. The mapping of the memory arbitrator signals, image signals and handshake signals require obtaining the entry corresponding to the value node in their respective arrays. This completes the mapping of the value node.

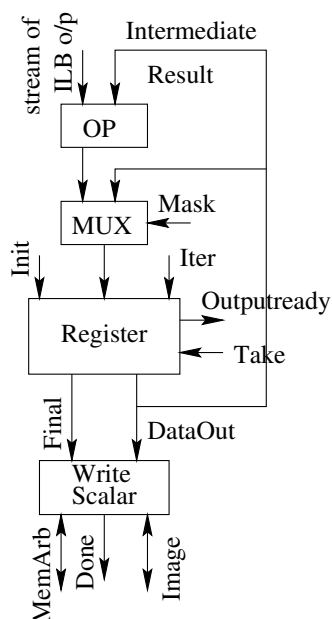


Figure 1.8: Mapping of value node in OnePE Model

1.8 Wrapper Mapping for TwoPE Model

Since the model uses two FPGAs, the translator generates two VHDL files (one file per FPGA). The model uses the CPE0 memory as its input memory and the PE1 memory as its output memory. All the nodes requiring access to input memory (*input* nodes and generator nodes) are mapped on CPE0 while the ILB, reduction nodes and circulate nodes are mapped on PE1. Also, the data being read in CPE0 is required by nodes mapped on PE1. This data is therefore transferred across the crossbar. There exists an entity which grabs the data arriving on the crossbar and distributes it to the run-time constant signals or the signals connected to the ILB. The translator groups all the *input* nodes and maps them into a single *constgrabber* entity which is placed on CPE0. It also groups the generator nodes and maps them into a *read* entity for CPE0 and a *distribute* entity for PE1. CPE0 handles only the *constgrabber* and *read* entity while PE1 deals with the *distribute* entity, memory arbitration for multiple output requests, handshake signals as well as node mapping for circulate and reduction nodes.

1.8.1 Loop Synchronization

Since the system is partitioned on to two FPGAs, direct handshaking between all the nodes (without cycle delays) is not possible. Hence, the translator performs some static analysis on the loop generators, reduction nodes and the actual ILB. Since the ILB is purely combinational, it takes only one cycle for execution. The translator needs to evaluate the number of cycles per loop iteration that are required by the generator nodes to make their loop-invariant ILB input available. Since they all access only one memory, the number of cycles for all the ILB loop-invariant inputs to become available is the summation of the cycles for each generator. The translator also finds the number of cycles required by the reduction nodes to clear their word-size buffers. The cycles required for all the output buffers to be cleared is the summation of cycles of each reduction node because they too access one memory (PE1s output memory). The generators need to slow themselves down so that the reduction nodes have enough time per loop-iteration to empty their buffers.

The number of cycles required by the generators may differ per loop iteration due to start of row conditions, window step sizes, etc. This is because partial data corresponding to the current iteration window may have been read in the previous iteration as shown in Figure 1.9. Similarly the cycles required by reduction nodes also vary because every node may not empty its buffer in each loop iteration. It may wait to fill its buffers completely before emptying them.

The worst case for any loop iteration will happen when the window generators require the minimum number of cycles (g_{min}) while the reduction nodes need the maximum number of cycles (r_{max}). In order to slow down the generators correctly, they must wait for at least $(r_{max} - g_{min})$ cycles after gathering their own data. The translator evaluates g_{min} and r_{max} by doing static analysis of the DFG.

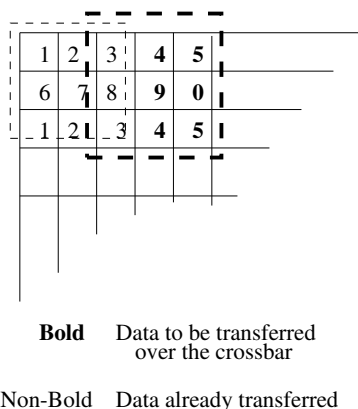


Figure 1.9: A 3x3 window of step 1x2

The minimum cycles for data availability happen when the data for the window has been prefetched from the memory for the generator node and it only needs to be transferred across the crossbar in words. The window/ data is transferred in a column major order to simplify the implementation of sliding the window across the image. The translator evaluates the number of cycles required to transmit one column of the window across the crossbar. Since the crossbar can transfer a word per cycle, the number of cycles required is equal to the number of words required to hold the pixels corresponding to a window column.

$$CyclesPerColumn[i] = \lceil (WindowRowSize[i] * DataSize[i]/32) \rceil$$

Since a window could have been sent in the previous iteration, partial data for the current iteration may be present on PE1. The minimum amount of data yet to be transferred is the 'column step size' number of columns as shown in Figure 1.9. Thus the minimum number of cycles for the generator nodes can be calculated by summation over all generator nodes.

$$g_{min} = \sum(CyclesPerColumn[i] * ColStepSize[i])$$

Similarly, r_{max} can be calculated by adding the maximum cycles required by all the write-tile nodes. The value nodes and write-scalar nodes require to empty their buffers only after the end of iterations and not at intermediate iterations like a write-tile node. Hence they are not used in the calculation of r_{max} . Since a write-tile node stores the output image in the row-major order, the translator evaluates the number of cycles required to store a row of the tile. Since memory is organised as 32-bit words, this is equal to the number of words that can hold a row of the tile.

$$CyclesPerRow[i] = \lceil (ColTileSize[i] * DataSize[i] / 32) \rceil$$

The total number of rows that the write-tile node needs to store is equal to the number of rows in the tile. Thus the formula for evaluating r_{max} can be summarised as follows:

$$r_{max} = \sum(CyclesPerRow[i] * RowTileSize[i])$$

If the evaluated r_{max} is equal to zero, it implies that there are no write-tile nodes. Since the ILB and other computations like circulate nodes need atleast one cycle to complete their execution, r_{max} is set to one. This completes the static loop synchronization analysis.

To handle the actual synchronization, the translator provides the *read* entity with the r_{max} and g_{min} values. Every iteration, the *read* entity counts the number of cycles it required to generate the data for the ILB. If this *cycle* value does not exceed the value of r_{max} , the *read* entity waits for $(r_{max} - cycle)$ cycles. This guarantees that the reduction nodes have enough number of cycles to complete emptying of their buffer for the current iteration.

1.8.2 Compile Time Constants

In order to simplify the translation for generator and write-tile nodes, the translator maps compile-time constants related to these nodes into a library package called *parameters*. This is similar to

the mapping for the OnePE Model. For each generator node, the translator defines constants for its window size (rows and columns), step size (in row and column dimension) and data size. For each write-tile node, the translator defines constants for the tile size (rows and columns) and data size. To handle multiple generators, the translator defines a subtype: a two-dimensional array which can hold the window of one generator node. It then declares a type *InBuff2D* which is an array of the window subtype. This final type is used to declare a signal to hold the windows of all the generators. The use of this type gives rise to multiple windows of the same window sizes. This works because currently the TwoPE model implementation of the *read* entity only handles identical multiple windows.

1.8.3 CPE0 Entity and Architecture

The translator includes all the libraries and packages included in the ILB. The library inclusions are followed by an entity declaration for the CPE0 system. This consists of the FPGA chip signals like clock, reset, interrupt request and acknowledge, crossbar and memory signals. It also gets a handshake signal from PE1. This signal is used to inform CPE0 when PE1 is ready to the start. The same signal is also used to inform CPE0 that PE1 has completed the tasks related to all the loop iterations. CPE0, on getting the second handshake, sends an interrupt to the host to indicate completion of execution.

The entity declaration is followed by the architecture which consists of a *constgrabber*, a *read* entity and some handshake signals between them. Handshaking is required between the nodes because the *read* entity must not begin its execution until all the run-time constants have been read by the *constantgrabber*. Although both the entities need access to memory (though not at the same time), only one of them is connected to the higher level CPE0 entity to prevent multiple drivers for the memory signals. The *constgrabber* is the entity that gets connected. The *constgrabber* receives the memory signals of the *read* entity. After reading the run-time constants, the *constgrabber* entity just passes the memory signals received from the *read* entity to the top-level entity signals to facilitate the memory arbitration.

To handle the *input nodes*, the translator provides the *constgrabber* with an array of memory locations (one for each run-time constant). The *constgrabber* outputs an array of the run-time constants read from the memory. The translator maps these read values to the output signal (*Id*) of their corresponding *input* DFG node. The translator then maps the *constgrabber* entity according to the prototype shown in Table 1.12.

	Signal Name	Signal Type	Signal Size	Description
Generic	nconstants	Natural Number	-	Number of runtime constants
Top-level Signals	-	-	-	Clock, Reset, memory lines
Passing Signals	-	-	-	Memory lines from <i>read</i> that pass through to top-level
Run-time Constants	Constaddrs	Address Array	NumConstants	Memory locations
	consts	Array of Words	NumConstants	Corresponding Values
HandShake	MemReady	Bit	1	An output signal set when run-time constants read

Table 1.12: Prototype for Constgrabber

The translator then maps all the generators nodes into a single *read* entity on CPE0. This entity is given the window size, step size and datasize for a generator node by using the *parameters* library. It also receives arrays of image parameters: one each for source address, number of rows and number of columns of the corresponding image. This image parameter mapping for the generator nodes is identical to the OnePE model and is shown in Table 1.6. The *read* entity acts as the arbitrator for the use of the crossbar. While transferring data (run time constants and/ or window pixels), the *read* entity tags the data. These bits are used to send PE1 handshake signals for the *distribute* entity (on PE1). It receives the array containing the run-time constant values from the *constgrabber*. The entity first transfers all the run-time constants across the crossbar before it starts the execution of loop iterations.

To obtain access to memory, the *read* entity is connected to the pass memory signals of the *constgrabber*. Also, it is connected to the top-level entity signals like clock, reset, crossbar, interrupt lines and handshake signal from PE1. It also receives the maximum number of cycles that are required to complete the emptying of buffers by the reduction nodes. It uses this value to slow the generator nodes. The translator maps the *read* entity according to the prototype shown in Table 1.13.

	Signal Name	Signal Type	Signal Size	Description
Generic	NArgs	Natural Number	-	Number of runtime constants
	Generator Size	-	-	Row, Cols, RowStep, ColStep, DataSize
	WriteCycles	Natural Number	-	Maximum number of cycles reqd for writes per iteration
TopLevelSignal	-	-	-	Clock, Reset, interrupt lines, crossbar, PE1 handshake
PassingSignal	-	-	-	Memory lines to <i>read</i> entity
RunTimeConsts	Args	Array of Words	NArgs	Corresponding Values
ImageParams	-	Multiple Arrays	Number of generators	Source address, rows, cols of corresponding images
HandShake	MemReady	Bit	1	Input from <i>read</i> , set when run-time consts read

Table 1.13: Prototype for Read Entity

1.8.4 PE1 Entity and Architecture

The translator includes all the libraries and packages included in the ILB. It also includes the *values* and *nextified* packages which contain routines that implement the value nodes and circulate nodes for the TwoPE system [1]. The library inclusions is followed by an entity declaration for the PE1 system. This consists of the FPGA chip signals and a handshake signal for informing CPE0 when PE1 is ready to the start and also when PE1 has completed the tasks related to all the loop iterations. The entity declaration is followed by the architecture mapping. This can be split into three subparts: memory arbitration, handshaking and run-time constant mapping, node mapping.

1.8.4.1 Memory Arbitration

Since multiple reduction nodes may require access to the output memory in the same cycle, the TwoPE model uses a memory arbitrator entity. The memory arbitrator is also responsible for setting the handshake signal for CPE0 informing it that PE1 has access to memory and that the loop iterations can start.

The reduction nodes communicate with memory via the memory arbitrator using a set of signals (similar to the OnePE model). The set of signals associated with each node contain the corresponding image *base address*, number of *words* of that image. The set also contains a *request* line, a *write* line, the *row* and *word* to be accessed from the corresponding image and an *acknowledge* line. It also contains lines for *DataIn* (data read) and *DataOut* (data to be written).

The prototype available to the translator for the memory arbitrator is shown in Table 1.14. It contains signals to handle requests from the reduction nodes. Finally, it contains signals that connect directly to the top-level wrapper entity chip signals and one signal for handshaking with CPE0.

	Signal Name	Signal Type	Signal Size	Description
Generic	NumPorts	Natural Number	-	Number of nodes requiring memory access
Chip Signals	-	-	-	Clock, Reset and memory lines
Reduction Nodes	MA_Base	Array of Words	NumIOPorts	Image Base Address
	MA_NumWords	Array of Words	NumIOPorts	Image Words
	MA_Req	Array of Bits	NumIOPorts	Request lines
	MA_Ack	Array of Bits	NumIOPorts	Acknowledge lines
	MA_Write	Array of Bits	NumIOPorts	Read/Write lines
	MA_Row	Array of Bits	NumIOPorts	Requested Row
	MA_Word	Array of Bits	NumIOPorts	Requested Word
	MA_DataIn	Array of Words	NumIOPorts	Data Read
	MA_DataOut	Array of Words	NumIOPorts	Data to be written
HandShake	MA_Ready	Bit	1	CPE0 Handshake (Output)

Table 1.14: Prototype for Memory Arbitrator in TwoPE Model

1.8.4.2 Handshaking and Runtime Constant Mapping

The translator maps the array of run-time constants obtained by the *distribute* entity into the output signals (*Id*) of the respective *input* DFG nodes. This mapping is identical to the one in CPE0. The translator also handles the grouping of various signals to obtain the handshakes between various nodes. One handshake that needs to be handled is the *done* signal that acts as a handshake to the CPE0. This handshake is made up of two parts: the *ready* signal from the memory arbitrator and the *done* signal from the *distribute* entity. The latter signal indicates when all the loop iterations are complete. Another handshake signal that the translator must handle exists between the reduction nodes and the *distribute* entity. The translator uses a bit vector with one bit allocated per reduction node. If this bit is set, it implies that the reduction node has not completed emptying its buffers. The translator ors these bits to obtain an *othersbusy* signal. This signal acts as an input to the *distribute* entity preventing it from setting the *done* signal until all buffers have been emptied. The translator also maps the tag bits of the crossbar data into their respective handshake signals. Bit 34 corresponds to StopProcess (iterations complete) signal. This signal acts as an input to the value nodes.

1.8.4.3 Node Mapping

The node mapping for the TwoPE system is fairly simple in comparison to the OnePE model. The translator maps all the generator nodes into one *distribute entity* whose job is to collect the data arriving on the crossbar and insert it in the correct signals. Table 1.15 show the prototype and mapping of the *distribute* entity. The circulate node which provides input for the ILB and also consumes its output is mapped as per the prototype shown in Table 1.16. The final value of the circulate node may be written into memory using a write-scalar node. Table 1.17 shows the prototype and mapping for the write-scalar node. The translator maps each value node into its corresponding routine. Though the function implemented internally for each value node is different, the prototype for each value node is identical and is shown in Table 1.18. The mapping of the write-tile node is shown in Table 1.19.

	Signal Name	Signal Type	Signal Size	Description
Generic	NArgs	Natural Number	-	Number of runtime consts
	Generator Size	-	-	Row, Cols, RowStep, ColStep, DataSize
TopLevelSignal	-	-	-	Clock, Reset, Crossbar
RunTimeConsts	Args	Array of Words	NArgs	Corresponding Values
ILB Data	WinDataOut	ArrayType declared in parameters	-	Array of Windows
HandShake	StoreData	Bit	-	Set when current loop iteration starts
	End of Row	Bit	-	Output to reduction node
	OthersBusy	Bit	-	(Input) Set to one if buffer emptying not complete
	Done	Bit	-	Handshake to CPE0

Table 1.15: Prototype for Distribute Entity

Signal Type	Signal Names	Mapping
Handshakes	Enable	StoreData from distribute
	StoreLastVal	Output for the write-scalar node
Data	Data_Init	Port 1 signal
	Iters	Port 2 signal
	Data_In, Data_Out	Input and Output from ILB
Chip Signals	Clock	Wrapper entity clock
	Reset	Wrapper entity reset

Table 1.16: Prototype and Mapping for Circulate Node in TwoPE Model

Signal Type	Signal Names	Mapping
MemArb Signals	Row, Word, DataOut, DataIn Req, Ack, Write	Entry corresponding to current node in respective MemArb signal arrays
Handshakes	StoreData	<i>StoreLastVal</i> from a circulate node
	ArrReductBusy	Corresponding entry in ArrReductBusy array
Data	ValToStore	Signal on input port0
Chip Signals	Clock	Wrapper entity clock
	Reset	Wrapper entity reset

Table 1.17: Prototype and Mapping for Write-Scalar Node in TwoPE Model

Signal Type	Signal Names	Mapping
MemArb Signals	Row, Word, DataOut, DataIn Req, Ack, Write	Entry corresponding to current node in respective MemArb signal arrays
Handshakes	StoreData	StoreData from distribute
	ArrReductBusy	Corresponding entry in ArrReductBusy array
	LastVal	StopProcess bit of tags
Data	ValToStore	ILB output corresponding to current node
	Mask	Input Port 1
Chip Signals	Clock	Wrapper entity clock
	Reset	Wrapper entity reset

Table 1.18: Prototype and Mapping for Value Node in TwoPE Model

Signal Type	Signal Names	Mapping
Generics	DataSize, Tile Size(row+column)	Constants defined as parameters
MemArb Signals	Row, Word, DataOut, DataIn Req, Ack, Write	Entry corresponding to current node in respective MemArb signal arrays
Image Signals	Columns, Words	Corresponding entries in respective arrays
Handshakes	StoreData	StoreData from distribute
	ArrReductBusy	Corresponding entry in ArrReductBusy array
	End of Row	EndOfRow from distribute
ILB output	ValToStore	ILB output for this node
Chip Signals	Clock	Wrapper entity clock
	Reset	Wrapper entity reset

Table 1.19: Prototype and Mapping for Write-Tile Nodes in TwoPE Model

1.9 VHDL Pragma

The translation process described in the above sections discusses the mapping of the DFG nodes that correspond to the functionality described in the corresponding SA-C program. Sometimes a user may want to use a function already written in VHDL to execute part of the ILB computation. The VHDL pragma allows a user to declare a function prototype for such a VHDL routine. Note that only combinational VHDL routines are currently supported because the ILB needs to be purely combinational.

The user partitions the VHDL routine signals into two categories: chip signals and DFG node signals. The chip signals are not included in the function declaration. This is because they have no mapping to other DFG node signals. They arrive from the top-level entity of the FPGA. The user declares the function with the same name as the VHDL routine. The function arguments correspond to the input signals of the VHDL routine while the return values correspond to the VHDL routine outputs. The user may need to modify the entity declaration of the VHDL routine such that all the inputs are written first, followed by the output and chip signals. Also the internal order of the inputs (and outputs) must be identical to the order in the function declaration arguments (and return values). The ordering is important because the translator maps the signals for the entity as per their relative position in the declaration.

Once the user has a function declaration for the routine, it is preceeded by the actual pragma, written as `//PRAGMA (VHDL string)`. The string is used to provide the translator with other information it may need about the VHDL routine like chip signals used, etc. The compiler maps a call to this function into a VHDL_CALL DFG node. It passes the name of the function and the string as parameters for this node. The remaining structure of this node is identical to any other DFG node, that is, it has a set of input ports and output ports with their bit-sizes and edges mapped. The translator needs information about three things to complete the mapping.

- **File Name:** The translator must know the file name (including the path) in which the VHDL routines entity and architecture are defined. This is required during the compilation of the VHDL files.
- **Package Name:** The user needs to declare the routine as a library package black-box in order to simplify the translation process. The translator includes this package in the ILB VHDL file.
- **Chip Signals:** This third part is present only if the routine uses some chip signals. Every chip signal used is separated from each other by a comma. Currently the system supports only the use of clock and reset lines. This is because there can be contention for other resources like memory lines. The contention issue can make the routine non-combinational which does not suit our current model.

Here is an example of the use of the VHDL pragma. Consider a VHDL component called *myadd* having two inputs (both 7 bit wide) and performing an unsigned addition. Let this VHDL routine be written in a file *tmp/add.vhdl* and included in a package *addlib*.

The function prototype for this routine can be written in an SA-C program as follows:

```
//PRAGMA (VHDL tmp/add.vhdl addlib)
uint7 myadd(uint7 a, uint7 b);
```

If the routine needed the reset line, the function declaration would be slightly different. The string would contain the third optional part for the chip signals.

```
//PRAGMA (VHDL tmp/add.vhdl addlib clock)
uint7 myadd(uint7 a, uint7 b);
```

Chapter 2

Translation Example

Consider the SA-C program shown in Figure 2.1. Figure 2.2 shows its corresponding DFG with its inner-loop body. These figures are identical to Figure 2.1 and Figure 2.2 and are repeated for convinience.

Figure 2.3 shows the corresponding ILB VHDL code. Lines 1 – 8 correspond to the library inclusions while lines 9 – 11 correspond to the entity declaration. Line 9 is the single input signal from the generator node, line 10 is the run-time input constant while line 11 is the output for the write-tile node. Lines 12 – 22 correspond to the intermediate signal declarations while the remaining lines declare the actual circuit. The splitting of the generator signal happens on lines 23–26. Lines 27–30 group the inputs for USUMMANY into one signal while lines 31 – 36 do the VHDL instantiation. Line 32 performs the unsigned addition while line 33 groups the signal for the write-tile node.

The architecture generated by the translator for the OnePE model is made up of multiple subparts. Figure 2.4 shows the code generated for loop synchronization and handshaking. Figure 2.5 shows the code generated for run-time constants and mapping the image parameters for the generator and reduction nodes. The node mapping is a straight forward as it is a basic VHDL instantiation of the node prototype and hence is not shown. Similarly, the architecture generated for the TwoPE model is mainly made up of VHDL instantiations. Since the run-time constants and image parameter mapping is identical to the OnePE model, the code generated is identical to the one in Figure 2.5.

```

uint8[:, :] main(uint8 A[:, :], uint8 x) {
    uint8 R[:, :] = for window w[2,2] in A {
        uint8 tmp = array_sum(w)+x;
    } return(array(tmp));
} return(R);

```

Figure 2.1: A simple SA-C program

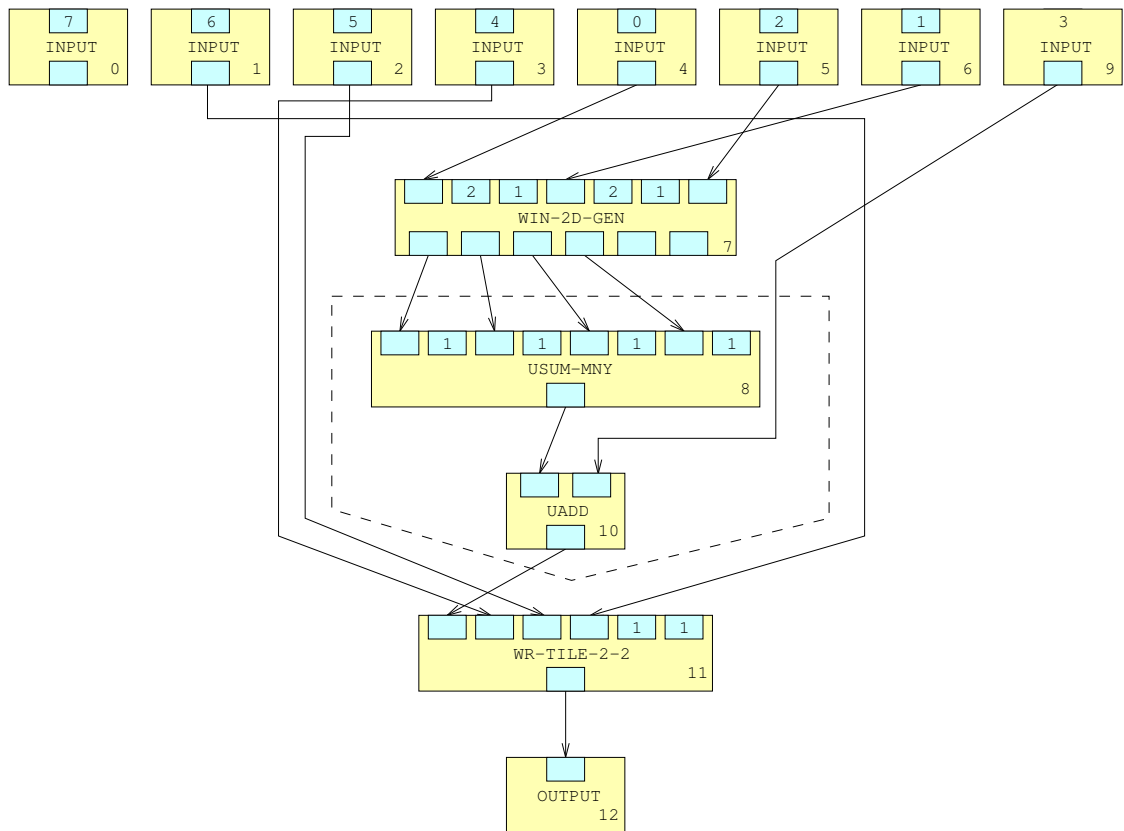


Figure 2.2: The resulting DFG (with the loop-body shown)


```

library ieee; --1
use ieee.std_logic_1164.all; --2
use ieee.std_logic_arith.all; --3
use ieee.std_logic_signed.all; --4
use ieee.std_logic_unsigned.all; --5
use work.cammacro.all; --6
use work.camtypes.all; --7
use work.parameters.all; --8

Entity Computation is
port(
    NODE7OUT : in InBuffArr; --9
    I9 : in std_logic_vector(7 downto 0); --10
    NODE11OUT : out WordArr(RowTileSize11*ColTileSize11-1 downto 0)); --11
end;

architecture byblocks of Computation is
    signal WINDOW_2D_GENERATOR_8_BIT7OUT0 : std_logic_vector(7 downto 0); --12
    signal WINDOW_2D_GENERATOR_8_BIT7OUT1 : std_logic_vector(7 downto 0); --13
    signal WINDOW_2D_GENERATOR_8_BIT7OUT2 : std_logic_vector(7 downto 0); --14
    signal WINDOW_2D_GENERATOR_8_BIT7OUT3 : std_logic_vector(7 downto 0); --15
    signal WINDOW_2D_GENERATOR_8_BIT7OUT4 : std_logic_vector(31 downto 0); --16
    signal WINDOW_2D_GENERATOR_8_BIT7OUT5 : std_logic_vector(31 downto 0); --17
    signal USUM_MANY8OUT0 : std_logic_vector(7 downto 0); --18
    signal UADD10OUT0 : std_logic_vector(7 downto 0); --19
    signal WRITE_TILE_2D_2D_8_BIT11OUT0 : std_logic_vector(0 downto 0); --20
    signal vals8 : std_logic_vector(31 downto 0); --21
    signal mask8 : std_logic_vector(3 downto 0); --22

begin
    WINDOW_2D_GENERATOR_8_BIT7OUT0 <= EXT(NODE7OUT(0), 8); --23
    WINDOW_2D_GENERATOR_8_BIT7OUT1 <= EXT(NODE7OUT(2), 8); --24
    WINDOW_2D_GENERATOR_8_BIT7OUT2 <= EXT(NODE7OUT(1), 8); --25
    WINDOW_2D_GENERATOR_8_BIT7OUT3 <= EXT(NODE7OUT(3), 8); --26
    vals8(7 downto 0) <= WINDOW_2D_GENERATOR_8_BIT7OUT0; --27
    vals8(15 downto 8) <= WINDOW_2D_GENERATOR_8_BIT7OUT1; --28
    vals8(23 downto 16) <= WINDOW_2D_GENERATOR_8_BIT7OUT2; --29
    vals8(31 downto 24) <= WINDOW_2D_GENERATOR_8_BIT7OUT3; --30
    NODE8: USUMMANY_NOMASK generic map( --31
        nVals => 4, insize => 8, outsize=> 8) --32
    port map( --33
        vals => vals8, --34
        result=>USUM_MANY8OUT0); --35
    UADD10OUT0 <= unsigned(USUM_MANY8OUT0) + unsigned(I9); --36
    NODE11OUT(0) <= SXT(UADD10OUT0, 32); --37
end;

```

Figure 2.3: The corresponding VHDL code of the ILB

```

-- Loop Synchronization
INPUTREADYAND : ANDMANY_NOMASK
generic map (
    nVals    => NumInReady,
    insize   => 1,
    outsize  => 1)
port map (
    vals     => PEX_InStores,
    Result   => PEX_InputReady);
OUTPUTREADYAND : ANDMANY_NOMASK
generic map (
    nVals    => NumOutReady,
    insize   => 1,
    outsize  => 1)
port map (
    vals     => PEX_OutStores,
    Result   => PEX_OutputReady);

--Handshake Signals
EOR_AND : ANDMANY_NOMASK
generic map (
    nVals    => NumInFiles,
    insize   => 1,
    outsize  => 1)
port map (
    vals     => PEX_EORS,
    Result   => PEX_EORIn);
EOF_AND : ANDMANY_NOMASK
generic map (
    nVals    => NumInFiles,
    insize   => 1,
    outsize  => 1)
port map (
    vals     => PEX_EOFS,
    Result   => PEX_EOFIn);
DONE_AND : ANDMANY_NOMASK
generic map (
    nVals    => NumOutFiles,
    insize   => 1,
    outsize  => 1)
port map (
    vals     => PEX_DONES,
    Result   => PEX_DONE);
PEX_Ready <= '0' when Reset = '1' or PEX_MARReady = '0' else '1';

```

Figure 2.4: Loop Synchronization and Handshake Code in OnePE Model

```

-- Run time Constants
PEX_AddrConstants(0) <= conv_std_logic_vector(7, AddrOnePESize);
I0 <= EXT(PEX_Constants(0),32);
PEX_AddrConstants(1) <= conv_std_logic_vector(6, AddrOnePESize);
I1 <= EXT(PEX_Constants(1),32);
PEX_AddrConstants(2) <= conv_std_logic_vector(5, AddrOnePESize);
I2 <= EXT(PEX_Constants(2),32);
PEX_AddrConstants(3) <= conv_std_logic_vector(4, AddrOnePESize);
I3 <= EXT(PEX_Constants(3),32);
PEX_AddrConstants(4) <= conv_std_logic_vector(0, AddrOnePESize);
I4 <= EXT(PEX_Constants(4),32);
PEX_AddrConstants(5) <= conv_std_logic_vector(2, AddrOnePESize);
I5 <= EXT(PEX_Constants(5),32);
PEX_AddrConstants(6) <= conv_std_logic_vector(1, AddrOnePESize);
I6 <= EXT(PEX_Constants(6),32);

-- Image Parameter Mapping for Write-Tile
PEX_Base(1) <= EXT(I3, AddrOnePESize);
PEX_TmpNumRows(1) <= unsigned(EXT(I2, WordSize))*
    unsigned(conv_std_logic_vector(RowTileSize11, AddrOnePESize));
PEX_NumRows(1) <= PEX_TmpNumRows(1)(AddrOnePESize-1 downto 0);
PEX_TmpNumCols(1) <= unsigned(EXT(I1, WordSize))*
    unsigned(conv_std_logic_vector(ColTileSize11, AddrOnePESize));
PEX_NumCols(1) <= PEX_TmpNumCols(1)(AddrOnePESize-1 downto 0);
PEX_NumWords(1) <= EXT(PEX_NumCols(1)(AddrOnePESize-1 downto 2), AddrOnePESize)
    when unsigned(PEX_NumCols(1)(1 downto 0)) =
        unsigned(conv_std_logic_vector(0,2))
    else EXT(unsigned(PEX_NumCols(1)(AddrOnePESize-1 downto 2))+1,AddrOnePESize);
-- Image Parameter Mapping for Window Generator
PEX_Base(0) <= EXT(I4, AddrOnePESize);
PEX_TmpNumRows(0) <= conv_std_logic_vector(0, BigOnePESize);
PEX_TmpNumCols(0) <= conv_std_logic_vector(0, BigOnePESize);
PEX_NumRows(0) <= EXT(I6, AddrOnePESize);
PEX_NumCols(0) <= EXT(I5, AddrOnePESize);
PEX_NumWords(0) <= EXT(I5(WordSize-1 downto 2), AddrOnePESize)
    when unsigned(I5(1 downto 0)) = unsigned(conv_std_logic_vector(0,2))
    else EXT(unsigned(I5(WordSize-1 downto 2)) + 1, AddrOnePESize);

```

Figure 2.5: Image Parameter Mapping

Chapter 3

Evaluation

The main purpose of the DFG to VHDL translator is to automate the process of generating VHDL code. The automated process takes a lot less time in comparison to manual implementation of the same algorithm in VHDL. For example, a simple SA-C program (*Prewitt*) shown in Figure 3.1 generates 541 lines of VHDL code for the TwoPE model. The translator also links in some static routines for the window generator nodes, etc. Table 3.1 (a) shows the number of the lines of code generated and used (per entity). The writing and testing of the prewitt SA-C program took about an hour. In contrast, Table 3.1 (b) also shows the lines of VHDL code for the hand-written prewitt program. The manual implementation and testing took about two months.

```
// Prewitt Gradient Edge detection of an image
uint8[:,:] main (uint8 image[:,:])
{
    // ***** defines the Prewitt's masks *****
    int3 vertmask[3,3] = {{-1, 0, 1}, {-1, 0, 1}, {-1, 0, 1}};
    int3 horzmask[3,3] = {{-1,-1,-1}, { 0, 0, 0}, { 1, 1, 1}};

    // ***** computes the gradient for the modified image *****
    uint8 res[:,:] =
        for window win[3,3] in image
        { int11 vert =
            for elem1 in win dot elem2 in vertmask
            return(sum((int11)elem1*elem2));

            int11 horz =
            for elem3 in win dot elem4 in horzmask
            return(sum((int11)elem3*elem4));
        }
        return(array(sqrt((int23)((int22)vert*vert)+(int22)horz*horz)));
} return (res);
```

Figure 3.1: Prewitt SA-C Program

Type	Entity	Lines of Code
Generated	ILB	123
	Parameters	37
	CPE0	163
	PE1	218
Used	WindowGenerator	887
	WriteTile	233
	ConstGrabber	156
	MemArb	142
Total		1959

(a) Automated System

Entity	Lines of Code
CPE0	901
PE1	767
Total	1668

(b) Manual Implementation

Table 3.1: Lines of VHDL Code

The automated process of DFG to VHDL translation was tested using a set of level one image processing codes listed in Table 3.2.

Program Name	Description
AddMonadic.sc	Add a run-time constant to each pixel in an image
AddDyadic.sc	Add two 8-bit images (pixel-wise)
SubtractMonadic.sc	Subtract a run-time constant from each pixel in an image
SubtractDyadic.sc	Subtract two 8-bit images (pixel-wise)
MultiplyMonadic.sc	Multiply a run-time constant with each pixel in an image
MultiplyDyadic.sc	Multiply two 8-bit images (pixel-wise)
RGBtoXYZ.sc	Convert RGB image to a XYZ image
XYZtoRGB.sc	Convert an XYZ image to a RGB image
convolution3x3.sc	Convolution operation over 3 * 3 windows
dilation.sc	Dilation Operation
erosion.sc	Erosion Operation
gaussianfilter.sc	Gaussian filter of size 3 * 3
max.sc	Find maximum value pixel in an image
maxfilter.sc	Maximum value of 4 * 5 windows in an image
mp4.sc	Part of MP4 Honeywell Benchmark
prewittmag.sc	Prewitt Edge Detection Algorithm
sobelmag.sc	Sobel Edge Detection Algorithm
threshold	Find pixels in an image above a threshold value
wavelet.sc	Wavelet Image Processing Application

Table 3.2: Level One Codes

REFERENCES

- [1] A. Patel. *Design of Library Modules for Automated Synthesis of Data Flow Graphs*, May 2000. MS Thesis, Colorado State University.
- [2] C. Ross. *A VHDL Runtime System for Dataflow Execution on Reconfigurable Systems*, April 2000. MS Thesis, Colorado State University.

CAMERON PROJECT: FINAL REPORT

Appendix G: An Automated Process for Compiling Dataflow Graphs into Reconfigurable Hardware

An Automated Process for Compiling Dataflow Graphs into Reconfigurable Hardware

Robert Rinker, Margaret Carter, Amitkumar Patel, Monica Chawathe, Charlie Ross, Jeffrey Hammes, Walid A. Najjar, Wim Böhm

Abstract—

We describe a system, developed as part of the Cameron project, which compiles programs written in a single-assignment subset of C called SA-C into dataflow graphs, and then into VHDL. The primary application domain is image processing. The system consists of an optimizing compiler which produces dataflow graphs, and a dataflow graph to VHDL translator. The method used for the translation is described here, along with some results on an application. The objective is not to produce yet another design entry tool, but rather to shift the programming paradigm from HDLs to an algorithmic level, thereby extending the realm of hardware design to the application programmer.

Keywords— Adaptive-computing, Configurable, Image-processing, Reconfigurable-components, Reconfigurable-computing, Reconfigurable-systems

I. INTRODUCTION

IMAGE processing (IP) applications are ideally suited for reconfigurable computing. They exhibit a large degree of fine and coarse grain parallelism: at the bit (or pixel), instruction, loop and task levels. Moreover, they require the repeated application of the same operation on successive sets of data (e.g. streaming video). Reconfigurable computing systems (RCS's) are therefore interesting candidates for special purpose IP acceleration hardware: they provide a large degree of fine-grained parallelism that can be configured to efficiently fit many simultaneous small-data-size (pixel) operations.

However, most reconfigurable computing systems are based on FPGAs, and therefore are programmed using hardware description languages where the user specifies the logical structure of the intended circuit. This programming paradigm is very different from the algorithmic programming languages that are typically used by IP application developers. Another difficulty is partitioning of the algorithm between a host processor and reconfigurable modules, and devising ways of producing efficient FPGA configurations – both of these steps require intimate knowledge of the hardware and host interface, which is not something that the typical IP programmer understands.

The goal of the Cameron project [1] is to shift the programming paradigm for reconfigurable computers from hardware-centered to software-centered, thereby making them accessible to IP application developers and portable across reconfigurable computing platforms. This

is achieved by creating a software infrastructure that translates a high level algorithmic language into a hardware description language. It consists of a graphical programming environment, a high-level language, an optimizing compiler, and debugging and performance monitoring tools for IP on reconfigurable computers. The objective of this system is to integrate the parts into a single design environment, and to allow the design to be carried out entirely by the application programmer, without requiring intimate knowledge of hardware or interface details.

The Cameron project includes the design of a language which is particularly suited for translation into hardware, called SA-C (*Single Assignment C*, pronounced *sassy*). SA-C programs are compiled into a dataflow graph (DFG) format. The compiler applies extensive expression, loop and array optimizations. The objective of the optimizations is to minimize the hardware cost of the program on the FPGA as well as to maximize locality by reusing data and expressions. The DFG format is then compiled into VHDL which is mapped, using commercial tools, onto the reconfigurable hardware. The focus of this paper is on this DFG to hardware mapping.

The rest of this paper is organized as follows. The next section highlights other related reconfigurable computing projects. Section III provides an overview of the SA-C language and DFGs, particularly those features which facilitate and impact the DFG-to-VHDL translation. Next, the development of an abstract architecture, which defines the target for the translation, is discussed. The actual translation process is described in Section V. Next, an example is described, along with some preliminary performance numbers. The paper concludes with a discussion of future work.

II. RELATED WORK

Reconfigurable computing is an active area of research – both hardware and software projects, and combinations of both, are ongoing. Hardware projects fall into two categories – those that use off-the-shelf components (in particular, FPGAs), and those which use custom designs.

The Splash-2 [2] is an early (circa 1991) implementation of an RCS, built from 17 Xilinx [3] 4010 FPGAs, and connected to a Sun host as a co-processor. Several different types of applications have been implemented on the Splash-2, including searching[4], [5], pattern matching[6], convolution [7] and image processing [8].

A commercial system which is loosely patterned after the Splash-2, but which utilizes larger but fewer FPGA's, is the Annapolis Microsystems Wildforce^(TM) board [9],

This work is supported by DARPA under US Air Force Research Laboratory contract F33615-98-C-1319.

Computer Science Department, Colorado State University, Ft. Collins, CO 80523-1873, E-mail: {rinkerr, carterm, amit, monica, rossc, hammes, najjar, bohm}@cs.colostate.edu

introduced in 1995 – this system was used in the implementation described in this paper, and is covered in some detail later.

Representing the current state of the art in FPGA-based RCS systems are the AMS WildStar[10] and the SLAAC project [11]. Both utilize Xilinx Virtex [12] FPGA’s, which offer over an order of magnitude more programmable logic, and provide a several-fold improvement in clock speed, compared to the earlier chips.

Several projects are developing custom hardware. The Morphosis project [13] marries an on-board RISC processor with an array of reconfigurable cells (RC’s). Each RC contains an ALU, shifter, and a small register file. The RAW Project [14] also consists of an array of computing cells, called *tiles*; it differs in that each tile is itself a complete processor, coupled with an intelligent network controller, and a section of FPGA-like configurable logic that is part of the processor data path, more like an on-chip network of workstations; there is no “host” processor. These designs represent a more coarse-grained, or *chunky* architecture [15], compared to FPGA-based logic cells; such architectures promise to be more manageable as complexity increases.

PipeRench [16] consists of a series of *stripes*, each of which is a pipeline stage – an input interconnection network, a lookup-table based PE, a results register, and an output network. During execution, a context loader places pipeline stages to be executed into the next available stripes – in most cases, the context switching can be completely hidden. the application appears to execute in an infinitely deep pipeline.

On the software front, several of the above hardware projects also involve software development. The RAW project includes a significant compiler effort [17] whose goal is to create a C compiler which treats the network of tiles as a single system, rather than as individual processor nodes as in conventional network programming. For PipeRench, a low-level language called DIL [18] has been developed for expressing an application as a series of pipeline stages, which can easily be mapped to stripes.

Several projects (including Cameron) focus on hardware-independent software for reconfigurable computing; the goal – still quite distant – is to make development of RCS applications as easy as for conventional processors, using commonly known languages or application environments. Several projects use C as a starting point for RCS development. Handel-C [19] both extends the C language to express important hardware functionality, such as bit-widths, explicit timing parameters, and parallelism, and limits the language to exclude C features that do not lend themselves to hardware translation, such as random pointers. Streams-C [20] does a similar thing, with particular emphasis on extensions to facilitate the expression of communication between parallel processes. SystemC [21] and Ocapi [22] provide C++ class libraries to add the functionality required of RCS programming to an existing language.

Finally, a couple of projects use higher-level application environments as input. The MATCH project [23], [24] uses

MATLAB as its input language – applications that have already been written for MATLAB can be compiled and committed to hardware, eliminating the need for re-coding them in another language. Similarly, CHAMPION [25] is using Khoros [26] for its input – common glyphs have been written in VHDL, so GUI-based applications can be created in Khoros and mapped to hardware.

III. THE SA-C COMPILER AND DATAFLOW GRAPHS

Rather than trying to extend or limit an existing language, SA-C[27] is designed specifically to make it easy for the compiler to analyze the code and extract both fine-grain and coarse-grain parallelism. SA-C is an expression-oriented, single assignment (functional) language that is designed to be translated into hardware descriptions. As the name implies, the syntax is loosely based on C; however, there are significant differences as well, mostly due to its use as hardware generation language.

A. Unique features of SA-C

- A flexible type system, including signed and unsigned integers of any bit width, as well as fixed point numbers.
- True multi-dimensional arrays, with a specific size and shape which may be inferred when the array is created.
- No pointers or other indirection, to eliminate side-effects,
- Loop generators, which are usually used in place of the more traditional “loop index used as an array subscript” to perform operations on arrays. Conceptually, there is no specified order to the operations performed on elements of the array, but rather it appears that the entire array is defined at one time. This gives the compiler the freedom to implement array operations in the most efficient way.
- Reduction operators, which perform commonly used operations on the data produced in loop bodies, such as `array sum` and `histogram`. This allows the programmer access to efficient VHDL implementations of these operations.

The language includes several features that make it especially suited for IP applications; however, it is a general purpose language that can be used for other applications as well.

A simple SA-C program is shown in figure 1(a). This program accepts a 2-D array (named `Arr`) of 8-bit unsigned integers (i.e., of type `uint8`) as input. A window generator statement (`for window...`) extracts all 3×3 sub-arrays from the image array, and sums the elements in each sub-array. A new array (named `r`) is formed such that each element is either the sum of the corresponding window or, if the sum is greater than 100, the sum minus 100. This simple program demonstrates several characteristics of the SA-C language; a complete language reference is available in [27]. Upon compilation, an intermediate form of the program, called a dataflow graph (DFG), is generated; a pictorial representation of the DFG for this program is shown in figure 1(b).

B. SA-C Compiler Optimizations

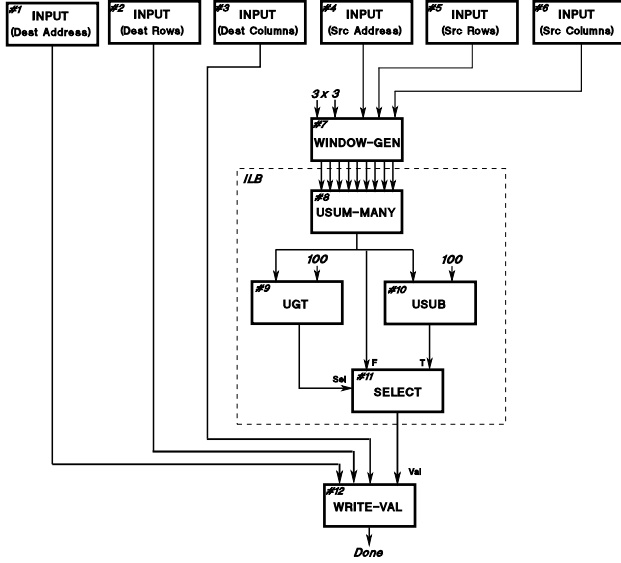
Numerous optimizations are performed by the compiler before creating the DFG. Several traditional optimizations

```

uint8[:,:] main (uint8 Arr[:,:]) {
  uint8 r[:,:] =
    for window W[3,3] in Arr {
      uint8 s = array_sum (W);
      uint8 v = if(s>100) return(s-100)
                else return(s);
    } return (array (v));
} return (r);

```

(a)



(b)

Fig. 1. (a) A simple SA-C program, and (b) the resulting dataflow graph

minimize the calculation required by the hardware; these include code motion, constant folding, array and constant value propagation, and common subexpression elimination. Function inlining and loop unrolling provide parallel computation opportunities, and often enable other optimizations.

Other, less traditional optimizations are included to reduce hardware size and/or increase execution speed:

- Array and loop size propagation, facilitated by the use of the loop generator statements, allow the compiler to automatically determine loop unrolling depth.
- Bit width narrowing works in conjunction with loop unrolling to insure that each instance of a loop body uses the smallest data sizes possible to perform a given calculation.
- *Stripmining* - splits a loop into a pair of nested loops, with the outer loop creating chunks of work which are performed by the inner loop. When implemented in hardware, parallelism is introduced by separately instantiating each inner loop body; the outer loop is converted into code which distributes data to these instances.
- *Tiling* - allows a large image to be split into smaller pieces that fit into the RCS memory.
- *Loop fusion* - forms a single loop from two or more consecutive loops in an algorithm. This helps to eliminate extra data communication between processing steps. The compiler can often perform this operation in situations where

such fusion is not obvious to the application programmer. Some of the optimizations may be detrimental to performance if applied in the wrong situation, or trade one hardware resource for another; in these cases, their application can be controlled by *pragmas*.

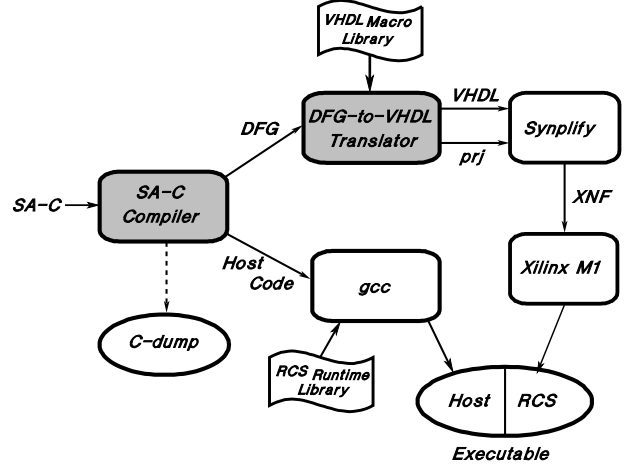


Fig. 2. Overview of the SA-C Compilation process

An overview of the current implementation of the compilation system is shown in figure 2. The compiler translates SA-C programs and performs optimizations as described above. It produces:

- A host-based C program, which controls the hardware, manages data transfer to/from the RCS, and performs execution tasks that cannot be performed by the hardware, such as file I/O.
- A DFG of that portion of the program which will execute on the RCS.
- Optionally, a *C-dump*, a C-version of the entire program which can be used for debugging and verification before committing the program to hardware.

By default, the compiler tries to move as much of the program as possible from host to hardware; in the vast majority of cases, its decision is reasonable. However, for those cases where it makes a bad decision, a *pragma* can be used to force the compiler to keep more of the program on the host.

The second component, a VHDL-to-DFG translator, extracts information from the generated DFG and produces a VHDL implementation of the program. This code is processed by VHDL synthesis and place-and-route tools to produce hardware configuration files for the RCS.

IV. AN ABSTRACT ARCHITECTURE FOR RECONFIGURABLE COMPUTING

Unlike “standard” instruction set architectures, which provide a relatively small set of well-defined instructions to the user, RCS’s are composed of an amorphous mass of logic cells which can be interconnected in any number of ways. To limit the number of possibilities available to the designer, an *abstract architecture* has been defined as a more manageable, hardware independent target. We currently define three types of functions:

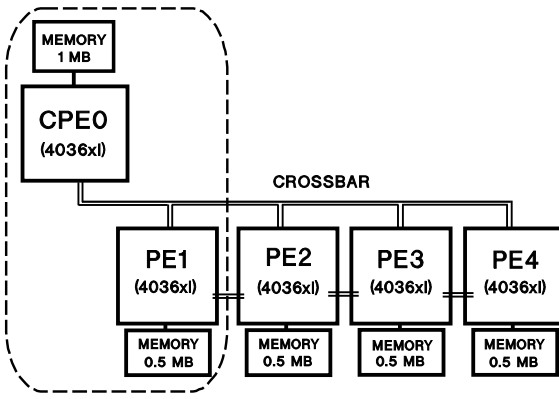


Fig. 3. Architecture of the Wildforce-XL Reconfigurable Computing Board, showing that part of the board being utilized by the DFG-to-VHDL translator

- Data transfer mechanisms. These include streams, block memory transfer, and systolic modes. These mechanisms are used both between host and RCS as well as within sections inside the RCS.
- Arithmetic and Logical Operations. These include common simple operators such as ADD, SHIFT, and COMPARE, as well as more complex operations such as SQRT, SUM, and MEDIAN. The set of operations that have been included in SA-C was influenced by the IP application domain.
- Data buffering and storage mechanisms, including FIFOs and arrays of buffers, which are used to implement more complex functions such as shift registers.

A. Implementation on the AMS Wildforce Board

The first version of the compilation system targets the AMS Wildforce board [9]. Figure 3 shows a simplified diagram of the board used as a target. This board consists of 5 FPGAs, connected such that one FPGA, named CPE0 (*Control Processing Element 0*), can broadcast data to the others, PE1-4, via a 36 bit crossbar. Each PE also has access to its own local memory, organized as 32 bit words. The board is connected to a host via a PCI connection – the host is responsible for downloading the configuration codes, and for sending and retrieving data to/from the board.

Our initial implementation of the compilation system uses only that portion of the board shown inside the dotted lines in the figure. This implementation subset was chosen primarily for simplicity – CPE0 retrieves image data from its local memory and sends it in the proper order over the crossbar to PE1, which buffers it, performs the necessary calculations, and stores the results in its local memory. Since two PE's and thus two memories are used, the design is simplified because there is no memory contention between reading and writing. It is expected that the next generation of the translator will use more of the PE's on this board, either for added functionality or to enhance parallelism in the present system. On the other hand, this subset is reasonable in its own right, since the trend in new board designs is to use fewer, larger FPGAs; thus, this scheme seems to be a reasonable model for future work when the system is ported to other hardware.

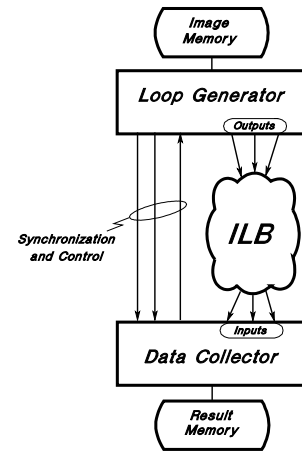


Fig. 4. Structure of the code generated by the DFG-to-VHDL translator

The choice of PE1 in the implementation is arbitrary – any of the PE's 1–4 could have been used, since they are identical chips and have nearly identical connections. In the following discussion we use the term *PE_x* to reference this PE.

V. TRANSLATING DFG'S TO VHDL

At first glance the DFG of figure 1(b) appears to describe a simple top to bottom execution that can be implemented by a combinational circuit; however, the operation of some of the nodes are more complicated. In particular, the WINDOW-GEN node retrieves data from RCS memory and presents it in the proper order to the inner loop body (ILB) of the program, and the WRITE-VAL node collects the results and stores them into memory to be retrieved later by the host. These operations require multiple clock cycles, several state machines, and coordination between the upper (loop generator) and lower (data collection) nodes. Figure 4 shows the resulting design partitioning.

A. Classification of DFG nodes

As a first step in converting the DFG to VHDL, the translator classifies the dataflow graph nodes into one of four types:

- Run-time input nodes. These nodes are not directly translated into VHDL, but rather specify addresses within CPE0's memory where run-time data has been stored. It includes information only known at run-time, such as the size, shape, and starting address of the data arrays. The translator uses this information to form a table of addresses which will be used by the ConstGrabber module to retrieve the data from memory at the start of execution.
- Generator nodes. The information in these nodes includes such things as window size, shape, and step size. In contrast to the run-time constants discussed above, this information is used during compilation to parameterize the instantiation of the various VHDL components.
- Loop body nodes. These nodes specify the operations to be performed by the ILB. These nodes are used to generate the VHDL code for the ILB.

```

entity Computation is -- 1
  port(NODE7OUT : in ByteArr(WinSize-1 downto 0); -- 2
        NODE12OUT: out Byte); -- 3
end; -- 4
architecture byblocks of Computation is -- 5
  signal USUM_MANY8OUT0:std_logic_vector(7 downto 0);-- 6
  signal UGT9OUT0: std_logic; -- 7
  signal USUB100OUT0: std_logic_vector(7 downto 0); -- 8
  signal SELECTOR110OUT0:std_logic_vector(7 downto 0);-- 9
begin --10
  NODE9: USUMMANY generic map( nVals => 9, --11
                              insize => 8, --12
                              outsize=> 8) --13
    port map( vals => NODE7OUT, --14
              result => USUM_MANY8OUT0);--15
  UGT9OUT0 <= '1' when USUM_MANY8OUT0 > 100 --16
            else '0'; --17
  USUB100OUT0 <= USUM_MANY8OUT0 - 100; --18
  SELECTOR110OUT0 <= USUB100OUT0 when UGT9OUT0 = '1' --19
                  else USUM_MANY8OUT0; --20
  NODE12OUT<= SXT(SELECTOR110OUT0, 8); --21
end; --22

```

Fig. 5. The generated VHDL for the ILB of figure 1(b) (some declarations and type casting have been omitted for clarity).

- Reduction nodes. Similar to the generator nodes, these specify the parameters needed to select and instantiate the reduction (collection) nodes.

Given this information, the translation process is divided into three main parts. First, the ILB is identified as being that part of the DFG that lies between the *outputs* of the loop generator nodes and the *inputs* of the data collection nodes, and consists entirely of loop body nodes. This section of the DFG is translated directly into a VHDL *component*. Then, the loop generator and collection nodes are implemented by selecting the proper VHDL components from a library, and by supplying these components with a parameters file containing information derived from the pertinent DFG nodes. Finally, the translator specifies the interconnections between the ILB and generator/collection components by generating two top-level VHDL modules, one for CPE0 and one for PEx, and a set of *project* files used by the Synplify VHDL compiler/synthesis tool. The files created in this last step serve to “glue” all the components together into a final design.

B. Translation of the ILB

The translation of the ILB involves a traversal of the dataflow graph. A VHDL *component* is created whose inputs are connected to the outputs of the loop generator, and whose outputs are connected to the inputs to the data collector. As an example, Figure 5 shows the VHDL that is generated for the ILB denoted by the dotted lines in the DFG of figure 1(b).

Many nodes implement simple operations, such as addition or logical operations; for these nodes, there is a one-to-one correspondence between DFG node and VHDL statement; for example, line 18 in figure 5 performs the subtract by 100. For more complicated operations, the translator generates a connection to a VHDL component; for example, lines 11-15 implement the SA-C *array sum* reduction

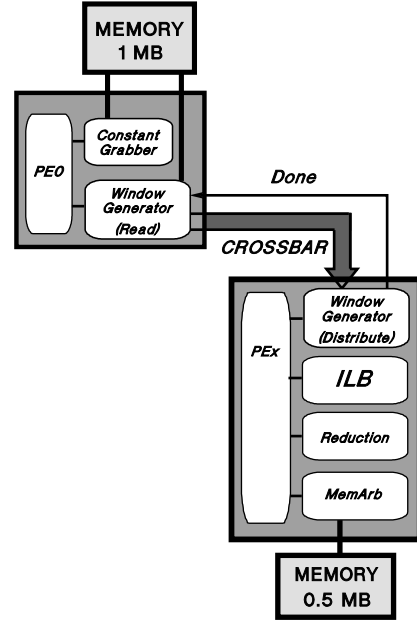


Fig. 6. Location of components in the 2 PE model

operation. A library of such components has been written directly in VHDL; this allows a SA-C program access to operators that either cannot be expressed in the high level language or that have efficient direct hardware implementations. To facilitate the tracing of signals through the ILB, the names of the signals used to interconnect nodes are derived from the DFG node type and number. For example, the signal name UGT9OUT0 in the VHDL example corresponds to the first (number 0) output of the UGT node numbered 9 in the DFG.

At present, the generated ILB is entirely combinational, although it is expected to eventually include multiple-cycle functions such as lookup tables, complex data reduction operations, and pipeline registers.

C. Implementation of the Other Components in a Design

In contrast to the ILB, which is generated from scratch by traversing the DFG, the data generators and collectors are created from VHDL components selected by the translator from a module library, and are parameterized with values extracted from the DFG.

An entire implementation, including the top-level glue modules mentioned earlier, is shown in figure 6. The operation of each component in the figure will be discussed in the following sections. The general flow of information is from top left to bottom right – the following discussion follows roughly the same order.

C.1 The ConstGrabber Component

Prior to initiating execution of the hardware, the host C program, created by the compiler as described earlier, downloads the run-time constants and input data to CPE0’s memory and then resets the hardware. The ConstGrabber component then reads the run-time constants from memory and makes them available to the rest

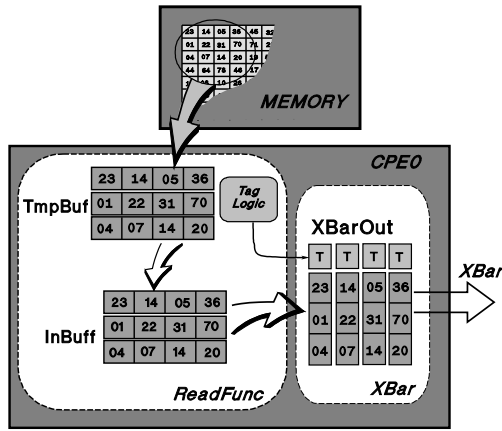


Fig. 7. Diagram of the read window generator function

of the system. When finished, the module sends a Ready signal to the rest of the system.

C.2 The Data Generator Components on CPE0

The data generator is responsible for retrieving data from memory in the proper order and buffering it, so it can serve as input to the inner loop body. At present, there are three types of data generators, selected because they seem to cover virtually all of the data access patterns required in IP applications. Scalar generators generate single values similar to the traditional for loops found in C and other languages. Element generators extract single values from an array. Window generators perform the more complex task of extracting small sub-arrays from a larger array. The discussion that follows specifically describes the window generator; the other generator types operate in a similar fashion (in fact, the element generator is currently implemented as a window generator with a 1×1 window).

The window generator function is distributed over the two PE's. The part that resides on CPE0 (called the *read* function) is the more complicated of the two parts; it is responsible for reading data from CPE0's memory and placing the data on the crossbar. It consists of two separate state machines – one which is responsible for retrieving data from memory (the *ReadData* state machine), and a second that sends the retrieved data out to the crossbar (the *XBar* state machine).

Initially, the *XBar* state machine waits for the Ready signal from the *ConstGrabber* module, then sends the retrieved run-time constants on the crossbar, so they are available to the components in PEx. Once this initial step occurs, the main operation of the read function starts. Figure 7 illustrates this operation for a 3×3 window example.

The *ReadData* state machine begins reading words of data from memory. The number of words read is equal to the number of columns in the window being used, and is called a *frame* of data. As the data is being read, it is stored in a buffer (*TmpBuf*); once all the words for a complete frame have been read, they are transferred to a second buffer called *InBuff*. This double buffering allows one frame to reside within CPE0 while the next frame is

being retrieved from memory. Once a frame has been input, *ReadData* sends a signal to the *XBar* state machine and then begins reading the next frame.

Upon signal from *ReadData*, the *XBar* machine starts sending data along the crossbar. Data is stored in memory and therefore retrieved in row-major order; however, it is sent along the crossbar in columns. An interconnection network, generated using the static parameters passed to the component at compilation, performs the transposition from rows to columns. The resulting buffer (*XbarOut*) holds the data that is output to the crossbar.

Several signals are generated and sent as tag bits as *XBar* sends the data; these signals are used by PEx to help interpret the data:

- **ValidData** (crossbar bit 35) - Indicates that the value on the crossbar is a legal data item. When set to 0, the value currently on the crossbar is undefined (i.e. a NULL). NULLs are sent during cycles when no valid data is available to be sent, such as when *XBar* is waiting for data from *ReadData*.
- **Start/Stop** (bit 34) - Set to indicate the beginning and end of data.
- **DontStore** (bit 33) - Used to indicate that the calculations calculated by the ILB at this time step should not be stored. This situation occurs at the beginning of each row, and when the window generator is using a step size other than 1, as described below.
- **LastCol** (bit 32) - set to one when the values being sent are from the last column in a row.

In addition to sending window data in the proper sequence, the window generator code on CPE0 must handle several special situations. It is possible for windows to be created faster than the result data can be stored. This might occur when an ILB produces several output values for each input. In this case, *XBar* places NULL values on the crossbar to “slow down” the window generation rate. To implement this, the translator determines the number of cycles required by each component to process a frame of data, and then finds the maximum of these. This value is used to determine the number of NULLs (if any) that must be sent after each frame to keep the data generation rate from overrunning the output bandwidth. No NULLs are sent if the writing of results can keep up with the generator.

The system must also handle window steps other than 1 – for example, a step of 2 specifies that only every other window is supposed to produce a value. In these situations, all of the data is still sent, and the ILB, being a combinational circuit, still calculates values at each time step; however, only some of the calculated results are stored. The *DontStore* signal determines when a value should not be stored. *DontStore* is also used to prevent the storing of results calculated at the beginning of each row, before the first complete window of data has been transmitted along the crossbar.

C.3 The Data Generator Components on PEx

That portion of the data generator that resides on PEx is called the *distribute* function. It retrieves the data from

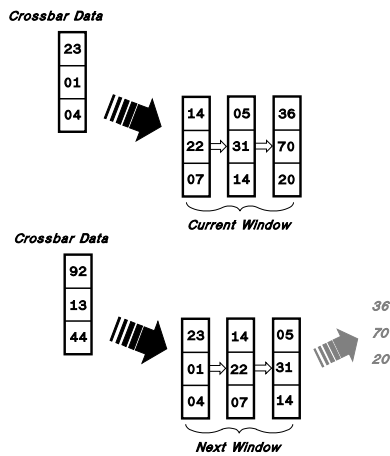


Fig. 8. Pictorial representation of a 3×3 sliding window generator for the current, and then next windows.

the crossbar and buffers it, so it can be presented to the inputs of the ILB. It also uses the tag bits sent on the crossbar to generate signals which control the other components on PEx. The distribute function consists of a single state machine.

In concert with operation of the code on CPE0, the *Distribute* state machine initially retrieves run-time data from the crossbar, then goes into its “normal” operation. The heart of the *Distribute* function is a shift register which buffers the window data. its operation is illustrated in figure 8. A “sliding window” effect is created by the shift register – when new data arrives from the crossbar, a new window is formed by taking the previous columns that had been sent, shifting them over, and then placing the new column of data into the space just vacated by the shift operation.

Distribute also generates a signal called *Storedata*, which is derived from the *ValidData* and *DontStore* tags bits from the crossbar. This signal inhibits the storing of data at the beginning of each row and during window stepping.

One special operator used with data generators is the *dot* (or dot product, although the operator does not imply that a product or any other arithmetic operation is to be performed). Consider the following SA-C statement:

```
for elem1 in image1 dot elem2 in image2
```

This syntax specifies that two (or more) data generators are to operate simultaneously, thereby providing two (or more) sets of data to the ILB at the same time. The latest version of the generator code has extended the window generator code to handle certain combinations of the *dot* operation. The operation of this generator is very similar to that of the single window. The primary difference is that a separate set of buffers on both CPE0 and PEx are instantiated for each generator. As the *XBar* machine sends data on the crossbar, first a column from one window is sent, then a column from the next window is sent, etc. The *Distribute* machine on PEx then places the data retrieved from the crossbar into the proper shifter register.

C.4 The Collector function

Once data has been presented to the input of the ILB, the resulting values must be stored in memory. The *Collector* component consists of a state machine which performs this function. The *StoreData* signal generated by *Distribute* causes the result value to be placed into a buffer; once an entire word of data has been collected, the buffer is sent to the *MemArb* component.

In addition to single values, the *collector* can also handle *tiles* of results. Tiles are small arrays of data, similar to windows, that can be produced by an ILB. Usually tiled output occurs as a result of the stripmining and tile optimizations applied by the compiler – an original (single) ILB is transformed into several instantiations, with the resulting output being a tile whose size and shape is determined by the order in which the compiler optimizations were applied. The tile shape is determined at compile time, so buffers are instantiated within *Collector* to hold the resulting tile values.

A different situation occurs for a single ILB which produces multiple (independent) values. In this case, a separate *Collector* component is instantiated for each output. This method is used so that the proper size and shape of buffers can be independently instantiated for each output. The PEx glue code is responsible for providing the interconnection for each instance of the *Collector* component.

With multiple and tiled outputs being produced by the ILB, it is possible that a single set of input values can produce multiple outputs, thus causing the output bandwidth to exceed the input. As discussed earlier, the *Collector* function does not need to worry about this, since the window generator adds wait cycles if necessary to guarantee that the input rate will not cause data overrun on output.

C.5 The MemArb function

The final component in the system is the *MemArb* component. This module receives words of data to be stored in memory, and performs the storing operation. In contrast to the *Collector*, which is instantiated once for each set of outputs, there is always only one *MemArb* component – it handles multiple store requests from possibly several *Collectors*. A simple priority encoder determines which output is stored first. However, since the window generators never produce data faster than the results can be stored, the exact order in which the stores occur is not important because even the lowest priority values will have time to store.

VI. AN EXAMPLE: THE PREWITT ALGORITHM

The Prewitt algorithm is a well known edge-detection algorithm used in many IP applications; its development and operation is described in [28], and in most IP texts, such as [29]. The algorithm involves the convolution of an image with two constant 3×3 masks – one which forms the X gradient, and one which forms the Y gradient; the two masks are shown in figure 9. The results of these convolutions form two vector components; the magnitude of the

$$\begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ +1 & +1 & +1 \end{pmatrix} \quad (a)$$

$$\begin{pmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{pmatrix} \quad (b)$$

Fig. 9. Constant convolution masks for the Prewitt algorithm. (a) the X gradient mask, and (b) the Y gradient mask.

```

uint8[:,:] prewitt(uint8 Image[:,:]) {
    int2 H[3,3] = {{-1,-1,-1},
                  { 0, 0, 0},
                  { 1, 1, 1}};
    int2 V[3,3] = {{-1, 0, 1},
                  {-1, 0, 1},
                  {-1, 0, 1}};

    uint8 res[:,:] =
        for window W[3,3] in Image {
            int11 sh, int11 sv =
                for h in H dot w in W dot v in V
                    return (sum( (int11)w*h ),
                            sum( (int11)w*v ));
        } return (array (magnitude(sh,sv)/8));
    } return (res);
    uint11 magnitude(int11 a, int11 b)
    return (sqrt( (int22)a*a + (int22)b*b ));
}

```

Fig. 10. The Prewitt edge-detection algorithm, written in SA-C

resulting vector forms the desired result. This fundamental operation is performed over the entire image, with the result being another 2D array called the *gradient array*.

While this algorithm is important in its own right as a fundamental IP operation, it is an interesting example for other reasons:

- It is inherently a parallel operation, with each 3×3 convolution being entirely independent of the others,
- It involves constant masks, which allows considerable optimization before being implemented in hardware,
- It presents some computational challenges, requiring a squaring (multiplication) and square root, which are difficult on FPGAs,
- It involves the use of a streaming data model from host to RCS back to host, which is common in IP applications,
- It is representative of a large number (perhaps even the majority) of other common IP applications. In fact, the SA-C language contains *window generators* to provide a simple means for expressing the extraction of a small sub-array from a larger one.

A SA-C program which performs the Prewitt calculation is shown in figure 10.

The SA-C compiler performs several of the optimizations described earlier on this program before producing the DFG. Since the convolutions involve multiplications with 3×3 masks that are composed of the constants 1, 0, and -1 , the compiler optimizes the calculation to a series of additions and subtractions. Multiplications with zero are eliminated completely. These optimizations eliminate all multiplies, and reduce the number of addition/subtractions from 16 down to 10.

The *magnitude* function is the most expensive part of the ILB, since it involves the squaring of the two results (requir-

TABLE I
STATISTICS FOR THE WILDFORCE IMPLEMENTATION OF THE PREWITT ALGORITHM USING THE SA-C COMPILER/TRANSLATOR.

(a) Lines of Code

SA-C	19
VHDL	
WINDOW-GEN - PEO	572
WINDOW-GEN - PE1	194
Generated Inner Loop Body	3744
WRITE-VAL	406
Glue Code and Misc	199
Total VHDL	5115

(b) FPGA Logic cell (CLB) Usage

WINDOW-GEN - PEO	251	(19.4%)
WINDOW-GEN/WRITE-VAL - PE1	236	(18.2%)
Inner Loop Body	281	(21.7%)
Glue Code and Misc	19	(1.5%)
Total - PE1	536	(41.3%)
Total CLBs	787	(30.4%)

(c) Propagation Delay - Inner Loop Body

ILB - Convolution	58.1 ns
ILB - Magnitude	295.9 ns
Total Propagation Delay	354.0 ns (2.8 MHz)

(d) Execution Times (msec)

	No stripmine	4×3 stripmine	Manual VHDL	Pentium
Data Download	0.54	0.54	?	—
Computation	59.14	30.10	4.66	28.4
Result Upload	0.93	0.93	?	—
Total time	60.61	31.57	4.66+	28.4
Freq(MHz)	2.82	2.78	16.9	450

ing a multiply), then finding the square root. An efficient square root routine is used which uses only shifts, adds, and bit operations. Nonetheless, the multiply/square-root operation consumes over 50% of the space, and requires more than 80% of the time required by the entire ILB.

The resulting DFG is processed by the DFG-to-VHDL translator, which extracts the ILB and translates it directly to VHDL, selects the appropriate generator and collector components from the VHDL library, and creates the top-level VHDL program which “glues” the entire system together. The translator also creates the script files needed by commercial design tools to compile and place-and-route the VHDL into FPGA configuration codes. These files, along with a compilation script that controls the numerous steps in the compilation process, allows the entire process, from high level language compilation down to the production of FPGA configuration codes and the host-based control program, to be fully automated. The user can execute the entire algorithm on the hardware like any other application (by typing a .out or something similar), without needing to worry about any of the operational details of the hardware.

A. Preliminary Performance Results

Table I shows some of the statistics of the entire compilation/translation process. The 19 line SA-C program for the Prewitt algorithm eventually requires over 5000 lines of VHDL, and occupies approximately 30% of the CLBs in the

two FPGAs used in the implementation. Table I(c) shows the long delay of the magnitude function discussed earlier. Finally, Table I(d) shows the execution performance of the algorithm, first with no stripmining (i.e., a single inner loop body), and then with 4×3 stripmining (which results in two inner loop bodies). As expected, increasing the parallelism from one to two essentially doubles the processing rate.

Two comparisons of the execution results obtained by the SA-C system are appropriate. First, the Prewitt algorithm was coded manually in VHDL, with two inner loop instantiations (equivalent to a 4×3 stripmine in the SA-C version), and with the magnitude computation replaced with a lookup table. The design was further optimized by pipelining the resulting inner computation. This design was meant to represent optimal performance of the algorithm on the RCS. The resulting design runs at 17 MHz, shown in the third column of Table I(d), compared to only 2.82 MHz achieved by the automated design. Clearly the manual version enjoys a major advantage with its lookup table version of magnitude – it executes more than 6 times faster than the equivalent (stripmined) SA-C version. We believe that it is reasonable to expect that the performance of the SA-C version can be improved several fold, to within a few percentage points of the manual version, by using the same optimizations used in the manual method.

The second comparison is with an equivalent algorithm running on a Pentium. A C version of prewitt was coded in C and compiled with the gcc -O6 option; the results of execution on a 450MHz Pentium is shown in the last column of Table I(d). The results are compared to the stripmined version of the SA-C version. We are encouraged by these results, since we believe we can improve the execution time several fold. Porting the SA-C system to more modern hardware will improve performance even more.

Perhaps more important than performance is the saving gained in development time by using the automated approach. The entire Prewitt algorithm was coded in SA-C and converted to hardware in a matter of a few hours, compared with several weeks required by the manual method. Equally important, the development process can be carried out by an application programmer, with little or no knowledge of hardware design or VHDL. Control of compiler operation via pragmas allows the programmer to optimize the design by changing the parallelization included in the application.

VII. FUTURE WORK AND CONCLUSION

Up to now, most of the project effort has focused on functionality, rather than optimal performance, as evidenced by the performance numbers. Work on the compiler always stays several steps ahead of the DFG-to-VHDL translation effort. Nonetheless, a large portion of the SA-C language can now be translated to hardware; unimplemented language features are added on an as-needed basis. Numerous compiler optimizations which control how an application is mapped to hardware, such as stripmining and loop fusion, have been completed.

Now attention is turning toward optimizing the translation from DFG to hardware. In particular, two areas of optimization hold considerable promise:

- Lookup tables - many common operations are inefficient on reconfigurable hardware. We are implementing a scheme which allows a SA-C function to be converted to a lookup table, via a pragma. The original function is then replaced with table lookup code, which is much faster, although it often requires more space.
- ILB Pipelining - calculations in the ILB's tend to create circuits with long propagation delays, which require slow clock speeds. The propagation delays can be reduced by strategically placing pipeline registers in the ILB.

New technology promises to provide impressive performance gains, both in speed and in the amount of space available for programming. New generations of FPGAs provide resources, such as on-chip memory, which can easily be utilized to enhance the current system performance.

We are currently porting the system to a Virtex-based AMS Starfire board. Preliminary results indicate that the new board can accommodate applications up to 30 times larger than those on the current board. With no board-specific optimizations, applications execute 3-5 times faster than the same application on the older board.

Reconfigurable computing holds the promise of significant performance gains over conventional computing for certain types of problems. The automated process for application development described here promises to greatly reduce software development times for such systems, and to bring the realm of hardware design to the application programmer.

REFERENCES

- [1] "The Cameron Project," Information about the Cameron Project, including several publications, is available at the project's web site, www.cs.colostate.edu/cameron.
- [2] D. Buell, J. Arnold, and W. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*, IEEE CS Press, 1996.
- [3] Xilinx, Inc., San Jose, CA., *The Programmable Logic Databook*, 1998, www.xilinx.com.
- [4] D.T. Hoang, "Searching genetic databases on Splash 2," in *IEEE Workshop on FPGAs for Custom Computing Machines*. 1993, pp. 185-192, CS Press, Los Alamitos, CA.
- [5] D. V. Pryor, M. R. Thistle, and N. Shirazi, "Text searching on Splash 2," in *IEEE Workshop on FPGAs for Custom Computing Machines*. 1993, pp. 172-178, CS Press, Los Alamitos, CA.
- [6] N. K. Ratha, D. T. Jain, and D. T. Rover, "Fingerprint matching on Splash 2," in *Splash 2: FPGAs in a Custom Computing Machine*, pp. 117-140. IEEE CS Press, 1996.
- [7] N. K. Ratha, D. T. Jain, and D. T. Rover, "Convolution on Splash 2," in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*. 1995, pp. 204-213, CS Press, Los Alamitos, CA.
- [8] P. M. Athanas and A. L. Abbott, "Processing images in real time on a custom computing platform," in *Field-Programmable Logic Architectures, Synthesis, and Applications*, R. W. Hartenstein and M. Z. Servit, Eds., pp. 156-167. Springer-Verlag, Berlin, 1994.
- [9] Annapolis Micro Systems, Inc., Annapolis, MD, *WILDFORCE Reference Manual*, 1997, www.annapmicro.com.
- [10] Annapolis Micro Systems, Inc., Annapolis, MD, *STARFIRE Reference Manual*, 1999, www.annapmicro.com.
- [11] B. Schott, S. Crago, Chen C., J. Czarnaski, M. French, I. Hom, T. Tho, and T. Valenti, "Reconfigurable architectures for systems level applications of adaptive computing," Available from <http://www.east.isi.edu/SLAAC/>.

- [12] Xilinx, Inc., *Virtex 2.5V Field programmable Gate Arrays: Preliminary Product Description*, Oct. 1999, www.xilinx.com.
- [13] G. Lu, H. Singh, M. Lee, N. Bagherzadeh, and F. Kurhadi, "The Morphosis parallel reconfigurable system," in *Proc. of EuroPar 99*, Sept. 1999.
- [14] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to software: RAW machines," *Computer*, September 1997.
- [15] W.H. Mangione-Smith, "Seeking solutions in configurable computing," *IEEE Computer*, vol. 30, pp. 38–43, Dec. 1997.
- [16] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "Piperench: A coprocessor for streaming multimedia acceleration," in *Proc. Intl. Symp. on Computer Architecture (ISCA '99)*, 1999, www.cs.cmu.edu/~mihaiib/research/isca99.ps.gz.
- [17] A. Agarwal, S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, and M. Srikrishna, D.and Taylor, "The RAW compiler project," in *Proc. Second SUIF Compiler Workshop*, August 1997.
- [18] S. C. Goldstein and M. Budiu, *The DIL Language and Compiler Manual*, Carnegie Mellon University, 1999, www.ece.cmu.edu/research/piperench/dil.ps.
- [19] OXFORD Hardware Compiler Group, "The Handel language," Tech. Rep., Oxford University, 1997.
- [20] M. Gokhale, "The Streams-C Language," 1999, www.darpa.mil/ito/psum1999/F282-0.html.
- [21] "SystemC. SystemC homepage," www.systemc.org/.
- [22] "IMEC. Ocapi overview," www.imec.be/ocapi/.
- [23] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Chang, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, and M. Walkden, "MATCH: a MATLAB compiler for configurable computing systems," Tech. Rep. CPDC-TR-9908-013, Center for Parallel and distributed Computing, Northwestern University, August 1999.
- [24] S. Periyayacheri, A. Nayak, A. Jones, N. Shenoy, A. Choudhary, and P. Banerjee, "Library functions in reconfigurable hardware for matrix and signal processing operations in MATLAB," in *Proc. 11th IASTED Parallel and Distributed Computing and Systems Conf. (PDCS'99)*, November 1999.
- [25] S. Natarajan, B. Levine, C. Tan, D. Newport, and D. Bouldin, "Automatic mapping of Khoros-based applications to adaptive computing systems," Tech. Rep., University of Tennessee, 1999, Available from <http://microsys6.engr.utk.edu:80/~bouldin/darpa/mapld2/mapld.paper.pdf>.
- [26] K. Konstantinides and J. Rasure, "The Khoros software development environment for image and signal processing," in *IEEE Transactions on Image Processing*, May 1994, vol. 3, pp. 243–252.
- [27] J. Hammes and W. Böhm, *The SA-C Language - Version 1.0*, 1999, Document available from www.cs.colostate.edu/cameron.
- [28] J. M. S. Prewitt, "Object enhancement and extraction," in *Picture Processing and Psychopictorics*, B. S. Lipkin and A. Rosenfeld, Eds. Academic Press, New York, 1970.
- [29] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, Addison-Wesley, Reading, MA, 1992.

CAMERON PROJECT: FINAL REPORT

Appendix H: A High Level, Algorithmic Programming Language and
Compiler for Reconfigurable Systems

A High Level, Algorithmic Programming Language and Compiler for Reconfigurable Systems *

Jeffrey P Hammes, Robert Rinker, Wim Böhm, Walid A. Najjar, Bruce Draper
Computer Science Department
Colorado State University
Ft. Collins, CO, U.S.A.

Abstract *This paper presents a high level, machine independent, algorithmic, single-assignment programming language SA-C and its optimizing compiler targeting reconfigurable systems, and intended for Image Processing applications. Language features are introduced and discussed. The intermediate forms DDCF and DFG, used in the optimization and code-generation phases are described. Conventional and reconfigurable system specific optimizations are briefly introduced. The code generation process, using an abstract target machine, is described. Finally the performance effects of combinations of various optimizations are compared to hand coded C, using an edge detection algorithm followed by a threshold operator. Timing results are encouraging. Improvements of the compilation and code generation route are discussed.*

Keywords: Reconfigurable Computing Systems, FPGA, Image Processing, High Level Languages, Optimizing Compilation

1 Introduction

This paper presents an algorithmic programming language, SA-C [6] (derived from “single-assignment C, and pronounced “sassy”) and its optimizing compiler targeting reconfigurable systems. SA-C has been initially designed for Image Processing applications, while being amenable to efficient compilation to fine grain parallel hardware systems.

*This work is supported by DARPA under US Air Force Research Laboratory contract F33615-98-C-1319.

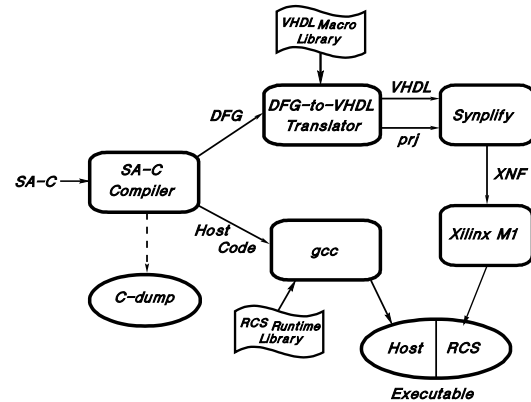


Figure 1. System overview

Currently, FPGAs are programmed in hardware description languages, such as VHDL or Verilog. While such languages are suitable for chip design, they are poorly suited for the kind of algorithmic expression that takes place in applications programming. Applications programmers, who want to exploit the potential of these reconfigurable systems, are discouraged by the difficulty of implementing algorithms in circuit design languages, and by the amount of hardware specific knowledge needed to use these systems.

Mapping an *algorithmic programming language* to reconfigurable hardware brings new challenges to language designers and compiler writers. The language must allow easy extraction of fine grain parallelism as well as aggressive optimization, both in terms of code space and execution time. The Cameron Project [7, 10] provides such a high level, algo-

rithmic language, SA-C, and optimizing compiler for the development of image processing algorithms on reconfigurable computing systems, see Figure 1.

2 The SA-C Language

The design goals of the SA-C language are

- high-level, algorithmic language
- single-assignment, for better compiler analysis and translation to DFGs
- no pointers or side effects, for better compiler analysis
- emphasis on loops and arrays
- high-level operators for IP applications
- operator syntax and precedences as in C
- variable bit-width data types
- user control of optimizations

The emphasis on loops and arrays accommodates the efficient expression of low and medium level Image Processing algorithms, which form the FPGA target application domain of the Cameron Project. As SA-C does not allow dynamic data structures and recursion, more irregular computations, e.g. involving trees and graphs, are not easily expressed.

SA-C differs significantly from other efforts to map higher level languages to FPGAs. Handel-C [1] programs are closer to hardware than SA-C programs. For example, timing is explicit in Handel-C. The parallelism in Handel-C is also more explicit: the user must declare processes and interconnecting channels. Ocapi [9] and SystemC [12] are C++ extensions that allow the user, through the use of class libraries, to begin creating an application at a high level and gradually migrate certain parts of the code toward a more explicit hardware description. By the time the user is down at the hardware level, the languages are in effect hardware description languages, but with a more familiar look. There is

no emphasis on aggressive automatic compiler optimizations as in SA-C. Another project using C++ is Streams-C [3]. The language model has processes and streams, and the compiler uses the SUIF infrastructure.

Data types in SA-C include signed and unsigned integers and fixed point numbers, with user-specified bit widths. Since SA-C is a single-assignment language, each variable's declaration occurs together with the expression giving it its value. (This approach prevents semantically unpleasant "dynamic single-assignment" situations such as declaring a variable in an outer code block and potentially assigning to it in only one part of a conditional.) SA-C has multidimensional rectangular arrays whose extents are determined dynamically or statically. The type declaration **int14 M[:,6]** is a declaration of a matrix **M** of 14-bit signed integers. The left dimension is determined dynamically; the right dimension is specified by the user. The most important part of SA-C is its treatment of **for** loops. A loop in SA-C returns one or more values (i.e., a loop is an expression), and has three parts: one or more generators, a loop body and one or more return values. The generators interact closely with arrays, providing array access expression that is concise for the user and easy for the compiler to analyze. Most interesting is the **window** generator, which extracts sub-arrays from a source array. Here is a median filter written in SA-C:

```
uint8 R[:,:] =
    for window W[3,3] in A {
        uint8 med = array_median (W);
    } return (array (med));
```

The **for** loop is driven by the extraction of 3x3 sub-arrays from array **A**. All possible 3x3 arrays are taken, one per loop iteration. The loop body takes the median of the sub-array, using a built-in SA-C operator. The loop returns an array of the median values, whose shape is derived from the shape of **A** and the loop's generator. In this example, if array **A** had a shape of [100,200], the result array **R**

would have a shape of [98,198]. SA-C’s generators can take windows, slices and scalar elements from source arrays, making it frequently unnecessary for source code to do any explicit array indexing whatsoever.

SA-C **for** loops may have “nextified” variables, a mechanism borrowed from other loop-oriented functional languages to allow loop-carried dependencies to be expressed. In the absence of nextified variables, a SA-C loop is fully parallel, and this loop-level parallelism can be exploited by the compiler in various ways. The presence of a nextified variable imposes an execution order on the loop. In SA-C this order is a row-major traversal of each array accessed by the loop’s generators.

A SA-C program compiles to a host machine executable that has calls to a reconfigurable coprocessor board. The system can also compile the entire program to a host executable for efficient program debugging. When compiling calls to reconfigurable hardware, it transforms bottom-level loops into dataflow graphs (DFGs) [8], suitable for mapping onto FPGAs. The host code includes interface code that automatically downloads FPGA configurations and source data, and uploads the results for further computation on the host.

The SA-C compiler attempts to translate every bottom-level loop (i.e., a loop that contains no loop) to a dataflow graph (DFG), a low-level, non-hierarchical and asynchronous program representation that will be mapped for execution on reconfigurable hardware. (In the present system, not all loops can be translated to DFGs. The most important limitation is the requirement that the sizes of a loop’s window generators be statically known.) DFGs can be viewed as abstract hardware circuit diagrams without timing considerations taken into account. Nodes are operators and edges are data paths. The dataflow graphs are designed to allow token driven simulation, used by the compiler writer and applications programmer for validation and debugging. There are four general classes of node types that can appear in a dataflow graph:

- arithmetic
- low level control (e.g. selective merge)
- data extraction and routing nodes that reflect the generators that drive a loop
- data collection nodes that accumulate a loop’s return values

3 Optimizations and pragmas

The SA-C compiler does a variety of optimizations, some traditional and some specifically designed to suit the language and its reconfigurable hardware targets. The compiler converts the entire SA-C program to an internal dataflow form called “Data Dependence and Control Flow” (DDCF) graphs [5], which it uses to perform all optimizations [4]. The traditional optimizations include Common Subexpression Elimination, Constant Folding, Invariant Code Motion, and Dead Code Elimination. The compiler also does specialized variants of Loop Stripmining, Array Value Propagation, Loop Fusion, Loop Unrolling, Function Inlining, Lookup Tables and Array Blocking, along with loop and array Size Propagation Analysis. Some of these interact closely and are now described briefly.

Since SA-C targets FPGAs, the compiler does aggressive full loop unrolling, which converts a loop to a non-iterative block of code more suitable for translating to a DFG. To help identify opportunities for unrolling, the compiler propagates array sizes through the DDCF graph, inferring sizes wherever possible. SA-C’s close association of arrays and loops makes this possible. Since the compiler converts only bottom-level loops to dataflow graphs, full loop unrolling can allow a higher-level loop to become a bottom-level loop, allowing it then to be converted to a DFG.

The SA-C compiler can do Loop Stripmining on parallel **for** loops, which when followed by full loop unrolling produces the effect of multi-dimensional partial loop unrolling. For example, a stripmine pragma can be added to the median filter:

```

uint8 R[:,:] =
  // PRAGMA (stripmine (6,4))
  for window W[3,3] in A {
    uint8 med = array_median (W);
  } return (array (med));

```

This wraps the existing loop in a new loop with a 6x4 window generator. Loop unrolling then replaces the inner loop with eight median code bodies. The resulting loop takes 6x4 sub-arrays and computes the eight 3x3 medians in parallel. Since the new loop has non-unit strides, there are fewer loop iterations.

The SA-C compiler can fuse many loops that have a producer/consumer relationship. For example, a Prewitt edge detector [11] might be followed by a threshold operator, as shown here

```

int3 vert[3,3] =
  {{-1, 0, 1}, {-1, 0, 1}, {-1, 0, 1}};
int3 horz[3,3] =
  {{-1,-1,-1}, { 0, 0, 0}, { 1, 1, 1}};

uint8 R0[:,:] =
  for window win[3,3] in image {
    int11 v =
      for elem1 in win dot elem2 in vert
        return(sum((int11)elem1*elem2));
    int11 h =
      for elem3 in win dot elem4 in horz
        return(sum((int11)elem3*elem4));
    int22 sqv = (int22)v*v;
    int22 sqh = (int22)h*h;
    uint8 mag = sqrt((int23)sqv+sqh);
  } return (array (mag));

uint8 R1[:,:] =
  for pix in R0 {
    uint8 v = pix>127 ? 255 : 0;
  } return (array (v));

```

where the bit-widths and casts, which behave the same as in C, cause the operations to be done correctly using the least number of bits. The compiler will fuse the loops into one new loop, eliminating the intermediate array R0. The SA-C compiler also is able to fuse a loop pair in which both loops have window generators, and is able to fuse a pipeline of such loops. The goal of fusion is primarily to reduce both host/coprocessor-board and local-memory/FPGA data communication. The

user can prevent fusion by placing a **no_fuse** pragma on the producer loop.

Lookup tables are often an attractive alternative to complex computations. SA-C allows a function to be given a pragma that tells the compiler to convert the specified function to a lookup table. The compiler computes all possible values of the function, building them into an array, and it converts all calls to the function to array lookups. This is feasible in SA-C, as the language allows small bit-width data types.

Though SA-C is a high-level language, it gives users control over the compilation process through the use of pragmas. The user can control function inlining, loop fusion, loop unrolling, array blocking, stripmining and lookup table conversion through the use of pragmas. In addition, the user can create a function prototype that is designated as an external VHDL plug-in; the SA-C compiler will pass calls to the designated function down through the DDCF and dataflow graphs, leaving “holes” that can be filled in at low level with a user’s own VHDL routine.

4 Low-level implementation

Unlike standard processors, which provide a relatively small set of well-defined instructions to the user, reconfigurable computing systems are composed of an amorphous mass of logic cells, which can be interconnected in a countless number of ways. To impose structure on the compilation process, an *abstract machine* has been defined, shown in Figure 2. The DFG for a SA-C program consists of a loop generator, an inner loop body (ILB), and a data collector. The loop generator reads data from local memory and presents it to the ILB. Values that are calculated by the ILB are then collected before being written to memory. The ILB is combinational; all timing and control of the computation process is handled by the loop generator and data collector.

The implementation of a window generator uses shift registers. In each loop iteration the oldest column of data is shifted out and a new

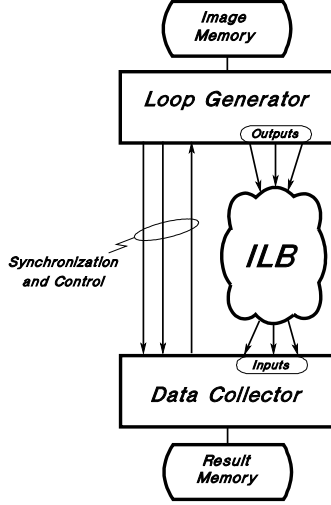


Figure 2. Abstract machine structure

column is shifted in. Words are read from the FPGA’s local memory as needed. The collector accepts the ILB outputs, buffers them into words, and writes them into the result memory. These steps are controlled by the window generator: if more than one value is produced by the ILB, timing signals within the window generator insure that the collector has enough time to write the data before the next window of data is produced.

5 Performance

This section discusses the effects of some of the SA-C compiler’s optimizations, using the Prewitt/threshold loop pair shown in Section 3, run on a Wildforce^(TM) board from Annapolis Microsystems[2]. The code is compiled in three ways: unfused (two separate loops, with two separate data round trips), fused, and fused with 4x3 stripmining. Each loop execution on the reconfigurable system requires the downloading of data, running of the loop, and uploading of the result. The downloads and uploads are done using DMA on the PCI bus. The loop execution takes one board cycle per iteration, regardless of the loop body size. However, the clock frequency varies from loop to loop, and is determined by the critical path length, i.e. propagation delay, of

	unfused		fused	fused, strip
	prew	thresh		
data download (msec)	0.54	0.51	0.54	0.54
execute (msec)	59.14	11.34	26.95	17.86
data upload (msec)	0.93	0.91	0.93	0.94
total time (msec)	60.61	12.76	28.42	19.34
freq (Mhz)	2.48	10.35	5.45	4.53
execute (K cycles)	146.9	117.8	147.0	80.9

Table 1. Performance of loop fusion and stripmining.

the circuit that is derived from the place-and-route step that creates the configuration file for the FPGA. Clock frequency on the Wildforce board also affects the speed of memory transactions between the FPGA and its local memory, since there is one memory access per clock cycle.

Table 1 shows the resulting times, as well as the board frequencies and number of execution cycles, run on an image of 198 by 300 pixels. The unfused loop pair takes two data round trips, and two loop executions. The Prewitt loop runs at only 2.5 MHz due to the long path length of the square root computation. The loop pair takes 73.37 msec (the sum of the two individual loop executions). When the loops are fused, the total execution time drops to 28.42 msec. The performance increase is due to a number of factors: First, the total number of iterations is cut in half. Second, the clock frequency is higher than that of the Prewitt alone¹. Third, there is one data round trip rather than two, though the DMA transfer

¹The rise in frequency is probably due to the fact that the threshold is comparing with 127. This requires looking at only the high-order bit, and allows the low-level FPGA mapping software to eliminate parts of the square root computation, thus reducing the critical path length.

times are so fast that this makes a very small difference.

Stripmining improves the time further, to 19.34 msec. This gain is primarily due to a reduction of FPGA reads from its local memory: when stripmined to a 4x3 window, the window has a vertical stride of two, so each data row is read twice instead of three times. There is also some improvement due to a further reduction in number of iterations, which saves some loop overhead. Studies in the Cameron group have shown that deep stripmining is able to give significant improvements in performance. Unfortunately the FPGAs on the WildForce board are small, and deeper stripmining of this example was not possible because of space constraints.

5.1 SA-C vs. a Hand-Coded VHDL Example

During development of the abstract model, several algorithms were manually-coded in VHDL. These hand-coded examples provided considerable insight into the issues that the compiler system must address in producing efficient hardware codes, and motivated many of the optimizations incorporated into the compiler. The performance of these manual implementations, which are optimized to use specific hardware resources, serve as a benchmark with which to judge the performance of the automated system.

Table 2 shows the execution times for a manually coded Prewitt design. It contains 2 ILB's, comparable to the 4×3 stripmined version described above. It differs in that it uses a lookup table for the magnitude computation (two multiplies and a square root) that accounts for most of the propagation delay of the loop body. As a result, the manual version is able to run more than two times faster than the automated one (10.1MHz vs. 4.53MHz). A second version of the manual design, which adds one stage of pipelining to the ILB, reduces the propagation delay by another 70%, allowing execution at nearly four times the speed of the automated version. Both lookup tables and pipelining are being added to the compilation system, and

	Hand-Coded Prewitt	
	Non-pipelined	Pipelined
execute (msec)	7.15	4.66
freq (Mhz)	10.1	17.0

Table 2. Performance of hand-coded Prewitt

should yield results that are similar to the manual version.

5.2 SA-C Performance vs. C

Performance comparisons across platforms and languages are always difficult and sometimes contentious, but it is nevertheless useful to put these execution times into some kind of perspective. The inner loop of a separately developed C program for the Prewitt-threshold example, hand-fused, is shown here.

```

for (i=0; i<ysz-2; i++)
  for (j=0; j<xsz-2; j++){
    pt = image + i*xsz + j;
    r0 = (int)pt[0] + pt[1] + pt[2];
    r2 = (int)pt[0+2*xsz] + pt[1+2*xsz]
        + pt[2+2*xsz];

    c0 = (int)pt[0] + pt[xsz]
        + pt[2*xsz];
    c2 = (int)pt[2] + pt[2+xsz]
        + pt[2+2*xsz];

    rsm = r0-r2;
    csm = c2-c0;
    mag = sqrt ((double)rsm*rsm + csm*csm);

    res = mag>127 ? 1 : 0;

    *(rpt++) = res;
  }

```

This code was compiled using *gcc -O6* and executed on both 266 MHz and 800 MHz Pentium based machines, yielding execution times of 42.6 msec and 14.0 msec, respectively. We are encouraged by the comparison of these results with the SA-C code's performance, especially since the FPGA technology being used in this comparison is a number of years old.

6 Conclusions and Future Work

The main thrust of the Cameron research project is to provide to applications programmers the ease of programming for reconfigurable systems that they have enjoyed for conventional architectures. This has already been achieved for that subset of programs that the compiler can currently map to FPGAs. The three versions of the Prewitt/threshold code differ only in one or two pragma lines, and were written, compiled and executed in a matter of hours. This contrasts to the days, sometimes weeks, of development time required for VHDL programs.

The optimizations currently available in the SA-C compiler have been shown to be highly effective for the kind of IP codes we have tested. In the above example, the original inner loop body was fully unrolled and array constants were propagated and folded. This gave rise to the unfused version. Loop fusion enlarged the loop body run on the FPGA, and reduced costly FPGA to local memory access, bringing the execution time down from 73.37 msec to 28.42 msec for an image of 198 by 300 pixels. This was further reduced to 19.34 msec by loop stripmining.

Work is currently underway to port the system to target boards containing Virtex [13] FPGAs. The order-of-magnitude space increase of this chip over the current FPGAs will allow deeper stripmining as well as fusion of longer loop pipelines. The chip's RAM Block memory will be used for lookup tables. Its higher clock frequencies, along with pipelining of the inner loop body, will allow significantly faster execution speeds.

In the compiler, work is currently underway to implement some novel optimizations designed to save space in the FPGAs. One program transformation will perform a kind of common subexpression elimination across loop iterations. Another will reduce window sizes by moving subexpressions across iterations. The saved space will allow still more aggressive application of the optimizations that have already been developed.

References

- [1] OXFORD hardware compiler group, the Handel Language. Technical report, Oxford University, 1997.
- [2] Annapolis Micro Systems, Inc., Annapolis, MD. *WILDFORCE Reference Manual*, 1997. www.annapmicro.com.
- [3] M. Gokhale. The Streams-C language. www.darpa.mil/ito/psum1999/F282-0.html.
- [4] J. Hammes. *Compiling SA-C to Reconfigurable Computing Systems*. PhD thesis, Colorado State University, 2000.
- [5] J. Hammes and W. Böhm. *The SA-C Compiler DDCF Graph Description*, 1999. Document available from www.cs.colostate.edu/cameron.
- [6] J. Hammes and W. Böhm. *The SA-C Language - Version 1.0*, 1999. Document available from www.cs.colostate.edu/cameron.
- [7] J. Hammes, R. Rinker, W. Böhm, and W. Najjar. Cameron: High level language compilation for reconfigurable systems. In *PACT'99*, Oct. 1999.
- [8] J. Hammes, R. Rinker, D. McClure, W. Böhm, and W. Najjar. *The SA-C Compiler Dataflow Description*, 1999. Document available from www.cs.colostate.edu/cameron.
- [9] IMEC. Ocapi overview. www.imec.be/ocapi/.
- [10] W. Najjar. The Cameron Project. Information about the Cameron Project, including several publications, is available at the project's web site, www.cs.colostate.edu/cameron.
- [11] J. M. S. Prewitt. Object enhancement and extraction. In B. S. Lipkin and A. Rosenfeld, editors, *Picture Processing and Psychopictorics*. Academic Press, New York, 1970.
- [12] SystemC. SystemC homepage. www.systemc.org/.
- [13] Xilinx, Inc. *Virtex 2.5V Field programmable Gate Arrays: Preliminary Product Description*, Oct. 1999. www.xilinx.com.

CAMERON PROJECT: FINAL REPORT

Appendix I: Mapping a Single Assignment Programming Language to Reconfigurable Systems

Mapping a Single Assignment Programming Language to Reconfigurable Systems *

W. Böhm[†], J. Hammes, B. Draper, M. Chawathe, C. Ross and R. Rinker
Colorado State University

W. Najjar
University of California Riverside

Abstract. This paper presents the high level, machine independent, algorithmic, single-assignment programming language SA-C and its optimizing compiler targeting reconfigurable systems. SA-C is intended for Image Processing applications. Language features are introduced and discussed. The intermediate forms DDCF, DFG and AHA, used in the optimization and code-generation phases, are described. Conventional and reconfigurable system specific optimizations are introduced. The code generation process is described. The performance for these systems is analyzed, using a range of applications from simple Image Processing Library functions to more comprehensive applications, such as the ARAGTAP target acquisition prescanner.

Keywords: Reconfigurable Computing Systems, FPGA, Image Processing, High Level Languages, Optimizing Compilation.

1. Introduction

Recently, the computer vision and image processing communities have become aware of the potential for massive parallelism and high computational density in FPGAs. FPGAs have been used for real-time point tracking (Benedetti and Perona, 1998), stereo vision (Woodfill and v. Herzen, 1997), color-based detection (Benitez and Cabrera, 1999), image compression (Hartenstein et al., 1995), and neural networks (Eldredge and Hutchings, 1994). The biggest obstacle to the more widespread use of reconfigurable computing systems lies in the difficulty of developing application programs. FPGAs are typically programmed using hardware description languages such as VHDL (Perry, 1993). Application programmers are typically not trained in these hardware description languages and usually prefer a higher level, algorithmic programming language to express their applications.

The Cameron Project (Hammes et al., 1999) has created a high-level algorithmic language, named SA-C (Hammes and Böhm, 1999), for ex-

* This work is supported by DARPA under US Air Force Research Laboratory contract F33615-98-C-1319.

[†] bohm@cs.colostate.edu

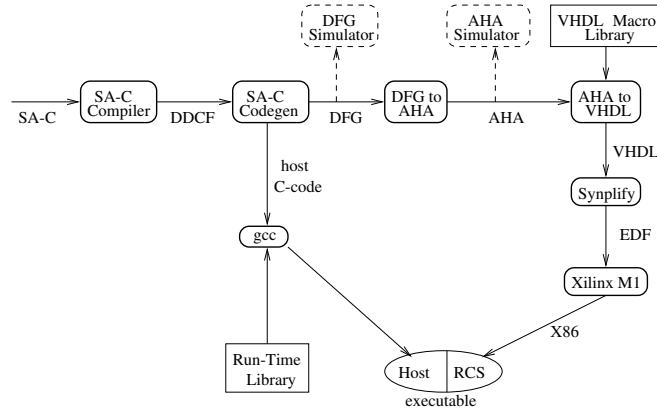


Figure 1. SA-C Compilation system.

pressing image processing applications and compiling them to FPGAs. The SA-C compiler provides one-step compilation to host executable and FPGA configurations. After parsing and type checking, the SA-C compiler converts the program to a hierarchical data dependence and control flow (DDCF) graph representation. DDCF graphs are used in many optimizations, some general and some specific to SA-C and its target platform. After optimization, the program is converted to a combination of dataflow graphs (DFGs) and host code. DFGs are then compiled to VHDL code via Abstract Hardware Architecture (AHA) graphs. The VHDL code is synthesized and place-and-routed to FPGAs by commercial software tools. Figure 1 shows a high-level view of the system.

To aid in program development, it is possible to view and simulate intermediate forms. For initial debugging the complete SA-C program can be executed on the host. All intermediate graph forms can be viewed, and DFG and AHA graphs can be simulated. The SA-C compiler can run in stand-alone mode, but it also has been integrated into the Khoros^(TM) (Konstantinides and Rasure, 1994) graphical software development environment.

The rest of this paper is organized as follows. An overview of the SA-C language is presented in section 2. Compiler optimizations and pragmas are discussed in section 3. Translations to dataflow graphs and then to VHDL via AHA are discussed in section 4. Applications and their performance data are presented in section 5. References to related work are given in section 6, and section 7 concludes and describes future work.

2. The SA-C Language

The design goals of SA-C are to have a language that can express image processing (IP) applications elegantly, and to allow seamless compilation to reconfigurable hardware. IP applications are supported by data parallel for loops with structured access to rectangular multidimensional arrays. Reconfigurable computing requires fine grain expression level parallelism, which is easily extracted from a SA-C program because of its *single assignment* semantics. Variables in SA-C are associated with wires, not with memory locations. This avoids von Neumann memory model complications and allows for better compiler analysis and translation to DFGs. Data types in SA-C include signed and unsigned integers and fixed point numbers, with user-specified bit widths. The extents of SA-C arrays can be determined either dynamically or statically. The type declaration `int14 M[:,6]` for example, is a declaration of a matrix M of 14-bit signed integers. The left dimension will be determined dynamically; the right dimension has been specified.

The most important aspect of SA-C is its treatment of for loops and their close interaction with arrays. SA-C is expression oriented, so every construct (including a loop) returns one or more values. A loop has three parts: one or more generators, a loop body and one or more return values. The generators provide parallel array access operators that are concise and easy for the compiler to analyze. There are four kinds of loop generators: *scalar*, *array-element*, *array-slice* and *window*. The scalar generator produces a linear sequence of scalar values, similar to Fortran's do loop. The array-element generator extracts scalar values from a source array, one per iteration. The array-slice generator extracts lower dimensional sub-arrays (e.g. vectors out of a matrix). Finally, window generators allow rectangular sub-arrays to be extracted from a source array. All possible sub-arrays of the specified size are produced, one per iteration. The dot product operator combines generators and runs them in lock step. A loop can return arrays and reductions built from values that are produced in the loop iterations, such as sum, product, min, and max.

Figure 2 shows SA-C code for the Prewitt edge detector (Prewitt, 1970), a standard IP operator. The outer for loop is driven by the extraction of 3x3 sub-arrays from array `limage`. Its loop body applies two masks to the extracted window `W`, producing a magnitude. An array of these magnitudes is collected and returned as the program's result. The shape of the return array is derived from the shape of `limage` and the loop's generator. If `limage` were a 100x200 array, the result array `M` would have a shape of 98x198.

```

int16[:,:] main (uint8 Image[:,:]) {
  int16 H[3,3] = {{-1,-1,-1}, { 0, 0, 0}, { 1, 1, 1}};
  int16 V[3,3] = {{-1, 0, 1}, {-1, 0, 1}, {-1, 0, 1}};
  int16 M[:,:] =
    for window W[3,3] in Image {
      int16 dfdy, int16 dfdx =
        for h in H dot w in W dot v in V
          return(sum(h*w), sum(v*w));
      int16 magnitude =
        sqrt(dfdy*dfdy+dfdx*dfdx);
    }return(array (magnitude));
}return(M);

```

Figure 2. Prewitt Edge detector code in SA-C.

Loop carried values are allowed in SA-C using the keyword `next` instead of a type specifier in a loop body. This indicates that an initial value is available outside the loop, and that each iteration can use the current value to compute a next value.

3. Optimizations and pragmas

The compiler’s internal program representation is a hierarchical graph form called the “Data Dependence and Control Flow” (DDCF) graph. DDCF subgraphs correspond to source language constructs. Edges in the DDCF express data dependencies, opening up a wide range of loop- and array-related optimization opportunities.

Figure 3 shows the initial DDCF graph of the Prewitt program of Figure 2. The `FORALL` and `DOT` nodes are compound, containing subgraphs. Black rectangles along the top and bottom of a compound node represent input ports and output ports. The outer `FORALL` has a single window generator operating on a two-dimensional image, so it requires window size and step inputs for each of the two dimensions. In this example, both dimensions are size three, with unit step sizes. The output of the `WINDOW_GEN` node is a 3x3 array that is passed into the inner `FORALL` loop. This loop has a `DOT` graph that runs three generators in parallel, each producing a stream of nine values from its source array. Each `REDUCE_SUM` node sums a stream of values to a single value. Finally, the `CONSTRUCT_ARRAY` node at the bottom of the outer loop takes a stream of values and builds an array with them.

Many IP operators involve fixed size and often constant convolution masks. A ***Size Inference*** pass propagates information about constant

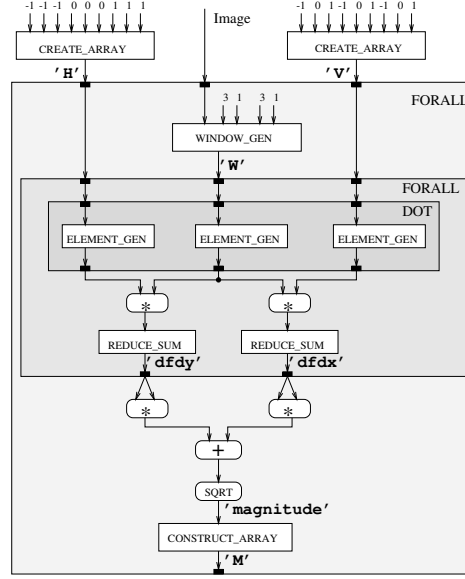


Figure 3. DDCF graph for Prewitt program.

size loops and arrays through the dependence graph. Array size information can propagate from a loop or source array to its target and vice versa. In addition, size information from one generator can be used to infer sizes of other generators. Effective size inference allows other optimizations, such as Full Loop Unrolling and Array Elimination, to take place.

Full Unrolling of loops with small, compile time assessable numbers of iterations can be important when generating code for FPGAs, because it spreads the iterations in code space rather than in time. Small loops occur frequently as inner loops in IP codes, for example in convolutions with fixed size masks.

Array Value Propagation searches for array references with constant indices, and replaces such references with the values of the array elements. When the value is a compile time constant, this enables constant propagation. In the Prewitt example, this optimization removes the arrays H and V entirely.

Loop Carried Array Elimination The most efficient representation of arrays in loop bodies is to have their values reside in registers. The important case is that of a loop carried array that changes values but not size during each iteration. To allocate a fixed number of registers for these arrays two requirements need to be met. 1) The compiler must be able to infer the size of the initial array computed outside the

loop. 2) Given this size, the compiler must be able to infer that the next array value inside the loop is of the same size.

N-dimensional Stripmining extends stripmining (Wolfe, 1996) and creates an intermediate loop with fixed bounds. The inner loop can be fully unrolled with respect to the newly created intermediate loop, generating a larger, more parallel circuit. The compiler generates code to compute left over fringes.

Some (combinations of) operators can be inefficient to implement directly in hardware. For example the computation of **magnitude** in Prewitt requires multiplications and square root operators. The evaluation of the whole expression can be replaced by an access to a ***Lookup Table***, which the compiler creates by wrapping a loop around the expression, recursively compiling and executing the loop, and reading the results.

The performance of many systems is limited by the time required to move data to the processing units. ***Fusion*** of producer-consumer loops is often helpful, since it reduces data traffic and may eliminate intermediate data structures. In simple cases, where arrays are processed element-by-element, this is straightforward. However, the windowing behavior of many IP operators presents a challenge. Consider the following loop pair:

```
uint8 R0[:, :] =
    for window W[2,2] in Image return (array (f(W)));
uint8 R1[:, :] =
    for window W[2,2] in R0 return (array (g(W)));
```

If **Image** is a $d_0 \times d_1$ array, **R0** is a $(d_0 - 1) \times (d_1 - 1)$ array, and **R1** will be $(d_0 - 2) \times (d_1 - 2)$, so the two loops do not have the same number of iterations. Nevertheless, it is possible to fuse such a loop pair by examining their data dependencies. One element of **R1** depends on a 2x2 sub-array of **R0**, and the four values in that sub-array together depend on a 3x3 sub-array of **Image**. It is possible to replace the loop pair with one new loop that uses a 3x3 window and has a loop body that computes one element of **R1** from nine elements of **Image**.

Common Subexpression Elimination (CSE) is a well known compiler optimization that eliminates redundancies by looking for identical subexpressions that compute the same value. This could be called “spatial CSE” since it looks for common subexpressions within a block of code. The SA-C compiler performs conventional spatial CSE, but it also performs ***Temporal CSE***, looking for values computed in one loop iteration that were already computed in previous loop iterations. In such cases, the redundant computation can be eliminated by holding such values in registers so that they are available later and need not be

recomputed. Here is a simple example containing a temporal common subexpression:

```
for window W[3,2] in A {
    uint8 s0 = array_sum (W[:,0]);
    uint8 s1 = array_sum (W[:,1]);
} return (array (s0+s1));
```

This code computes a separate sum of each of the two columns of the window, then adds the two. Notice that after the first iteration of the loop, the window slides to the right one step, and the column sum `s1` in the first iteration will be the same as the column sum `s0` in the next iteration. By saving `s1` in a register, the compiler can eliminate one of the two column sums, nearly halving the space required for the loop body.

A useful phenomenon often occurs with Temporal CSE: one or more columns in the left part of the window are unreferenced, making it possible to eliminate those columns. *Narrowing* the window lessens the FPGA space required to store the window's values.

In many cases the performance tradeoffs of various optimizations are not obvious; sometimes they can only be assessed empirically. The SA-C compiler allows many of its optimizations to be controlled by *user pragmas* in the source code. This allows the user to experiment with different approaches and evaluate the space-time tradeoffs.

4. Compiler Backend

A dataflow graph (DFG) is a non-hierarchical and asynchronous program representation. DFGs can be viewed as abstract hardware circuit diagrams without timing or resource contention taken into account. Nodes are operators and edges are data paths. DFGs have token driven semantics.

The SA-C compiler attempts to translate every innermost loop to a DFG. The innermost loops the compiler finds may not be the innermost loops of the original program, as loops may have been fully unrolled or stripmined. In the present system, not all loops can be translated to DFGs. The most important limitation is the requirement that the sizes of a loop's window generators be statically known.

In the Prewitt program shown earlier, the DDCF graph is transformed to the DFG shown in Figure 4. The SUM nodes can be implemented in a variety of ways, including a tree of simple additions. The window generator also allows a variety of implementations, based on the use of shift registers. The CONSTRUCT_ARRAY node streams its values out to a local memory.

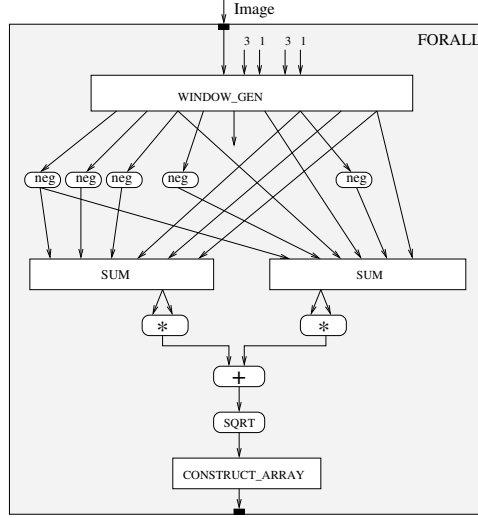


Figure 4. DFG for Prewitt after optimizations.

DFGs are translated into a lower level form called Abstract Hardware Architecture (AHA). This is also a graph form, but with nodes that are more fine-grained than DFG nodes and that can be translated to simple VHDL components. AHA graphs have *clocked*, *semi-clocked* and *non-clocked* nodes. The clocked and semi-clocked nodes have internal state but only the clocked nodes participate in the *handshaking* needed to synchronize computations and memory accesses. Some clocked nodes communicate via an arbitrator with a local memory. An AHA graph is organized as a sequence of *sections*, each with a top and a bottom boundary. A section boundary consists of clocked nodes, whereas its internal nodes are non-clocked or semi-clocked. In the AHA model, a section fires when all clocked nodes at its top boundary can produce new values and all clocked nodes at its bottom boundary can consume new values. This contrasts with DFGs, where each node independently determines when it can fire.

The AHA graph of the Prewitt code is too large and complex to display in this paper (1568 nodes). Figure 5 shows a dataflow graph on the left and an AHA graph on the right of a much simpler code fragment, which copies a one dimensional array A to another array B:

```
uint32 B[:] = for a in A return(array(a))
```

The DFG consists of an Element Generator node extracting elements out of A, and a Construct Tile node collecting the elements of B. The top three nodes in the AHA graph implement the Element Generator

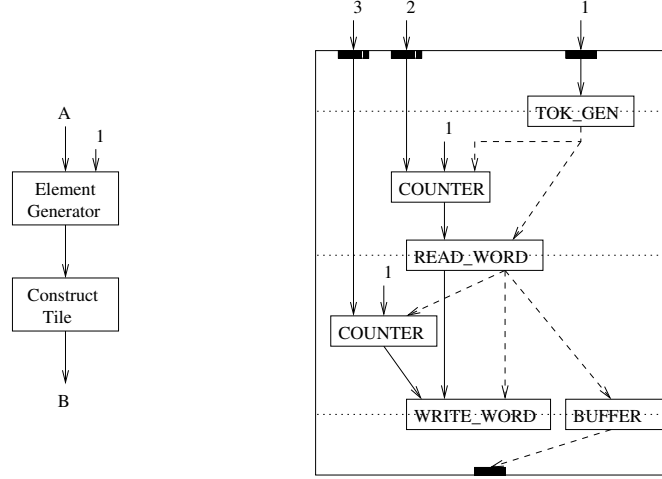


Figure 5. One-dimensional array copy in dataflow and AHA.

node, whereas the bottom three AHA nodes implement the Construct Tile node. The inputs 1, 2 and 3 in the AHA graph represent the extents of A, the start address of A, and the start address of B, respectively. Dotted horizontal lines represent section boundaries. Upon an input n , a TOK_GEN node produces $n+1$ control signals (dashed edges): n 0-s and a 1. A COUNTER is a semi-clocked node, starting at its left input and incrementing with its middle input (1 here). BUFFER, READ_WORD and WRITE_WORD speak for themselves.

Some low-level optimizations take place in this stage. A **Bitwidth Narrowing** phase is performed just before AHA graphs are generated. **Dead code elimination** and **graph simplification** sweeps are applied on the AHA graph. A **Pipelining** phase uses node propagation delay estimates to compute the delay for each AHA section, and adds layers of pipeline registers in sections that have large propagation delays. The maximum number of pipeline stages added to the AHA graph is user controlled. Reducing propagation delays is important because it increases clock frequency.

AHA graph simulation allows the user (or, more likely, the compiler or system developer) to verify program behavior. The AHA simulator strictly mimics the hardware behavior with respect to clock cycles and signals traveling over wires. This removes the need for time consuming VHDL simulation and hardware level debugging. AHA-to-VHDL translation is straightforward; AHA nodes translate to VHDL components, which are connected according to the AHA edges.

5. Applications

The current test platform in Cameron is the WildStar Board, produced by Annapolis Micro Systems (Annapolis Micro Systems, 1999). The WildStar has three XCV2000E Virtex FPGAs made by Xilinx (Xilinx Incorporated, 1999). The WildStar board is capable of operating at frequencies from 25 MHz to 180 MHz. It communicates over the PCI bus with the host computer at 33 MHz. In our system, the board is housed in a 266-MHz Pentium-based PC. This section compares the performance of SA-C codes running one WildStar FPGA chip to the performance of C or assembly code running on an 800 MHz Pentium III. A 512x512 8 bit image is used for input.

5.1. INTEL IMAGE PROCESSING LIBRARY

When comparing simple IP operators one might write corresponding SA-C and C codes and compare them on the WildStar and Pentium. However, neither the Microsoft nor the Gnu C++ compilers fully exploit the Pentium's MMX technology. Instead, we compare SA-C codes to corresponding operators from the Intel Image Processing Library (IPL). The Intel IPL library consists of a large number of low-level Image Processing operations. Many of these are simple point- (pixel-) wise operations such as square, add, etc. These operations have been coded by Intel for highly efficient MMX execution.

For example, in case of a pointwise add, naively implemented SA-C runs four times slower on the FPGA than MMX assembly code on the Pentium. However, if care is taken by the SA-C programmer to strip-mine and interleave both input and output memories to take advantage of 64 bit memory access, the FPGA is twice as fast as the Pentium. Overall, the difference in run times for simple operators is not significant. This result is not surprising. The compute/communicate ratio in these codes is very low, and thus the memory bandwidth governs the performance.

However, the Prewitt edge detector is more complex. It requires convolving the image with two 3x3 masks, squaring the results, summing the squares, and finally computing the square root of the sum (see figures 2 and 3). Prewitt written in C using a single IPL routine (`iplConvolve`) runs on the Pentium, for our 512x512 test image, in 158 milliseconds. In comparison the equivalent SA-C code on the WildStar runs in less than 2 milliseconds, a speedup of 80 over the Pentium.

5.2. ARAGTAP

The ARAGTAP pre-screener (Raney et al., 1993) was developed by the U.S. Air Force at Wright Labs as the initial focus-of-attention mechanism for a SAR automatic target recognition application. Aragtap's components include down sampling, dilation, erosion, positive differencing, majority thresholding, bitwise-and, percentile thresholding, labeling, label pruning, and image creation. All these components, apart from label pruning and image creation, have been written in SA-C. Most of the computation time is spent in a sequence of eight gray-scale morphological dilations, and a later sequence of eight gray-scale erosions. Four of these dilations are with a 3x3 mask of 1's in the shape of a square, the other four are with a 3x3 mask with 1's in the shape of a cross and with zeros at the edges.

The dilate and erode loops allow temporal CSE and window narrowing. A hand optimized C implementation of a dilate sequence running on the Pentium takes 66 milliseconds. The SA-C compiler fuses the whole dilate sequence into one loop, which takes 3 milliseconds to execute on the WildStar, delivering a speedup of 22 compared to the Pentium.

5.3. CANNY

In section 5.1 we discussed the Prewitt operator. A more sophisticated edge detector is the Canny operator, which comprises a four step process. The four steps are 1) image smoothing, 2) computing edge magnitudes and (discretized) orientations, 3) non-maximal suppression in the gradient direction, and 4) hysteresis labeling, which is a connected components algorithm. For a clear and simple explanation of Canny and the reasoning behind it, see (Trucco and Verri, 1998), Chapter 4. The first three steps of Canny were implemented in SA-C and run on the reconfigurable hardware. Although connected components can be written in SA-C, it can currently not be compiled to FPGAs. Therefore, we assume that the last step will be performed on the host. The compiler performed eight fold vertical stripmining, among other optimizations. SA-C execution time on the WildStar is 6 milliseconds.

Comparing the performance to a Pentium is hard. A version of the same program was written in C, using Intel's IPL whenever possible. The resulting program took 850 milliseconds on the Pentium. However, Intel's OpenCV library has a hand-optimized assembly-coded version of Canny that includes the fourth (connected components) step. By setting the high and low thresholds to be the same (so that connected components will not iterate), the OpenCV routine takes 135 millisec-

onds. So SA-C was 22 times faster than assembly code and 140 times faster than a C plus IPL implementation.

5.4. WAVELET

Wavelets are commonly used for multi-scale analysis in computer vision, as well as for image compression. Honeywell has defined a set of benchmarks for reconfigurable computing systems, including a wavelet-based image compression algorithm (Kumar, 2000). This code takes one image and returns four quarter sized images, three of which are derivatives of the original. The SA-C code takes 2 milliseconds execution time on the FPGA, whereas on the Pentium the Honeywell C code took 75 milliseconds. SA-C was 37 times faster.

Concluding, apart from very simple point wise operations, SA-C on the WildStar using one Virtex 2000E chip runs between 20 and 75 times faster than the fastest code we could run on the 800 MHz Pentium III.

6. Related work

Hardware and software research in reconfigurable computing is active and ongoing. Hardware projects fall into two categories – those using commercial off-the-shelf components (e.g. FPGAs), and those using custom designs.

The Splash-2 (Buell et al., 1996) is an early (circa 1991) implementation of an RCS, built from 17 Xilinx (Xilinx Incorporated, 1998) 4010 FPGAs, and connected to a Sun host as a co-processor. Several different types of applications have been implemented on the Splash-2, including searching (Hoang, 1993; Pryor et al., 1993), pattern matching (Ratha et al., 1996), convolution (Ratha et al., 1995) and image processing (Athanas and Abbott, 1994).

Representing the current state of the art in FPGA-based RCS systems are the AMS WildStar (Annapolis Micro Systems, 1999) and the SLAAC project (Schott et al., 1997). Both utilize Xilinx Virtex (Xilinx Incorporated, 1999) FPGAs, which offer over an order of magnitude more programmable logic, and provide a several-fold improvement in clock speed, compared to the earlier chips.

Several projects are developing custom hardware. The Morphosys project (Lu et al., 1999) combines an on-chip RISC processor with an array of reconfigurable cells (RCs). Each RC contains an ALU, shifter, and a small register file.

The RAW Project (Waingold et al., 1997) also uses an array of computing cells, called *tiles*; each tile is itself a complete processor,

coupled with an intelligent network controller and a section of FPGA-like configurable logic that is part of the processor data path. The PipeRench (Goldstein et al., 1999) architecture consists of a series of *stripes*, each of which is a pipeline stage with an input interconnection network, a lookup-table based PE, a results register, and an output network. The system allows a virtual pipeline of any size to be mapped onto the finite physical pipeline.

On the software front, a framework called “Nimble” compiles C codes to reconfigurable targets where the reconfigurable logic is closely coupled to an embedded CPU (Li et al., 1999). Several other hardware projects also involve software development. The RAW project includes a significant compiler effort (Agarwal et al., 1997) whose goal is to create a C compiler to target the architecture. For PipeRench, a low-level language called DIL (Goldstein and Budiu, 1999) has been developed for expressing an application as a series of pipeline stages mapped to stripes.

Several projects (including Cameron) focus on hardware-independent software for reconfigurable computing; the goal – still quite distant – is to make development of RCS applications as easy as for conventional processors, using commonly known languages or application environments. Several projects use C as a starting point. DEFACTO (Hall et al., 1999) uses SUIF as a front end to compile C to FPGA-based hardware. Handel-C (Oxford Hardware Compiler Group, 1997) both extends the C language to express important hardware functionality, such as bit-widths, explicit timing parameters, and parallelism, and limits the language to exclude C features that do not lend themselves to hardware translation, such as random pointers. Streams-C (Gokhale et al., 2000) does a similar thing, with particular emphasis on extensions to facilitate the expression of communication between parallel processes. SystemC (SystemC, 2000) and Ocapi (IMEC, 2000) provide C++ class libraries to add the functionality required of RCS programming to an existing language.

Finally, a couple of projects use higher-level application environments as input. The MATCH project (Banerjee et al., 2000; Banerjee et al., 1999; Periyayacheri et al., 1999) uses MATLAB as its input language – applications that have already been written for MATLAB can be compiled and committed to hardware, eliminating the need for re-coding them in another language. Similarly, CHAMPION (Natarajan et al., 1999) is using Khoros (Konstantinides and Rasure, 1994) for its input – common glyphs have been written in VHDL, so GUI-based applications can be created in Khoros and mapped to hardware.

7. Conclusions and Future Work

The Cameron Project has created a language, called SA-C, for one-step compilation of image processing applications that target FPGAs. Various optimizations, both conventional and novel, have been implemented in the SA-C compiler.

The system has been used to implement routines from the Intel IPL, as well as more comprehensive applications, such as the ARAG-TAP target acquisition prescreener. Compared to Pentium III/MMX technology built at roughly the same time, the SA-C system running on an Annapolis WildStar board with one Virtex 2000 FPGA has similar performance when it comes to small IPL type operations, but shows speedups up to 75 when it comes to more complex operators such as Prewitt, Canny, Wavelet, and Dilate and Erode sequences. Performance evaluation of the SA-C system has just begun. As performance issues become clearer, the system will be given greater ability to evaluate various metrics including code space, memory use and time performance, and to evaluate the tradeoffs between conventional functional code and lookup tables.

Currently, the VHDL generated from the AHA graphs ignores the structural information available in the AHA graph. We will soon be investigating the use of Relatively Placed Macros (RPM) as a method to make some of the structural information explicit to the synthesis tools. Providing constraints to specify the placement of nodes relative to each other may prove to decrease synthesis and place and route time.

Also, stream data structures are being added to the SA-C language, which will allow multiple cooperating processes to be mapped onto FPGAs. This allows expression of streaming video applications, and partitioning of a program over multiple chips.

References

- Agarwal, A., S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, and M. Srikrishna, D. and Taylor: 1997, 'The RAW Compiler Project'. In: *Proc. Second SUIF Compiler Workshop*.
- Annapolis Micro Systems: 1999, 'STARFIRE Reference Manual'. Annapolis Micro Systems, Inc., Annapolis, MD. www.annapmicro.com.
- Athanas, P. M. and A. L. Abbott: 1994, 'Processing Images in Real Time on a Custom Computing Platform'. In: R. W. Hartenstein and M. Z. Servit (eds.): *Field-Programmable Logic Architectures, Synthesis, and Applications*. Springer-Verlag, Berlin, pp. 156–167.
- Banerjee, P. et al.: 2000, 'A MATLAB Compiler For Distributed, Heterogeneous, Reconfigurable Computing Systems'. In: *The 8th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*.

- Banerjee, P., N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Chang, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, and M. Walkden: 1999, 'MATCH: A MATLAB Compiler for Configurable Computing Systems'. Technical Report CPDC-TR-9908-013, Center for Parallel and distributed Computing, Northwestern University.
- Benedetti, A. and P. Perona: 1998, 'Real-time 2-D Feature Detection on a Reconfigurable Computer'. In: *IEEE Conference on Computer Vision and Pattern Recognition*. Santa Barbara, CA.
- Benitez, D. and J. Cabrera: 1999, 'Reactive Computer Vision System with Reconfigurable Architecture'. In: *International Conference on Vision Systems*. Las Palmas de Gran Canaria, Spain.
- Buell, D., J. Arnold, and W. Kleinfelder: 1996, *Splash 2: FPGAs in a Custom Computing Machine*. IEEE CS Press.
- Eldredge, J. and B. Hutchings: 1994, 'RRANN: A Hardware Implementation of the Backpropagation Algorithm Using Reconfigurable FPGAs'. In: *IEEE International Conference on Neural Networks*. Orlando, FL.
- Gokhale, M. et al.: 2000, 'Stream Oriented FPGA Computing in Streams-C'. In: *The 8th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*.
- Goldstein, S. C. and M. Budiu: 1999, 'The DIL Language and Compiler Manual'. Carnegie Mellon University. www.ece.cmu.edu/research/piperench/dil.ps.
- Goldstein, S. C., H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer: 1999, 'PipeRench: A Coprocessor for Streaming Multimedia Acceleration'. In: *Proc. Intl. Symp. on Computer Architecture (ISCA '99)*. www.cs.cmu.edu/~mihaib/research/isca99.ps.gz.
- Hall, M., P. Diniz, K. Bondalapati, H. Ziegler, P. Duncan, R. Jain, and J. Granacki: 1999, 'DEFACTO: A Design Environment for Adaptive Computing Technology'. In: *Proc. 6th Reconfigurable Architectures Workshop (RAW'99)*. Springer-Verlag.
- Hammes, J. and W. Böhm: 1999, 'The SA-C Language - Version 1.0'. www.cs.colostate.edu/cameron.
- Hammes, J., R. Rinker, W. Böhm, and W. Najjar: 1999, 'Cameron: High Level Language Compilation for Reconfigurable Systems'. In: *PACT'99*.
- Hartenstein, R. et al.: 1995, 'A Reconfigurable Machine for Applications in Image and Video Compression'. In: *Conference on Compression Technologies and Standards for Image and Video Compression*. Amsterdam, Holland.
- Hoang, D.: 1993, 'Searching Genetic Databases on Splash 2'. In: *IEEE Workshop on FPGAs for Custom Computing Machines*. pp. 185–192, CS Press, Los Alamitos, CA.
- IMEC: 2000, 'Ocapi overview'.
- Konstantinides, K. and J. Rasure: 1994, 'The Khoros Software Development Environment For Image And Signal Processing'. In: *IEEE Transactions on Image Processing*, Vol. 3. pp. 243–252.
- Kumar, S.: 2000, 'A Benchmark Suite for Evaluating Configurable Computing Systems - Status, Reflections, and Future Directions'. In: *FPGA2000 Eighth International Symposium on FPGAs*. Feb. 10-12, Monterey, CA.
- Li, Y., T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood: 1999, 'Hardware-Software Co-Design of Embedded Reconfigurable Architectures'. In: *Proc. 37th Design Automation Conference*.
- Lu, G., H. Singh, M. Lee, N. Bagherzadeh, and F. Kurhadi: 1999, 'The Morphosis Parallel Reconfigurable System'. In: *Proc. of EuroPar 99*.

- Natarajan, S., B. Levine, C. Tan, D. Newport, and D. Bouldin: 1999, 'Automatic Mapping of Khoros-based Applications to Adaptive Computing Systems'. Technical report, University of Tennessee.
- Oxford Hardware Compiler Group: 1997, 'The Handel Language'. Technical report, Oxford University.
- Periyayacheri, S., A. Nayak, A. Jones, N. Shenoy, A. Choudhary, and P. Banerjee: 1999, 'Library Functions in Reconfigurable Hardware for Matrix and Signal Processing Operations in MATLAB'. In: *Proc. 11th IASTED Parallel and Distributed Computing and Systems Conf. (PDCS'99)*.
- Perry, D.: 1993, *VHDL*. McGraw-Hill.
- Prewitt, J. M. S.: 1970, 'Object Enhancement and Extraction'. In: B. S. Lipkin and A. Rosenfeld (eds.): *Picture Processing and Psychopictorics*. Academic Press, New York.
- Pryor, D. V., M. R. Thistle, and N. Shirazi: 1993, 'Text Searching on Splash 2'. In: *IEEE Workshop on FPGAs for Custom Computing Machines*. pp. 172–178, CS Press, Los Alamitos, CA.
- Raney, S., A. Nowicki, J. Record, and M. Justice: 1993, 'ARAGTAP ATR system overview'. In: *Theater Missile Defense 1993 National Fire Control Symposium*. Boulder, CO.
- Ratha, N. K., D. T. Jain, and D. T. Rover: 1995, 'Convolution on Splash 2'. In: *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*. pp. 204–213, CS Press, Los Alamitos, CA.
- Ratha, N. K., D. T. Jain, and D. T. Rover: 1996, 'Fingerprint Matching on Splash 2'. In: *Splash 2: FPGAs in a Custom Computing Machine*. IEEE CS Press, pp. 117–140.
- Schott, B., S. Crago, C. C., J. Czarnaski, M. French, I. Hom, T. Tho, and T. Valenti: 1997, 'Reconfigurable Architectures for Systems Level Applications of Adaptive Computing'. Available from <http://www.east.isi.edu/SLAAC/>.
- SystemC: 2000, 'SystemC Homepage'.
- Trucco, E. and A. Verri: 1998, *Introductory Techniques for 3-D Computer Vision*. Prentice Hall.
- Waingold, E., M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal: 1997, 'Baring it all to Software: RAW Machines'. *Computer*.
- Wolfe, M.: 1996, *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company.
- Woodfill, J. and B. v. Herzen: 1997, 'Real-time Stereo Vision on the PARTS Reconfigurable Computer'. In: *IEEE Symposium on Field-Programmable Custom Computing Machines*. Napa, CA.
- Xilinx Incorporated: 1998, 'The Programmable Logic Databook'. Xilinx, Inc., San Jose, CA. www.xilinx.com.
- Xilinx Incorporated: 1999, 'Virtex 2.5V Field programmable Gate Arrays: Preliminary Product Description'. Xilinx, Inc. www.xilinx.com.

Address for Offprints: Colorado State University, Computer Science Department, Fort Collins, Colorado, USA

CAMERON PROJECT: FINAL REPORT

Appendix J: Accelerated Image Processing on FPGAs

Accelerated Image Processing on FPGAs¹

**Bruce A. Draper, J. Ross Beveridge, A.P. Willem Böhm, Charles Ross,
Monica Chawathe**

Department of Computer Science
Colorado State University
Fort Collins, CO 80523, U.S.A.
draper,ross,bohm,rossc,chawathe@cs.colostate.edu

ABSTRACT

The Cameron project has developed a language and compiler for mapping image-based applications to field programmable gate arrays (FPGAs). This paper tests this technology on several applications and finds that FPGAs are between 8 and 800 times faster than comparable Pentiums for image based tasks.

1) Introduction

Although computers keep getting faster and faster, there are always new image processing (IP) applications that need more processing than is available. Examples of current high-demand applications include real-time video stream encoding and decoding, real-time biometric (face, retina, and/or fingerprint) recognition, and military aerial and satellite surveillance applications. To meet the demands of these and future applications, we need to develop new techniques for accelerating image-based applications on commercial hardware.

Currently, many image processing applications are implemented on general-purpose processors such as Pentiums. In some cases, applications are implemented on digital signal processors (DSPs), and in extreme cases (when economics permit) applications can be implemented in application-specific integrated circuits (ASICs). This paper presents another technology, field programmable gate arrays (FPGAs), and shows how compiler technology can be used to map image processing algorithms onto FPGAs, achieving 8 to 800 fold speed-ups over Pentiums.

2) Field Programmable Gate Arrays

Field-programmable gate arrays (FPGAs) are non-conventional processors built almost entirely out of lookup tables. In particular, FPGAs contain grids of logic blocks, connected by programmable wires, as shown in Figure 1. Each logic block has one or more lookup tables (LUTs) and several bits of memory. As a result, logic blocks can implement arbitrary logic functions (up to a few bits), or be combined together to form registers. FPGAs as a whole can be used to implement circuit diagrams, by mapping the gates and registers onto logic blocks.

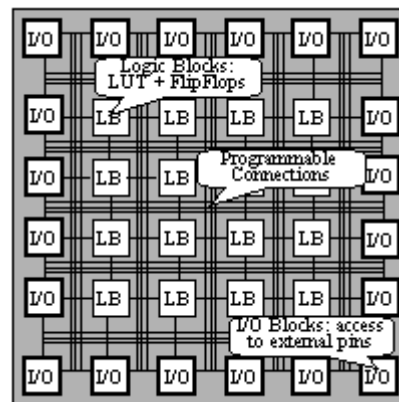


Figure 1: A conceptual illustration of an FPGA. Every logic block contains one or more LUTs, plus a bit or two of memory. The contents of the LUTs are (re)programmable, as are the grid connections. I/O blocks provide access to external pins, which usually connect to local memories.

¹ This work was funded by DARPA through AFRL under contract F33615-98-C-1319.

FPGAs were originally developed to serve as test vehicles for hardware circuit designs. Recently, however, FPGAs have become so dense and fast that they have evolved from simple test and “glue logic” circuits into the central processors of powerful reconfigurable computing systems [1]. A Xilinx XCV-2000E, for example, contains 38,400 logic blocks, and can operate at up to 180 MHz (depending on the latency of the embedded circuit). The logic blocks can be configured so as to exploit data, pipeline, process, I/O parallelism, or all of the above. In computer vision and image processing, FPGAs have already been used to accelerate real-time point tracking [2], stereo [3], color-based object detection [4], video and image compression [5], and neural networks [6].

The economics of FPGAs are fundamentally different from the economics of other parallel architectures. Because of the comparatively small size of the image processing market, most special-purpose image processors have been unable to keep pace with advances in general purpose processors. As a result, researchers who adopt them are often left with obsolete technology. FPGAs, on the other hand, enjoy a multi-billion dollar market as low-cost ASIC replacements. Consequently, increases in FPGA speeds and capacities have followed or exceeded Moore’s law for the last several years, and researchers can continue to expect them to keep pace with general-purpose processors [7].

Unfortunately, FPGAs are very difficult to program. Algorithms must be expressed as detailed circuit diagrams, including clock signals, etc., in hardware description languages such as Verilog or VHDL. This discourages most computer vision researchers from exploiting FPGAs; the intrepid few who do are repeatedly frustrated by the laborious process of modifying or combining FPGA circuits.

The goal of the Cameron project is to change how reconfigurable systems are programmed from a circuit design paradigm to an algorithmic one. To this end, we have developed a high-level language (called SA-C) for expressing image processing algorithms, and an optimizing compiler that targets FPGAs. Together, these tools allow programmers to quickly write algorithms in a high-level language, compile them, and run them on FPGAs.

Detailed descriptions of the SA-C language and optimizing compiler can be found elsewhere (see [8, 9], or <http://www.cs.colostate.edu/~cameron/> for a complete set of documents and publications). This paper only briefly introduces SA-C and its

compiler before presenting experiments comparing SA-C programs compiled to a Xilinx XCV-2000E FPGA to equivalent programs running on an Intel Pentium III processor. Our goal is to familiarize applications programmers with the state of the art in compiling high-level programs to FPGAs, and to show how FPGAs implement a wide range of image processing applications.

3) SA-C

SA-C is a single-assignment dialect of the C programming language designed to exploit both coarse-grained (loop-level) and fine-grained (instruction-level) parallelism. Roughly speaking, there are three major differences between SA-C and standard C: 1) SA-C adds variable bit-precision data types and fixed point data types. This exploits the ability of FPGAs to form arbitrary precision circuits, and compensates for the high cost of floating point operations on FPGAs by encouraging the use of fixed-point representations. 2) SA-C includes extensions to C that provide data parallel looping mechanisms and true multi-dimensional arrays. These extensions make it easier to express operations over sliding windows or slices of data (e.g. pixels, rows, columns, or sub-images), and also make it easier for the compiler to identify and optimize data access patterns. 3) SA-C restricts C by outlawing pointers and recursion, and restricting variables to be single assignment. This creates a programming model where variables correspond to wires instead of memory addresses, and functions are sections of a circuit, rather than entries on a program stack.

To illustrate the differences between SA-C and traditional C, consider how the Prewitt edge detector might be written in SA-C, as shown in Figure 2.

```
int16[:,:] main (uint8 image[:,:]) {
    int16 H[3,3] = {{-1,-1,-1}{0,0,0}{1,1,1}};
    int16 V[3,3] = {{-1,0,1}{-1,0,1}{-1,0,1}};
    int16 M[:,:] =
        for window W[3,3] in image {
            int16 dfdy, int16 dfdx =
                for w in W dot h in H dot v in V
                    return(sum(w*h),sum(w*v);
            int16 magnitude =
                sqrt(dfdy*dfdy+dfdx*dfdx);
        } return(array(magnitude));
    } return(M);
```

Figure 2: SA-C source code for the Prewitt edge detector. The Prewitt edge detector convolves the image with two masks (H & V above), and then

computes the square root of the sum of the squares.

At first glance, one is struck by the data types and the looping mechanisms. “int16” simply represents a 16-bit integer, while “uint8” represents an unsigned 8-bit integer. Unlike in traditional C, integers and fixed point numbers are not limited to 8, 16, 32 or 64 bits; they may have any precision (e.g. int11), since the compiler can construct circuits with any precision². Also, arrays are true multi-dimensional objects whose size may or may not be known at compile time. For example, the input argument “uint8 image[:,:]” denotes a 2D array of unknown size.

Perhaps the most significant differences are in the looping constructs. “for window W[3,3] in image” creates a loop that executes once for every possible 3x3 window in the image array. Such windows can be any size, although their size must be known at compile time. In addition to stepping through images, SA-C’s looping constructs also allow new arrays to be constructed in their return statements. In this case, “return (array (magnitude))” makes a new array out of the edge magnitudes calculated at each 3x3 window.

Perhaps the least C-like element of this program is the interior loop “for w in W dot h in H dot v in V”. This creates a single loop that executes once for every pixel in the 3x3 window W. Since H and V are also 3x3 arrays, each loop iteration matches one pixel in W with the corresponding elements of H and V. This “dot product” looping mechanism is particularly handy for convolutions, but requires that the structures being combined have the same shape and size. A more thorough description of SA-C can be found in [8] or at the Cameron web site.

4) THE SA-C COMPILER

The SA-C compiler translates high-level SA-C code into dataflow graphs, which can be viewed as abstract hardware circuit diagrams without timing information [10]. The nodes in a data flow graph are generally simple arithmetic operations whose inputs arrive over edges. There are also control nodes (e.g. selective merge) and array access/storage nodes.

Dataflow graphs are a common internal representation for optimizing compilers. The SA-C

compiler performs standard optimizations including common subexpression elimination, constant folding, operator strength reduction, invariant code motion, function in-lining, and dead code elimination. It also performs specialized optimizations for hardware circuits that reduce I/O bandwidth (e.g. loop fusion, partial loop unrolling), reduce circuit size (e.g. bitwidth narrowing, window narrowing, and temporal common subexpression elimination,) and increase the clock rate (pipelining, lookup tables). These optimizations are described in [11].

After optimization, the SA-C compiler translates data flow graphs into VHDL; commercial tools³ synthesize and place and route the VHDL to create FPGA configurations. The SA-C compiler also generates host code to download the FPGA configuration, data, and parameters, to trigger the FPGA, and to upload the results.

5) IMAGE PROCESSING ON FPGAs

The SA-C language and compiler allow FPGAs to be programmed in the same way as other processors. Programs are written in a high-level language, and can be compiled, debugged, and executed from a local workstation. It so happens that for SA-C programs, the host executable off-loads the processing of loops onto an FPGA, but this is invisible. SA-C therefore makes reconfigurable processors accessible to applications programmers with no hardware expertise.

The empirical question in this paper is whether image processing tasks run faster on FPGAs than on conventional general-purpose hardware, in particular Pentiums. The tests presented below in Table 1 suggest that in general, the answer is yes. Simple image operators are faster on reconfigurable processors because of their greater capabilities for I/O between the FPGA and local memory, although this speed-up is modest (a factor of ten or less). More complex tasks result in larger speed-ups, up to a factor of 800 in one test, by exploiting the parallelism within FPGAs and the strengths of an optimizing compiler.

The reconfigurable processor used in our tests is an Annapolis Microsystems WildStar with 3 Xilinx XV-2000E FPGAs and 12 local memories. Our conventional processor is a Pentium III running at 800 MHz. The chips in both processors are of a

² Earlier versions of SA-C limited variables to no more than 32 bits, but this limitation has been removed.

³ Synplicity and the Xilinx Foundation Tools.

similar age and were the first of their respective classes fabricated at 0.18 microns.

Routine	Pentium III	XV-2000E	Ratio
AddS	0.00595	0.00067	8.88
Prewitt	0.15808	0.00196	83.16
Canny	0.13500	0.00606	22.5
Wavelet	0.07708	0.00208	38.5
Dilates (8)	0.06740	0.00311	21.6
Probing	65.0	0.08	812.5

Table 1. Execution times in seconds for routines with 512x512 8-bit input images. The comparison is between a 800Mhz Pentium III and an AMS WildStar with Xilinx XV-2000E FPGAs.

5.1) The Simplest Image Operator: Scalar Addition

The simplest program we tested adds a scalar argument to every pixel in an image. For the WildStar, we wrote the function in SA-C and compiled it to a single FPGA. We compared its performance to the matching routine from the Intel Image Processing Library (IPL) running on the Pentium. In so doing, we compare the performance of compiled SA-C code on an FPGA to hand-optimized (by Intel) Pentium assembly code. As shown in Table 1, the WildStar outperforms the Pentium by a factor of 8.

Why is the WildStar faster? The clock rate of an FPGA is a function of the latency of the programmed circuit, but in general FPGAs run at lower clock rates than Pentiums. For this program, the WildStar ran at 51.7 MHz, compared to 800 MHz for the Pentium. Unfortunately for the Pentium, however, memory response times have not kept up with processor speeds, and the 512x512 source image is too large to fit in its primary cache. As a result, the Pentium is not able to read or write data at anything close to 800MHz, so its primary advantage is squandered.

FPGAs, on the other hand, are capable of parallel I/O. The WildStar gives the FPGAs 32-bit I/O channels to each of four local memories, so the FPGA can both read and write 8 8-bit pixels per cycle. This is four times the I/O bandwidth of the Pentium. Also, the operator's pixel-wise access pattern is easily identified by the SA-C compiler, which is able to optimize the I/O so that the program reads and writes almost 8 pixels per cycle.

The FPGA outperforms the Pentium on the scalar addition task by slightly more than I/O considerations alone would predict. This is because the SA-C compiler exploits both data and pipeline parallelism. On every cycle, the FPGA (1) reads eight 8-bit pixels, (2) adds eight copies of the scalar to the eight pixels read on the previous cycle (in parallel), and (3) writes back to memory the sums of the scalar with the eight pixels read on the cycle before that.

This program represents one extreme in the FPGA vs. Pentium comparison. It is a simple, pixel-based operation that fails to fully exploit the FPGA, since only 9% of the lookup tables (and 9% of flip-flops) were used. However, it demonstrates that FPGAs will outperform Pentiums on simple image operators because of their parallel I/O capabilities. The exact amount of the speed-up will depend on the number of local memories the FPGA has access to and the speeds of the memories, but in general one expects a small speed-up of less than a factor of ten.

5.2) Edge Operators

The Prewitt edge operator shown in Figure 2 is more complex than simple scalar addition. Every 3x3 window in the image is convolved with horizontal and vertical edge masks; and the edge magnitude at a pixel is the square root of the sum of the square of the convolution responses. When the Prewitt program written in SA-C is compiled for the WildStar, it computes the edge magnitude response image for a 512x512 input image in 1.96 milliseconds. In comparison, the equivalent Intel IPL function⁴ on the Pentium takes 158.08 milliseconds, or approximately 80 times longer (see Table 1).

Why is the Prewitt edge detector so much faster on an FPGA? One of the reasons, as before, is I/O: the FPGA can read the input image and write the output image faster than the Pentium can. The advantage is magnified by the fact that edge detection is a window-based operator. A naive implementation of a 3x3 window will slide the window horizontally across the image until it reaches the end of the row, at which point it will drop down one row and repeat the process. While shift registers can be used to avoid reading a pixel more than once on any given horizontal sweep, every pixel is still read three times,

⁴ `iplConvolve2D` with two masks (the Prewitt horizontal and vertical edge masks) and `IPL_SUMSQROOT` as the combination operator.

once for each row in the window. The SA-C compiler avoids this by partially unrolling the loop and computing eight vertical windows in parallel (see Figure 3). This reduces the number of input operations needed to process the image by almost a factor of three by exploiting the parallelism of the FPGA.

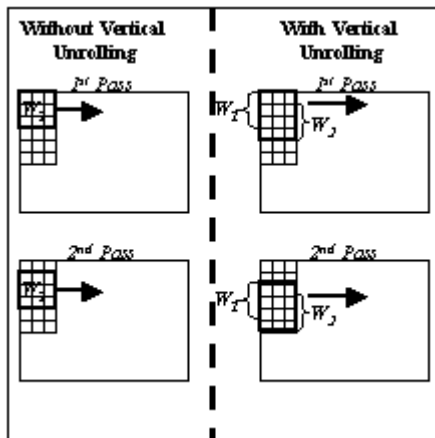


Figure 3: How partial vertical unrolling optimizes I/O. As a 3x3 image window slides across an image, each pixel is read 3 times (assuming shift registers hold values as it slides horizontally). Under partial unrolling, two or more vertical windows are computed in parallel, allowing the passes to skip rows and reducing I/O.

Of course, the parallelism of the FPGAs does more than just reduce the number of I/O cycles. We mentioned above that the FPGA computes eight image windows in parallel. It also exploits parallelism within the windows. Convolutions, in general, are ideal for data parallelism. The multiplications can be done in parallel, while the additions are implemented as trees of parallel adders. Pipeline parallelism is equally important, since square root is a complex operation that leads to a long circuit (on an FPGA) or a complex series of instructions (on a Pentium). The SA-C compiler lays out the circuit for the complete edge operator including the square root, and then inserts registers to divide it into approximately twenty stages, each of which operate in parallel.

Finally, the SA-C compiler has the advantage of being a compiler, not a library of subroutines. Thus, while the IPL convolution routine must be general enough to perform arbitrary convolutions, the SA-C implementation of Prewitt can take advantage of compile-time constants. In particular, the Prewitt edge masks are composed entirely of ones, zeroes and minus ones, so all of the multiplications in these

particular convolutions can be optimized away or replaced by negation. Furthermore, the eight windows being processed in parallel contain redundant additions, the extra copies of which are removed by common subexpression elimination.

The Canny edge detector is similar to Prewitt, only more complex. It smooths the image and convolves it with horizontal and vertical edge masks. It then computes edge orientations as well as edge magnitudes, performs non-maximal suppression in the direction of the gradient, and applies high and low thresholds to the result. (A final connected components step was not implemented; see [12] pp. 76-80.)

We implemented the Canny operator in SA-C and executed it on the WildStar. The result was compared to two versions of Canny on the Pentium. The first version was written in VisualC++, using IPL routines for the convolutions. This allowed us to compare compiled SA-C on the WildStar to compiled C on the Pentium; the WildStar was 120 times faster. We then tested the hand-optimized assembly code version of Canny in Intel's OpenCV, setting the high and low thresholds equal to prevent the connected components routine from iterating. The Pentium's performance improved five fold, but the FPGA still outperformed the Pentium by a factor of 22. Table 1 shows the comparison with OpenCV.

Why is performance relatively better for Prewitt than for Canny? There are two reasons. First, the Canny operator uses fixed convolution masks, so the OpenCV Canny routine has the same opportunities for constant propagation and common subexpression elimination that SA-C has. Second, the Canny operator does not include a square root operation. Square roots can be pipelined on an FPGA but require multiple cycles on a Pentium.

5.3) Wavelet

In a test on the Cohen-Daubechies-Feauveau Wavelet [13], the WildStar beat the Pentium by a factor of 35. Here a SA-C implementation of the wavelet was compared to a C implementation provided by Honeywell as part of a reconfigurable computing benchmark [14].

5.4) The ARAGTAP Pre-screener

We also compared FPGAs and Pentiums on two military applications. The first is the ARAGTAP

pre-screener [15], a morphology-based focus of attention mechanism for finding targets in SAR images. The pre-screener uses six morphological subroutines (downsample, erode, dilate, bitwise and, positive difference, and majority threshold), all of which were written in SA-C. Most of the computation in the pre-screener, however, is in a sequence of 8 erosion operators with alternating masks, and a later sequence of 8 dilations. We therefore compared these sequences on FPGAs and Pentiums. (Just the dilate is shown in Table 1; results are similar for erosion).

The SA-C compiler fused all 8 dilations into a single pass over the image; it also applied temporal common subexpression elimination, pipelining, and other optimizations. The result was a 20 fold speed-up over the Pentium running automatically compiled (but heavily optimized) C. We had expected a greater speed-up, but were foiled by the simplicity of the dilation operator: after optimization, it reduces to a collection of max operators, and there is not enough computation per pixel to fully exploit the parallelism in the FPGA.

5.5) Probing

The second application is an ATR probing algorithm [16]. The goal of probing is to find a target in a LADAR or IR image. A target (as seen from a particular viewpoint) is represented by a set of probes, where a probe is a pair of pixels that straddle the silhouette of the target. The idea is that the difference in values between the pixels in a probe should exceed a threshold if there is a boundary between them. The match between a template and an image location is measured by the percentage of probes that straddle image boundaries.

Probe sets must be evaluated at every window position in the image. What makes this application complex is the number of probes. In our example there are approximately 35 probes per view, 81 views per target, and three targets. In total, the application defines 7,573 probes per window position. Fortunately, many of these probes are redundant in either space or time, and the SA-C optimizing compiler is able to reduce the problem to computing 400 unique probes (although the summation trees remain complex). This is still too large to fit on one FPGA, so to avoid dynamic reconfiguration we distribute the task across all three FPGAs on the WildStar. When compiled using VisualC++ for the Pentium, probing takes 65

seconds; the SA-C implementation on the WildStar run in 0.08 seconds.

These times can be explained as follows. For the configuration generated by the SA-C compiler for the probing algorithm, the FPGAs run at 41.1 MHz. The program is completely memory IO bound: every clock cycle each FPGA reads one 32 bit word, containing two 12 bit pixels. As there are $(512-13+1) \times (1024) \times 13$ pixel columns to be read, the FPGAs perform $(512-13+1) \times (1024) \times (13/2) = 3,328,000$ reads. At 41.1 MHz this takes 80.8 milliseconds.

The Pentium performs $(512-13+1) \times (1024-34+1)$ window accesses. Each of these window accesses involves 7573 threshold operations. Hence the inner loop that performs one threshold operation is executed $(512-13+1) \times (1024-34+1) \times 7573 = 3,752,421,500$ times. The inner loop body in C is:

```
for(j=0; j<sizes[i]; j++){
    diff = ptr[set[i][j][2]*in_width+set[i][j][3]] -
           ptr[set[i][j][0]*in_width+set[i][j][1]];
    count += (diff>THRESH || diff<-THRESH);
}
```

where in_width and THRESH are constants. The VC++ compiler infers that ALL the accesses to the set array can be done by pointer increments, and generates an inner loop body of 16 instructions. (This is, by the way, much better than the 22 instructions that the gcc compiler produces at optimization setting -O6.) The total number of instructions executed in the inner loop is therefore $3,752,421,500 \times 16 = 60,038,744,000$. If one instruction were executed per cycle, this would bring the execution time to about 75 seconds. As the execution time of the whole program is 65 seconds, the Pentium (a super scalar architecture) is actually executing more than one instruction per cycle!

6) Practical Timing Considerations

So far, we have reported only run-times. This is misleading, if the reconfigurable processor is being used as a co-processor. To run an operator on a co-processor, the image has to be downloaded to the reconfigurable systems' memory, and the results must be returned to the host processor. A typical upload or download time on a PCI bus for a 512x512 8-bit image is about 0.022 seconds. As a result, the FPGA is slower than a Pentium at adding a scalar to an image, if data communication times are taken into account. The other programs listed in Table 1 are still faster on the FPGA, although their speed-up factors are reduced.

In addition, current FPGAs cannot be reconfigured quickly. It takes about 0.14 seconds to reconfigure an XV-2000E over a PCI bus. Any function compiled to an FPGA configuration must save enough time to justify the reconfiguration cost. The simplest way to do this in practice is to select one operator sequence to accelerate per FPGA, and to pre-load the FPGAs with the appropriate configurations, thus eliminating the need for dynamic reconfiguration. However, non real time applications may find it useful to reconfigure and FPGA, so long as the benefit of each configuration is great enough.

7) Related Work

Researchers have tried to accelerate image processing on parallel computers for as long as there have been parallel computers. Some of this early work tried to map IP onto commercially available parallel processors (e.g. [17]), while other research focused on building special-purpose machines (e.g. [18]). Unfortunately, in both cases the market did not support the architecture designs, which were eclipsed by general-purpose processors and Moore's law. More recent work has focused on so-called "vision chips" that build the sensor into the processor [19]. Another approach (advocated here) is to work at the board level and integrate existing chips – either DSPs or FPGAs -- into parallel processors that are appropriate for image processing. Focusing on FPGAs, Splash-2 [20] was the first reconfigurable processor based on commercially available FPGAs (Xilinx 4010s) and applied to image processing. The current state of the art in commercially available reconfigurable processors is represented by the AMS WildStar⁵, the Nallatech Benblue⁶ and the SLAAC project⁷, all of which use Xilinx FPGAs. (The experimental results in this paper were computed on an AMS WildStar.) Research projects into new designs for reconfigurable computers include PipeRench [21], RAW [22] and Morphosis [23].

To exploit new hardware, researchers have to develop software libraries and/or programming languages. One of the most important software libraries is the Vector, Signal, and Image Processing Library (VS IPL)⁸, proposed by a

consortium of companies, universities and government laboratories as a single library to be supported by all manufacturers of image processing hardware. The Intel Image Processing Library (IPL)⁹ and OpenCV¹⁰ are similar libraries that map image processing and computer vision operators onto Pentiums. It is also possible to build graphical user interfaces (GUIs) to make using libraries easier. AMS provides such a GUI to a library of primitive operators for programming the WildStar; CHAMPION [24] uses the Khoros [25] GUI, having implemented all the primitive Khoros routines in VHDL. SA-C has also been integrated with Khoros, which can be used both to call pre-written SA-C routines or to write new ones.

One of the first programming languages designed to map image processing onto parallel hardware was Adapt [26]; C\ [27] and C_T++ [28] are more recent languages designed for the same purpose. Other languages are less specialized and try to map arbitrary computations onto fine-grained parallel hardware; several of these projects focus on reconfigurable computing. Handel-C [29] is similar to SA-C in that it both extends C to express important hardware functionality, such as bit-widths, explicit timing parameters, and parallelism, and limits the language to exclude C features that do not lend themselves to hardware translation. It is lower-level than SA-C, however, in that programmers need to consider clock signals and other timing considerations explicitly. Streams-C [30] emphasizes streams to facilitate the expression of parallel processes. Finally, the MATCH project [31] uses MATLAB as its input language, while targeting reconfigurable processors.

8) Future Work

For many years, real-time applications on traditional processors had to be written in assembly code, because the code generated by compilers was not as efficient. We believe there is an analogous progression happening with VHDL and the SA-C compiler. At the moment, applications written directly in VHDL are more efficient (albeit more difficult to develop), but we expect future improvements to the compiler to narrow this gap.

In particular, the FPGA configurations generated by the SA-C compiler currently use only one clock signal. This limits the I/O ports to operate at the

⁵ www.annapmicro.com

⁶ www.nallatech.com

⁷ www.east.isi.edu/SLAAC/

⁸ www.vsipl.org

⁹ www.intel.com/software/products/perflib/ipl/

¹⁰ www.intel.com/software/products/opensource/libraries/cvfl.htm

same speed as the computational circuit. Xilinx FPGAs, however, support multiple clocks running at different speeds, and include internal RAM blocks that can serve as data buffers. Future versions of the compiler will use two clocks, one for internal computation and one for I/O. This should double (or more) the speed of I/O bound applications.

We also plan to introduce streams into the SA-C language and compiler. This will support new FPGA boards with channels for direct sensor input, and will also make it easier to implement applications where the run-times of subroutines are strongly data dependent, for example connected components. We also plan to introduce new compiler optimizations to support trees and other complex data structures in memory, and to improve pipelining in the presence of nextified (a.k.a. loop carried) variables.

The goal of these extensions is to support stand-alone applications on FPGAs. Imagine, for example, reconfigurable processor boards with one or more FPGAs, local memories, A/D converters (or digital camera ports), and internet access. Such processors could be incorporated inside a camera, and would consume very little power. A security application running on the FPGAs could then inspect images as they came from the camera, and notify users via the internet whenever something irregular occurred. The application could be as simple as motion detection or as complex as human face recognition. A single host processor could then monitor a large number of cameras/FPGAs from any location.

8) CONCLUSION

FPGAs are a class of processor with a two billion dollar per year market. As a result, they obey Moore's law, getting faster and denser at the same rate as other processors. The thesis of this paper is that most image processing applications run faster on FPGAs than on general-purpose processors, and that this will continue to be true as both types of processors become faster.

In particular, complex image processing applications do enough processing per pixel to be compute bound, rather than I/O bound. In such cases, FPGAs dramatically outperform Pentiums by factors of up to 800. Simpler image processing operators tend to be I/O bound. In these cases, FPGAs still outperform Pentiums because of their

greater I/O capabilities, but by smaller margins (factors of 10 or less).

References

- [1] A. DeHon, "The Density Advantage of Reconfigurable Computing," *IEEE Computer*, vol. 33, pp. 41-49, 2000.
- [2] A. Benedetti and P. Perona, "Real-time 2-D Feature Detection on a Reconfigurable Computer," presented at IEEE Conference on Computer Vision and Pattern Recognition, Santa Barbara, CA, 1998.
- [3] J. Woodfill and B. v. Herzen, "Real-Time Stereo Vision on the PARTS Reconfigurable Computer," presented at IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, 1997.
- [4] D. Benitez and J. Cabrera, "Reactive Computer Vision System with Reconfigurable Architecture," presented at International Conference on Vision Systems, Las Palmas de Gran Canaria, 1999.
- [5] R. W. Hartenstein, J. Becker, R. Kress, H. Reinig, and K. Schmidt, "A Reconfigurable Machine for Applications in Image and Video Compression," presented at Conference on Compression Technologies and Standards for Image and Video Compression, Amsterdam, 1995.
- [6] J. G. Eldredge and B. L. Hutchings, "RRANN: A Hardware Implementation of the Backpropagation Algorithm Using Reconfigurable FPGAs," presented at IEEE International Conference on Neural Networks, Orlando, FL, 1994.
- [7] N. Tredennick, "Moore's Law Shows No Mercy," in *Dynamic Silicon*, vol. 1: Gilder Publishing, LLC, 2001, pp. 1-8.
- [8] J. P. Hammes, B. A. Draper, and A. P. W. Böhm, "Sassy: A Language and Optimizing Compiler for Image Processing on Reconfigurable Computing Systems," presented at International Conference on Vision Systems, Las Palmas de Gran Canaria, Spain, 1999.
- [9] A. P. W. Böhm, J. Hammes, B. A. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar, "Mapping a Single Assignment Programming Language to Reconfigurable Systems," *Supercomputing*, vol. 21, pp. 117-130, 2002.
- [10] J. B. Dennis, "The evolution of 'static' dataflow architecture," in *Advanced Topics in Data-Flow Computing*, J. L. Gaudiot and L. Bic, Eds.: Prentice-Hall, 1991.

- [11] J. Hammes, W. Bohm, C. Ross, M. Chawathe, B. Draper, R. Rinker, and W. Najjar, "Loop Fusion and Temporal Common Subexpression Elimination in Window-based Loops," presented at 8th Reconfigurable Architectures Workshop, San Francisco, 2001.
- [12] E. Trucco and A. Verri, *Introductory Techniques for 3-D Computer Vision*. Saddle River, NJ: Prentice-Hall, 1998.
- [13] A. Cohen, I. Daubechies, and J. C. Feauveau, "Biorthogonal bases of compactly supported wavelets," *Communications of Pure and Applied Mathematics*, vol. 45, pp. 485-560, 1992.
- [14] S. Kumar, "A Benchmark Suite for Evaluating Configurable Computing Systems - Status, Reflections, and Future Directions," presented at International Symposium on FPGAs, Monterey, CA, 2000.
- [15] S. D. Raney, A. R. Nowicki, J. N. Record, and M. E. Justice, "ARAGTAP ATR system overview," presented at Theater Missile Defense 1993 National Fire Control Symposium, Boulder, CO, 1993.
- [16] J. E. Bevington, "Laser Radar ATR Algorithms: Phase III Final Report," Alliant Techsystems, Inc. May 1992.
- [17] P. J. Narayanan, L. T. Chen, and L. S. Davis, "Effective Use of SIMD Parallelism in Low- and Intermediate-Level Vision," *IEEE Computer*, vol. 25, pp. 68-73, 1992.
- [18] C. C. Weems, E. M. Riseman, and A. R. Hanson, "Image Understanding Architecture: Exploiting Potential Parallelism in Machine Vision," *IEEE Computer*, vol. 25, pp. 65-68, 1992.
- [19] A. Moini, *Vision Chips*, vol. 526. Boston: Kluwer, 1999.
- [20] D. Buell, J. Arnold, and W. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*: IEEE CS Press, 1996.
- [21] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "PipeRench: A Coprocessor for Streaming Multimedia Acceleration," presented at International Symposium on Computer Architecture, 1999.
- [22] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to Software: RAW Machines," *IEEE Computer*, vol. 30, pp. 86-93, 1997.
- [23] G. Lu, H. Singh, M. Lee, N. Bagherzadeh, and F. Kurhadi, "The Morphosis Parallel Reconfigurable System," presented at EuroPar, 1999.
- [24] S. Natarajan, B. Levine, C. Tan, D. Newport, and D. Bouldin, "Automatic Mapping of Khoros-based Applications to Adaptive Computing Systems," University of Tennessee 1999.
- [25] K. Konstantinides and J. Rasure, "The Khoros Software Development Environment for Image and Signal Processing," *IEEE Transactions on Image Processing*, vol. 3, pp. 243-252, 1994.
- [26] J. A. Webb, "Steps Toward Architecture-Independent Image Processing," *IEEE Computer*, vol. 25, pp. 21-31, 1992.
- [27] A. Fatni, D. Houzet, and J. Basille, "The C\\ Data Parallel Language on a Shared Memory Multiprocessor.," presented at Computer Architectures for Machine Perception, Cambridge, MA, 1997.
- [28] F. Bodin, H. Essafi, and M. Pic, "A Specific Compilation Scheme for Image Processing Architecture.," presented at Computer Architecture for Machine Perception, Cambridge, MA, 1997.
- [29] O. H. C. Group, "The Handel Language," Oxford University 1997.
- [30] M. Gokhale, "Stream Oriented FPGA Computing in Streams-C," presented at IEEE Symposium on Field-Programmable Custom Computing Machines, Nap, CA, 2000.
- [31] P. Banerjee, "A MATLAB Compiler for Distributed, Heterogeneous, Reconfigurable Computing Systems," presented at IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, 2000.

CAMERON PROJECT: FINAL REPORT

Appendix K: One-Step Compilation of Image Processing Applications to FPGAs

One-step Compilation of Image Processing Applications to FPGAs *

A.P.W. Böhm, B. Draper, W. Najjar, J. Hammes, R. Rinker, M. Chawathe, C. Ross
Computer Science Department
Colorado State University
Ft. Collins, CO, U.S.A.

Abstract *This paper describes a system for one-step compilation of image processing (IP) codes, written in the machine-independent, algorithmic, high-level single assignment language SA-C, to FPGA-based hardware. The SA-C compiler performs a variety of optimizations, some conventional and some specialized, before generating dataflow graphs and host code. The dataflow graphs are then compiled, via VHDL, to FPGA configuration codes. This paper introduces SA-C and describes the optimization and code generation stages in the compiler. The performance of a target acquisition prescanner (ARAGTAP), the Intel Image Processing Library, and an iterative tri-diagonal solver running on a reconfigurable system are compared to their performance on a Pentium PC with MMX.*

1 Introduction

Recently, the computer vision and image processing communities have become aware of the potential for massive parallelism and high computational density in FPGAs. FPGAs have been used for, e.g., real-time point tracking [9], stereo vision [40], color-based detection [10], image compression [22], and neural networks [14]. Unfortunately, the biggest obstacle to the more widespread use of reconfigurable computing systems lies in the difficulty of developing application programs. FPGAs are typically programmed using behavioral hardware description languages such as VHDL [31] and Verilog. These languages require great attention to detail such as timing issues and low level synchronization. However, application programmers are typically not trained in these hardware description languages and may be reluctant to learn them. They usually prefer a higher level, algorithmic programming language to express their applications.

The Cameron Project [21, 28] has created a high-level algorithmic language, named SA-C [20], for expressing image processing applications and compiling them to FPGAs. For a SA-C program the compiler provides one-step compilation to a ready-to-run host executable and FPGA configurations. The compiler uses dataflow graphs to perform a variety of program optimizations. While some of these transformations are conventional, others are novel and have been developed specifically for mapping to FPGAs.

After parsing and type checking, the SA-C compiler converts the program to a hierarchical graph representation called DDCF (Data Dependence and Control Flow) graphs. DDCF graphs are used in many optimizations, some general and some specific to SA-C and its target platform. After optimization, the program is converted to a combination of low-level dataflow graphs (DFGs) and host code. DFGs are then compiled to VHDL code, which is synthesized and place-and-routed to FPGAs by commercial software tools.

To aid in program development, the SA-C compiler has two other modes. In the first, the entire SA-C code is compiled to a host executable for traditional debugging. In the second, the DFGs are executed by a simulator for validation. Dataflow execution is animated, allowing the user to view the flow of data through the graph.

Figure 1 shows a high-level view of the system. The SA-C compiler can run in stand-alone mode, but it also has been integrated into the Khoros^(TM) [25] graphical software development environment.

The rest of this paper presents an overview of the SA-C language in section 2. Compiler optimizations and pragmas are discussed in section 3. Translations to low-level dataflow graphs and then to VHDL are discussed in sections 4 and 5. The Cameron Project's test platform is described in section 6, and some applications with performance data are presented in section 7. References to related work are given in section 8, and section 9 concludes and describes future work.

*This work is supported by DARPA under US Air Force Research Laboratory contract F33615-98-C-1319.

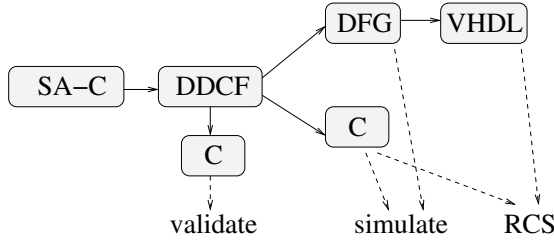


Figure 1. SA-C Compilation system.

2 The SA-C Language

The design goals of SA-C are to have a language that can express image processing applications elegantly, and to allow seamless compilation to reconfigurable hardware. Variables in SA-C are associated with wires, not with memory locations. SA-C is a single-assignment side effect free language; each variable's declaration occurs together with the expression defining its value. This avoids pointer and von Neumann memory model complications and allows for better compiler analysis and translation to DFGs. The IP applications that have been coded in SA-C transform images to images and are easily expressed using single assignment. Data types in SA-C include signed and unsigned integers and fixed point numbers, with user-specified bit widths. SA-C has multidimensional rectangular arrays whose extents can be determined either dynamically or statically. The type declaration `int14 M[:,6]` for example, is a declaration of a matrix *M* of 14-bit signed integers. The left dimension will be determined dynamically; the right dimension has been specified by the user.

The most important aspect of SA-C is its treatment of **for** loops and their close interaction with arrays. SA-C is expression oriented, so every construct including a loop returns one or more values. A loop has three parts: one or more generators, a loop body and one or more return values. The generators provide parallel array access operators that are concise and easy for the compiler to analyze. There are four kinds of loop generators: *scalar*, *array-element*, *array-slice* and *window*. The scalar generator produces a linear sequence of scalar values, similar to Fortran's **do** loop. The array-element generator extracts scalar values from a source array, one per iteration. The array-slice generator extracts lower dimensional sub-arrays (e.g. vectors out of a matrix). Finally, window generators allow rectangular sub-arrays to be extracted from a source array. All possible sub-arrays of the specified size are produced, one per iteration. Generators can be combined through **dot** and **cross** products. The **dot** product runs the gener-

ators in lock step, whereas **cross** products produce all combinations of components from the generators.

SA-C loops provide a simple and concise way of processing arrays in regular patterns, often making it unnecessary to create nested loops to handle multi-dimensional arrays or to refer explicitly to the array's extents or the loop's index variables. They make compiler analysis of array access patterns significantly easier than in C or Fortran, where the compiler must analyze index expressions in loop nests and infer array access patterns from these expressions. In SA-C, the index generators and the array references have been unified; the compiler can reliably infer the patterns of array access.

A loop can return arrays and reductions built from values that are produced in the loop iterations, including **sum**, **product**, **min**, **max**, **mean**, and **median**. The **histogram** operator returns a histogram of loop body values in a one-dimensional array.

```

int16[:,:] main (uint8 Image[:,:]) {
    int16 H[3,3] = {{ -1, -1, -1 },
                   {  0,  0,  0 },
                   {  1,  1,  1 }};

    int16 V[3,3] = {{ -1,  0,  1 },
                   { -1,  0,  1 },
                   { -1,  0,  1 }};

    int16 M[:,:] =
        for window W[3,3] in Image {
            int16 dfdy, int16 dfdx =
                for h in H dot w in W dot v in V
                    return (sum (h*w), sum (v*w));
            int16 magnitude =
                sqrt (dfdy*dfdy+dfdx*dfdx);
        } return (array (magnitude));
} return (M);
  
```

Figure 2. Prewitt Edge detector code.

Figure 2 shows SA-C code for the Prewitt edge detector, a standard IP operator. The outer **for** loop is driven by the extraction of 3x3 sub-arrays from array *Image*. All possible 3x3 arrays are taken, one per loop iteration. Its loop body applies two masks to the extracted window *W*, producing a magnitude. An array of these magnitudes is collected and returned as the program's result. The shape of the return array is derived from the shape of *Image* and the loop's generator. If *Image* were a 100x200 array, the result array *M* would have a shape of 98x198.

Loop carried values are allowed in SA-C using the keyword **next** instead of a type specifier in a loop body. This indicates that an initial value is available outside

the loop, and that each iteration can use the current value to compute a next value. The following code fragment shows an initial value of an array Y computed outside a loop driven by a scalar generator, array elements used to compute next elements inside the loop, and the final version of Y being returned by the loop. In general, the size of Y may change from iteration to iteration.

```
fix16.14 Y[:] = ...
fix16.14 res[:] =
  for uint8 i in [SIZE]
    next Y = for y in Y return(array(f(y)));
  }return(final(Y));
```

3 Optimizations and pragmas

The compiler's internal program representation is a hierarchical graph form called the "Data Dependence and Control Flow" (DDCF) graph. DDCF subgraphs correspond to source language constructs. Edges in the DDCF express data dependencies, opening up a wide range of loop- and array-related optimization opportunities.

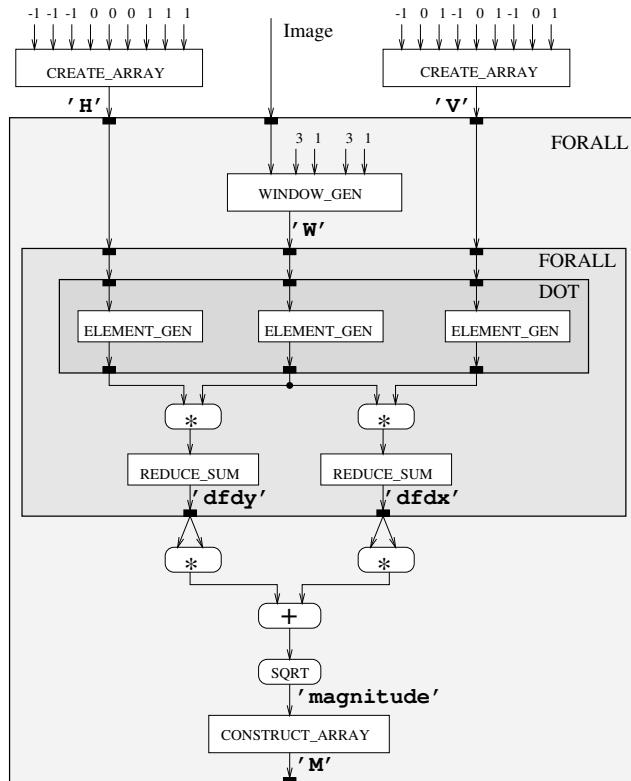


Figure 3. DDCF graph for Prewitt program.

Figure 3 shows the initial DDCF graph of the Prewitt program of figure 2. The FORALL and DOT nodes

are compound, containing subgraphs. Black rectangles along the top and bottom of a compound node represent input ports and output ports. The outer FORALL has a single window generator. The WINDOW_GEN is operating on a two-dimensional image, so it requires window size and step inputs for each of the two dimensions. In this example, both dimensions are size three, with unit step sizes. The output of the WINDOW_GEN node is a 3x3 array that is passed into the inner FORALL loop. This loop has a DOT graph that runs three generators in parallel, each producing a stream of nine values from its source array. Each REDUCE_SUM node sums a stream of values to a single value. Finally, the CONSTRUCT_ARRAY node at the bottom of the outer loop takes a stream of values and builds an array with them.

Dataflow analysis is performed on the DDCF graph, and a number of conventional optimizations [3] are performed as DDCF-to-DDCF transformations, including Invariant Code Motion, Function Inlining, Switch Constant Elimination, Constant Propagation and Folding, Algebraic Identities, Dead Code Elimination and Common Subexpression Elimination. Another set of optimizations is more specific to the single assignment nature of SA-C and to the target hardware. Many of these are controlled by user pragmas.

Many IP operators involve fixed size and often constant convolution masks. A *Size Inference* pass propagates information about constant size loops and arrays through the dependence graph. This pass is vitally important in this compiler, since it exploits the close association between arrays and loops in the SA-C language. Source array size information can propagate through a loop to its result arrays (downward flow), and result array size information can be used to infer the sizes of source arrays (upward flow). In addition, since the language requires that the generators in a dot product have identical shapes, size information from one generator can be used to infer sizes in the others (sideways flow).

Effective size inference allows other optimizations to take place. Examples are Full Loop Unrolling and array elimination. Full Unrolling of loops is discussed next. Array elimination can occur in two contexts described below. In Array Value Propagation, when the array is fixed size and all references to elements have fixed indices, the array is broken up into its elements, i.e. storage is avoided and elements are represented as scalar values. In Loop Carried Array Elimination the array is again replaced by individual elements, but here the values are carried from one iteration to the next.

Full Unrolling of loops with small, compile time assessable numbers of iterations can be important when generating code for FPGAs, because it spreads the iterations in code space rather than in time. Small loops

occur frequently as inner loops in IP codes, for example in convolutions with fixed size masks.

Array Value Propagation searches for array references with constant indices, and replaces such references with the values of the array elements. When the value is a compile time constant, this enables constant propagation. As will be shown in the Prewitt example, this optimization may remove entire user defined (or compiler generated) arrays. When all references to the array have constant indices, the array is eliminated completely.

Loop Carried Array Elimination The most efficient representation of arrays in loop bodies is to have their values reside in registers. This eliminates the need for array allocation in memory and array references causing delays. This requires that the array size be statically known. The important case is that of a loop carried array that changes values but not size during each iteration. To allocate a fixed number of registers for these arrays two requirements need to be met. 1) The compiler must be able to infer the size of the initial array computed outside the loop. 2) Given this size, the compiler must be able to infer that the next array value inside the loop is of the same size.

To infer sizes, the compiler clones the DDCF graph representing the loop, and speculates that the size of the array is as given by its context. It then performs analysis and transformations based on this assumption. There are three possible outcomes of this process. If the size of the next array cannot be inferred, the optimization fails. If the size of the next array can be inferred, but is different from the initial size, it is shown that the array changes size dynamically, and the optimization fails again, but now for stronger reasons. If the size of the next array is equal to the initial size, the sizes of the arrays in all iterations have been proven, by induction, to be equal and the transformation is allowed, the important transformation being array elimination. This form of speculative optimization requires the suppression of global side effects, such as error messages.

N-dimensional Stripmining extends stripmining [39] and creates an intermediate loop with fixed bounds. The inner loop can be fully unrolled with respect to the newly created intermediate loop. As an example the following code shows the convolution of an image with a three by three mask:

```
uint8[:,:] Conv (uint8 I[:,:], uint8 M[3,3]) {
    uint16 Res[:,:] =
        for window W[3,3] in I {
            uint16 ip =
                for w in W dot m in M
                    return (sum (w*m));
            } return (array (ip));
        } return (Res);
```

The inner loop computing ip gets unrolled. However,

this unrolled loop is still rather small and it is often more efficient to execute a number of these inner loops in parallel. If the outer loop is stripmined with a user specified size of eight by eight, the compiler transforms it to the following:

```
uint16 Res[:,:] =
    for window WT[8,8] in I step (6,6) {
        uint16 ResTile[6,6] =
            for window W[3,3] in WT {
                uint16 ip =
                    for w in W dot m in M
                        return (sum (w*m));
            } return (array (ip) );
        } return (tile (ResTile));
```

This is important because the two nested inner loops can now be unrolled, generating a larger more parallel circuit. The **tile** loop return operator concatenates equal size N-dimensional sub-arrays into one N-dimensional array. The compiler generates code to compute left over fringes.

Some (combinations of) operators can be inefficient to implement directly in hardware. For example the computation of magnitude in Prewitt requires multiplications and square root operators. The evaluation of the whole expression can be replaced by an access to a **Lookup Table**, which the compiler creates by wrapping a loop around the expression, recursively compiling and executing the loop, and reading the results.

The performance of many systems today, both conventional and specialized, is often limited by the time required to move data to the processing units. **Fusion** of producer-consumer loops is often helpful, since it reduces data traffic and may eliminate intermediate data structures. In simple cases, where arrays are processed element-by-element, this is straightforward. However, the windowing behavior of many IP operators presents a challenge. Consider the following loop pair:

```
uint8 R0[:,:] =
    for window W[2,2] in Image
        return (array (f (W)));
uint8 R1[:,:] =
    for window W[2,2] in R0
        return (array (g (W)));
```

If the **Image** array has extents $d_0 \times d_1$, then **R0** will have extents $(d_0 - 1) \times (d_1 - 1)$ (determined by the number of 2x2 windows that can be referenced in **Image**.) Similarly, the extents of **R1** will be $(d_0 - 2) \times (d_1 - 2)$. This means that these loops do not have the same number of iterations. Nevertheless, it is possible to fuse such a loop pair by examining their data dependencies. One element of **R1** depends on a 2x2 sub-array of **R0**, and the four values in that sub-array together depend on a

3x3 sub-array of `Image`. It is possible to replace the loop pair with one new loop that uses a 3x3 window and has a loop body that computes one element of `R1` from nine elements of `Image`. Sub-arrays of size 2x2 are extracted from the window, and each feeds a copy of the upper loop body `f`. The values emerging from the copies of `f` feed the lower loop body `g`. This kind of fusion can create large loop bodies since the loop body of the upper loop is replicated in the new loop. These space problems are tackled by Temporal CSE, as described next.

Common Subexpression Elimination (CSE) is an old and well known compiler optimization that eliminates redundancies by looking for identical subexpressions that compute the same value [3]. The redundancies are removed by keeping just one of the subexpressions and using its result for all the computations that need it. This could be called “spatial CSE” since it looks for common subexpressions within a block of code. The SA-C compiler performs conventional spatial CSE, but it also performs *Temporal CSE*, looking for values computed in one loop iteration that were already computed in previous loop iterations. In such cases, the redundant computation can be eliminated by holding such values in registers so that they are available later and need not be recomputed.

Here is a simple example containing a temporal common subexpression:

```
for window W[3,2] in A {
    uint8 s0 = array_sum (W[:,0]);
    uint8 s1 = array_sum (W[:,1]);
} return (array (s0+s1));
```

This code computes a separate sum of each of the two columns of the window, then adds the two. Notice that after the first iteration of the loop, the window slides to the right one step, and the column sum `s1` in the first iteration will be the same as the column sum `s0` in the next iteration. By saving `s1` in a register, the compiler can eliminate one of the two column sums, nearly halving the space required for the loop body. This is the essence of Temporal CSE performed by the SA-C compiler. This optimization requires some special handling to take care of the problem of getting the registers “primed” with initial values, a level of detail that is beyond the scope of this paper.

A useful phenomenon often occurs with Temporal CSE: one or more columns in the left part of the window are unreferenced, making it possible to eliminate those columns. *Narrowing* the window lessens the FPGA space required to store the window’s values. Figure 4 shows the dataflow graph that results after Temporal CSE and Window Narrowing have been applied to the column-sum example above. The register is produced by the TCSE step. Window narrowing then removes one column of the window.

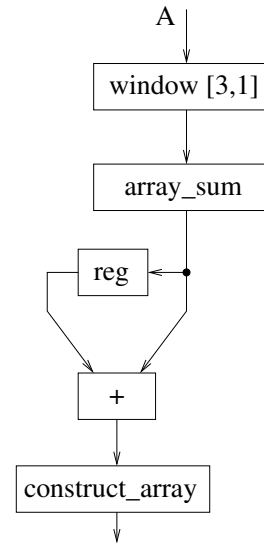


Figure 4. Effect of Temporal CSE and Window Narrowing.

Another optimization that sets the stage for window narrowing moves the computation of expressions fed by window elements into earlier iterations by moving the window references rightward, and uses a register delay chain to move the result to the correct iteration. This can remove references to left columns of the window, and allows window narrowing. This optimization trades window space for register delay chain space. Hence, whether this optimization actually provides a gain depends on the individual computation. We call this *Window Compaction*.

In many cases the performance tradeoffs of various optimizations are not obvious; sometimes they can only be assessed empirically. The SA-C compiler allows many of its optimizations to be controlled by *user pragmas* in the source code. This makes it relatively painless for a user to experiment with different approaches and evaluate the space-time tradeoffs that inevitably exist.

After multiple applications of fusion and unrolling, the resulting loop often has a long critical path, resulting in a low clock frequency. Adding stages of *pipeline registers* can break up the critical path and thereby boost the frequency. The SA-C compiler uses propagation delay estimates, empirically gathered for each of the DFG node types, to determine the proper placement of pipeline registers. The user specifies the number of pipeline stages.

4 Conversion to DFGs

A dataflow graph (DFG) is a low-level, non-hierarchical and asynchronous program representation. DFGs are used for mapping SA-C program parts onto reconfigurable hardware. DFGs can be viewed as abstract hardware circuit diagrams without timing considerations taken into account. Nodes are operators and edges are data paths. Dataflow graphs allow token driven simulation, used by the compiler writer and applications programmer for validation and debugging.

After optimizations have been performed on a program's DDCF graph, the compiler searches for loops that meet the criteria for execution on reconfigurable hardware. The SA-C compiler attempts to translate every innermost loop to a DFG. The innermost loops the compiler finds may not be the innermost loops of the original program, as loops may have been fully unrolled or stripped.

In the present system, not all loops can be translated to DFGs. The most important limitation is the requirement that the sizes of a loop's window generators be statically known.

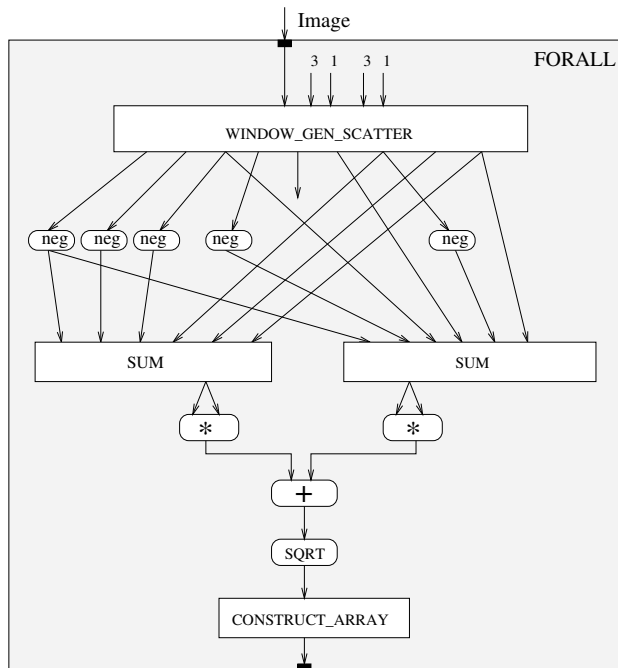


Figure 5. DFG for Prewitt after optimizations.

In the Prewitt program shown earlier, the DDCF graph is transformed to the DFG shown in figure 5. The SUM nodes can be implemented in a variety of ways, including a tree of simple additions. The window generator also allows a variety of implementations, based on

the use of shift registers. The CONSTRUCT_ARRAY node streams its values out to a local memory or to the host, as appropriate.

5 Translation of DFGs to VHDL

The DFG to VHDL translator produces synchronous circuits, expressed in VHDL, from asynchronous DFGs [12]. Each DFG *node* corresponds to a VHDL operation; *edges* correspond to the input and output signals that the operation requires. The simple DFG nodes, such as the arithmetic and logical operators, are strictly combinational. Complex nodes like generator nodes and collector nodes are synchronous and contain registers or state machines that use a clock.

The translator handles the two types of nodes differently. The combinational nodes form the inner loop body (ILB) of the circuit, and are translated either into a single VHDL statement or into the instantiation of a pre-built VHDL component. Complex nodes are implemented by selecting the proper VHDL component from a library of pre-built modules; they are parameterized by information gleaned from the DFG. These clock driven nodes are connected together into a “wrapper” around the ILB, resulting in the structure shown in Figure 6. For each iteration of the loop, the generator produces data at the top of the ILB, and the ILB's output is available a short time later, the exact time being determined by the circuit propagation delay. This delay determines the maximum execution frequency of the resulting circuit.

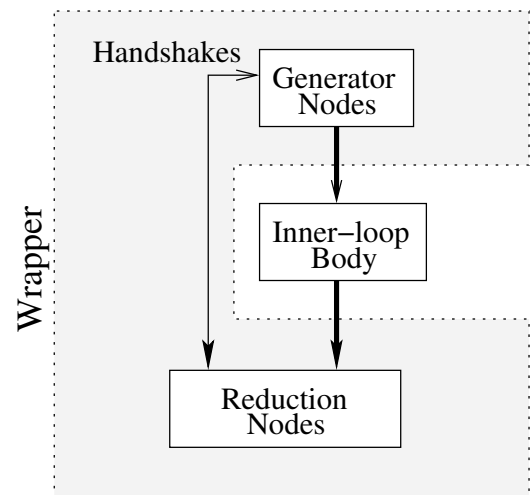


Figure 6. Abstract System Structure

A generator node retrieves the input data in blocks

from the external memory in the order required by the ILB, buffers it, and presents it at the proper time to the inputs of the ILB. A state machine controls the generation of the proper memory access and buffer signals, and provides the timing for the operation of the entire circuit. In the case of a “2D window-generator” node, a 2D buffer array is instantiated, of the same size as the window specified in the generator node. This buffer is formed as a shift register; after the current window of data has been used by the ILB, the oldest column is shifted out, and a new column is shifted in, to form the next window of data. This shift register operation produces the “sliding window” effect required by the SA-C language. Other types of generators provide 1D windows, single array element, and scalars to the ILB in a similar fashion. The generator code also handles the prefetching of data from memory, so the buffers can be kept full and the ILB can be kept as busy as possible.

In a complementary fashion, the reduction nodes take the output from the ILB, buffer this output, and write completed buffers to memory. As with the input, ILB output may be a single element, several single elements, or tiles (sub-arrays.)

The translator selects the proper VHDL components for the generator and reductions and creates VHDL glue logic that “wires” the components together. A parameter file, which specifies the sizes and shapes of the various buffers, as well as any additional timing information required to synchronize the wrapper components, is also created. Finally, the translator generates the script files needed by the commercial VHDL compiler and Place-and-Route tools to synthesize the final design.

6 System Description

As stated earlier, the SA-C system can be used in three modes:

1. Compile the entire program to a host executable. This is useful in the early stages of code development, for quick compiles and functional debugging.
2. Compile the program to dataflow graphs, called by a host executable. The system has a token-driven dataflow simulator that allows validation of the dataflow graphs produced by the compiler. There is an optional “view” mode that displays the DFGs and allows the user to single-step the execution and watch the values flowing through the graph, windows stepping through input images, and creation of result images..
3. Compile the program to FPGA codes, called by a host executable. The Cameron compiler produces

VHDL code [13] that is compiled and place-and-routed by commercial software tools.

The SA-C run-time system has I/O formats that are compatible with standard image processing formats PBM, PGM and PPM. This means that after compilation to host and FPGA codes, the program is ready to run immediately on standard image files.

The current test platform in Cameron is the StarFire Board, produced by Annapolis Microsystems [4], which has a single XCV1000-BG560-4 Virtex FPGA made by Xilinx [41]. The board contains six local memories of one megabyte each. Each of the memories is addressable in 32 bit words; all six can be used simultaneously if needed. The StarFire board is capable of operating at frequencies from 25 MHz to 180 MHz. It communicates over the PCI bus with the host computer at 66 MHz. In our system, the board is housed in a 266-MHz Pentium-based PC.

7 Applications

This section reports the performance of SA-C codes running on the StarFire system described earlier.

7.1 Intel Image Processing Library

When comparing simple IP operators one might write corresponding SA-C and C codes and compare them on the Starfire and Pentium II. However, neither the Microsoft nor the Gnu C++ compilers exploit the Pentium’s MMX technology. Instead, we compare SA-C codes to corresponding operators from the Intel Image Processing Library (IPL). The Intel IPL library consists of a large number of low-level Image Processing operations. Many of these are simple point- (pixel-) wise operations such as square, add, etc. These operations have been coded by Intel for highly efficient MMX execution. Comparing these on a 450 Mhz Pentium II (with MMX) to SA-C on the StarFire, the StarFire is 1.2 to six times slower than the Pentium. This result is not surprising. Although the FPGA has the ability to exploit fine grain parallelism, it operates at a much slower clock rate than the Pentium. These simple programs are all I/O bound, and the slower clock of the FPGA is a major limitation when it comes to fetching data from memory. However, the Prewitt edge detector written in C using IPL calls and running on the 450 Mhz Pentium II, takes 0.053 seconds as compared to 0.017 seconds when running the equivalent SA-C code on the StarFire. Thus, non I/O bound SA-C programs running on the StarFire board are competitive with their hand optimized IPL counterparts.

7.2 ARAGTAP

SA-C running on the StarFire compares even better when we look at more complex operations. The ARAGTAP pre-screener [33] was developed by the U.S. Air Force at Wright Labs as the initial focus-of-attention mechanism for a SAR automatic target recognition application. Aragtap's components include down sampling, dilation, erosion, positive differencing, majority thresholding, bitwise And, percentile thresholding labeling, label pruning, and image creation. All these components, apart from image creation, have been written in SA-C. Most of the computation time is spent in a sequence of eight gray-scale morphological dilations, and a later sequence of eight gray-scale erosions. Four of these dilations are with a 3x3 mask of 1's (the shape of a square), the other four are with a 3x3 mask with 1's in the shape of a cross and zeros at the edges.

First, the compiler can fuse dilate and erode loops in groups of four. (Fusing loop sequences longer than four currently brings the frequency of the resulting FPGA configurations below 25 Mhz, the minimum frequency required by the StarFire.) The dilate and erode loops also allow temporal CSE, window compaction, and window narrowing. Combined with the fusion of four loops, this results in an FPGA code driven by a 9x1 window generator.

A straight C implementation of ARAGTAP running on the Pentium takes 1.07 seconds. Running down sampling, eight unfused dilations, and eight unfused erosions, positive differencing, majority thresholding, and bitwise And on the StarFire board and the rest of the code on the PC takes 0.096 seconds, a more than ten fold speedup over the Pentium. Fusing erode and dilate loops into groups of four brings the execution time down to 0.065 seconds: about 50% gain over the unfused FPGA execution time, and a sixteen fold speedup over the Pentium.

7.3 An Iterative Tri-Diagonal Solver

```
fix16.14 X[:] = DiB; // initial X = D_inverse.B
fix16.14 res[:] =
  for uint8 iter in [2*SIZE] {
    fix16.14 Xperim[:] =
      for uint8 i in [SIZE+2]
        return(array(i==0 || i==(SIZE+1)
          ? (fix16.14)(0.0): X[i-1]));
    // next X = D_inverse.B - D_inverse.(L+U).X
    next X =
      for b in DiB dot l in DiL dot u in DiU
        dot window WX[3] in Xperim
        return(array(b - (l*WX[0] + u*WX[2])));
  }return(final(X));
```

Solving systems of tri-diagonal matrices is important in many applications in e.g. earth sciences. A tri-diagonal system $Ax = b$, where $A = L + D + U$ and L , D and U are a lower diagonal, diagonal and upper diagonal matrices, respectively, can be solved iteratively: $x_n = D^{-1}b - D^{-1}(L + U)x_{n-1}$.

The SA-C code above shows this iterative process. **SIZE** is a compile time constant. The context of this loop allows the size of the initial value of **X** to be inferred to be equal to **SIZE**. The compiler then infers the size of **next X** to be **SIZE** also, so array elimination can occur and the loop computing **next X** can be fully unrolled, allowing the loop computing **res** to be run on the FPGA. A **SIZE = 8** problem runs in 9 microseconds on the Pentium and 18 times faster, in .5 microseconds, on the StarFire.

8 Related work

Hardware and software research in reconfigurable computing is active and ongoing. Hardware projects fall into two categories – those using commercial off-the-shelf components (e.g. FPGAs), and those using custom designs.

The Splash-2 [11] is an early (circa 1991) implementation of an RCS, built from 17 Xilinx [42] 4010 FPGAs, and connected to a Sun host as a co-processor. Several different types of applications have been implemented on the Splash-2, including searching[23, 32], pattern matching[35], convolution [34] and image processing [6].

Representing the current state of the art in FPGA-based RCS systems are the AMS WildStar[5] and the SLAAC project [36]. Both utilize Xilinx Virtex [41] FPGAs, which offer over an order of magnitude more programmable logic, and provide a several-fold improvement in clock speed, compared to the earlier chips.

Several projects are developing custom hardware. The Morphosys project [27] marries an on-board RISC processor with an array of reconfigurable cells (RCs). Each RC contains an ALU, shifter, and a small register file.

The RAW Project [38] also uses an array of computing cells, called *tiles*; each tile is itself a complete processor, coupled with an intelligent network controller and a section of FPGA-like configurable logic that is part of the processor data path. The PipeRench [18] architecture consists of a series of *stripes*, each of which is a pipeline stage with an input interconnection network, a lookup-table based PE, a results register, and an output network. The system allows a virtual pipeline of any size to be mapped onto the finite physical pipeline.

On the software front, a framework called “Nimble” compiles C codes to reconfigurable targets where the

reconfigurable logic is closely coupled to an embedded CPU [26]. Several other hardware projects also involve software development. The RAW project includes a significant compiler effort [2] whose goal is to create a C compiler to target the architecture. For PipeRench, a low-level language called DIL [17] has been developed for expressing an application as a series of pipeline stages mapped to stripes.

Several projects (including Cameron) focus on hardware-independent software for reconfigurable computing; the goal – still quite distant – is to make development of RCS applications as easy as for conventional processors, using commonly known languages or application environments. Several projects use C as a starting point. DEFACTO [19] uses SUIF as a front end to compile C to FPGA-based hardware. Handel-C [1] both extends the C language to express important hardware functionality, such as bit-widths, explicit timing parameters, and parallelism, and limits the language to exclude C features that do not lend themselves to hardware translation, such as random pointers. Streams-C [15, 16] does a similar thing, with particular emphasis on extensions to facilitate the expression of communication between parallel processes. SystemC [37] and Ocapi [24] provide C++ class libraries to add the functionality required of RCS programming to an existing language.

Finally, a couple of projects use higher-level application environments as input. The MATCH project [7, 8, 30] uses MATLAB as its input language – applications that have already been written for MATLAB can be compiled and committed to hardware, eliminating the need for re-coding them in another language. Similarly, CHAMPION [29] is using Khoros [25] for its input – common glyphs have been written in VHDL, so GUI-based applications can be created in Khoros and mapped to hardware.

9 Conclusions and Future Work

The Cameron Project has created a language, called SA-C, for one-step compilation of image processing applications that target FPGAs. Various optimizations, both conventional and novel, have been implemented in the SA-C compiler. These optimizations are focused on reducing execution time and/or reducing FPGA space. The system has been used to implement some simple primitive IP operations, such as the routines from the Intel IPL, as well as more comprehensive applications, such as the ARAGTAP target acquisition prescanner.

Performance evaluation of the SA-C system has just begun. As performance issues become clearer, the system will be given greater ability to evaluate various metrics including code space, memory use and time perfor-

mance, and to evaluate the tradeoffs between conventional functional code and lookup tables.

More compiler optimizations are under way, including further manipulations of window generators. Some more optimizations at the Dataflow graph level are also being implemented. One optimization involves the replacement of delay-intensive portions of the DFG with lookup tables, which often trade FPGA space for a more time-efficient solution.

Also, stream data structures are being added to the SA-C language, which will allow multiple cooperating processes to be mapped onto FPGAs.

References

- [1] OXFORD hardware compiler group, the Handel Language. Technical report, Oxford University, 1997.
- [2] A. Agarwal, S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, and M. Srikrishna, D. and Taylor. The RAW compiler project. In *Proc. Second SUIF Compiler Workshop*, August 1997.
- [3] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.
- [4] Annapolis Micro Systems, Inc., Annapolis, MD. *WILDFORCE Reference Manual*, 1997. www.annapmicro.com.
- [5] Annapolis Micro Systems, Inc., Annapolis, MD. *STARFIRE Reference Manual*, 1999. www.annapmicro.com.
- [6] P. M. Athanas and A. L. Abbott. Processing images in real time on a custom computing platform. In R. W. Hartenstein and M. Z. Servit, editors, *Field-Programmable Logic Architectures, Synthesis, and Applications*, pages 156–167. Springer-Verlag, Berlin, 1994.
- [7] P. Banerjee et al. A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems. In *The 8th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2000.
- [8] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Chang, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, and M. Walkden. MATCH: a MATLAB compiler for configurable computing systems. Technical Report CPDC-TR-9908-013, Center for Parallel and distributed Computing, Northwestern University, August 1999.
- [9] A. Benedetti and P. Perona. Real-time 2-d feature detection on a reconfigurable computer. In *IEEE Conference on Computer Vision and Pattern Recognition*, Santa Barbara, CA, 1998.
- [10] D. Benitez and J. Cabrera. Reactive computer vision system with reconfigurable architecture. In *International Conference on Vision Systems*, Las Palmas de Gran Canaria, Spain, 1999.

- [11] D. Buell, J. Arnold, and W. Kleinfelder. *Splash 2: FPGAs in a Custom Computing Machine*. IEEE CS Press, 1996.
- [12] M. Chawathe. Dataflow graph to VHDL translation. Master's thesis, Colorado State University, 2000.
- [13] M. Chawathe, M. Carter, C. Ross, R. Rinker, A. Patel, and W. Najjar. Dataflow graph to VHDL translation. Technical report, Colorado State University, Dept. of Computer Science, 2000.
- [14] J. Eldredge and B. Hutchings. Rrann: A hardware implementation of the backpropagation algorithm using reconfigurable fpgas. In *IEEE International Conference on Neural Networks*, Orlando, FL, 1994.
- [15] M. Gokhale. The Streams-C Language, 1999. www.darpa.mil/ito/psum1999/F282-0.html.
- [16] M. Gokhale et al. Stream oriented PFGA computing in Streams-C. In *The 8th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2000.
- [17] S. C. Goldstein and M. Budiu. *The DIL Language and Compiler Manual*. Carnegie Mellon University, 1999. www.ece.cmu.edu/research/piperench/dil.ps.
- [18] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. Piperench: A coprocessor for streaming multimedia acceleration. In *Proc. Intl. Symp. on Computer Architecture (ISCA '99)*, 1999. www.cs.cmu.edu/~mihaib/research/isca99.ps.gz.
- [19] M. Hall, P. Diniz, K. Bondalapati, H. Ziegler, P. Duncan, R. Jain, and J. Granacki. DEFACTO: A design environment for adaptive computing technology. In *Proc. 6th Reconfigurable Architectures Workshop (RAW'99)*. Springer-Verlag, 1999.
- [20] J. Hammes and W. Böhm. *The SA-C Language - Version 1.0*, 1999. Document available from www.cs.colostate.edu/cameron.
- [21] J. Hammes, R. Rinker, W. Böhm, and W. Najjar. Cameron: High level language compilation for reconfigurable systems. In *PACT'99*, Oct. 1999.
- [22] R. Hartenstein et al. A reconfigurable machine for applications in image and video compression. In *Conference on Compression Technologies and Standards for Image and Video Compression*, Amsterdam, Holland, 1995.
- [23] D. Hoang. Searching genetic databases on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–192. CS Press, Los Alamitos, CA, 1993.
- [24] IMEC. Ocapi overview. www.imec.be/ocapi/.
- [25] K. Konstantinides and J. Rasure. The Khoros software development environment for image and signal processing. In *IEEE Transactions on Image Processing*, volume 3, pages 243–252, May 1994.
- [26] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *Proc. 37th Design Automation Conference*, 1999.
- [27] G. Lu, H. Singh, M. Lee, N. Bagherzadeh, and F. Kurhadi. The Morphosis parallel reconfigurable system. In *Proc. of EuroPar 99*, Sept. 1999.
- [28] W. Najjar. The Cameron Project. Information about the Cameron Project, including several publications, is available at the project's web site, www.cs.colostate.edu/cameron.
- [29] S. Natarajan, B. Levine, C. Tan, D. Newport, and D. Bouldin. Automatic mapping of Khoros-based applications to adaptive computing systems. Technical report, University of Tennessee, 1999. Available from <http://microsys6.engr.utk.edu:80/~bouldin/darpa/mapld2/mapld-paper.pdf>.
- [30] S. Periyayacheri, A. Nayak, A. Jones, N. Shenoy, A. Choudhary, and P. Banerjee. Library functions in reconfigurable hardware for matrix and signal processing operations in MATLAB. In *Proc. 11th IASTED Parallel and Distributed Computing and Systems Conf. (PDCS'99)*, November 1999.
- [31] D. Perry. *VHDL*. McGraw-Hill, 1993.
- [32] D. V. Pryor, M. R. Thistle, and N. Shirazi. Text searching on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 172–178. CS Press, Los Alamitos, CA, 1993.
- [33] S. Raney, A. Nowicki, J. Record, and M. Justice. Aragtat atr system overview. In *Theater Missile Defense 1993 National Fire Control Symposium*, Boulder, CO, 1993.
- [34] N. K. Ratha, D. T. Jain, and D. T. Rover. Convolution on Splash 2. In *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pages 204–213. CS Press, Los Alamitos, CA, 1995.
- [35] N. K. Ratha, D. T. Jain, and D. T. Rover. Fingerprint matching on Splash 2. In *Splash 2: FPGAs in a Custom Computing Machine*, pages 117–140. IEEE CS Press, 1996.
- [36] B. Schott, S. Crago, C. C., J. Czarnaski, M. French, I. Hom, T. Tho, and T. Valenti. Reconfigurable architectures for systems level applications of adaptive computing. Available from <http://www.east.isi.edu/SLAAC/>.
- [37] SystemC. SystemC homepage. www.systemc.org/.
- [38] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: RAW machines. *Computer*, September 1997.
- [39] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.
- [40] J. Woodfill and B. v. Herzen. Real-time stereo vision on the parts reconfigurable computer. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 1997.
- [41] Xilinx, Inc. *Virtex 2.5V Field programmable Gate Arrays: Preliminary Product Description*, Oct. 1999. www.xilinx.com.
- [42] Xilinx, Inc., San Jose, CA. *The Programmable Logic Databook*, 1998. www.xilinx.com.

CAMERON PROJECT: FINAL REPORT

Appendix L: Precision vs. Error in JPEG Compression

Precision vs. Error in JPEG Compression

José Bins, Bruce A. Draper, Willem A.P. Böhm, and Walid Najjar

Computer Science Department, Colorado State University,
Fort Collins, CO, USA

ABSTRACT

By mapping computations directly onto hardware, reconfigurable machines promise a tremendous speed-up over traditional computers. However, executing floating-point operations directly in hardware is a waste of resources. Variable precision fixed-point arithmetic operations can save gates and reduce clock cycle times. This paper investigates the relation between precision and error for image compression/decompression. More precisely, this paper investigates the relationship between error and bit-precision for the Discrete Cosine Transform (DCT) and JPEG.

The present work is part of the Cameron project at the Computer Science Department of Colorado State University. This project is roughly divided in three areas: an C-like parallel language called SA-C that is targeted for image processing on reconfigurable computers, an implementation of the VSIP library for image processing in SA-C, and an optimizing compiler for SA-C that targets FPGAs.

Keywords: Image processing, Fixed-point arithmetic, Reconfigurable computer

1. INTRODUCTION

Field-programmable gate arrays (FPGAs) and other reconfigurable systems offer a fundamentally new model of computation in which programs are mapped directly onto hardware. FPGAs are composed of massive arrays of simple logic units, each of which can be programmed to compute arbitrary functions over small numbers of bits (usually four or five). In addition, the wires connecting the logic units have programmable interconnections. As a result, programs are mapped into circuits, which can then be dynamically loaded into the FPGA hardware. When the program is finished, the hardware is reconfigured into a new circuit for the next program. (Mangione-Smith^{1,2} gives an introduction to reconfigurable computing, while Rose, et al³ provide a general, if now somewhat dated, survey of reconfigurable processors, and Buell, et al⁴ describe the application of FPGAs in practice.)

By mapping computations directly onto hardware, reconfigurable machines promise a tremendous speed-up over traditional computers. Part of this speed-up comes from *parallelism*, while the rest comes from increased *computational density*. The potential for parallelism is obvious; FPGAs are composed of thousands of logic blocks, and with reprogrammable interconnections the degree of parallelism is limited only by the data dependencies within the program (and I/O limitations). Computational density is equally important, however. Traditional processors implement a complex fetch-and-execute cycle that requires many special purpose units. Only a small fraction of the hardware (and time) is actually dedicated to the computation being performed. The computational density of these processors – the amount of space consumed per useful operation – is therefore very low. FPGA circuits, on the other hand, can be designed so that data “flows through” the circuit without repeated storage and retrieval, and without wasting resources on unused circuitry. This leads to more efficient processing and a potential speed-up of up to two orders of magnitude.^{5,6}

One example of increased computational density in FPGAs is the ability to do variable precision arithmetic. While traditional processors have arithmetic units designed for fixed-width operations (either 16, 32 or 64 bits), FPGA circuits can be written for any bit precision. For example, if an algorithm calls for multiplying two seven-bit numbers, there is no need to allocate the resources of a 32-bit multiplier; a circuit for multiplying seven-bit numbers is created instead. Such variable precision arithmetic can save gates and reduce clock cycle times.

Variable precision arithmetic can be very powerful in image processing, where one-bit (binary), eight-bit (byte), and twelve-bit pixel values are common. It is less clear, however, whether variable precision arithmetic is useful in frequency-space applications where floating point numbers are typically used. One important example is JPEG image

E-mail: (bins,draper,bohm,najjar)@cs.colostate.edu

compression/decompression, which uses the Discrete Cosine Transform (DCT) and its inverse to convert images from the spatial domain to the frequency domain and back again. This paper investigates the relation between precision and error for JPEG image compression/decompression.

More precisely, this paper investigates the relationship between error and bit-precision first for the DCT and then for JPEG. In order to simplify the FPGA circuits (and because there is no floating point standard for anything shorter than 32 bits), we have implemented DCT and JPEG using fixed point, rather than floating point, arithmetic, and we measure the increase in reconstruction error as the precision of the fixed point values is decreased. Because DCT and JPEG depend on the frequency components of an image, we measure the precision/accuracy tradeoff for sets of real, artificial, and synthetic images created with different spectral components. Reconstruction error is measured in terms of total gray-level error, RMS gray-level error, RMS signal-to-noise ratio, and peak signal-to-noise ratio.⁷ The results indicate that traditionally sized numbers are necessary for the addition tree in the DCT algorithm, but that a more moderate number of bits can be used for the multiplications. More significantly, far fewer bits are needed when the DCT is used within JPEG. These results are consistent across the spectrum of images tested.

2. THE DISCRETE COSINE TRANSFORM (DCT)

2.1. Definition of DCT

The DCT maps images from the intensity domain to the frequency domain. A simple, one-dimensional DCT is defined as:

$$C(u) = \alpha(u) \sum_{x=0}^{N-1} f(x) \cos \left[\frac{(2x+1)u\pi}{2N} \right], \quad (1)$$

where

$$\alpha(u) \begin{cases} \sqrt{\frac{1}{N}} & \text{for } u = 0 \\ \sqrt{\frac{2}{N}} & \text{for } u = 1, 2, \dots, N-1. \end{cases} \quad (2)$$

The inverse DCT is similarly defined as:

$$f(x) = \sum_{u=0}^{N-1} \alpha(u) C(u) \cos \left[\frac{(2x+1)u\pi}{2N} \right]. \quad (3)$$

For image compression, we need the two-dimensional form of the DCT, which is written as:

$$C(u, v) = \alpha(u)\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x, y) \cos \left[\frac{(2x+1)u\pi}{2N} \right] \cos \left[\frac{(2y+1)v\pi}{2N} \right]. \quad (4)$$

Note that JPEG first divides the image into 8×8 sub-images, so that in the context of JPEG N always equals eight. The two dimensional inverse DCT (IDCT) is written as:

$$f(x, y) = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} \alpha(u)\alpha(v) C(u, v) \cos \left[\frac{(2x+1)u\pi}{2N} \right] \cos \left[\frac{(2y+1)v\pi}{2N} \right]. \quad (5)$$

Blinn⁸ provides an intuitive explanation of the DCT and some of its most important properties.

2.2. Minimum Error for DCT

Some readers may object to the idea of tolerating error in the DCT/IDCT. After all, we are taught that the DCT is an exactly invertible mapping between the spatial and frequency domains. Unfortunately, as commonly used the DCT/IDCT *does* introduce error. The question is how much error can be tolerated for a particular application.

To get some intuition about the error introduced by DCT, the equation for the one-dimensional DCT (Eq. 1) can be rewritten as:

$$c = Ax \quad (6)$$

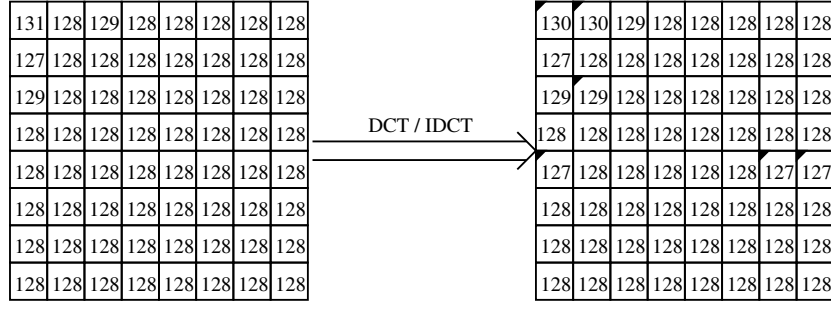


Figure 1. An 8 x 8 source image and the reconstructed image after DCT and IDCT. Errors are introduced into five pixels (shown with marked corners) as a result of rounding frequencies to the nearest integer.

where c is the vector (length = N) of frequency values, x is the initial vector of data, and A is a constant matrix of products of cosines and alphas. Note that the terms in A depend only on N , not on x , so A is constant as long as N is fixed.

As is clear from Eq. 6, the relationship between the spatial-domain pixels x and the frequency-domain values c is linear. Since A is non-degenerate, it can be inverted, leading to the inverse DCT. In fact, A is orthonormal, so $A^{-1} = A^T$, which explains the relationship between Eqs. 1 and 3. Unfortunately, as defined in the Vector, Signal and Image Processing Library (VSIP^{*}), the Intel Image Processing Library[†] and JPEG,⁹ the output of the DCT is a frequency-domain image with integer pixels. This requires that the values of c be rounded to the nearest integer and introduces error into the reconstruction, even if infinite precision was used to calculate Ax . In particular, if a 1D signal is converted into the frequency domain by the DCT, has its values rounded to the nearest integer, and then is converted back into the spatial domain using the inverse DCT, there is a worst-case error in term x_i of $0.5 \times \sum_{j=0}^{N-1} |a_{i,j}^{-1}|$, where $a_{i,j}^{-1}$ is the i th row, j th column element of A^{-1} . For $N = 8$, this implies an error of up to 1.32 gray levels for one dimensional data. If the spatial-domain result of the IDCT is rounded back to the nearest integer (assuming the source data was integer), that still leaves the possibility of a reconstruction error of one gray level for any element x_i .

In two dimensions, the situation is similar. Once again, the frequency-space values are rounded to the nearest integer. Since rounding introduces an error of up to ± 0.5 in each frequency term, the maximum error $E_{x,y}$ for a pixel in the reconstructed image is

$$E_{x,y} \leq 0.5 \times \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} |\alpha(u)\alpha(v)\cos\left(\frac{(2x+1)u\pi}{2N}\right)\cos\left(\frac{(2y+1)v\pi}{2N}\right)| = 3.489$$

creating a possible final error of up to three gray levels. Figure 1 shows an image that is degraded by DCT/IDCT as a result of rounding the frequency values.

2.3. Implementation of DCT/IDCT

Because Eq. 4 is typically applied to 8×8 subwindows (and there are many such windows in any image), an efficient implementation is to precompute $\cos\left[\frac{(2x+1)u\pi}{2N}\right]\cos\left[\frac{(2y+1)v\pi}{2N}\right]$ and $\alpha(u)\alpha(v)$, storing them as $8 \times 8 \times 8 \times 8$ and 8×8 matrices, respectively. To compute the DCT, we then multiply the elements of these matrixes by the corresponding image pixels and sum the intermediate values, as shown in Fig. 2. Note that this implementation is designed to be fast on a parallel machine, where all the multiplications can be done in parallel, and the results can be added in a tree. A so-called Fast DCT algorithm,¹⁰ which minimizes the total number of multiplications at the cost of a longer sequence of steps, is optimized for sequential processors.

In terms of precision, the variables in Fig. 2 can be grouped into two classes. The first are the stored cosine and α terms. Since these terms range from one to minus one, in fixed point notation they require at least two bits to

^{*}see <http://www.vsipl.org>

[†]see <http://developer.intel.com/vtune/perfbist/ipl>

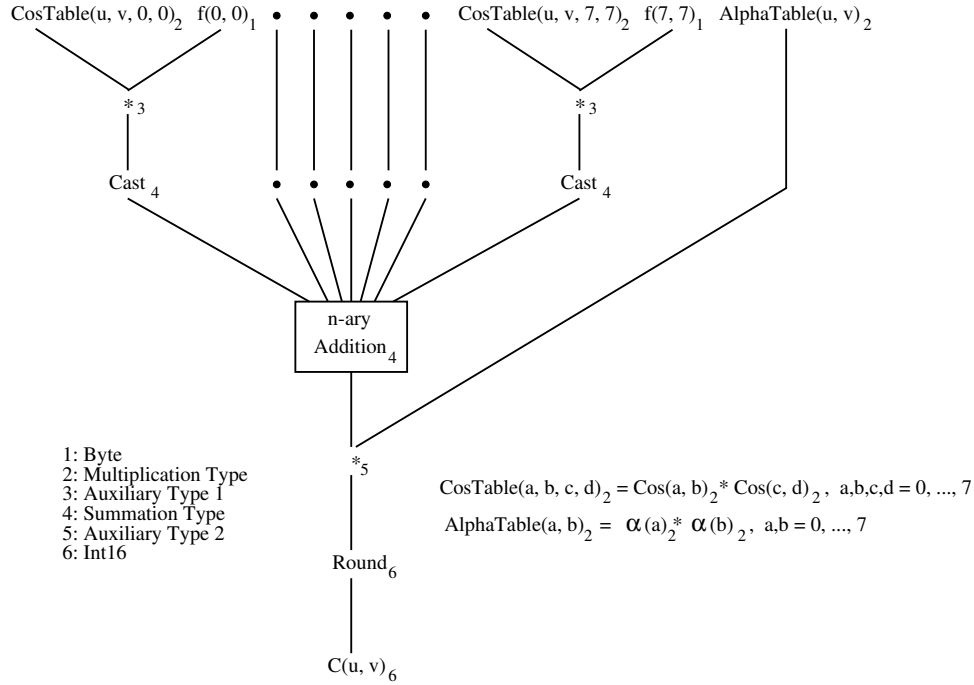


Figure 2. Data flow of the DCT computation for $C(u, v)$ showing types of variables and intermediate results. Auxiliary type 1 is the result of multiplying a byte with *multiplication type*, and auxiliary type 2 is the result of multiplying *summation type* and *multiplication type*. The type inference rules are given in Sect..?

the left of the binary point to represent their sign and integer magnitude. The number of bits to the right can be varied, however, to trade precision against accuracy. The precision of the cosine and α terms in turn determines the complexity of the multipliers in the circuit shown in Fig. 2. In the case of the multipliers at the top of the figure, these will be special-purpose circuits designed to multiply 8 bit integer pixel values to fixed point cosine terms. The higher the precision of the cosine terms, the greater the complexity of the multipliers, and therefore the overall circuit. We therefore refer to the precision of the cosine and α terms as the *multiplication type*.

The second group of variables are the terms used for the n-ary addition in Fig. 2. In hardware, the n-ary addition is implemented as a tree of binary additions. Since there are $8 \times 8 = 64$ terms being added, the n-ary addition is a depth-six binary tree of adders. In the forward DCT, the terms being added must have at least 16 bits to the left of binary point to hold the sign and integer magnitude of the final result, but we can once again vary the number of bits used to the right of the binary point to exchange precision for accuracy. We refer to the precision of these terms as the *summation type*.

For these experiments, we implemented the DCT and inverse DCT in SA-C,^{11,12} a high-level programming language designed for FPGAs as part of the Cameron project.¹³ The Cameron project is dedicated to making FPGAs available to non-circuit designers by creating a high-level language and optimizing compiler that target reconfigurable processors. SA-C is a single assignment dialect of C created for Cameron that, among other things, includes variable precision fixed point numbers. For example, in SA-C the datatype *fix14.8* indicates a signed fixed point number with a total of 14 bits, eight of which are to the right of the binary point. (By subtraction, the other six bits are to the left of the binary point and signify the sign and integer size of the fixed point number.) In general, all fixed point types of the form *fixX.Y*, where $0 \leq Y \leq X \leq 32$, are supported by SA-C. (Unsigned fixed point numbers, written as *ufixX.Y*, are also supported.) SA-C's type inference rules define the result of an operation (e.g. addition or multiplication) between two fixed point values as a fixed point number with an integral size equal to the maximum integral sizes of the operands, and a fractional size equal to the maximum fractional size of the operands. The only limitation is that if the sum of the integral and fractional sizes exceeds 32 bits, the fractional size is reduced until the total size is 32.

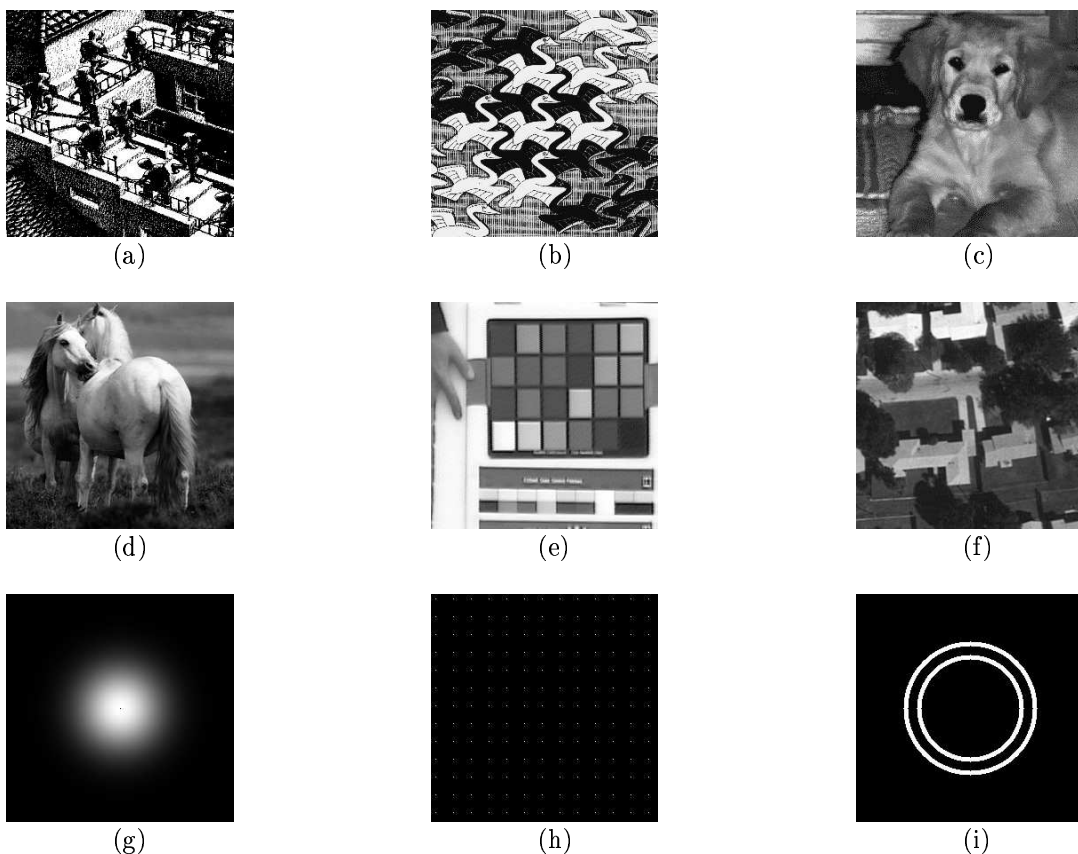


Figure 3. Subset of images used: (a) compacted/uncompacted drawing; (b) drawing; (c) compacted/uncompacted animal; (d) animal; (e) color palette; (f) houses at Fort Hood (TX); (g) Gaussian filter (Variance = 1000); (h) impulse images (impulse spacing = 20 pixels); and (i) concentric circumferences.

2.4. Evaluating DCT/IDCT

The data sets used to measure the relationship between precision and error in the DCT were composed of natural, artificial and synthetic images, some of which are shown in Fig. 3[‡]. The natural and artificial images were collected from the web. The artificial images are drawings extracted from Escher’s work (five drawings). The natural ones are: six animal images, one color palette, one image of a seed, one aerial image of Fort Hood (TX), and one specular microscope image of cornea endothelial cells. All images were clipped to 256 x 256 and when necessary converted to black and white.

Because the DCT maps between the spatial and frequency domains, we also tested synthetic images containing controlled frequencies. We tested three Gaussian filter images with different variances, three noise images (uniform, Gaussian and exponential noise), two impulse images with different spacing between the impulses, one sinusoid image, one image of a constant-intensity circle against a constant background, and one image of two concentric rings.

Figure 4 shows the histogram for images (a), (f), and (g) of Fig. 3. As can be seen, image (a) (which had been previously compressed/uncompressed before we retrieved it off the web) has far fewer peaks. This image, together with image (c) (which was also previously compressed/uncompressed) were included to evaluate the influence of a sparse histogram on the reconstruction error. Some of the synthetic images also have sparse values due to the way they were constructed. At the extreme, images (h) and (i) are bitonal images (with values of 0 and 255).

To evaluate the effect of precision on the reconstruction error, we converted each image into the frequency domain using the DCT, rounded the frequency values to the nearest integer, and then converted them back into the spatial

[‡]The complete set of images can be seen at <http://www.cs.colostate.edu/cameron/SPIE99.html>

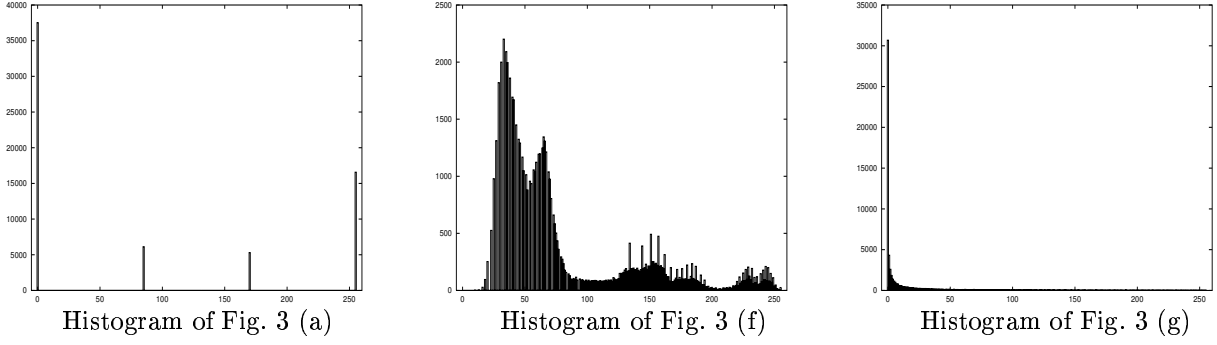


Figure 4. Histogram of images (a), (f), and (g) of Fig. 3.

Table 1. Reconstruction Error Measures: total error (upper left), RMS error (upper right), RMS signal-to-noise ratio (lower left), and peak signal-to-noise ratio (lower right).

$\sum_{x=0}^{N-1} \sum_{y=0}^{N-1} [\hat{f}(x, y) - f(x, y)]$	$\sqrt{\frac{\sum_{x=0}^{N-1} \sum_{y=0}^{N-1} [\hat{f}(x, y) - f(x, y)]^2}{N^2}}$
$\sqrt{\frac{\sum_{x=0}^{N-1} \sum_{y=0}^{N-1} \hat{f}(x, y)^2}{\sum_{x=0}^{N-1} \sum_{y=0}^{N-1} [\hat{f}(x, y) - f(x, y)]^2}}$	$10 \log_{10} \frac{[L-1]^2}{\sum_{x=0}^{N-1} \sum_{y=0}^{N-1} [\hat{f}(x, y) - f(x, y)]^2}$

domain using the inverse DCT. We then compared the original images to the reconstructed versions. We repeated this process for all 26 images and for all combinations of the multiplication and summation types (i.e. precisions). The multiplication types tested were: float, fix32.30, fix28.26, fix26.22, fix18.16, fix17.15, fix16.14, fix15.13, fix14.12, fix12.10, fix10.8, fix8.6, fix6.4, and fix4.2. (Remember that the cosine and α terms always require two bits to the left of the binary point to represent their sign and integer magnitude.) The summation types require 16 bits to the left of the binary point to represent their integer magnitude and sign, so the summation types tested were: float, fix32.16, fix28.12, fix24.8, fix22.6, fix20.4, fix18.2 and integer. All 112 of these combinations were executed. The reconstructed images were compared to the originals using four measures (defined in Tab. 1)⁷ : total error, RMS error, RMS signal-to-noise ratio, and peak signal-to-noise ratio.

Figure 5 shows the errors resulting from the DCT/IDCT reconstruction process on image (f) of Fig. 3. Each curve in Fig. 5 corresponds to one precision level for the summation type. The horizontal axes represent precisions associated with the multiplication type. In this case, restricting the multiplication type to sixteen bits to the right of the binary point (with two bits to the left) introduces a negligible amount of error[§], implying that 18 bits of precision are enough for these terms. This is significant, since hardware multipliers require more resources than adders. Unfortunately, the DCT is more sensitive to the summation precision. Using fixed point precision for the summation terms creates a significant amount of error. This implies that in a strict implementation of the DCT, the tree of additions (represented by the n-ary addition in Fig. 2) *must use floating point addition*.

Significantly, these results were consistent across the images tested. Figure 6 shows the results of the total error for three more images. In general, the natural and artificial images had almost indistinguishable error curves to each other, whether or not the images had been previously compressed. A small number of synthetic images presented slightly different results. The most significant difference was less error for some images where the background dominates. This is expected, since most of the 8×8 background windows are constant and equal to zero in this

[§]Where “negligible” means less than 0.01 gray levels per pixel on average.

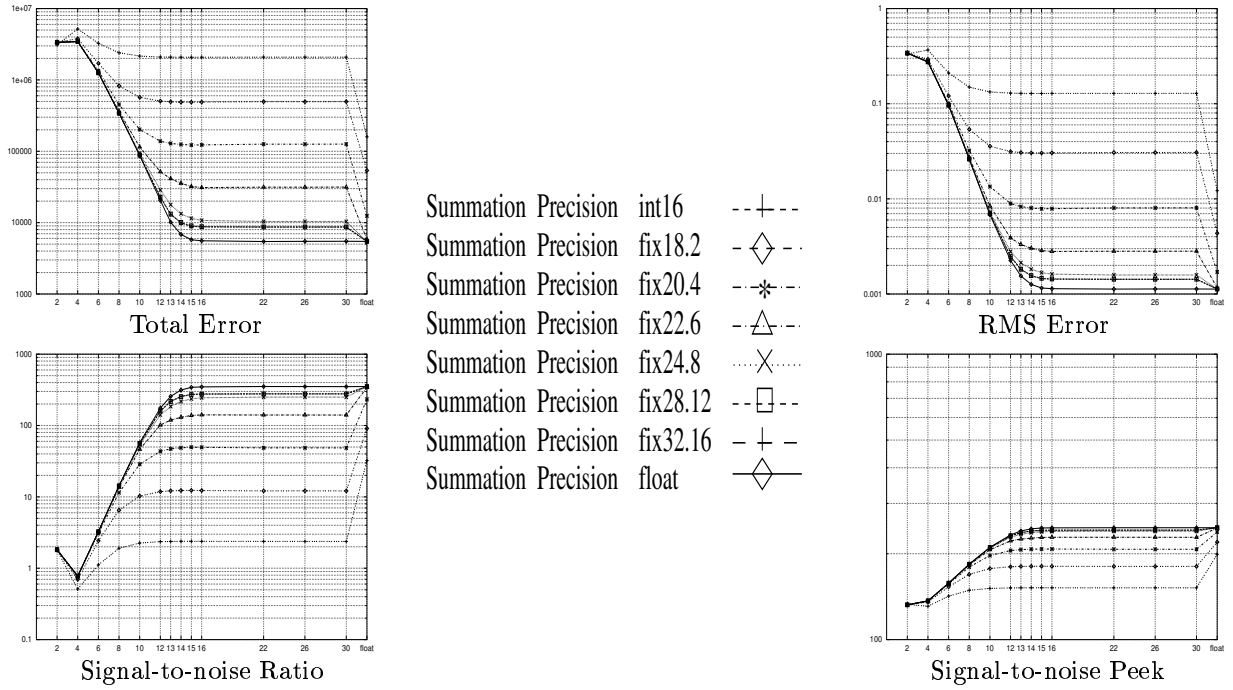


Figure 5. Result of DCT reconstruction for image (*f*) of Fig. 3 for each of the four measures.

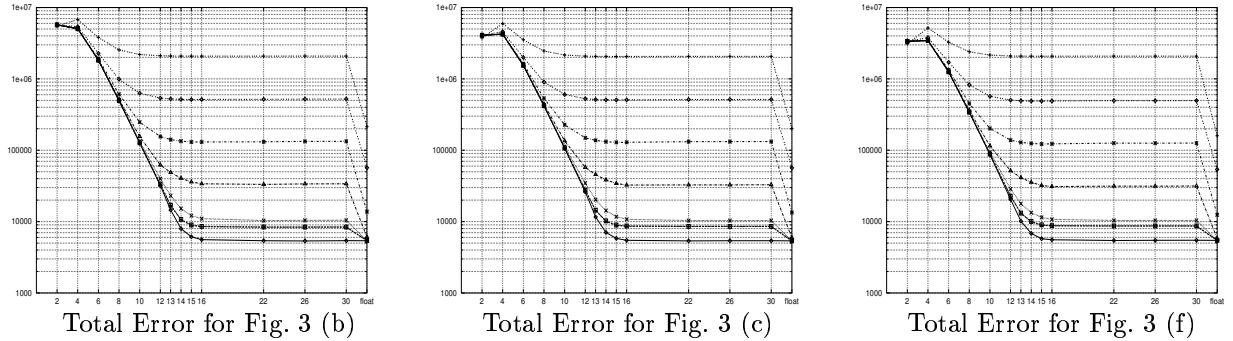
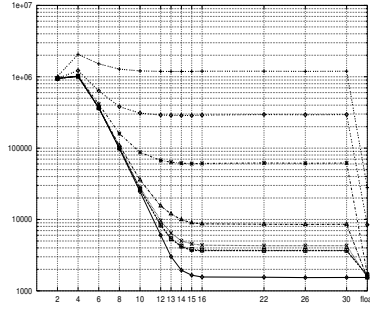


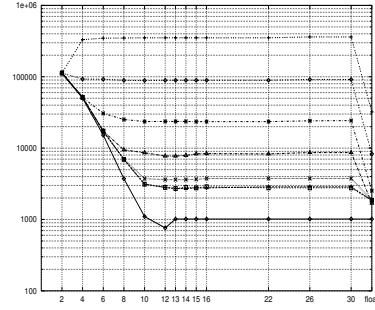
Figure 6. DCT Total Error for images (b), (c), and (f) of Fig. 3.

case. Nevertheless, the shape of the curves remain the same. As shown in Fig. 7, there is an unexplained anomaly in image (*i*) of Fig. 3, where the best result occurs for 12 bits of precision for the multiplication type.

In applications where a slight increase in error is tolerable for the DCT, faster circuits can still be constructed. Using floating point computation as the best case, we computed the difference between floating point arithmetic and combinations of fixed point precisions. This measures how much error is added by each combination of precisions. Figure 8 and Tab. 2 show the average increase in error for each pixel for all artificial and natural images. As can be seen in graph (b), although there is error, the error is very small for most precision combinations. For example, if an average increased error of one gray level or less is acceptable, a fractional precision of 12 bits for the multiplication type and 6 for the summation type are enough. That means 14 bits (2+12) for the cosine and α variables and 22 (16+6) for the internal summation variables, a savings of 18 and 10 bits respectively over a 32 bit floating point representation. Moreover, fixed point arithmetic requires fewer logic blocks and less time than floating point arithmetic does. The synthetic images have similar results, but the increase in error was even smaller. Figure 9 shows the average results for the synthetic images.

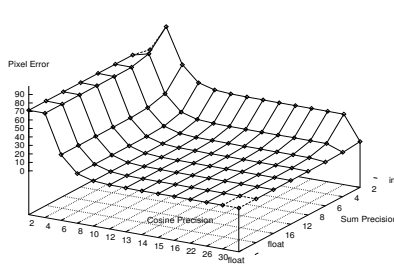


Total Error for Fig. 3 (g)

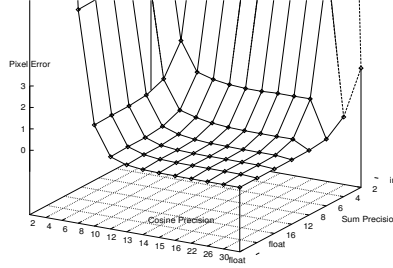


Total Error for Fig. 3 (i)

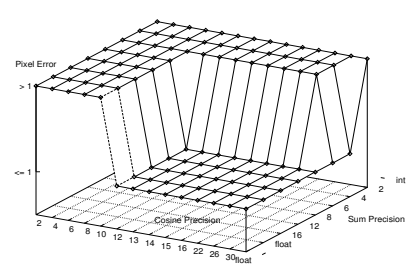
Figure 7. DCT Total Error for images (g) and (i) of Fig. 3.



(a)



(b)



(c)

Figure 8. Average increase in pixel error for all artificial and natural images. (a) shows the full graph, (b) shows the expanded graph, and (c) shows the thresholded graph (threshold error = 1).

Table 2. Average increase in pixel error for all artificial and natural images after DCT/IDCT. Rows represent multiplication types and columns represent summation types.

	float	fix32.16	fix28.14	fix24.8	fix22.6	fix20.4	fix18.2	int16
float	0.0000	0.0056	0.0056	0.0052	0.0108	0.1099	0.7142	2.5607
fix32.30	0.0000	0.0552	0.0558	0.0854	0.3994	1.8538	7.6533	31.4997
fix28.26	0.0001	0.0552	0.0558	0.0852	0.3974	1.8465	7.6313	31.4545
fix24.22	0.0000	0.0552	0.0559	0.0848	0.3953	1.8411	7.6246	31.4610
fix18.16	0.0020	0.0568	0.0584	0.0933	0.3985	1.8084	7.5531	31.4266
fix17.15	0.0079	0.0622	0.0639	0.1082	0.4258	1.8047	7.5216	31.3574
fix16.14	0.0326	0.0865	0.0886	0.1502	0.4920	1.8547	7.5523	31.3954
fix15.13	0.1187	0.1682	0.1703	0.2611	0.6195	1.9602	7.6588	31.5136
fix14.12	0.3638	0.4003	0.4003	0.4982	0.8215	2.1686	7.8722	31.7296
fix12.10	1.7140	1.7440	1.7440	1.8243	2.1502	3.5058	9.2145	33.0276
fix10.8	6.9594	6.9908	6.9908	6.9908	7.3059	8.6650	14.3759	38.1041
fix8.6	25.8551	25.8796	25.8796	25.8796	25.8796	27.1488	32.6963	55.9756
fix6.4	71.2694	71.2547	71.2547	71.2547	71.2547	71.2547	76.3535	97.3901
fix4.2	71.3656	71.5043	71.5043	71.5043	71.5043	71.5043	71.5043	66.9298

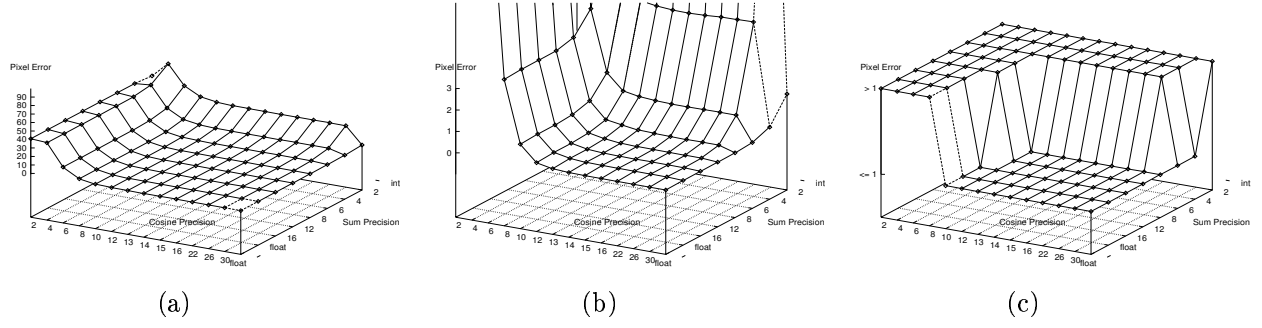


Figure 9. Average increase in pixel error for all synthetic images. (a) shows the full graph, (b) shows the expanded graph, and (c) shows the thresholded graph (threshold error = 1).

For each 8x8 sub-image:

1. subtract 128 from pixel values
2. apply Discrete Cosine Transform (DCT)
3. normalize: $newval(x,y) = round(f(x,y)/table(x,y))$
4. linearize values into a vector of length 64 (S-pattern)
5. discard trailing zeroes
6. apply Huffman code
7. mark end of sub-image

Figure 10. Steps of the JPEG image compression algorithm⁹

3. JPEG

3.1. Definition of JPEG

The transmission and storage of large images has become more and more common, increasing the need for fast compression algorithms. The most widely-used compression algorithm for still images is JPEG,⁹ a lossy compression technique common on the world wide web. As shown in Fig. 10, JPEG divides images into 8x8 sub-images. For each sub-image, JPEG applies the discrete cosine transform, converting the sub-window into frequency space. It then divides the frequency values according to a user-selected normalization table (rounding the result), and encodes the scaled frequencies with a Huffman code.¹⁴ Images are uncompressed by inverting this process.

Of the seven steps in the JPEG algorithm, only two introduce noise. As discussed above, the discrete cosine transform introduces a small but measurable amount of noise, even when computed with 32 or 64 bit floating point numbers. Most of the error (and most of the compression), however, comes from the normalization table in step 3. In effect, this table allocates a fixed range of numbers for every frequency, dividing values accordingly. Users may select a compression level from 0 to 100, where compression level 100 is lossless (in terms of the normalization step) and level 0 is maximum compression. All the experiments in this paper were conducted at compression level 50.

3.2. Precision vs. Error in JPEG

When the DCT and inverse DCT are used within the context of JPEG, more error can be tolerated and therefore less precision is required. This occurs because of the normalization step of the JPEG algorithm (step 3 in Fig. 10). For compression purposes, this step allocates a fixed range of values for every frequency, in essence discarding the least significant bits on a frequency-by-frequency basis. As a result, small errors that may be significant when DCT/IDCT are considered in isolation become insignificant because of normalization within the context of JPEG.

Figure 11 presents the error curves for image (*f*) of Fig. 3 for different precisions of the DCT/IDCT terms when used in the context of JPEG (compression level 50). Suddenly, the bottom six curves are indistinguishable. This implies that floating point precision is no longer needed for the summation type. In fact four bits of precision to the

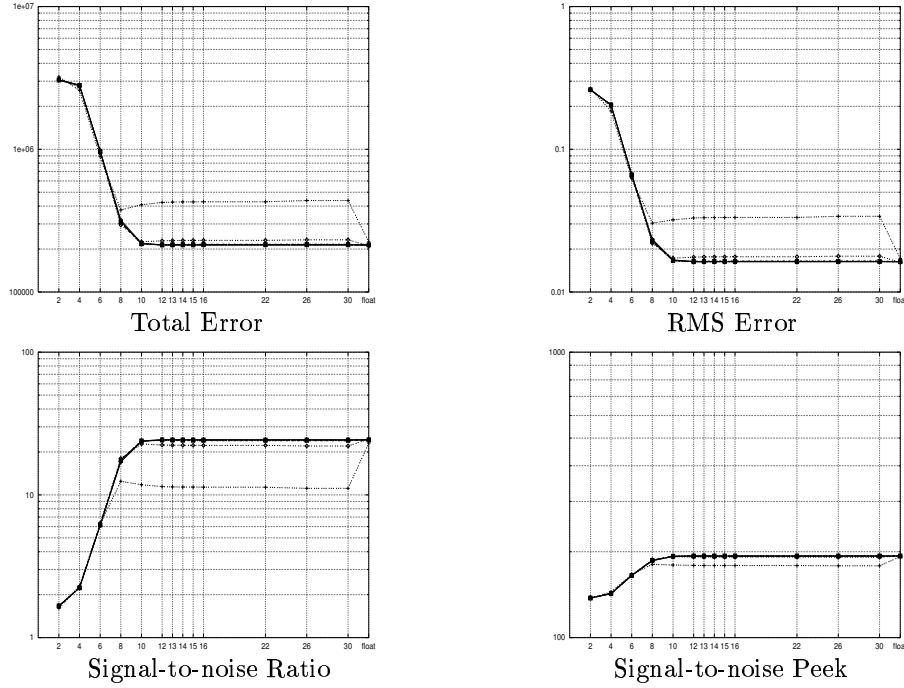


Figure 11. Result of JPEG reconstruction for image (*f*) of Fig. 3 for each of the four measures.

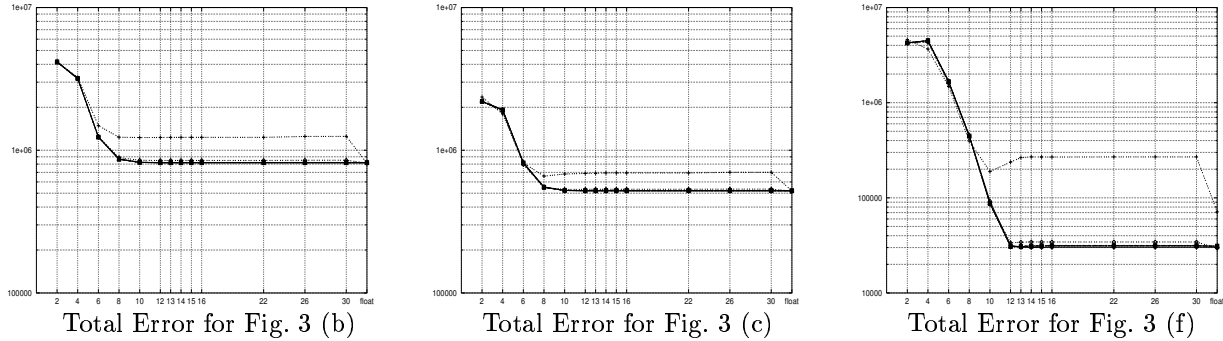


Figure 12. JPEG Total error for images (b), (c), and (f) of Fig. 3.

right of the binary point (20 bits total) is enough – the rest of the information is discarded during normalization anyway! Similarly, the multiplication type now require only 10 bits to the right of the binary points (12 total).

As shown in Fig. 12, although the absolute error varies between images, the shape of the curves and the consequences remain the same as for image (*f*) of Fig. 3. On the other hand, these results are highly dependent on the compression level being used. On one extreme, (level 100) no normalization is applied and float and fix18.16 precisions (respectively for summation and multiplication types) are required, as for the DCT in isolation. At level 50, precisions of fix20.4 and fix12.10 are sufficient (see Fig. 13 and Tab. 3), a saving of 12 and 20 bits respectively. Presumably, higher levels of compression would permit even smaller representations. A web site (www.cs.colostate.edu/cameron/SPIE99.html) presents all four error tables for all the precision combinations tested.

4. CONCLUSION

Reconfigurable processors can be more efficient than traditional machines in part because of increased computational density. Variable precision arithmetic is one mechanism for further increasing the computation density of reconfigurable processors. To test whether less precise arithmetic could be used for frequency-space computations, we

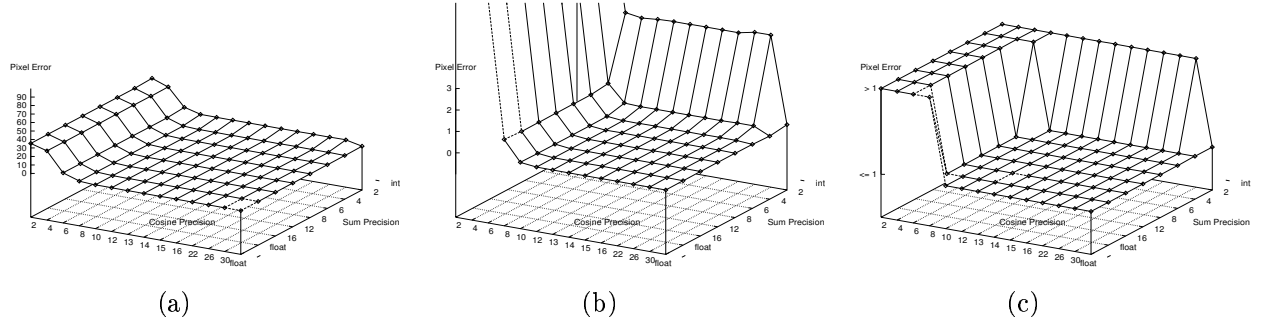


Figure 13. Average increase in pixel error for all artificial and natural images. (a) shows the full graph, (b) shows the expanded graph, and (c) shows the thresholded graph (threshold error = 1).

Table 3. Average increase in pixel error for all artificial and natural images after compression/uncompression using JPEG. Rows correspond to multiplication types and columns correspond to summation types. Values marked with † indicate that there was a decrease in error instead of an increase; these decreases are the result of random variation and suggest that the change in error as a result of reduced precision is not statistically significant.

	float	fix32.16	fix28.14	fix24.8	fix22.6	fix20.4	fix18.2	int16
float	0.0000	0.0071	0.0070	0.0058	0.0015	0.0133†	0.0589†	0.0490
fix32.30	0.0000	0.0115	0.0116	0.0125	0.0162	0.0382	0.2919	4.0913
fix28.26	0.0001	0.0115	0.0116	0.0124	0.0161	0.0380	0.2899	4.0891
fix24.22	0.0001†	0.0114	0.0116	0.0123	0.0156	0.0350	0.2636	3.9335
fix18.16	0.0002†	0.0111	0.0110	0.0119	0.0153	0.0347	0.2628	3.9331
fix17.15	0.0005†	0.0107	0.0107	0.0114	0.0151	0.0341	0.2613	3.9307
fix16.14	0.0004†	0.0100	0.0101	0.0102	0.0144	0.0336	0.2595	3.9258
fix15.13	0.0005†	0.0087	0.0087	0.0101	0.0135	0.0323	0.2564	3.9190
fix14.12	0.0021	0.0083	0.0083	0.0091	0.0123	0.0311	0.2498	3.8936
fix12.10	0.0946	0.0778	0.0778	0.0776	0.0781	0.0916	0.2866	3.8197
fix10.8	1.0245	0.9748	0.9748	0.9748	0.9711	0.9653	1.0685	3.9346
fix8.6	7.4684	7.3601	7.3601	7.3601	7.3601	7.3250	7.2531	8.7778
fix6.4	29.9439	29.8395	29.8395	29.8395	29.8395	29.8395	29.5013	29.5608
fix4.2	35.3315	35.3536	35.3536	35.3536	35.3536	35.3536	35.3536	36.3420

tested the results of various precisions on DCT and JPEG. We concluded that some savings can be applied to DCT by using 18 bit fixed point numbers for the multiplication type, but that floating point addition is still required. Within the context of JPEG (compression level 50), however, a much greater savings can be achieved by using 20 bit (summation) and 12 bit (multiplication) fixed point numbers for DCT. This should result in significantly smaller and faster circuits. Presumably, higher levels of compression would permit even smaller representations. Moreover the results are consistent over a great range of images, over different histogram and frequency spectra what indicates that the results are general.

REFERENCES

1. W. Mangione-Smith, "Seeking solutions in configurable computing," *IEEE Computer* **30**, pp. 38–43, Dec. 1997.
2. W. Mangione-Smith, "Application design for configurable computing," *Computer* **30**, pp. 115–117, Oct. 1997.
3. J. Rose, A. E. Gamal, and A. Sangiovanni-Vincentelli, "Architecture of field-programmable gate arrays," *Proceedings of the IEEE* **81**(7), pp. 1013–1029, 1993.
4. D. A. Buell, J. M. Arnold, and W. J. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*, IEEE CS Press, 1996.
5. A. DeHon and J. Wawrzynek, "The case for reconfigurable processors." 1997.
6. A. DeHon, "Comparing computing machines: Technology and applications," in *Proceedings of SPIE 3526*, Nov. 1998.
7. S. E. Umbaugh, *Computer Vision and Image Processing: a practical approach using CVPtools*, Prentice-Hall, London, 1998.
8. J. F. Blinn, "What's the deal with the dct?," *IEEE Computer Graphics and Applications* **13**, pp. 78–83, July 1993.
9. G. K. Wallace, "The jpeg still picture compression standard," *Communications of the ACM* **34**(4), pp. 30–44, 1991.
10. C. Loeffler, A. Ligtenberg, and G. S. Moschytz, "Practical fast 1-d dct algorithms with 11 multiplications," in *International Conference on Acoustics, Speech and Signal Processing*, pp. 988–992, 1989.
11. J. P. Hammes and W. Bohm, *The Sassy Language - Version 1.0*, 1999.
12. J. Hammes, B. Draper, and W. Bohm, "Sassy: A language and optimizing compiler for image processing on reconfigurable computing systems," in *Proceedings: International Conference on Vision Systems*, pp. 522–537, (Las Palmas de Gran Canaria, Spain), January 1999.
13. W. Najjar, B. A. Draper, W. Bohm, and J. R. Beveridge, "The cameron project: High-level programming of image processing applications on reconfigurable computing machines," in *Workshop on Reconfigurable Computing*, pp. 83–88, (Paris), Oct. 1998.
14. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, Addison-Wesley, Reading, MA, 1992.

CAMERON PROJECT: FINAL REPORT

Appendix M: Compiling ATR Probing Codes for Execution on FPGA Hardware

Compiling ATR Probing Codes for Execution on FPGA Hardware *

W. Böhm, R. Beveridge, B. Draper, C. Ross and M. Chawathe
Colorado State University, Fort Collins, CO, USA

W. Najjar
University of California, Riverside, CA, USA

Abstract

This paper describes the implementation of a significant Image Processing application, Probing, on a reconfigurable system, using the SA-C programming language and optimizing compiler. The factors leading to an impressive speedup compared to a C implementation of the same algorithm executing on a Pentium are analyzed.

1. Introduction

The Cameron Project has developed a compiler that maps programs written in a high level language, SA-C, onto FPGA-based reconfigurable hardware. One useful test of this technology is to compile a significant Automatic Target Recognition (ATR) algorithm to run on a commercially-available reconfigurable processor. This paper describes how a silhouette probing algorithm was written in SA-C and compiled and run on FPGA hardware. In particular, we report time and space requirements of the probing algorithm running on an Annapolis Microsystems (AMS) WildStar, with three Xilinx XCV2000E FPGAs. Execution times are compared with those achieved using a C code version of the same algorithm running on a Pentium PC.

The rest of this paper is organized as follows. Section 2 introduces SA-C and its optimizing compiler. Section 3 provides background to the prob-

ing algorithm. Section 4 discusses implementation details. Section 5 analyses the performance of the algorithm when running on an Annapolis WildStar board, and compares it to the performance on a Pentium PC. Section 6 concludes.

2 SA-C and its optimizing compiler

FPGAs are typically programmed using hardware description languages such as VHDL [8]. Application programmers are typically not trained in these hardware description languages and usually prefer a higher level, algorithmic programming language to express their applications. The Cameron Project [6] has created a high-level algorithmic language, named SA-C [5], for expressing image processing applications and compiling them to FPGAs. The most important aspect of SA-C is its treatment of for loops and their close interaction with arrays. A loop has three parts: one or more generators, a loop body and one or more return values. The generators provide parallel array access operators that are concise and easy for the compiler to analyze. In particular, window generators allow rectangular sub-arrays to be extracted from a source array. All possible sub-arrays of the specified size are produced, one per iteration. A loop can return arrays and reductions built from values that are produced in the loop iterations. SA-C has integers, unsigned integers, and fixed point numbers with user defined bitwidths for each of these. Floats are supported in the language but currently not implemented on FPGAs. A more complete introduction to the SA-C language can

*This work is supported by DARPA under US Air Force Research Laboratory contract F33615-98-C-1319.

be found in [4] and a reference manual in [5].

The SA-C compiler performs both conventional and FPGA-specific optimizations. Some of these, pertinent to the probing algorithm will be quickly reviewed here. The performance tradeoffs of various optimizations are not obvious; sometimes they can only be assessed empirically. Therefore, the SA-C compiler allows many of its optimizations to be controlled by *user pragmas* in the source code.

Full Unrolling of loops is important when generating code for FPGAs, because it spreads the iterations in code space rather than in time. By default, the SA-C compiler fully unrolls loops anytime the number of iterations through the loop can be determined at compile-time. **Array Value Propagation** searches for array references with constant indices, and replaces such references with the values of the array elements. When the value is a compile time constant, this enables constant propagation.

Common Subexpression Elimination (CSE) is a well known compiler optimization that eliminates redundancies by looking for identical subexpressions that compute the same value. This could be called “spatial CSE” since it looks for common subexpressions within a block of code. The SA-C compiler not only performs spatial CSE, but also performs **Temporal CSE**, looking for values computed in one loop iteration that were already computed in previous loop iterations. In such cases, the redundant computation can be eliminated by holding such values in a chain of registers so that they are available later and do not need to be recomputed.

A useful phenomenon often occurs with Temporal CSE: one or more columns in the left part of the window are unreferenced, making it possible to eliminate those columns. **Narrowing** the window lessens the number of shift registers (and therefore space) required to store the window elements. After temporal CSE and narrowing, the compiler may lessen the window space even further by shifting window computations as far to the right as possible and inserting shift registers to move the results back to the correct iteration. This is called window **compaction**, and is designed to save window registers. It is only effective if the registers

needed to shift the results are smaller than the shift registers in the window. In the case of the probing algorithm this is the case, since the inputs are twelve bit pixels, and the outputs are hits: one bit results of threshold operations.

Some operators, such as division, can be inefficient to implement directly in hardware, and can be replaced by a **Table Lookup**. A pragma indicates that a function or an array needs to be compiled as a lookup table. Another low level optimization, **Bitwidth Narrowing** exploits the user defined bitwidths of variables to infer the minimal bitwidths of intermediate variables.

The portions of the program that are to be executed on the reconfigurable hardware are translated, via data flow graphs, to VHDL. Commercial tools map the VHDL to configuration codes.

3 Probing Algorithm Background

The probing code presented here represents a portion of the LARS [2] algorithm developed by Alliant Techsystems in the late 1980’s for identifying ground vehicles in LADAR imagery. The probing algorithm was used as part of a larger multi-sensor ATR system developed to find vehicles in visible light, IR and LADAR imagery. This work was done at Colorado State University working with Alliant Techsystems under the DARPA Reconnaissance, Surveillance and Target Acquisition program (RSTA) [7].

A probe in this context is a pair of pixels and an associated true/false question. Typically, a probe returns true when the absolute value of the difference in pixel values exceeds a threshold. Loosely speaking, the probe may be thought of as answering the question: “Is one pixel inside an object of interest and the other outside?” A probeset is a set of probes arrayed along the silhouette of an object. When the proper probeset is placed in the correct location over an object of interest, one expects all of the probes to return true. Likewise, if the probesets are sufficiently detailed and the objects sufficiently distinct, then no other probeset will do as well. Figure 2 is an example from our RSTA project of a typical ATR problem to which probing may be applied. It is a histogram

equalized color image of an M113 Target and its associated LADAR image. Part a) shows a full color image, part b) shows the portion roughly corresponding to a LADAR field of view, part c) shows the LADAR image with the winning probe set overlaid on top of LADAR pixels. The image is “nov31100L1” from the Fort Carson Dataset. The winning probe set identifies an M113 viewed at 35 degree aspect angle and 5 degree depression angle. For this case, all 29 probes returned true.

Probesets are generated by regularly sampling target views. Three-dimensional models of targets are used in conjunction with a LADAR simulator to generate synthetic LADAR images at different aspect and depression angles. Subsequently, probes that straddle the boundary of the target and are placed in accordance with heuristics developed by the research team at Alliant Techsystems are selected to form the probeset representing that target at that view.

When a probing algorithm is applied to an image to recognize targets, each probeset must be evaluated at every position in the image. Probing is a simple and robust procedure that has proven to work well in some contexts. Later important refinements were made to the simple exhaustive application of probesets described here by Bienenstock and the Geman brothers [3], working in conjunction with Alliant Techsystems and the Army’s Night Vision Lab. This effort led to the development of the ARTM algorithm, that combined probing with a decision tree in order to reduce the total number of probe sets applied to an image. The initial success of LARS on LADAR ATR problems, and the subsequent success of ARTM on FLIR ATR problems, is largely responsible for the interest in probing as an ATR technique.

In our multi-sensor ATR effort probing of LADAR imagery is used to queue a more complex target verification process that fuses information from visible light, IR and LADAR imagery relative to the actual 3D target model. Unlike previous uses where probing was used to make a final determination of target type and viewing angle, in our application probing generates a list of possible target types and viewing angles. This relaxes considerably the demands placed upon the prob-

ing algorithm itself: it does not have to always rank the true target type and viewing angle first. This, in turn, allows us to use probing successfully in more difficult ATR contexts where it would fail on its own.

The exhaustive application of probe sets for all possible objects, all possible viewing angles, and all possible placements in an image is at first blush a tremendously burdensome computation. However, as we are about to show, it is also a computation that it is ripe for optimization. It is thus both an algorithm of considerable practical interest and a powerful demonstration of what can be done using the compilation capabilities developed in the Cameron project.

4 Probing Implementation Details

Here is a pseudo code version of the Probing algorithm.

```
for each window in image {
  best_score, probe_set_index =
    for all probe_sets {
      hit_count =
        for each probe in probe_set
          return(sum(threshold(probe)))
      score = hit_count/probe_set_size
    } return(max(score), probe_set_index)
} return( array(best_score),
          array(probe_set_index) )
```

The probing algorithm applies each probe set to each image position and returns two result images (outer loop in above code). The first and second images together indicate the highest score of a probe set in a certain position, and the probe set index identifying the winning probe set. A probe set score is calculated by adding all threshold results and dividing this by the probe set size. A threshold operation subtracts two pixel values and compares the absolute difference to a threshold value. The result is a hit: zero, if the absolute difference between the two probed pixels is below the threshold, and one otherwise.

The two inner loops computing the scores and probe set indices can be fully unrolled, because the probe sets are statically known. This turns the code into a singly nested loop driven by a window

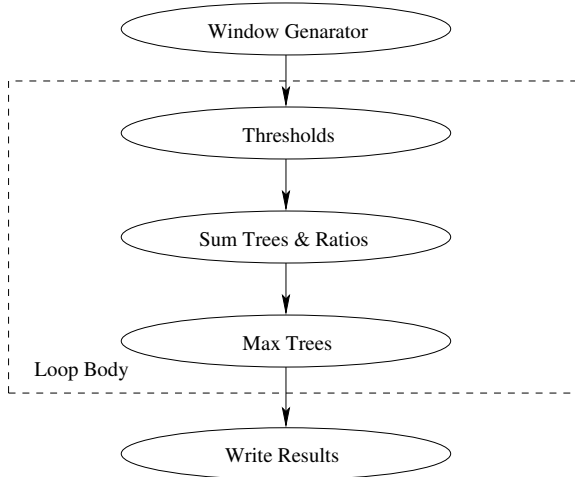


Figure 1. Program Flow

generator extracting windows in row major order from the LADAR input image. The loop body has become an expression consisting of threshold operators computing hits, sum trees adding the hits for each probe set, division operators computing the scores for each probe set, and max trees selecting the winner, see figure 1. This giant expression allows for spatial CSE, temporal CSE, and window compaction optimizations, as pictured in figure 3.

In figure 3(a) the effect of spatial CSE is demonstrated. Probes common to two (or more) probe sets are computed once and their result is shared by the sum trees computing the hit counts. If probe sets share two or more probes, then parts of the sum tree can be shared as well. Figure 3(b) shows temporal CSE. The right most of the four top probes is computed and its result is reused in three computations that are three, five and seven iterations ahead. In this example this replaces three threshold expressions by a simple one bit register chain of length seven. Window compaction is demonstrated in figure 3(c). The window is compacted from the width of the silhouette, in the example case fifteen, to the width of the horizontally widest probe, in this case four.

The computation of the score of a probe set in a window requires the hit count to be divided by the probe set size. Floating point division is a very costly operator on FPGAs, while fixed point division causes truncation errors which influence

the ordering of the scores. Since the goal of computing the scores is to find the maximum score, the division can be replaced by a table lookup that maps hit counts and probe set sizes onto rank numbers. The maximum rank number of all probe sets indicates the winner. Once a rank number is used, scores below a certain score threshold (e.g. 80%) can be given rank zero. This drastically reduces the number of bits in the rank, and therefore reduces the size of the lookup table, and the size of the comparison operators in the max trees. Because the size of every probe set is statically known, the two dimensional lookup table can be replaced by a set of one dimensional lookup tables, one for each probe set size. The division computing the score is thus replaced by a one dimensional table lookup using the hit count as lookup index.

The bit width narrowing optimization is highly effective on the sum trees computing the hit counts. The inputs to the trees are one bit unsigned integers and the resulting sums are at most six bits, as the number of probes per probe set is between 20 and 42.

5 Probing Performance and Analysis

The test suite for the probing application consists of three vehicles (an M60 tank, an M113 armored personnel carrier, and an M901 armored personnel carrier with missile launcher), each represented by 81 probe sets (27 aspect angles times three depression angles), totalling 7573 probes in windows of sizes up to 13x34. The input is a 512x1024 LADAR image of 12 bit pixels.

The reconfigurable platform used is a WildStar Board, produced by Annapolis Micro Systems [1]. The WildStar has three XCV2000E Virtex FPGAs made by Xilinx [9]. The WildStar board is capable of operating at frequencies from 25 MHz to 180 MHz. It communicates over the PCI bus with the host computer at 33 MHz. In our system, the board is housed in a 266-MHz Pentium-based PC. In this paper we compare the performance of SA-C codes running on the WildStar to the performance of C code running on an 800 MHz Pentium III. (The XCV2000E and 800 MHz PIII are of similar age, and were both fabricated at .18 microns.)

Unoptimized				Optimized		
	Pbs	Adds	Win.	Pbs	Adds	Win.
m60	2832	2751	12x34	151	1413	12x4
m113	2315	2234	11x26	106	967	11x4
m901	2426	2345	13x25	143	1196	13x4
total	7573	7330	13x34	400	3576	13x4

Table 1. DFG level statistics: Probes , Additions, Window sizes before and after optimization

In both cases, execution times do not include the time required to read the image from the disk or host into local memory, but do include I/O time between the processor and local memory.

The SA-C code is partitioned in the most straightforward way: each vehicle is mapped onto one FPGA. Each FPGA scans the input image and produces an image of winning scores and probe set indices for its particular vehicle. The host gathers the resulting data and creates a result image: an 8 bit version of the input image with the probe set of the highest scoring winner superimposed over it (see figure 2(c)).

Table 1 provides dataflow graph level statistics for the test suite. It shows the number of probes, the number of additions in the sum trees, and the window size, before and after optimization. This shows that optimization reduces the number of probes about nineteen fold and the number of additions about two fold. About 50% of these additions are one bit additions. The number of columns in the window is compacted from the width of the largest probe set (34) to the horizontal width of the widest probe (4).

Program execution time on the Wildstar is 81 milliseconds. Executing the equivalent C code on the Pentium, using the Microsoft VC++ compiler optimized for speed, takes 65 seconds. Hence the Wildstar is about 600 times faster than the Pentium.

These times can be explained as follows. For the configuration generated by the SA-C compiler for the probing algorithm, the FPGAs run at 41.1 MHz. The program is completely memory IO bound: every clock cycle each FPGA

reads one 32 bit word, containing two 12 bit pixels. As there are $(512 - 13 + 1) * (1024)$ 13 pixel columns to be read, the FPGAs perform $(512 - 13 + 1) * (1024) * (13/2) = 3,328,000$ reads. At 41.1 MHz this takes 80.8 milliseconds.

The Pentium performs $(512 - 13 + 1) * (1024 - 34 + 1)$ window accesses. Each of these window accesses involves 7573 threshold operations. Hence the inner loop that performs one threshold operation is executed $(512 - 13 + 1) * (1024 - 34 + 1) * 7573 = 3,752,421,500$ times. The innerloop body in C is:

```
for(j=0; j<sizes[i]; j++){
    diff =
        ptr[set[i][j][2]*in_width+set[i][j][3]] -
        ptr[set[i][j][0]*in_width+set[i][j][1]];
    count += (diff>THRESH || diff<-THRESH);
}
```

where in_width and THRESH are constants. The VC++ compiler infers that ALL the accesses to the array set can be done by pointer increments, and generates an inner loop body of 16 instructions. (This is, by the way, much better than the 22 instructions that the gcc compiler produces at optimization setting -O6.) The total number of instructions executed in the inner loop is therefore $3,752,421,500 * 16 = 60,038,744,000$. If one instruction were executed per cycle, this would bring the execution time to about 75 seconds. As the execution time of the whole program is 65 seconds, the Pentium (a super scalar architecture) is actually executing more than one instruction per cycle!

The factors contributing to the speed difference between the Wildstar and the Pentium can be broken down as follows.

- Analysis and optimization: the compiler has reduced the number of probes nineteen fold.
- Coarse grain parallelism: the Wildstar executes three processes, each process corresponding to a vehicle, in parallel without any interference.
- Massive fine grain parallelism: each FPGA performs all its threshold operations, hit summations, table lookups, and comparisons in parallel, whereas the Pentium performs slightly more than one instruction per cycle.

- Clock frequency: The Pentium runs at a 40 times higher clock rate than the FPGAs.

It can be argued that an optimizing and parallelizing compiler for a parallel von Neumann machine could achieve the same improvements in terms of the first two factors: analysis and optimization, and coarse grain parallelism. However, the largest factor, the fine grain parallelism, is a defining FPGA characteristic. Taking the first two factors out of the equation still gives the FPGAs a factor of 10 speedup over the Pentium.

6 Conclusion and Future Work

In this paper we have introduced the optimizing SA-C compiler and have studied a complex, relevant, image processing application, Probing, its implementation in SA-C, its mapping to reconfigurable hardware, and its performance in comparison to a C version of the program executing on a Pentium. We have analysed the factors comprising the difference in FPGA and Pentium performance. We have shown that the FPGAs show a considerable speedup as compared to the Pentium, even when programmed in a high level language.

In future work we will compare the performance of SA-C generated FPGA code to hand coded VHDL implementations of relevant image applications. Also, high level language approaches to FPGA debugging will be considered.

References

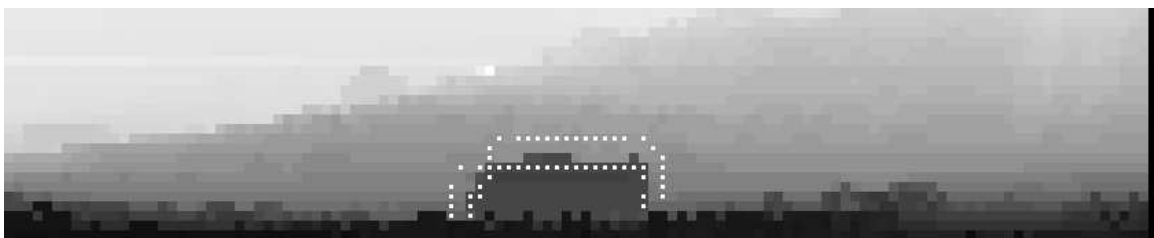
- [1] Annapolis Micro Systems, Inc., Annapolis, MD. *STARFIRE Reference Manual*, 1999. www.annapmicro.com.
- [2] J. E. Bevington. Laser Radar ATR Algorithms: Phase III Final Report. Technical report, Alliant Techsystems, Inc., May 1992.
- [3] E. Bienstock, D. Geman, S. Geman, and D. E. McClure. Development of laser radar atr algorithms: Phase ii - military objects. Technical report, Mathematical Technologies Inc., Providence, Rhode Island, October 1990. Prepared under Harry Diamond Laboratories Contract No. DAAL02-89-C-0081.
- [4] W. Böhm, B. Draper, W. Najjar, J. Hammes, R. Rinker, M. Chawathe, and C. Ross. One-step compilation of image processing applications to FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '01*, May 2001.
- [5] J. Hammes and W. Böhm. *The SA-C Language - Version 1.0*, 1999. www.cs.colostate.edu/cameron.
- [6] J. Hammes, R. Rinker, W. Böhm, and W. Najjar. Cameron: High level language compilation for reconfigurable systems. In *PACT'99*, Oct. 1999.
- [7] J. Ross Beveridge, Bruce Draper, Mark R. Stevens, Kris Siejko and Allen Hanson. A Coregistration Approach to Multisensor Target Recognition with Extensions to Exploit Digital Elevation Map Data. In O. Firschein, editor, *Reconnaissance, Surveillance, and Target Acquisition for the Unmanned Ground Vehicle*, pages 231 – 265. Morgan Kaufmann, 1997.
- [8] D. Perry. *VHDL*. McGraw-Hill, 1993.
- [9] Xilinx, Inc. *Virtex 2.5V Field programmable Gate Arrays: Preliminary Product Description*, Oct. 1999. www.xilinx.com.



Full “color” image (a)



Portion proportional to LADAR view (b)



Winning probe set (c)

Figure 2. Probing example

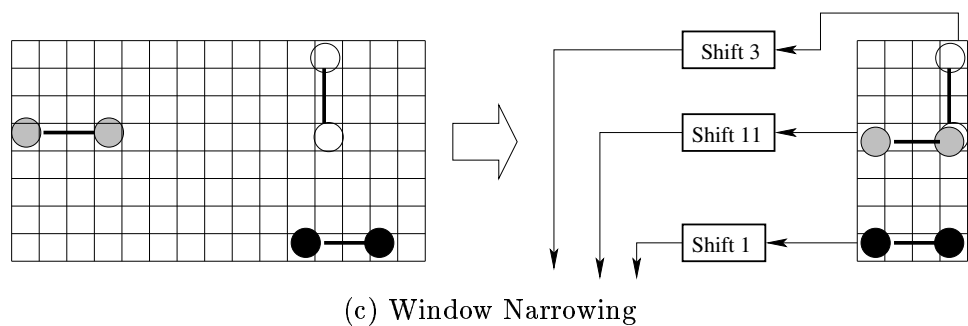
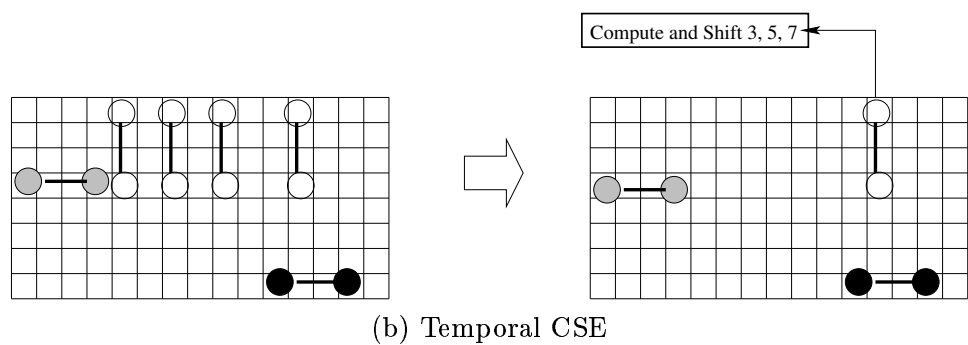
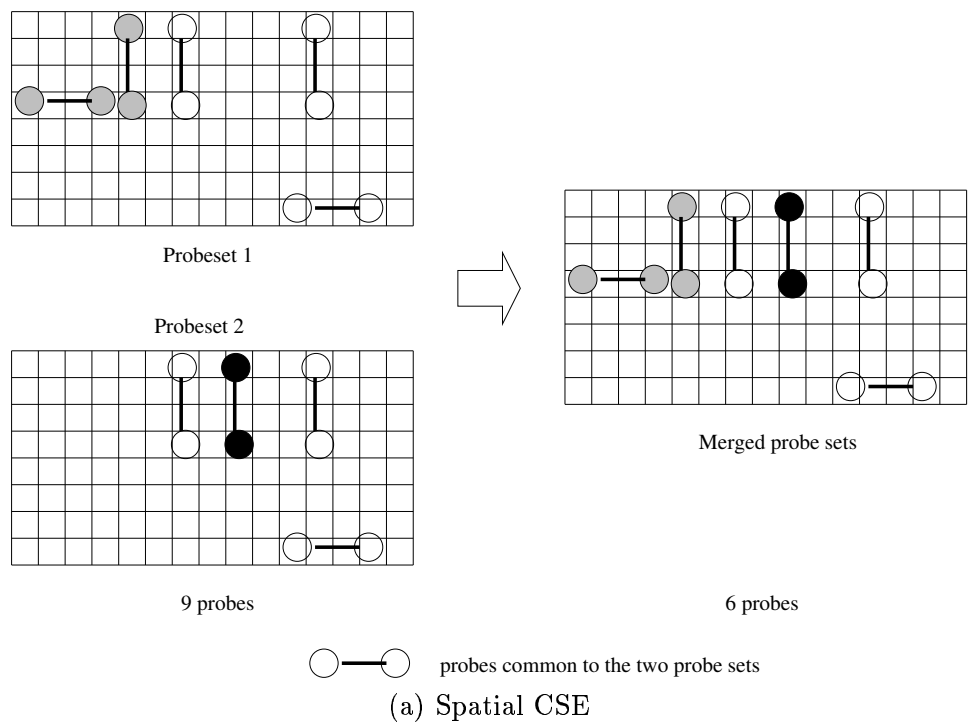


Figure 3. Optimizations as applied to Probing

CAMERON PROJECT: FINAL REPORT

Appendix N: University of California Riverside Subcontract



CAMERON PROJECT FINAL REPORT

*Subcontract to the
University of California Riverside*

Walid A. Najjar

January 2002

Introduction

The research effort of the Cameron Project was expanded from Colorado State University to the University of California Riverside when Prof. Walid Najjar accepted a faculty position in the Department of Computer Science and Engineering at UCR in July 2000.

The research effort at UCR has been done in very close cooperation with the whole Cameron team at CSU. It has continued some of the work that had already been initiated at CSU and started some new avenues of research. The main areas of research activity in the Cameron Project at UCR are:

1. Compile-time area estimation for FPGAs.
2. Compilation of SA-C programs to the UCI MorphoSys architecture.
3. Board-to-board transfer of video images using the Annapolis Microsystems WSDP I/O cards.

The first two areas have been supported jointly by the CSU Cameron sub-contract (\$137,357 from 11/1/2000 to 3/31/2002) and by a joint NSF Grant with UCI (PI: F. Kurdahi, Co-PI: W. Najjar and N. Bagherzadeh) Award No. 0083080 "ITR: Synthesis of Adaptive Mission-Specific Processors" (\$499,996 from 09-15-00 to 08-31-03, UCR sub-contract \$250,000). The third area has been supported exclusively by the CSU sub-contract.

Compile-Time Area Estimation

The SA-C compiler performs extensive optimizations whose objective is to improve the overall execution time. Most of these optimizations have a very serious impact on the area that would be utilized by the circuit being generated. For example, loop unrolling increases the required area while bit-width narrowing and loop fusion tend to reduce it. The total area on the FPGA, however, is finite and therefore the compiler needs to get some feedback as to how much area is being used. An estimate of the area is usually provided by the synthesis tool. However, the run-time of the synthesis tool for an average program is of the order of minutes which is much too long for a compiler. Furthermore, using the synthesis tool assumes that the compiler has completed its code generation of VHDL, whereas we would like the estimation to be coupled with the optimization stage.

As part of the Cameron Project, we have developed at UCR a technique that can provide a compile-time estimate of the area used by a program. The estimation is based on the parameterized analysis of area utilized on the FPGA by each type of nodes in the dataflow graph. The average error obtained by the estimation tool is less than 5% for large programs and under 3% for small image processing operators. The estimator tool is integrated with the SA-C compiler.

Compiling SA-C Programs for the MorphoSys Architecture

The rapid growth of silicon densities has made it feasible to deploy reconfigurable hardware as a highly parallel computing platform. However, one of the obstacles to its wider acceptance is their programmability. The application needs to be programmed in hardware description languages or an assembly equivalent, whereas most application programmers are used to the algorithmic programming paradigm. SA-C has been proposed as an

expression-oriented language designed to implicitly express data parallel operations. The Morphosys project proposes a computer architecture, which consists of reconfigurable hardware that supports a data-parallel, SIMD computational model. This paper describes a compiler framework to analyze SA-C programs, perform optimizations, and map the application onto the Morphosys architecture. The mapping process is static and it involves operation scheduling, processor allocation and binding and register allocation in the context of the Morphosys architecture. The compiler also handles issues concerning data streaming and caching in order to minimize data transfer overhead. We have compiled some important image-processing kernels, and the generated schedules reflect an average speed-up (in execution times) of up to 6x compared to the execution on 800 MHz Pentium III machines.

Board-to-Board Video Streaming

The objective of this project is to simulate the input of a video camera into the FPGA of a reconfigurable board (we are using the Annapolis MicroSystems WildStar board) by streaming video images directly to the board from a file. The main advantage of this technique is that allows the exact same image streams to be repeated which is extremely valuable in debugging a system. Our experimental setup is as follows:

1. A WildStar board (with three FPGAs Virtex E 2000 and 20MB SRAM) is hosted in PC #1. A WSDP I/O Card (with a Virtex E 600 and 4MB SRAM) is attached to the WildStar board as a daughter card.
2. A PCI Carrier card is plugged in PC #2 and has the same identical daughter card attached to it.
3. A 24" Mictor cable connects the two WSDP cards.

The main idea is to transfer a video file from the disk on PC #2 to the FPGAs on the WildStar board via the two daughter cards.

The main problem so far has been that the drivers for the AMS WSDP boards are written for Microsoft Windows OS and we are using Linux OS for the Cameron Project as a whole. We have been collaborating with the staff at AMS to fix that but so far we have been unsuccessful.

A Compiler Framework for Mapping Applications to a Coarse-grained Reconfigurable Computer Architecture

Girish Venkataramani, Walid Najjar
University of California Riverside
{girish, najjar}@cs.ucr.edu

Fadi Kurdahi, Nader Bagherzadeh
University of California Irvine
{kurdahi, nader}@ece.uci.edu

Wim Bohm, Jeff Hammes
Colorado State University
bohwm@cs.colostate.edu

ABSTRACT

The rapid growth of silicon densities has made it feasible to deploy reconfigurable hardware as a highly parallel computing platform. However, one of the obstacles to its wider acceptance is their programmability. The application needs to be programmed in hardware description languages or an assembly equivalent, whereas most application programmers are used to the algorithmic programming paradigm. SA-C has been proposed as an expression-oriented language designed to implicitly express data parallel operations. The Morphosys project proposes a computer architecture, which consists of reconfigurable hardware that supports a data-parallel, SIMD computational model. This paper describes a compiler framework to analyze SA-C programs, perform optimizations, and map the application onto the Morphosys architecture. The mapping process is static and it involves operation scheduling, processor allocation and binding and register allocation in the context of the Morphosys architecture. The compiler also handles issues concerning data streaming and caching in order to minimize data transfer overhead. We have compiled some important image-processing kernels, and the generated schedules reflect an average speed-up (in execution times) of up to 6x compared to the execution on 800 MHz Pentium III machines.

1. Introduction

Computer architecture and microprocessor growth has resulted in rapid increase in circuit densities and speed of VLSI systems. Some of the CPU-intensive applications that were previously feasible only on supercomputers, have now entered the realm of workstations and PCs. The viability of reconfigurable hardware has also increased in the past decade due to such improvements in VLSI technology. The systems rely on the dynamic mapping of a program segment directly onto the hardware in order to take advantage of the inherent data parallelism in the program. A common reconfigurable platform deployed widely today is Field Programmable Gate Array (FPGA). DeHon [1][27] shows that the computational density of FPGAs is much greater than that of traditional processors. Consequently, the performance achieved by FPGA-based reconfigurable architectures is a few orders of magnitude greater than that of processor-based alternatives for some applications.

One of the application domains that can realize the advantages of reconfigurable computing systems is image processing. A typical image-processing application is characterized by inherent data parallelism, regular data structures with regular access patterns. Such applications are very well suited for execution on reconfigurable hardware.

However, fine-grained reconfigurable platforms like FPGAs have a number of inherent disadvantages:

- **Ease of programmability:** In most cases, reconfigurable computing systems still require the manual translation of program into a circuit using a hardware description language (HDL). This process is a significant hindrance to the wider acceptance of this technology by application program developers, since most application programmers are used to expressing the application at the algorithmic level in some high-level programming language.
- **Logic Granularity:** FPGAs are designed for logic replacement. Consequently, for applications where the data path is coarse-grained (8 bits or more), the performance on FPGAs is inefficient.
- **Compilation and Reconfiguration Time:** Applications meant to execute on FPGAs are, typically, written in a hardware description language like VHDL or Verilog. Mapping such code onto FPGAs has to go through a number of compilation passes which include logic synthesis, technology mapping, and placing and routing on the target FPGA. This process takes a few hours to days for some applications.

Many coarse-grained reconfigurable systems [5], [17], [18], [19], [20], [21] have been proposed as an alternative between FPGA-based systems and fixed logic CPUs. The reconfigurable computing element in such systems is typically a specialized hardware that is (usually) deployed as a co-processor.

The reconfigurable computing paradigm poses some new challenges and difficulties. Hence, conventional compiler techniques may not apply directly to this situation. The

research work in this thesis addresses the problem of compiling a program written in a high-level language SA-C to the Morphosys architecture, consisting of a general-purpose processor core and an array of reconfigurable processing elements.

The remainder of this chapter presents some background for this work, including a brief overview of reconfigurable computing architectures, the SA-C language, the Morphosys architecture, and some previous relevant research. The next chapter presents a brief description of the approach to compilation adopted in this work. The third chapter describes the entire mapping process in detail. The fourth chapter presents the performance evaluation of the compiler-generated code. Finally, the last chapter presents the conclusions and possible future directions.

1.1 Reconfigurable Computing Systems

The main idea behind reconfigurable computing is to avoid the “von Neumann bottleneck” (the limited bandwidth between processor and memory) by mapping computation directly into hardware. Such a system also has the ability to dynamically change the hardware logic that it implements. Hence, an application can be temporally partitioned for execution on the hardware. After one partition completes its execution the hardware is reconfigured to execute the next partition. Thus, system designers can execute more hardware than they have gates to fit. Reconfigurable computing systems represent an intermediate approach between Application Specific Integrated Circuits (ASICs) and general-purpose processors.

The most common way to deploy reconfigurable computing systems is to combine a reconfigurable hardware-processing unit with a software programmable processor. Reconfigurable processors have been widely associated with Field Programmable Gate Array (FPGA)-based system designs. An FPGA consists of a matrix of programmable logic cells with a grid of interconnect lines running between them. In addition, there are I/O pins on the perimeter that provide an interface between the FPGA and the interconnect lines and the chip’s external pins.

However, reconfigurable hardware is not limited to FPGAs. Several projects have investigated and successfully built systems where the reconfiguration is coarse-grained and is performed within a processor or amongst processors. In such cases the reconfigurable unit is a specialized hardware architecture that supports dynamic logic reconfiguration.

1.2 The SA-C Language

SA-C [9], [10], [11], [13], [12], [14], [15], [16] is a language with functional features, and is a single assignment variant of the C programming language. It has been designed with the following objectives:

- Easy expression of image processing applications with a high degree of abstraction

- Efficient compilation to hardware: The language constructs are such that they expose the inherent parallelism in the program

The main features of SA-C can be summarized as follows:

- It is an expression-oriented language. Hence, every construct in the language will have to return a value. Alternatively, every statement in the language is an assignment statement.
- Its data types support variable bit-width precision for integer and fixed-point numbers.
- The arrays in SA-C are true multi-dimensional arrays. Hence, any plane, slice, row, column, window or element of the array can be accessed directly.
- The language is based on single-assignment. This means that variables cannot be re-assigned. This feature makes it possible for the compiler to perform extensive optimizations when mapping algorithms to hardware.
- There are no pointers in SA-C. However, since the language supports such flexible array accessing patterns, other features of the language have satisfied the usefulness of pointers. A number of standard image processing applications, libraries and benchmarks have been written in SA-C in spite of this restriction.
- SA-C does not support recursion. This feature ensures that the algorithms can be more easily mapped to hardware since there is no concept of “recursive hardware”. Moreover, almost all recursive functions can be re-written as iterative functions with loops.
- Multiple-value returns and assignments. In addition to being expression-oriented, the language allows multiple values to be returned from an expression/statement.
- Image Processing Reduction Operators. The language supports a number of useful image processing reduction operators (like histogram, median etc) that can be applied to loops and arrays.
- Loops in SA-C are the most commonly used constructs. They are very special in that they tend to inherently specify how the loop may traverse a particular array and what kind of parallel operations it performs. The following section describes SA-C loops in a little more detail.

A compiler for SA-C has already been developed that maps applications to multi-FPGA based systems. The SA-C compiler supports a wide range of optimizations aimed at producing an efficient hardware execution model. This is achieved by re-using previous computations, eliminating unnecessary computations, reducing the storage area required on the FPGA, reducing the number of reconfigurations, exploiting the locality of data and therefore reducing the required data bandwidth from the host to the FPGA and finally improving the clock rate of the circuit. These include traditional optimizations such as constant folding, operator strength reduction, dead code elimination, invariant code

motion and common sub-expression elimination. Other optimizations have been either developed or adapted from vectorizing and parallelizing compilers as well as synthesis tools. These include bit-width narrowing, loop unrolling, stripmining and loop fusion.

1.2.1. SA-C Loops

Every loop in SA-C has three components to it – the loop generator, the loop body and the loop collector. A loop generator specifies what values are generated in each iteration, and how many iterations the loop will perform. The loop collector generates a return value for the loop expression, by combining, in various ways, values that are produced within the loop.

There are 2 main types of loop generators – *array-element* and *window* generators. An *element generator* produces a scalar value from the source array per iteration. A *window generator* produces a sub-array (of specified size) of the same dimensions as the source image per iteration. Fig. 1.1 shows how a window generator in SA-C works. The example shows windows of size 3x3 being generated from a source array (*Image*) of size 4x5. Shaded portions of Fig 1.1(b) represent the different windows generated in different iterations (the figure does not show all the iteration windows).

for window w[3,3] in Image

(a)

I ₁₁	I ₁₂	I ₁₃	I ₁₄	I ₁₅	I ₁₁	I ₁₂	I ₁₃	I ₁₄	I ₁₅	I ₁₁	I ₁₂	I ₁₃	I ₁₄	I ₁₅	I ₁₁	I ₁₂	I ₁₃	I ₁₄	I ₁₅
I ₂₁	I ₂₂	I ₂₃	I ₂₄	I ₂₅	I ₂₁	I ₂₂	I ₂₃		I ₂₅	I ₂₁	I ₂₂	I ₂₃	I ₂₄	I ₂₅		I ₂₁	I ₂₂	I ₂₃	I ₂₄
I ₃₁	I ₃₂	I ₃₃	I ₃₄	I ₃₅	I ₃₁	I ₃₂	I ₃₃	I ₃₄	I ₃₅	I ₃₁	I ₃₂	I ₃₃	I ₃₄	I ₃₅		I ₃₁	I ₃₂	I ₃₃	I ₃₄
I ₄₁	I ₄₂	I ₄₃	I ₄₄	I ₄₅	I ₄₁	I ₄₂	I ₄₃	I ₄₄	I ₄₅	I ₄₁	I ₄₂	I ₄₃	I ₄₄	I ₄₅		I ₄₁	I ₄₂	I ₄₃	I ₄₄

(b)

Fig 1.1: (a) Syntax of a SA-C loop generating 3x3 windows from the source array, *Image*
(b) The shaded portion of *Image* represents the value of *w* in various iterations

Every loop in SA-C must return one or more values that are described by the loop's collectors. There are two kinds of collectors – *ConstructArrayCollector* and *ReductionCollector*. A *ConstructArrayCollector* returns an array whose elements correspond to results from each iteration of the loop execution. A *ReductionCollector* applies some sort of an arithmetic reduction operation (like sum, product etc.) on the range of values produced from each iteration, and returns the reduced value.

1.3 The Morphosys Architecture

Morphosys [2], [3], [4], [5], [6], [7] is a model for reconfigurable computing systems that is targeted at applications with inherent data parallelism, high regularity and high

throughput information. Most applications that fit this profile fall under the domain of image processing.

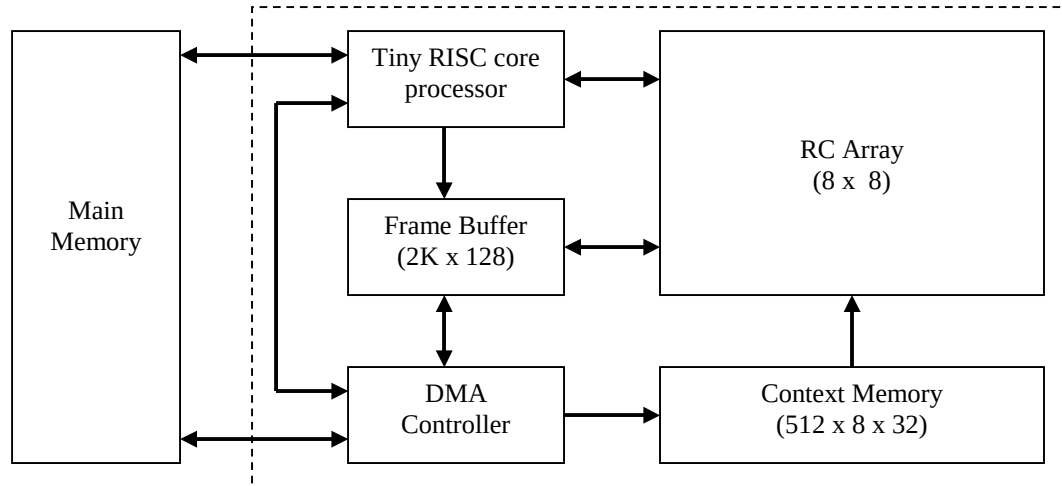


Fig 1.2: The Morphosys Architecture

The Morphosys architecture consists of five main components – the Tiny RISC core processor, the Reconfigurable Cell Array (RC Array), the context memory, the frame buffer and the DMA controller.

1.3.1. Tiny RISC

Tiny RISC is a MIPS-like core processor with a 4-stage pipeline. It has 16 32-bit registers and three functional units – a 32-bit ALU, a 32-bit shift unit and a memory unit. An on-chip data cache memory minimizes accesses to external memory. The Tiny RISC processor handles general-purpose operations and controls the execution of the RC Array through special instructions added to its ISA [6]. Through DMA instructions, it also initiates all data transfers to or from the frame buffer and the loading of configuration program into the Context Memory. RC Array instructions specify one of the internally stored configuration programs and how it is broadcast to the RC Array. The Tiny RISC processor is not intended to be used as a stand-alone, general-purpose processor. Although Tiny RISC performs sequential tasks of the application, performance is mainly determined from data-parallel processing in the RC Array.

1.3.2. RC Array

The RC Array consists of an 8x8 matrix of processing elements called the reconfigurable cells. Each RC cell consists of an ALU-Multiplier, a shift unit, input multiplexers, and the context register. In addition to standard arithmetic and logical operations, the ALU-Multiplier can perform a multiply-accumulate operation in a single cycle. The input multiplexers select from one of several inputs for the ALU-Multiplier:

- (1) One of the four nearest neighbors in the RC Array,
- (2) Other RCs in the same row/column within the same RC Array quadrant,
- (3) The operand data bus, or
- (4) The internal register file.

The context register provides control signals for the RC components through the context word. The bits of the context word directly control the input multiplexers, the ALU/Multiplier and the shift unit. The context word determines the destination of a result, which can be a register in the register file and/or the express lane buses. The context word also has a field for an immediate operand value.

1.3.3. Context Memory

The Context Memory stores the configuration program (the contexts) for the RC Array. It is logically organized into two partitions, called Context Block 0 and Context Block 1. By its turn, each Context Block is logically subdivided into eight partitions, called Context Set 0 to Context Set 7. Finally, each Context Set has a depth of 512 context words. A context plane comprises the context words at the same depth across the Context Block. As the Context Sets are 512 words deep, this means that up to 512 context planes can be simultaneously resident in each of the two Context Blocks. A context plane is selected for execution by the Tiny RISC core processor, using the RC Array instructions.

Context words are broadcast to the RC Array on a row/column basis. Context words from Context Block 0 are broadcast along the rows, while context words from Context Block 1 are broadcast along the columns. Within Context Block 0 (1), Context Set n is associated with row (column) n , $0 \leq n \leq 7$, of the RC Array. Context words from a Context Set are sent to all RCs in the corresponding row (column). All RCs in a row (column) receive the same context word and therefore perform the same operation. It is also possible to selectively enable a single context set (and therefore, row/column) to be active in any given clock cycle. This demonstrates the SIMD/MIMD hybrid model of the RC Array. It supports SIMD-style execution within each row/column, while different rows (columns) can execute different operations.

1.3.4. Frame Buffer

This is a streaming buffer and is part of a high-speed memory interface. It has two sets and two banks. It enables streamlined data transfers between the RC Array and main memory, by overlapping computation with data load and store, alternately using the two sets.

The frame buffer is an internal data memory logically organized into two sets, called Set 0 and Set 1. Each set is further subdivided into two banks, Bank A and Bank B. A 128-bit operand bus carries data operands from the Frame Buffer to the RC Array. This bus is connected to the RC Array columns, allowing eight 16-bit operands to be loaded into the eight cells of an RC Array row/column (i.e., one operand for each cell) in a single cycle. Therefore, the whole RC Array can be loaded in eight cycles.

The operand bus has a single configuration mode, called interleaved mode. In this mode the operand bus carries data from the Frame Buffer banks in the order A0, B0, A1, B1, ..., A7, B7, where A_n and B_n denote the n th byte from Bank A and Bank B, respectively. Each cell in an RC Array column receives two bytes of data, one from Bank A, and the

other from Bank B. Results from the RC Array are written back to the Frame Buffer through a special “*result bus*”.

1.3.5. DMA Controller

The DMA controller performs data transfers between the Frame Buffer and the main memory. It is also responsible for loading contexts into the Context Memory. The Tiny RISC core processor uses DMA instructions to specify the necessary data/context transfer parameters for the DMA controller.

1.4 Motivation and Problem Definition

The SA-C language has been designed specifically for easy expression of image processing applications, and for efficient, automatic translation of high-level programs to reconfigurable computing systems. However, the current SA-C compiler addresses FPGAs as the reconfigurable computing element. Morphosys has been developed as a reconfigurable computing architecture targeted at image processing applications as well. Currently, all applications must be hand-coded in assembly and loaded into the processor.

This research work aims at alleviating the ease of programmability of the Morphosys architecture by allowing automatic mapping of SA-C programs onto the architecture. In the process, the compiler must be able to analyze the source SA-C program, partition the program for execution on the Tiny RISC processor core and the RC Array, and generate a near-optimal assembly-level schedule for execution on the Morphosys architecture, while taking advantages of the features of the architecture.

1.5 Previous work

Computing systems with reconfigurable architectures can be classified by the kind of reconfigurable computing fabric that they use. Field Programmable Gate Arrays (FPGAs) have been widely used as the reconfigurable fabric in many research efforts [11], [24], [28], [29], [30], [31]. There has also been significant work in the area of non-FPGA based reconfigurable computing architectures. In the latter case, the reconfigurable computing element is some specially designed form of configurable computing hardware [5], [17], [18], [19], [20], [21].

1.5.1. Compiling to FPGA-based systems

Many research efforts have focused on automatically trying to map algorithms that are written in a high-level language to FPGAs. Some of these works have focused on defining a new language that can be more easily mapped to FPGAs, while still maintaining the level of abstraction that is important to algorithm writers. In most of the work in this field, the main focus of the compiler is to partition the high-level algorithms temporally, and then, spatially, in order to fit them on the FPGAs for execution.

In [28], [32], the approach is to generalize the reconfigurable elements of the architecture. The base platform is a multi-FPGA system (Annapolis Microsystems Wildforce platform). The resources available in this base architecture can be parameterized through an architectural description file. The system consists of a host processor and the

reconfigurable system as specified above. The source language is Graph Description Language (GDL). This language is closer to the data flow graph representation of the program and is intended to be an intermediate form as opposed to a human interface. Given the source (GDL) program and the resource constraints as input, the compiler temporally partitions the program to satisfy resource constraints. Spatial partitioning will then map the partitions produced by temporal partitioning onto the multiple FPGAs. Simulated annealing algorithms are used to select the most desirable partitions.

Streams-C [33] is a language that is a restricted version of C. The framework proposes a compilation system, based on the SUIF compiler infrastructure, for automated mapping of algorithms onto FPGAs. There is particular emphasis on extensions that facilitate the expression of communication between parallel processes.

In Cameron [11], [12], [10], a program written in SA-C (a high-level language) is compiled directly to a multi-FPGA system (like AMS Wildstar). The approach is to convert the program into an equivalent dataflow graph. Compiler optimizations like loop unrolling, loop strip-mining, and loop-carried common sub-expression reuse are applied in addition to other traditional compiler optimizations like copy propagation, common sub-expression elimination. Frequently executed codes (mostly loops) are identified as the partitions that will be mapped onto the FPGAs. Finally, the program is automatically converted to VHDL code that can be mapped onto the FPGA system.

In [29], the approach is to leverage parallelizing compiler technology based on the Stanford SUIF compiler. The architecture is made of a general-purpose processor (GPP) core and several configurable computing units (CCUs). The source program can be either C or MATLAB. The compiler identifies those portions of the program that can be executed on the CCUs and partitions the program accordingly based on resource and timing requirements. The code that can be executed on the CCUs is usually identified as parallelizing loops, and vector-style SIMD computations. Each partition is then scheduled to execute on the CCUs and control for the partition is usually a finite state machine (FSM) that executes on the GPP.

A lot of work has also been applied in solving particular problems involved in mapping algorithms to FPGAs. These efforts focus on a specific problem in the mapping process and propose heuristics and algorithms that attempt to optimally solve these problems.

The NIMBLE [24] compiler is a framework for compiling C codes to VHDL targeted at FPGAs. The work addresses the problem of temporal partitioning of applications intended to run on FPGAs as a hardware-software partitioning problem. The NIMBLE compiler is designed to preprocess the application to extract candidate loops (kernels) that can be scheduled to execute on the FPGAs. In addition to preprocessing, profiling the kernels is necessary to determine the optimizations that are best suited for the targeted hardware. The work proposes a heuristic algorithm to select from the candidate kernels; those that will execute on the FPGAs and those that will execute on the general-purpose host CPU, such that the execution time for the whole application is minimized.

In [30], [34], the work addresses the problem of mapping loop constructs to a generic reconfigurable architecture. In particular, the approach aims at minimizing reconfiguration overhead by optimally scheduling the reconfigurations. The loop is represented as a kernel with a set of operations each of which is associated with a configuration cost. The work is aimed at coming up with an optimal solution that searches the solution space in polynomial time using dynamic programming.

On a related front, [31] proposes a model to perform near-optimal temporal partitioning of a given application that is intended to execute on a multi-FPGA system like the AMS Wildforce board. The application is specified as a task graph, which is really a dataflow graph whose nodes represent tasks/operations with given execution latencies. The target reconfigurable hardware is parameterized in terms of resource constraints and reconfiguration costs. Integer Linear Programming Model is used to find near-optimal (in terms of execution time) temporal partitions of the application that can be mapped to the target hardware.

1.5.2. Compiling to Non-FPGA based Systems

Most non-FPGA systems, are based on a special configurable computing hardware component that is generally attached as a co-processor to a general-purpose core processor. Compiling to these kind of systems is a not as generic a problem as compiling to FPGA-based systems. This is because FPGAs have almost become standard hardware elements. In non-FPGA based systems, the reconfigurable computing element would pose special problems that are specific to the concerned hardware. Hence, the compilation approach needs to be closely related to the kind of reconfigurable architecture.

The Garp [19], [35] architecture consists of a general-purpose architecture and a reconfigurable array of computing elements (common logic blocks or CLBs), and is designed to function as a general-purpose architecture. This approach draws heavily from compiling techniques for VLIW processors. The compiler aims at exploiting fine-grained parallelism in applications by scheduling frequently executed instruction sequences (the trace-scheduling technique from VLIW compilers) for execution on the array. The source program is converted to an equivalent data flow graph, which is then partitioned into modules and hyperblocks, which are a group of basic blocks that expose ILP. This data flow graph is further optimized and is implemented as a fully spatial network of modules in the array; hence, every operation gets its own hardware. Further compiler analysis is performed which can then pipeline loops and add pipeline registers where necessary.

CHIMAERA [21], [36] is a RISC processor with a reconfigurable functional unit (RFU). The compiler recognizes frequently executed sequences of instructions that can be performed on the RFU, and creates new operations (RFUOPs) based on them. To do this, three important compiler optimizations are performed – control localization (to remove branches), SIMD within a register (to maximize parallelism by identifying loop bodies and optimizing the data access within the loop) and finally, frequently executed basic

blocks are transformed into an RFUOP. The SIMD-within-a-register optimization is interesting because it partitions a register into fields that perform the same operation but on different data. However, this approach has a lot of limitations because of restricted bit-width and the kind of operations that it can perform.

PipeRench [20] is an interconnection network of configurable logic and storage elements. The PipeRench compiler introduces the idea of pipelined reconfiguration where the application's virtual pipe stages are first analyzed and then optimally mapped to the architecture's physical pipe stages to maximize execution throughput. The source language, Dataflow Intermediate Language (DIL), is characterized by the single-assignment paradigm and configurable bit-widths. The compiler flattens the application's dataflow graph and then uses a greedy place-and-route algorithm (that runs in polynomial time) to map the application onto the reconfigurable fabric.

The RAW microarchitecture [18], [37] is a set of interconnected tiles, each of which contains its own program and data memories, ALUs, registers, configurable logic and a programmable switch that can support both static and dynamic routing. The tiles are connected with programmable, tightly integrated interconnects. The proposed compiler is meant to partition program execution into multiple, coarse-grained parallel regions. Each parallel region may execute on a collection of tiles. The size and the number of these regions are determined by compiler analyses that take into account the resource restrictions. Then, static schedules are generated for each such execution thread. These schedules are designed to exploit fine-grained parallelism and minimize communication latencies. The compiler is implemented using the Stanford SUIF compiler infrastructure.

The RaPiD architecture [17], [38] is a field-programmable architecture that allows pipelined computational structures to be created from a linear array of ALUs, registers and memories. These are interconnected and controlled using a combination of static and dynamic control. RaPiD-C is proposed as a programming language to specify the application that is to be executed on the RaPiD architecture. The language, however, requires the programmer to explicitly specify the parallelism, data movement and partitioning. Hence, partitioning is inherent in the language itself – outer loops specify time and inner loops specify space. It turns out that an application written in RaPiD-C is very close to a structural, hardware description of the algorithm. Hence, compiling a RaPiD-C program essentially involves mapping this RaPiD-C description onto a complete structural description consisting entirely of components in the target architecture.

The Morphosys group has done some work [8] in the area of synthesizing algorithms (in the form of data flow graphs) to reconfigurable hardware. Like [26], the work is aimed at addressing the problem of scheduling the kernels within an algorithm in such way as to minimize reconfiguration overhead and maximize data reuse. A typical complex application is composed of a sequence of kernels. Each kernel is characterized by the amount of configurations it needs and the data it executes over. Almost every kernel has

its own unique configuration (context) data. However, the (input) data it works on can be common to more than one kernel - hence, a lot of opportunity for data reuse. Also, some kernels will have to be executed after others because of data dependences. Given an application with the above characteristics, an exploration algorithm is proposed to search the space (composed of different execution orders (or schedules) of the kernels) in order to produce the best linear schedule (of kernels to be executed) in terms of minimum execution time. The approach aims at minimizing context re-loading, and maximizing data re-use while keeping execution time low.

1.5.3. Other Work

Kennedy et. al. [26] have focused on automatic translation of Fortran programs to Fortran 8x programs that are meant to be executed on vector computers like Cray-1. Fortran 8x allows the programmer to explicitly specify vector and array operations. Although their work is similar to our work with respect to exploiting implicit SIMD parallelism, the architecture of vector computers is very different from the Morphosys architecture. In particular, the reconfigurable element of Morphosys is an array (RC Array) of processors. Each processor of the array has its own register file, which can be accessed by other processors via an interconnection network. Hence, issues in instruction scheduling and register allocation are more complex.

1.5.4. This Work

The main focus of this work is to build a compiler framework to translate SA-C programs into an execution schedule that can be mapped to the Morphosys reconfigurable architecture. A typical image-processing application consists of a number of kernels (loops). A kernel is, typically, a set of computationally intensive operations performed in a loop. We focus on discrete (non-overlapping) loops (or kernels) in the program, and map them for execution on the reconfigurable element. This will provide the execution characteristics (execution latency, and configuration memory requirements) that are necessary to optimally schedule multiple kernels in the same application as per the techniques described in [8].

The mapping process, itself, is similar in nature to the architectural synthesis of algorithms. Each loop is analyzed as a set of operations, which require a certain number of resources for execution. Algorithms that perform operation scheduling, processor binding, and register allocation in the context of the Morphosys computational model are applied to produce a complete execution schedule.

Data transfer and caching can make significant differences in the overall execution of the program. The compiler uses a simple strategy to pre-fetch data, so as to overlap most data fetching (and data storing) with computation.

The compiler is evaluated by comparing the execution times of some common image-processing kernels on Morphosys to execution on an 800 MHz Pentium III. An average speed-up of 6x is observed among the benchmarks used.

The compiler presented here aims at maximizing the benefits of the computation model presented by the Morphosys architecture, under the restrictions and resource constraints presented by the architecture and the language.

This work concentrates on producing an instruction schedule that exploits the SIMD computational model of Morphosys, and identifies and exploits parallelism at a fine- and coarse-grained level. The focus of the compiler is to build a framework to map a *single kernel* onto the reconfigurable hardware for efficient execution. This objective is orthogonal to those addressed in [24], [8], [22], where the focus is on optimal *inter*-kernel scheduling. Also, the techniques proposed in [17], [20] can be used to optimally pipeline the schedule generated by our compiler.

2. Compiler Framework Overview

This chapter gives a brief overview of the entire compilation process.

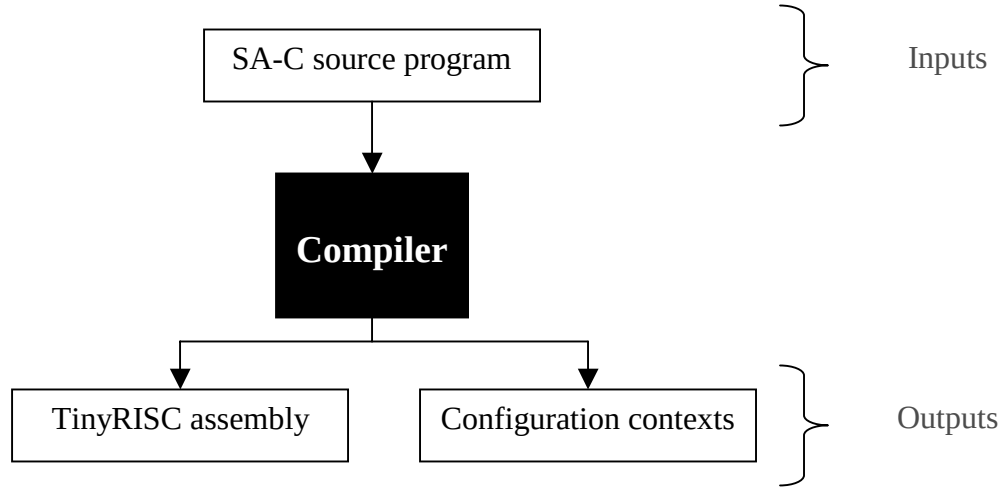


Fig 2.1: Overall objective of the compiler

Fig 2.1 shows a top-level view of the functions of the compiler. The compiler takes a SA-C source program as input and generates two files as output – the Tiny RISC assembly code and the configuration contexts. The configuration contexts represent the list of operations that will be performed by the RC Array during the execution of the program. These contexts are stored in the context memory of the Morphosys architecture. The Tiny RISC instruction set architecture (ISA) contains instructions to direct the rows/columns of the RC Array to perform the operation represented by a particular context. The Tiny RISC assembly code that is generated will contain such instructions that will control the execution flow of the RC Array based on the contexts that are generated. Hence, the Tiny RISC assembly code represents the control code that drives the execution of the whole program, while the configuration contexts are just a list of the operations that need to be performed by the RC Array to execute the given program. This chapter briefly describes the entire compilation process.

2.1 Flow of Compilation

Code partitioning determines which segments of the program will execute on the RC Array and which will execute on the Tiny RISC processor. The focus of this work is to completely map a given kernel (Fig. 2.2) for execution on the RC Array. All sequential code (outside loops) and code for synchronization and control are mapped for execution on the Tiny RISC.

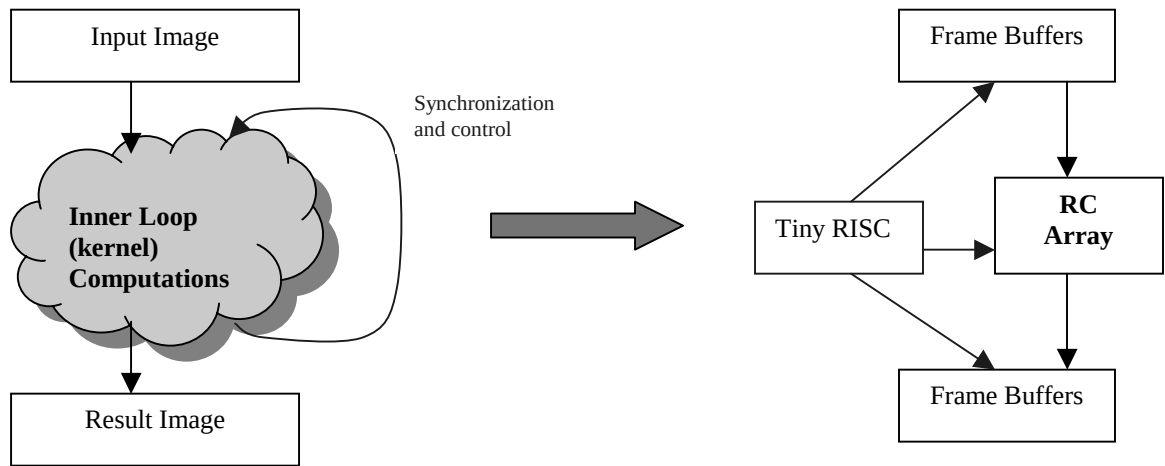


Fig 2.2: Mapping kernels to Morphosys

Fig. 2.3 shows the flow of compilation. The right-side branch of compilation after code partitioning (Fig. 2.3) represents the compilation of code that is not within loops. This phase of code generation is, essentially, similar to that of traditional compilers. The left-side branch represents the heart of this compiler.

The process of generating the detailed execution schedule is referred to as “loop synthesis” throughout this document and is described in detail in the next chapter. The compiler, first, performs a number of tasks that prepare the program graph for loop synthesis.

2.1.1. Function Inlining

Since SA-C does not support pointers or recursion, every function in SA-C can be inlined. Inlining a function ensures that the context, within which a function is called, is exposed. This makes a difference during code partitioning as a particular function can be called from either within a loop or from outside a loop, and this will determine whether

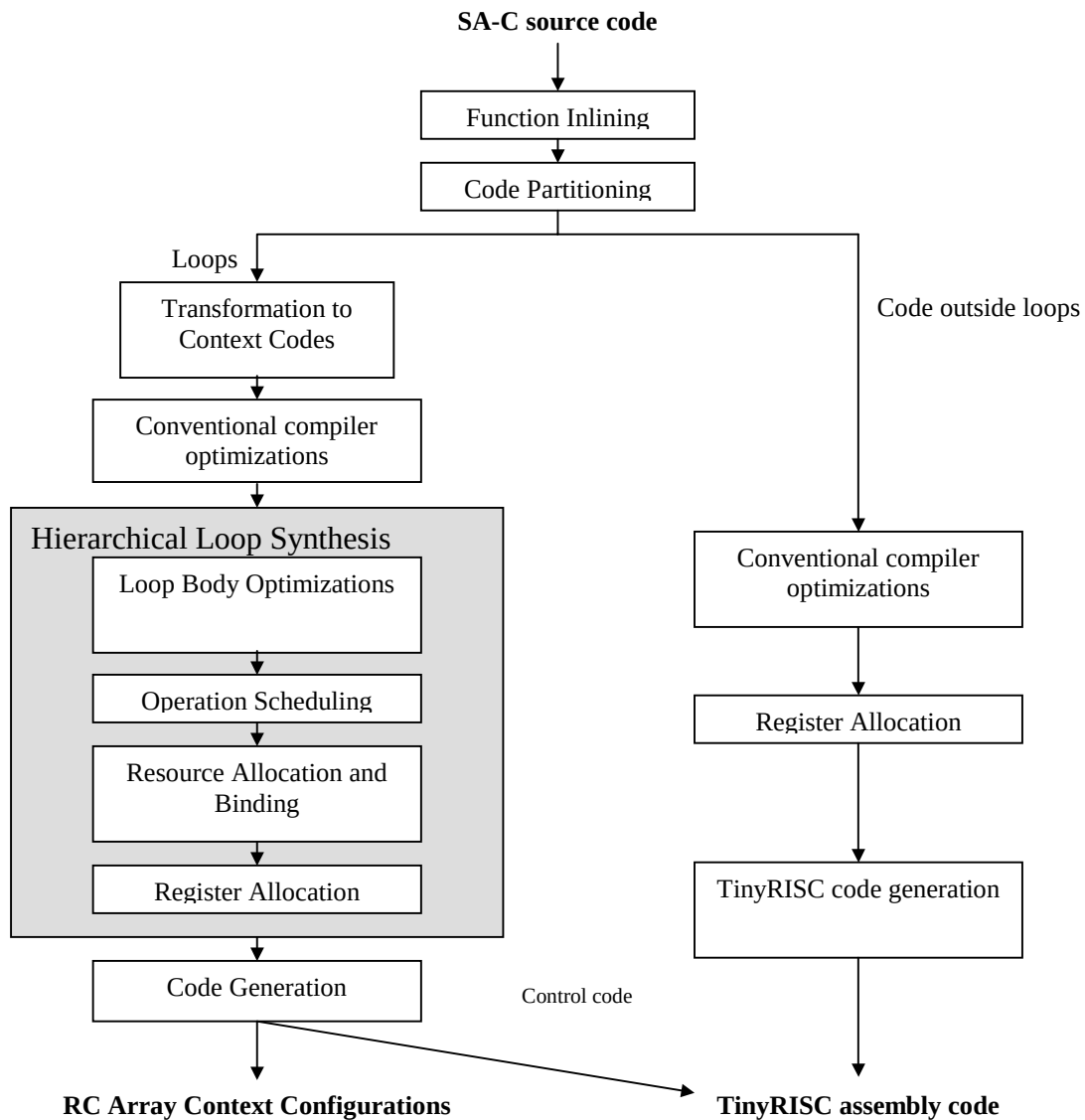


Fig. 2.3: Flow of Compilation

the particular function will be mapped to the RC Array or to the Tiny RISC processor.

The loops in the SA-C program get mapped onto the RC Array for execution. Hence, as a requirement, every simple node within a loop must have a one-to-one correspondence to a RC Array Context code.

2.1.2. Transformation to Context Codes

Most of the operations within a loop will usually directly correspond to an RC Array context code. However, at times, the operation is implicit and may be associated with a group of graph nodes. For example, the ABSOLUTE context code is designed to determine the absolute value of a given number. In the HDFG representation, this operation does not present itself directly. The two most common ways of writing the ABSOLUTE operation in code are:

```
Int8 x = a > 0? a: -a;          .. (1)
```

```
Int8 x = a > b? a-b: b-a;      .. (2)
```

During the optimization phase, the compiler essentially performs a pattern-matching pass to find such candidate groups of nodes that can represent an ABSOLUTE operation, and transforms such nodes.

On the other hand, there may be certain nodes in the HDFG representation that do not directly correspond to any of the RC Array context codes. For examples, there are no context codes that correspond to MAX, MIN or SQRT. These operations can, however, be represented as a sequence of context codes that have the same effect. In such cases, the operation execution latencies of these nodes are updated to reflect the time required to execute this sequence of contexts. Ordinarily, for all other operations that directly correspond to an RC Array context code, the execution latency of the operation is 1 clock cycle.

For example, a MIN(A, B) operation can be computed using RC Array context codes as follows:

```
D=A+B;
```

```
C=A-B;
```

```
If sign(C) then E=D+C else E=D-C
```

Hence, the latency of a MIN (and similarly MAX) operation is assumed to be 3 clock cycles.

The interesting operation to implement is SQRT. The compiler uses the Friden Algorithm [39] to implement the square root functionality. It assumes that all numbers in the applications that execute on Morphosys are 8-bit numbers. Given that we need to compute the square root of an 8-bit number, the Friden Algorithm computes in constant

time. This algorithm has been converted to a sequence of RC Array context codes to compute the square root of any 8-bit number. The execution latency of the algorithm is 50 clock cycles. Although this latency value may seem large, it can easily be amortized over multiple computations (remember that the RC Array functions as per the SIMD computation model).

2.1.3. Conventional Compiler Optimizations

Apart from the optimizations mentioned above, the compiler also performs certain conventional optimizations. Note that these optimizations are again directed at the code within loops.

- Conversion of Multiplications and Divisions to Shifts
- Common Sub-expression elimination
- Constant Folding and Constant Propagation
- Dead Code Elimination

2.2 Hierarchical Data Flow Graph

The Hierarchical Data flow graphs (HDFG) are used as intermediate representation in the compiler. These graph representations are similar in structure to the data dependence control flow (DDCF) graphs [15], with certain differences that reflect the nature of the Morphosys architecture. It is a convenient representation for performing compiler optimizations and for analysis in the mapping process.

An HDFG is an acyclic, directed, data flow graph, where some nodes can have sub-graphs within them. This hierarchical property preserves the program semantics. Fig. 2.4(a) shows an example of a SA-C program that computes the following function:

$$R[x][y] = \sum_{a=x}^{x+2y+2} \sum_{b=y} A[a][b]$$

```

Int8[8,8] f(int8[8,8] A) {
  Int8[8,8] R =
    For window w[3,3] in A {
      Int8 x = For e in w
        return (sum(e));
    } return (array(x));
} return R;

```

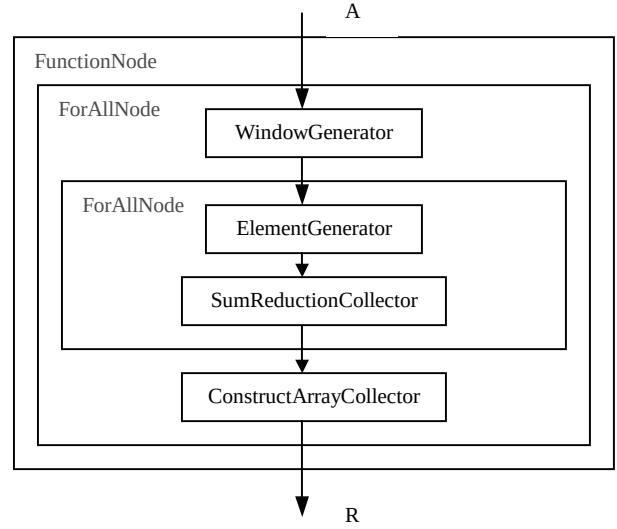
(a)

```

For (I=0; I<M; I++) {
  For (J=0; J<N; J++) {
    For (X=I; X<(I+3); X++) {
      For (Y=J; Y<(J+3); Y++) {
        R[I][J] += A[X][Y];
      }
    }
  }
}

```

(b)



(c)

Fig 2.4: SA-C Loop Example that performs the following function:

$$R[x][y] = \sum_{a=x}^{x+2} \sum_{b=y}^{y+2} A[a][b]$$

(a) The SA-C source code

(b) equivalent C code

(c) equivalent HDFG representation for a program that performs the following

Fig. 2.4(b) shows the equivalent C program and Fig. 2.4(c) shows the equivalent HDFG representation. The SA-C program has 2 loops. The outer loop contains a window generator that produces 3x3 windows from the source image, A. The inner loop contains an element generator, which produces scalar values from the generated window. Its loop collector is a *ReductionCollector* that performs a summation. Essentially, the inner loop computes the sum total of each window generated. The outer loop creates an array whose elements are the summation values produced from the inner loop. Hence, the outer loop contains a *ConstructArrayCollector*.

3. Hierarchical Loop Synthesis

The objective of this phase is to analyze each loop, perform optimizations, and generate a complete, efficient execution schedule that specifies the temporal ordering of each operation, where on the RC Array each of the operation will execute, and which results are written to which registers within the RC Array. The RC Array provides extensive support for the SIMD computation model. Hence, the goal of the compiler is to exploit the benefits of the RC Array by scheduling the loop code for execution on the Morphosys architecture, while adhering to its computational model.

In general, the RC Array can be visualized as an array of programmable computing elements. Hence, in the trivial case, a data flow graph can be mapped to the RC Array using traditional architectural synthesis techniques. However, the number of computing elements is limited and the computation model is not as flexible as a general programmable logic array. Also, it is important to be aware of the structure and the restrictions of the source language in which the application is expressed – this will really govern the kind of programs the compiler has to deal with. Hence, a framework is designed, based on the characteristics of the language as well as the RC Array, to automatically map the loops for execution on the RC Array.

3.1 Hierarchical Approach

All code in the SA-C program is statically scheduled for execution by the compiler. The compiler adopts a hierarchical approach to solve the problem of mapping SA-C loops. Loops are synthesized based on their relative position in the *loop hierarchy*. The innermost loop is defined to be at the bottom of the loop hierarchy. The compiler's approach is to synthesize the inner most loop, and then recursively move up the loop hierarchy until the outermost loop is synthesized. The compiler framework defines different execution models based on the loop's generator. This section examines different loop generators and describes the strategy used in synthesizing their loops.

3.2 Problem Analysis

In general, the problem of mapping loops to the RC Array is treated as a generic architectural synthesis problem, which is concerned with mapping algorithms to hardware. Hence, given the SA-C program, the compiler determines which operations will be executed on which resources and in which execution cycle. In this context, a *resource* is defined to be one row (or column) of the RC Array. This is due to the SIMD nature of the RC Array – in a particular clock cycle, all the cells in a particular row (or column) shall only perform the same operation. Hence, there is a total of 8 resources (8 rows) that is available in any given clock cycle. The objective of the compiler scheduling is to maximize the resource usage in every clock cycle. Hence, loops are always unrolled when executing on the RC Array so that multiple iterations are executing in any given clock cycles.

Each node (or operation) within the loop is marked with its execution latency times and resource requirements. For the loop body of an innermost loop, these numbers are pre-defined. Once an inner loop is synthesized, these numbers for the loop itself can be recursively inferred.

3.3 Loops with Element Generators

Loops with element generators are generally the innermost loop of any loop hierarchy. Its loop body is a function of a particular element of the source array. There are no data dependences between iterations, and there are no common computations between iterations. The loop is unrolled in both the horizontal and vertical direction so as to process 64 loop iterations in a single RC Array iteration. Execution of every loop iteration is performed on a single RC Array cell. Hence, the resource-binding problem is trivial and is obviated.

3.3.1. Operation Scheduling

The operation-scheduling problem reduces to scheduling a data flow graph onto a single, sequential processor. There is only one constraint that needs to be considered in scheduling these loops. Certain operations within the loop may be data fetch operations, whose source data inputs reside in the Frame Buffer. As per the Morphosys architecture, only one row (or column) can perform a data fetch operation in a given cycle. Alternately, only 8 elements can be fetched from the Frame Buffer in any given cycle. To accommodate this constraint, such operations are identified, and their operation latency numbers are tweaked to eight times their actual latencies. Finally, the compiler uses the ASAP (“As Soon As Possible”) scheduling algorithm to schedule operations. This algorithm schedules operations as soon as their source data inputs are available.

3.3.2. Register Allocation

A simple register allocation strategy is used which keeps track of the free registers, and allocates registers to intermediate results of operations, as and when required. Register spills are handled by writing the values to the Frame Buffer.

3.4 Loops with Window Generators

Perhaps the most important part of the SA-C language is its *window generators*, which are useful in expressing a number of common image processing applications in an elegant way. This kind of a loop allows a window to “slide” over the source array producing sub-arrays of the same rank (dimensions) as the source array.

Fig. 3.1 shows a snapshot of a windowing loop from the example in Fig. 2.4. The loop

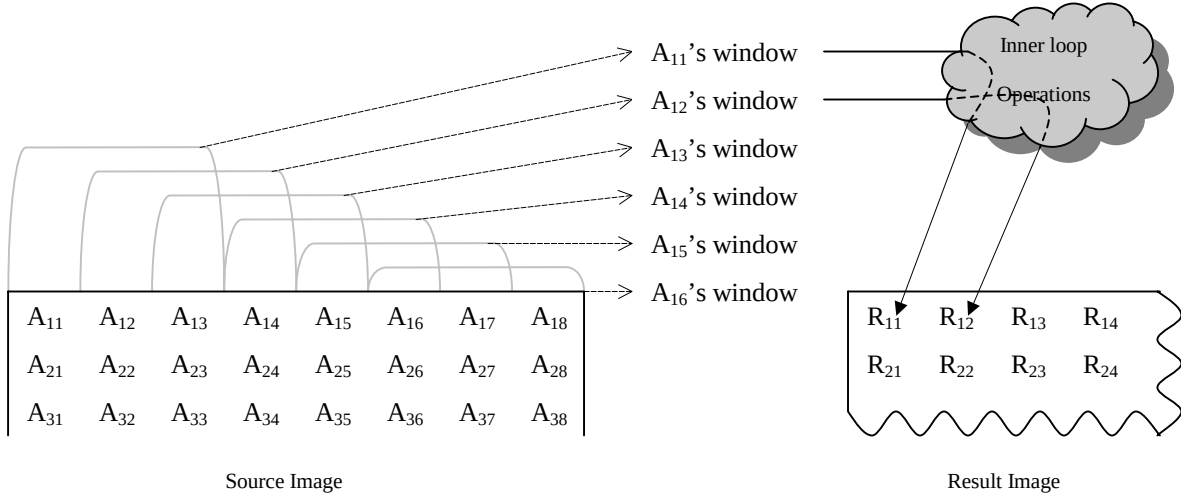


Fig. 3.1: Snapshot of windowing

generates a 3x3 window in each iteration of the loop. Hence, every 3x3 window in the loop is the input data for a separate iteration. The inner loop body transforms this window into a single pixel (corresponding to the sum total of elements in the iteration's window) in the resultant image.

In spite of the SIMD computational model of the RC Array, all the iteration windows present in the RC Array cannot be computed concurrently. This is because some of the elements are part of multiple iteration windows. For example, element A_{13} is a member of 3 iteration windows – A_{11} , A_{12} and A_{13} . However, execution of windows A_{11} and A_{14} can be performed concurrently. Similarly, windows A_{12} and A_{15} can also be executed concurrently.

Hence, non-overlapping windows can be computed in parallel. Specifically, all windows that are separated by whole multiples of the window size are computed concurrently. Hence, iteration windows corresponding to elements A_{11} , A_{14} , A_{41} , and A_{44} are computed concurrently. Then, iteration windows corresponding to elements A_{12} , A_{15} , A_{42} , and A_{45} are computed concurrently, and so on. There are a total of 36 iteration windows in the RC Array, and sets of 4 iterations can be executed concurrently. After all the 36 iterations are completed, the next chunk of 64 elements is brought into the RC Array.

This framework can be generalized for any loop generating windows of size $M \times N$. The RC Array processes $(8-M+1)$ iterations in the horizontal dimension and $(8-N+1)$ iterations in the vertical dimension, for a total of $[(8-N+1) \times (8-M+1)]$ iterations between successive data fetches. The following sections describe how this strip-mined version of the loop is synthesized. The compiler assumes that all window generators produce windows that are smaller than or equal to an 8x8 window in size. Since most standard image-processing applications work within this constraint, this is a reasonable assumption to make.

3.4.1. Windowing Loop Optimizations

Fig. 3.2 shows a simple program that computes the resultant array, R , for any two given arrays, A and B . The program can be summarized by the following function:

$$R[x][y] = \sum_{a=x}^{x+2} \sum_{b=y}^{y+2} (A[a][b] * B[a][b])$$

<pre> Int8[:, :] R = For window wa[3,3] in A dot window wb[3,3] in B { Int8 asum = For a in wa dot b in wb return (sum(a * b)); } return (array(asum)); </pre>	<pre> For (I=0; I<M; I++) { For (J=0; J<N; J++) { R[I][J] = 0; For (X=I; J<(I+3); X++) { For (Y=J; Y<(J+3); Y++) { R[I][J] += A[X][Y] * B[X][Y]; } } } } </pre>
(a)	(b)

Fig 3.2: Windowing Loop example

(a) SA-C code

(b) C code

The windows generated in two separate iterations of this loop have some common sub-expressions. Fig. 3.3 shows the pictorial view of two iterations of this loop. The computations “ $A_{12} * B_{12}$ ” and “ $A_{13} * B_{13}$ ” are performed in both iterations and are common sub-expressions.

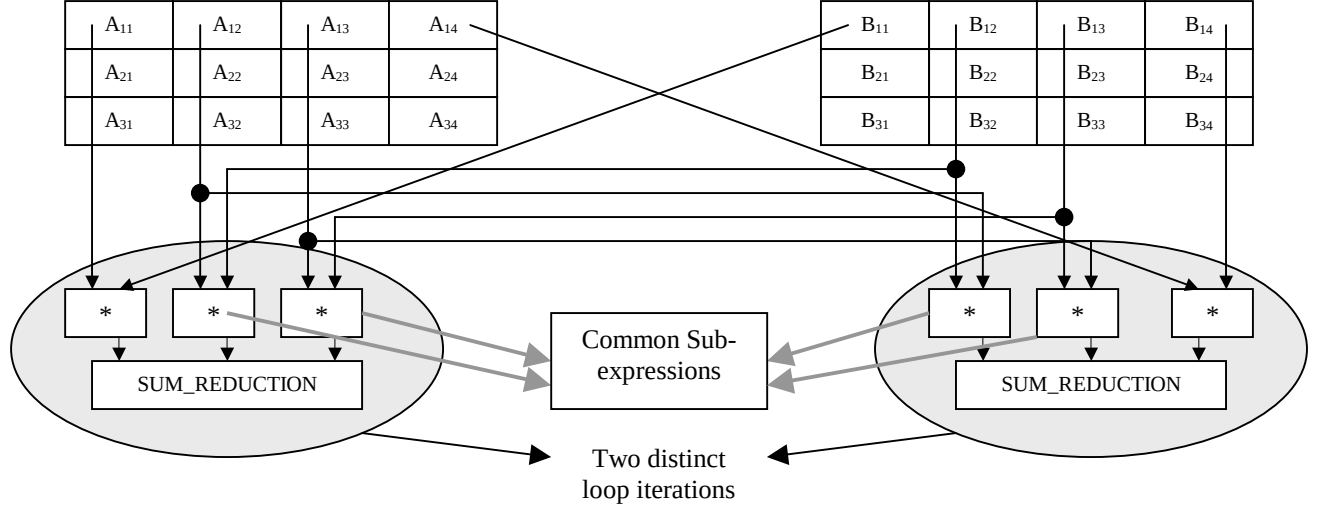


Figure 3.3: Common Sub-expressions

In general, whenever a particular element of the source array appears in multiple windows generated, there could potentially be common sub-expressions. In order to eliminate these common sub-expressions, the windowing loop must be unrolled so as to expose all dependencies across iterations.

The number of iterations of the windowing loop that need to be unrolled is equivalent to the number of overlapping iterations. For a loop generating $M \times N$ window with steps of sh and sv in the horizontal and vertical dimensions respectively, the number of overlapping iterations, NI , is given by:

$$NI = \text{ceil}(N/sh) * \text{ceil}(M/sv)$$

Where $\text{ceil}(n)$ returns the largest integer lesser than or equal to n .

However, for window sizes greater than 4 in either direction, it is not possible to fetch all the data corresponding to NI windows into the RC Array. Consider a window size of 5×5 . The first window in each row corresponds to column 1, and the last window will begin on column 5 and end in column 9. Hence, this requires a total of 9×9 elements, whereas the total size of the RC Array is 8×8 . For such windows, there will be a total of $(8-N+1)$ iterations that need to be analyzed. However, if the source array (of size $IX * IY$, say) is smaller than the RC Array itself, then the number of windows is equivalent to $(IY-N+1)$. Hence, the number of iterations, NI , is modified as follows:

$$\begin{aligned} X &= \text{MIN}(IX, 8) \\ Y &= \text{MIN}(IY, 8) \\ H &= \text{ceil}[\{\text{MIN}(N, Y - N + 1)\}/sh] \\ V &= \text{ceil}[\{\text{MIN}(M, X - M + 1)\}/sv] \\ NI &= H * V \end{aligned}$$

The compiler analyzes these NI iteration windows and eliminates all redundant sub-expressions. This gives rise to dead code, which is eliminated as well.

At the end of this optimization pass, there will be NI distinct data flow graphs corresponding to each iteration. However, there may be some *cross-edges* between these data flow graphs that represent the re-use of computation. These edges are synthesized into registers during the register allocation phase.

This optimization pass only unrolls an element-generating loop that may be embedded within it. If another windowing loop is embedded within this windowing loop, then the inner loop is not unrolled, and is treated as a compound node and is not considered as a candidate node during this optimization pass. This is because analyzing a windowing loop in this manner produces iterations of the order $O(n^2)$. A windowing loop that encloses another windowing loop will have to analyze $O(n^2)$ iterations, where each iteration is an iteration of the windowing loop and contains another $O(n^2)$ iterations. Hence, opening inner window-generating loops in this manner would result in exponential growth.

3.4.2. Loop Synthesis

Once the optimizations specified in the previous section are performed, each windowing loop will be associated with NI different loop iterations. These iterations may be different because of redundant computations that may be eliminated. To synthesize the inner loop of the windowing loop, each of these iterations must be separately synthesized. Hence, the synthesis techniques discussed in the following sections will be applied to **each** of the NI iterations. These iterations are never executed concurrently. Hence, the final schedule would just be a linear ordering of each iteration's schedule.

3.4.2.1 Inference of Operation Latencies and Resource Requirements

Before using the synthesis techniques mentioned in this section, it is required to assign operation latencies and resource requirements to each node (operation) within the loop. All simple nodes will already have been assigned pre-determined latencies and resource requirements.

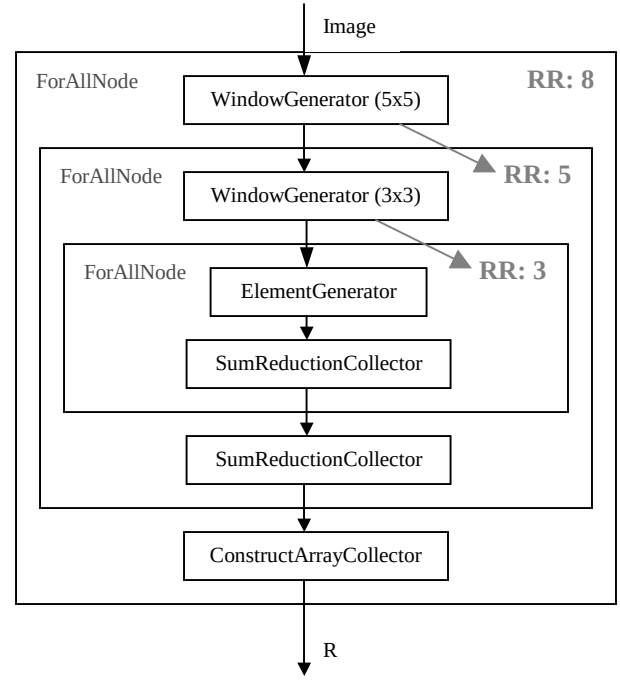
The top-most loop in the loop hierarchy is always assigned a resource requirement of 8 – this will ensure that the loop will attempt to fully use all the resources available in the RC Array. A windowing loop can have an element-generating loop and/or another windowing loop embedded within it. The resource requirement for an inner loop is defined to be “the vertical dimension of the window generated by its parent loop”. This will ensure that the inner loops will attempt to fully utilize the data exposed by the outer loop. Figure 3.4 shows an example program (a), and its HDFG representation (b). Each loop in the HDFG is annotated with its resource-requirement (RR) assignment.

```

Int8[:, :] R =
  For window win[5, 5] in Image {
    Int8 res =
      For window w[3, 3] in win {
        Int8 x =
          For elem in w
            Return (sum(elem));
        } return (sum(x));
      } return (array(res));
  }

```

(a)



(b)

Fig. 3.4: Resource Allocation: for Loop Hierarchies is based on the vertical dimension of the window generated by the parent loop (is 8 for outer-most loops. Example of: (a) SA-C program, and (b) its equivalent HDFG representation

3.4.2.2 Operation Scheduling

The operation-scheduling problem for a windowing loop is defined as finding a schedule that executes in minimum time under two constraints - the availability of resources and the RC Array execution mode. There are two modes of execution on the RC Array – row mode and column mode. In any given clock cycle, only one mode of operation can be active. However, a node could be scheduled to execute over both modes through multiple clock cycles. Concurrent operations must all execute in the same modes throughout each operation's lifetimes.

The operation scheduling algorithm itself is known to be NP-complete, and usually heuristic algorithms are used. One popular heuristic is the *List Scheduling* algorithm. The compiler uses an extension of this algorithm that attempts to schedule nodes on the critical path as early as possible. A node is scheduled only when there are sufficient resources available and when the node's execution mode does not conflict with the execution mode of the schedule generated thus far.

The schedules thus generated (for each of the NI iterations) are then linearly ordered to complete the execution of all the iterations that are present in the RC Array. Each set of data fetched into the RC Array is subjected to these execution schedules. For a

windowing loop generating $M \times N$ windows, the total execution time, T , of the loop over an image of size, $[h, w]$, is given by:

Let S = Size of the source image in any dimension

Let D_t = Distance between first element of two successive data fetches

= Number of windows in that dimension, $NW * \text{Window step in that dimension, } st$

Then, number of data fetches in that dimension = S/D_t

Number of windows in any dimension, $NW = \text{ceil}[(X - W + 1)/st]$

Where $X = \text{MIN}(W_p, 8)$

W = window size in that dimension

W_p = source image size in that dimension (= 8 if outermost loop)

Hence, total Data fetches, $D = D_h * D_v$

Where D_h = Data fetches in horizontal dimension

D_v = Data fetches in vertical dimension

Execution time of window loop, $T = D * \sum_{i=1}^{NI} k_i$ schedule.
where k_i = Latency of the i^{th} iteration's

3.4.2.3 Resource Allocation and Binding

The RC Array is divided into four quadrants each of size 4×4 . A given RC cell can directly access only the cells in the same row and column as itself. Further, these other cells need to be in the same quadrant as itself. In Fig. 3.5, for example, cell R_{22} can directly access (in the same clock cycle) cells R_{12} , R_{32} , and R_{42} in the vertical dimension, and cells R_{21} , R_{23} , and R_{24} in the horizontal dimension. Accessing any other cell would incur a communication penalty.

The resource to minimize	R_{11}	R_{12}	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}	R_{18}	objective of allocation is these
	R_{21}	R_{22}	R_{23}	R_{24}	R_{25}	R_{26}	R_{27}	R_{28}	
	R_{31}	R_{32}	R_{33}	R_{34}	R_{35}	R_{36}	R_{37}	R_{38}	
	R_{41}	R_{42}	R_{43}	R_{44}	R_{45}	R_{46}	R_{47}	R_{48}	
	R_{51}	R_{52}	R_{53}	R_{54}	R_{55}	R_{56}	R_{57}	R_{58}	
	R_{61}	R_{62}	R_{63}	R_{64}	R_{65}	R_{66}	R_{67}	R_{68}	
	R_{71}	R_{72}	R_{73}	R_{74}	R_{75}	R_{76}	R_{77}	R_{78}	
	R_{81}	R_{82}	R_{83}	R_{84}	R_{85}	R_{86}	R_{87}	R_{88}	

Fig 3.5: RC Array Connectivity

communication latencies. To solve this problem, a special graph is created, where the nodes are operations and edges between nodes indicate “affinity to sharing a resource” between the two nodes. These edges, called *shareable* edges, are added as follows:

- Operation-pairs scheduled to execute concurrently don’t share any edge.

- All other operation-pairs (that are not scheduled for concurrent execution) share an edge.
- Two nodes that have a direct data-dependence (i.e. an edge in the data flow graph) are assigned a higher weight (say k) than all other nodes (default weight is 1). This is because the result of one operation is the input operand of the other. There will be no communication penalty if the two operations share the same resource. Hence, a weight on an edge gives more importance to it.

Another type of edges, called *closeness* edges, is also added to the graph. These edges reflect the condition when two nodes are assigned to different resources; however, these resources must be as close to each other as possible. Consider an operation, *op*, which needs two operands that are produced as results of operations, *op1* and *op2*. Then, *op1* and *op2* must be scheduled as close to each other as possible in order to avoid the communication penalty. These edges are added as follows:

- If the operands of a node are produced by two different operations, then these two (source) operations will share a *closeness* edge between them
- The weight on this *closeness* edge is accumulated if more *closeness* edges are generated between the same two nodes.

The graph thus generated is subject to CLIQUE_PARTITIONING¹. There are two different objectives that need to be satisfied during resource allocation – resource sharing (based on the *shareable* edges) and assignment of resources close to each other (*closeness* edges). To satisfy these seemingly orthogonal objectives, the compiler performs two levels of clique partitioning:

- Perform CLIQUE_PARTITIONING based on the *shareable* edges. A cluster of nodes thus formed will indicate the nodes that should share a resource.
- Create a new graph by collapsing each clique into a single, unique node.
- Perform CLIQUE_PARTITIONING on this new graph based on the *closeness* edges. *Super-clusters* now formed represent a group of nodes, which need to be assigned resources as close together as possible.

One of the components of the CLIQUE_PARTITIONING problem is to find the maximal clique in the graph (MAX_CLIQUE). This problem is known to be NP-complete. The compiler uses a heuristic to solve it – the clique containing the node with the maximum number of edges is assumed to be the best candidate for the maximal clique.

In the end, the graph is a set of “super-cliques”, where each node in the super-clique represents a clique from the first level of clique partitioning. When every clique in the super-clique has been assigned a resource, all the operations within that clique will share this resource. The compiler uses a heuristic in assigning resources to the cliques within a

¹ CLIQUE_PARTITIONING is a popular graph-partitioning algorithm. A *clique* is defined as a fully connected sub-graph.

super-clique. It tries to keep the clique with largest “closeness requirements” (equal to the sum total of all weights on its *closeness* edges) as close as possible to every other clique within its super-clique.

3.4.2.4 Register Allocation

Register Allocation strategy for windowing loops use the same strategies as used by element-generating loops. However, after performing common sub-expression elimination, values (represented by *cross-edges*) may be forwarded to other iterations. Register allocation is performed in two phases. First, common computation results that are forwarded between iterations are allocated to registers. These registers will be required throughout the entire loop execution between data fetches. Then, registers are allocated to each (of the NI) iterations. However, these registers are alive only during the particular iteration’s execution.

3.5 Data Pre-Fetching and Caching

One of the principal hindrances to achieving higher performance is the memory-CPU bandwidth. The factor has been taken into consideration in the design of the Morphosys architecture. The Frame Buffer in the architecture is designed to perform the function of a “streaming cache”. It consists of two sets, each of which, in turn, is made up two banks. The DMA controller sets up direct data transfers between the main memory and the Frame Buffer.

The design of the Frame Buffer is such that its two sets can be independently accessed at the same time. Hence, while data is being loaded into one set, the RC Array can fetch data from the other set. Thus, computation can be overlapped with data transfers, thereby reducing the memory latency.

To implement this kind of strategy, the compiler analyzes the concerned kernel’s rate of data consumption (implying data loading), its rate of data production (implying data storing), and its execution latency, and inserts data pre-fetch instructions in appropriate places in order to reduce the memory latency.

Consider a kernel with N_i streaming inputs, and N_o streaming outputs. Assume that the compiler generated schedule for the kernel has an overall execution latency of T cycles. Further, assume that this kernel consumes a data slab of size $M \times N$ and produces output slabs of sizes $P_1, P_2, \dots, P_o$ for each of the N_o outputs. Memory latency for a typical execution cycle² is made of the time to fetch data for the next execution cycle and the time to write back the results of the previous execution cycle. Hence, the total execution time for an execution cycle is given by:

$$\text{Actual Execution Time, } T_{\text{actual}} = \text{MAX}(T, T_{\text{mem}}(N_i * (M * N) + \sum P))$$

where $T_{\text{mem}}(X)$ is the memory latency for fetching X data elements.

² A cycle in this context does not refer to the clock cycle. It refers to an “execution cycle” during which the kernel performs all its computations on the given input data slab. It can be thought of as an equivalent of one (or more) loop iterations.

The compiler computes these latencies and inserts the data-fetch instructions in the execution cycle prior to the one that needs them, and write-back instructions in the execution cycle following the one that produces them.

The delay due to memory latency exists only if the T_{mem} component of the *MAX* function is bigger than T . An interesting observation is that the memory latency does not depend directly on the size of the Frame Buffer. The Frame buffer need only be big enough to house all input data required in and output data produced by an execution cycle (which is easily satisfied for most benchmarks). This implies that for the execution of a single kernel, the speed of data transfer, and not the size of the cache buffers, is the principal bottleneck. However, [8] discusses an exploration algorithm that analyzes many such kernels, and based on their execution latencies, and data consumption and production rates, determines a schedule for kernel execution as well as data transfer. In this context, a particular data structure (element) may be accessed multiple times by multiple kernels. Hence, the algorithm examines the trade-offs between executing multiple kernels using the same set of data, and running each kernel to completion, one at a time. In the former case, once a data set is loaded, it satisfies all its consumers (kernels). However, each kernel's configuration code may have to be loaded multiple times. In the latter case, each kernel's configuration code needs to be loaded only once, but data sets may need to be re-loaded. In this context, the size of the pre-fetch cache will make a significant difference.

4. Performance Measurements

This chapter discusses the performance measurements of applications compiled for the Morphosys architecture. In addition, it also evaluates the efficiency and benefits of the loop optimizations that are performed by the compiler. It first describes the framework and methodology used to evaluate the performance of the compiler. Then, the representative applications that have been used for the purpose of measurement are described, and finally, the results are presented.

4.1 Experimental Framework

The approach used to test the efficiency of the compiler is to examine the performance of the instruction schedules generated by the compiler for certain sample applications written in SA-C. These applications are also executed separately, using the same test data, under Windows 2000 on an 800 MHz Pentium III platform. For this purpose, the applications are written in native C code and are compiled using the VC++ 6.0 compiler with the highest level of optimizations turned on. The results of the two execution times are compared and contrasted. The chapter also presents comparison between optimized and unoptimized schedules to examine the usefulness of the compiler loop optimizations.

4.2 The Applications

The test applications used for the purpose of performance evaluation represent important application kernels from the image-processing domain. These applications are chosen to describe a variety of behavior. Hence, these applications differ in the type of the loops they contain and also in the kind of inner loop computations that need to be performed.

4.2.1. Wavelet Compression

Wavelets are commonly used for multi-scale analysis in computer vision, as well as for image compression. Honeywell has defined a set of benchmarks for reconfigurable computing systems, including a wavelet-based image compression algorithm. The wavelet program has been translated into SA-C, generalized to operate on any size image. The algorithm works on 5x5 windows of the source image.

4.2.2. Prewitt Edge Detection

This is an edge detection program that calculates the square root of the sum of the squares of responses to horizontal and vertical Prewitt edge masks. Since this same task can be performed using the Intel Image Processing Library (IPL), the results can be compared with that of a hand-optimized Pentium program.

The Prewitt edge detection masks are one of the oldest and best understood methods of detecting edges in images. Basically, there are two masks, one for detecting image derivatives in X and one for detecting image derivatives in Y. To find edges, a user convolves an image with both masks, producing two derivative images (dx & dy). The strength of the edge at any given image location is then the square root of the sum of the squares of these two derivatives. (The orientation of the edge is the arc tangent of dy/dx).

This particular implementation of the algorithm works on 3x3 windows using 3x3 horizontal and vertical masks.

4.2.3. Motion Estimation for MPEG

Motion estimation helps to identify redundancy between frames in an MPEG video stream. The most popular technique for motion estimation is the block-matching algorithm [22]. This algorithm is one of the kernels in the MPEG-4 compression algorithms.

4.2.4. 2D Convolution

This is a common application used in digital signal processing. It computes the linear convolution of every 3x3 window in the source image with a 3x3 kernel size. This algorithm is a part of the Intel Image Processing Library (IPL) as well as the Vector, Signal and Image Processing Library (VSIPL). VSIPL forum is a volunteer organization made up of industry, government, users, and academia representatives who are working to define an industry standard API for vector, signal, and image processing primitives for embedded real-time signal processing systems.

4.3 Results

All applications were executed using 8-bit 512x512 source image (s). The Morphosys processor runs at a clock frequency of 200 MHz. Figure 4.1 compares the compiled codes running on Morphosys with execution on the Pentium platforms. The execution time is specified in seconds.

While the runtime performance of the *Convolution* and *Motion Estimation* on Morphosys is comparable to that on Pentium, Morphosys outperforms Pentium in *Wavelet* and *Prewitt*. This is because *Convolution* and *Motion Estimation* are small applications that are pre-dominantly I/O-bound. In contrast, *Wavelet* and *Prewitt* are computationally intensive kernels, and they reap big benefits from SIMD computational model of

Morphosys.

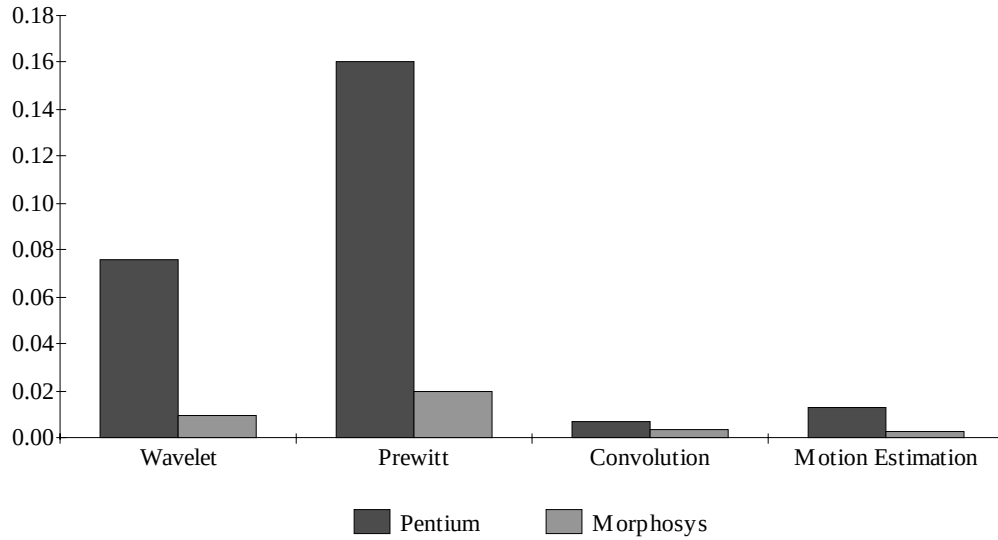


Figure 4.1: Runtime Performance Comparison

Fig 4.2 shows the effect of performing loop optimizations (the execution time is specified as number of clock cycles). The only application that does not benefit is the *Motion Estimation* kernel. This is because the application works on an element-generating loop, and the iterations of element-generating loops are all independent of each other. Hence, there is no opportunity to perform any of the inner loop optimizations.

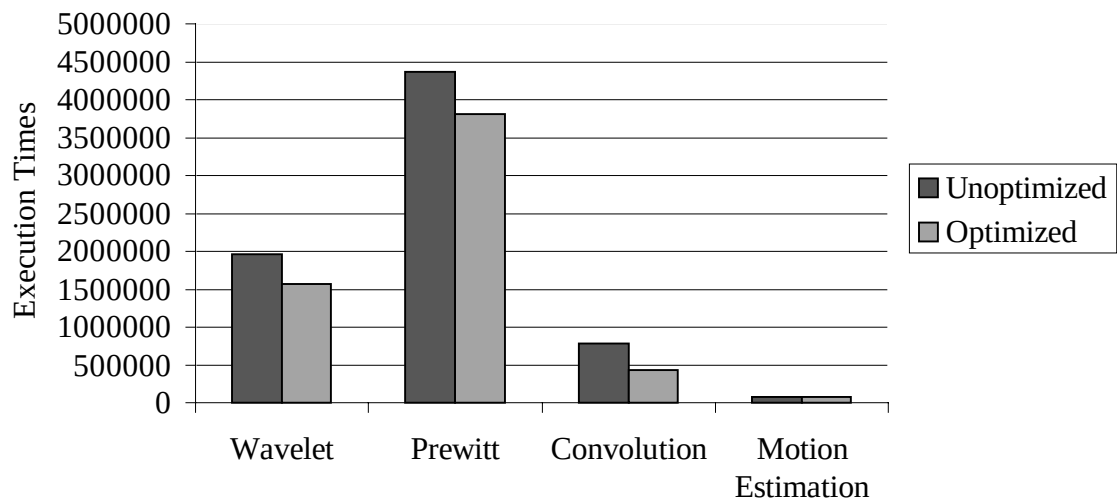


Fig 4.2: Evaluation of Loop Optimizations

5. Conclusions and Future Directions

5.1 Conclusions

This thesis presents a methodology and framework that enables the efficient compilation of applications written in a high-level language to a reconfigurable computing architecture. In particular, the compiler aims at extracting the data parallelism at a coarse- and fine-grained level in a given application, and producing an instruction schedule that explicitly reflects a SIMD computational model. It describes how an image-processing application written in SA-C is partitioned and how it can be executed on the Morphosys architecture. In doing so, it describes how data parallel semantics of the program are identified, analyzed and mapped for execution on the RC Array of the Morphosys architecture.

It describes the synthesis approach of the mapping process, which performs operation scheduling, resource binding and register allocation, in order to produce an execution schedule. In the process, a number of algorithms that need to be used in the mapping process are proposed. Also, different compiler optimizations are proposed that could potentially improve the execution time of applications on the target platform. It also discusses the data transfer and caching issues that could greatly alleviate memory latencies.

The previous chapter presents performance results that show that the compiler-generated schedule can achieve an average speedup of up to 6x for the tested benchmarks. Also, it shows that the loop optimizations performed by the compiler could potentially produce significant improvements in the execution times.

5.2 Future Work

The work presented here is a first step in the direction of automatic compilation for the Morphosys platform. It presented an approach to efficiently analyze the computation-intensive kernels in image-processing applications, and map them onto the RC Array of the platform. There are, however, a number of issues that could be addressed as the next possible steps for improving this process:

- **Inter-kernel Analysis:** The current model is designed to analyze a single computational kernel in a given image-processing application, and then map it to the Morphosys architecture for near-optimal execution. A standard image processing application like Automatic Target Recognition (ATR), for example, may have a number of different kernels, which may interact with one and other. Issues like maximizing data re-use, and minimizing reconfiguration time are critical to an optimal execution schedule. Some research efforts [21, 23, 28] have already looked into this, but special analysis is required in the context of the Morphosys architecture. Hence, a future step would be to incorporate these issues into the current compiler model.

- Exploration: The current compilation model to map loops is to fit as many loop iterations in the RC Array as is possible. This is done by unrolling and stripmining the loop iterations as much as necessary. However, it may sometimes be beneficial to restrict this unrolling, so that other operations within the loop body could be performed concurrently. The compiler has to decide the optimal unrolling/stripmining factor that might produce the best schedule in terms of execution times. For this purpose, the compiler may have to produce a number of different schedules and compare their execution times.
- Pipelining: In the current model, the compiler adopts a bottom-up approach to synthesize the different nodes within a loop. In the process, compound nodes are created that are annotated with execution times and resource requirements. When scheduling these nodes, the compiler devotes all the resources that the compound node needs to it for the entire duration of its execution latency. However, it may be the case, that the node may not need all of these resources for the entire duration of the operation. As a next step, the compiler could analyze such situations, and either pipeline such operations to optimally use all the resources.
- Extending the Compilable domain: Currently, only a restricted subset of the whole SA-C language can be mapped to the Morphosys architecture. A number of reduction operators, loop generators and language semantics have not been analyzed. The main reason for this is that currently only the most essential features that need to be mapped onto the RC Array has been analyzed. Moreover, some of the language features cannot be directly mapped to the Morphosys architecture as yet. For example, the SA-C language supports variable bit-width precision and extracting a column slice or a plane from an array. Currently, these features cannot be implemented on the Morphosys architecture.

References

- [1] A. DeHon, "The Density Advantage of Configurable Computing". *IEEE Computer*, vol. 33, pp. 41-49, April 2000.
- [2] H. Singh, et. al., "Morphosys: An Integrated Reconfigurable Architecture,". *NATO Symposium on Systems Concepts and Integration*, Monterey, CA, 1998.
- [3] H. Singh, et. al., "Morphosys: Case study of a reconfigurable computing system targeting multimedia applications,". *37th Design Automation Conference*, Los Angeles, CA, 2000.
- [4] H. Singh, et. al., "Morphosys: A Parallel Reconfigurable System,". *Euro-Par*, Toulouse, France, 1999.
- [5] H. Singh, et. al., "Morphosys: A Reconfigurable Architecture for Multimedia Applications,". *Workshop on Reconfigurable Computing at PACT*, Paris, France, 1998.
- [6] E. M. C. Filho, "The TinyRISC Instruction Set Architecture, Version 2," University of California, Irvine, Irvine, CA November 1998.
<http://www.eng.uci.edu/morphosys/docs/isa.pdf>.
- [7] M. Lee, et. al., "Design and Implementation of the Morphosys Reconfigurable Computing Processor,". *Journal of VLSI and Signal Processing-Systems for Signal, Image and Video Technology*, 2000.
- [8] R. Maestre, et. al., "Kernel Scheduling in Reconfigurable Computing,". *DATE*, Munich, Germany, 1999.
- [9] R. Rinker et. al., "An Automated Process for Compiling Dataflow Graphs into Hardware,". *IEEE Transactions on VLSI Systems*, vol. 9, 2001.
- [10] R. Rinker, et. al., "Compiling Image Processing Applications to Reconfigurable Hardware," *IEEE International Conference on Application-specific Systems, Architectures and Processors*, Boston, MA, 2000.
- [11] J. Hammes, et. al., "Cameron: High Level Language Compilation for Reconfigurable Systems," *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Newport Beach, CA, 1999.
- [12] J. Hammes, et. al., "Compiling a High-level Language to Reconfigurable Systems," *Compiler and Architecture Support for Embedded Systems (CASES)*, Washington, DC, 1999.
- [13] J. Hammes, et. al., "Sassy: A Language and Optimizing Compiler for Image Processing on Reconfigurable Computing Systems," *International Conference on Vision Systems*, Las Palmas de Gran Canaria, Spain, 1999.
- [14] W. Bohm, "The SA-C Language - Version 1.0," Colorado State University, Fort Collins, CO, Technical Report June 2001.
<http://www.cs.colostate.edu/cameron/Documents/sassy.pdf>.
- [15] W. Bohm, "The SA-C Compiler Data-Dependence-Control-Flow (DDCF)," Colorado State University, Fort Collins, CO, Technical June 2001.
<http://www.cs.colostate.edu/cameron/Documents/ddcf.pdf>.

- [16] B. Draper, et. al., "Compiling and Optimizing Image Processing Applications for FPGAs," *International Workshop on Computer Architecture for Machine Perception (CAMP)*, Padova, Italy, 2000.
- [17] C. Ebeling, et. al., "Mapping Applications to the RaPiD Configurable Architecture," *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, 1997.
- [18] Elliot Waingold, et. al., "Baring it all to software: RAW machines," *IEEE Computer*, vol. 30, pp. 86-93, 1997.
- [19] J. Wawrzynek and T.J. Callahan, "Instruction-level Parallelism for Reconfigurable Computing," *8th International Workshop on Field-Programmable Logic and Applications*, Berlin, 1998.
- [20] S.C. Goldstein, et. al., "PipeRench: A Reconfigurable Architecture and Compiler," *IEEE Computer*, vol. 33, pp. 70-77, 2000.
- [21] Z. A. Ye, et. al., "CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit," *27th Annual International Symposium on Computer Architecture (ISCA)*, Vancouver, British Columbia, 2000.
- [22] V. K. Prasanna et. al., "Mapping Applications onto Reconfigurable Architectures using Dynamic Programming," *Military and Aerospace Applications of Programmable Devices and Technologies*, Laurel, Maryland, 1999.
- [23] Annapolis Microsystems Inc., "Overview of the WILDFORCE, WILDCHILD and WILDSTAR Reconfigurable Computing Engines," Annapolis Microsystems Inc., Maryland 1998.
- [24] Y. Li, et. al., "Hardware-software co-design of embedded reconfigurable architectures," *37th Design Automation Conference (DAC)*, Los Angeles, CA, 2000.
- [25] Jack W. Crenshaw, "MATH toolkit for REAL-TIME Programming". Kansas: CMP Books, Inc., 2000.
- [26] K. Kennedy and R. Allen, "Automatic Translation of FORTRAN programs to vector forms". *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 4, pp. 491-542, October 1987.
- [27] A. DeHon and J. Wawrzynek, "Reconfigurable Computing: What, Why and Implications for Design Automation," presented at *36th Design Automation Conference '99*, New Orleans, Louisiana, 1999.
- [28] R. D. Hudson, D. Lehn, J. Hess, J. Atwell, D. Moye, K. Shiring, and P. M. Athanas, "Spatio-Temporal Partitioning of Computational Structures onto Configurable Computing Machines," presented at *SPIE*, Bellingham, WA, 1998.
- [29] Mary Hall, Pedro Diniz, Kiran Bondalapati, Heidi Ziegler, Philip Duncan, Rajeev Jain and John Granacki, "DEFACTO: A Design Environment for Adaptive Computing Technology," presented at *6th Reconfigurable Architectures Workshop*, San Juan, Puerto Rico, 1999.
- [30] K. Bondalapati, G. Papavassilopoulos and V. K. Prasanna, "Mapping Applications onto Reconfigurable Architectures using Dynamic Programming," presented at *Military and Aerospace Applications of Programmable Devices and Technologies*, Laurel, Maryland, 1999.

- [31] M. Kaul, R. Vemuri, S. Govindarajan and I. E. Ouais, "An Automated Temporal Partitioning and Loop Fission approach for FPGA Based Reconfigurable Synthesis of DSP Applications," presented at *36th Design Automation Conference*, New Orleans, Louisiana, 1999.
- [32] J. B. Peterson, R. B. O' Connor and P. M. Athanas, "Scheduling and Partitioning ANSI-C Programs onto Multi-FPGA CCM Architectures," presented at *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, 1996.
- [33] J. Frigo, M. Gokhale, and D. Lavenier, "Evaluation of the streams-C C-to-FPGA compiler: An applications perspective," presented at *9th International Symposium on Field Programmable Gate Arrays*, Monterey, CA, 2001.
- [34] Kiran Bondalapati and Viktor K. Prasanna, "Mapping Loops onto Reconfigurable Architectures," presented at *International Workshop on Field Programmable Logic and Applications*, Estonia, 1998.
- [35] Timothy J. Callahan and John Wawrzynek, "Adapting Software pipelining for reconfigurable computing," presented at *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, San Jose, CA, 2000.
- [36] Z. A. Ye, N. Shenoy and P. Banerjee, "A C Compiler for a processor with a reconfigurable functional unit," presented at *ACM/SIGDA Symposium on Field Programmable Gate Arrays*, Monterey, CA, 2000.
- [37] Saman Amarasinghe, Anant Agarwal, Rajeev Barua, Matthew Frank, Walter Lee, Vivek Sarkar, Devabhaktuni Srikrishna, and Michael Taylor, "The RAW compiler project," presented at *2nd SUIF Compiler Workshop*, Stanford, CA, 1997.
- [38] D. C. Cronquist, P. Franklin, S. G. Berg and C. Ebeling, "Specifying and Comiling Applications for RaPiD," presented at *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, 1998.
- [39] Jack W. Crenshaw, *MATH toolkit for REAL-TIME Programming*. Kansas: CMP Books, Inc., 2000.

ASAP Scheduling

Inputs:

- A flat data flow graph, G , with annotated nodes³.

Constraints:

- Respect data dependences and program semantics.

Outputs:

- An execution schedule⁴.

Method:

```
ASAPSchedule( $G$ ) {  
     $Cycle = 0$ ;  
    While ( $G$  is not empty) {  
        Determine Ready Ops5,  $Op_{ready}$ ;  
        Randomly pick an op,  $O$ , from  $Op_{ready}$ ;  
        Schedule  $O$  for execution at  $cycle$ ;  
         $Cycle = Cycle + O$ 's execution latency;  
        Remove  $O$  from the graph;  
    }  
}
```

³ An *annotated node* is defined to be a node whose operation latency is available.

⁴ An *execution schedule* specifies which clock cycle each node in the graph will begin execution.

⁵ A *ready op* is defined to be a node (or operation) whose data dependences have been satisfied.

Windowing Loop Optimizations

Common Sub-Expression Elimination

Inputs:

- Data flow graph, G , generated as described above.

Outputs:

- Optimized graph, G' .

Constraints:

- Maintain program semantics

Method:

```
CSE( $G$ ) {  
     $Change = TRUE$ ;  
    While ( $Change$  is  $TRUE$ ) {  
         $Change = FALSE$ ;  
        Find a set of identical nodes6,  $S$ ;  
        If ( $S$  is not empty) {  
            From  $S$ , find node  $N$  from the earliest iteration.  
            Replace every node in  $S$  (except  $N$ ) as follows:  
                Delete all input edges coming into the node  
                Replace every output edge going out of this node  
                    with an edge that starts at  $N$   
                    and ends where the replaced edge ends.  
                Delete the node itself  
             $Change = TRUE$ ;  
        }  
    }  
}
```

Dead Code Elimination

Inputs:

- Data flow graph, G' , generated after performing common sub-expression elimination.

Outputs:

- Optimized graph, G'' .

Constraints:

- Maintain program semantics

Method:

```
DeadCodeElimination( $G$ ) {  
     $Change = TRUE$ ;
```

⁶ Two nodes are identical if the nodes represent the same operation and their operands are the same. Note that this does not apply to loop nodes.

```

While (Change is TRUE) {
    Change = FALSE;
    Find node, N that is not a data write-back node and
        that has no output edges.
    If (N exists) {
        Delete all incoming edges of N.
        Delete N.
        Change = TRUE;
    }
}

```

Extended List Scheduling Algorithm

Inputs:

- Data Flow Graph of the windowing loop's body, G , where each node is annotated with its execution schedule (RC Array context mode information), and resource requirements.

Outputs:

- Execution schedule, S , for an iteration of the windowing loop.

Constraints:

- Only 8 resources (rows) available each cycle
- All operations scheduled in a given cycle must execute in the same RC Array mode (either row or column).

Method:

```
OpScheduling( $G$ ) {  
    Cycle = 0;  
    Resource Usage Vector,  $V = 0$ ;  
    RC Array Execution Mode Vector,  $CM = \text{undefined}$  for all cycles;  
    While ( $G$  is not empty) {  
        Determine list of Ready Ops7,  $U$ ;  
        List of examined ops,  $E = 0$ ;  
        Do {  
            Determine op (not present in  $E$ ),  $op$ , with largest slack8;  
            If (resource availability9 to execute  $op$  not satisfied) {  
                 $E = E \cup op$ ;  
                Break;  
            }  
            If (all alternate schedules of  $op$  produce context clashes10) {
```

⁷ A *ready op* is defined to be a node (or operation) whose data dependences have been satisfied.

⁸ *Slack* of an operation is defined as the longest path from the operation to then of the dataflow graph. This can be done by performing ALAP (as late as possible) scheduling; and subtracting the current cycle from the node's position in the ALAP schedule.

⁹ Resource Availability for an operation can be checked by ensuring that the operation's resource requirements are less than the available resources (resource usage vector V) in every cycle starting from the current cycle to the (current cycle + operation's execution latency) cycle.

```

         $E = E \cup op;$ 
        Break;
    }
    Schedule  $op$  for execution in cycle  $Cycle$ ;
     $L = op$ 's execution latency;
     $R = op$ 's resource requirements;
    For  $I = Cycle$  to  $(Cycle + L)$  {
         $V[I] = V[I] + R;$ 
         $CM[I] = op$ 's execution mode in cycle  $I$ ;
    }
    Remove  $op$  from  $G$  and  $U$ ;
} While ( $U$  or  $E$  has changed);
Cycle = Cycle + 1;
}
}

```

¹⁰ Checking for RC Array context code clashes is simple – in each cycle, the RC Array is said to be executing in a certain mode (this is reflected in vector CM in the algorithm). If the op 's execution schedule matches with the RC Array's execution mode in every cycle starting from current cycle to the (current cycle + operation's execution latency) cycle, then there is no conflict. Initially the RC Array's mode is set as *undefined* in every cycle. A comparison of the op 's context with such an *undefined* value always results in "no conflict".

Clique Partitioning Algorithms

Clique Partitioning

Inputs:

- A graph, $G(V, E)$

Outputs:

- Set of partitioned clusters, C .

Constraints:

- Each cluster is a clique

Method:

```
CliquePartitioning( $G$ ) {  
     $C = 0$ ;  
    While ( $G$  is not empty) {  
         $K = \text{MAX\_CLIQUE}(G)$ ;  
         $C = C \cup K$ ;  
        Remove  $K$  from  $G$ ;  
    }  
}
```

Although this algorithm is pretty straightforward, finding the maximum clique¹¹ (MAX_CLIQUE) is known to be an NP-complete problem. The compiler uses a heuristic to solve the problem, as an optimal solution is not essential.

MAX CLIQUE HEURISTIC

Inputs:

- Graph $G(V, E)$

Outputs:

- Clique C

Non-strict Constraints:

- C is a (near-) maximum clique.

Method:

```
Max_Clique_Heuristic( $G, V$ ) {  
     $C = \text{Vertex, } u$ , with largest out-degree;  
    Remove  $u$  from  $V$ ;  
    Remove from  $V$  all vertices that are not adjacent to  $u$ ;  
    While ( $V$  is not empty) {  
         $N = \text{List of nodes with largest out-degree, and connected}$   
        To every node in  $C$ ;  
        If ( $N$  is empty) {  
            exit;  
        }  
    }
```

¹¹ The *maximum clique* is the biggest fully connected sub-graph in the source graph.

Select node n from N such that the sum of the weights on the edges between n and every node in C is highest.
 Remove n from V ;

}
 }

Compile-time Area Estimation for LUT-based FPGAs

Dhananjay Kulkarni
Walid A. Najjar

University of California, Riverside
Dept. of Computer Science
Riverside, CA 92521
{kulkarni, najjar}@cs.ucr.edu

Robert Rinker

University of Idaho
Computer Science Dept.
Moscow, ID 83844
rinker@cs.uidaho.edu

Fadi J. Kurdahi

University of California, Irvine
Dept. of Electrical and
Computer Engineering
Irvine, CA 92717
kurdahi@ece.uci.edu

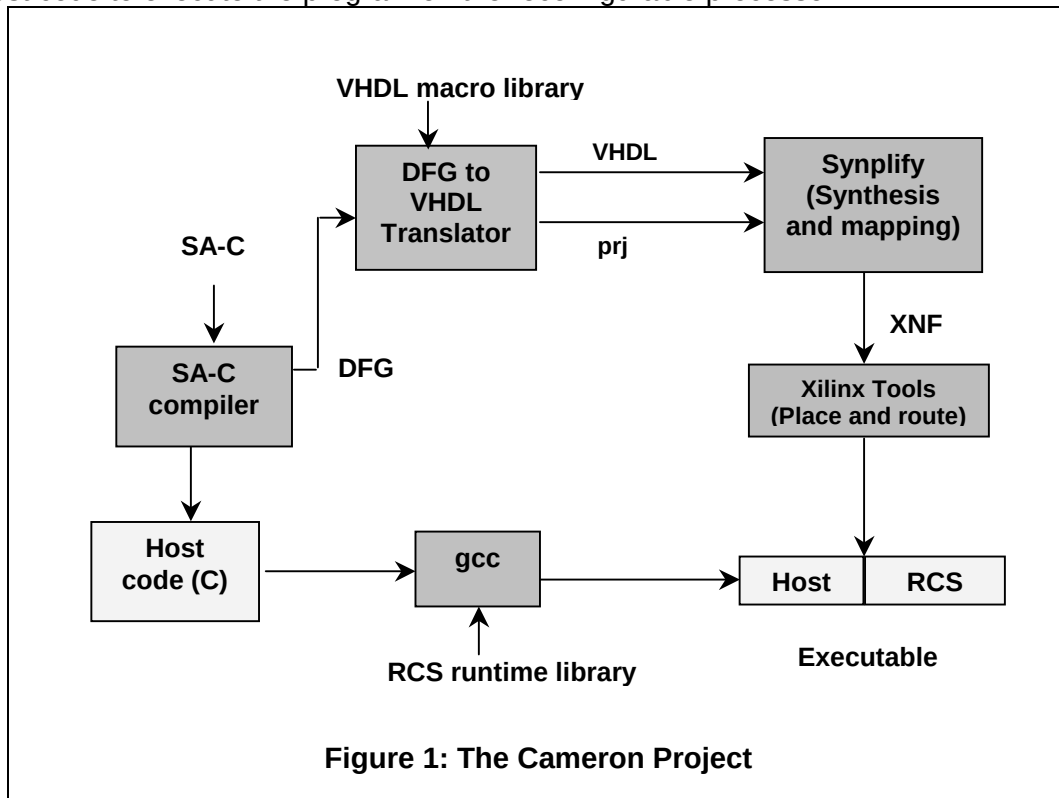
ABSTRACT

The Cameron Project has developed a system for compiling codes written in a high-level language called SA-C, to FPGA-based reconfigurable computing systems. In order to exploit the parallelism available on the FPGAs, the SA-C compiler performs a large number of optimizations such as full loop unrolling, loop fusion and strip-mining. However, the area on the FPGA is limited and therefore the compiler needs to know the effect of compiler optimizations on the FPGA area. In this paper we present a compile-time area estimation technique to guide the SA-C compiler optimizations. We demonstrate our technique for a variety of benchmarks written in SA-C. Experimental results show that our technique achieves an error rate of 2.5 % for small image-processing operators and 5.0 % for larger benchmarks, as compared to the synthesis/mapping. The estimation time is in the order of 1 millisecond as compared to order of minutes for a synthesis tool.

1 INTRODUCTION

Even though FPGAs present a computational density advantage over general-purpose processors [1] for certain applications, the biggest obstacle to the widespread use of FPGA-based reconfigurable computing lies in the difficulty of programming them. A typical design cycle for programming FPGAs starts with a behavioral or structural description of the design, using hardware description languages such as VHDL¹ or Verilog. These hardware description languages require the programmer to explicitly handle the issues of timing and synchronization of the complete design. Most application program developers are not experts in hardware description languages and are more familiar with the algorithmic programming paradigm. The goal of the Cameron Project [2, 3] is to bridge the semantic gap between applications and FPGAs by developing an algorithmic language called SA-C (Single Assignment C, pronounced *sassy*), that is suitable for mapping image processing applications onto reconfigurable systems. The ease of programming in SA-C makes FPGAs and other adaptive computer systems available to more application programmers.

As shown in Figure 1, the SA-C compiler translates the high-level SA-C code into dataflow graphs, which can be viewed as abstract hardware circuits without timing information. The SA-C compiler applies extensive expression, loop and array optimizations at the dataflow graph level before generating a VHDL hardware description. These compiler optimizations might lead to a different structural design than what is implied in the SA-C source program. The SA-C DFG to VHDL translator [2, 4] translates the dataflow graph into VHDL. The VHDL code is then synthesized, placed and routed using commercial synthesis tools. The SA-C compiler also automatically generates all the necessary run-time host code to execute the program on the reconfigurable processor.



¹ VHDL stands for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language

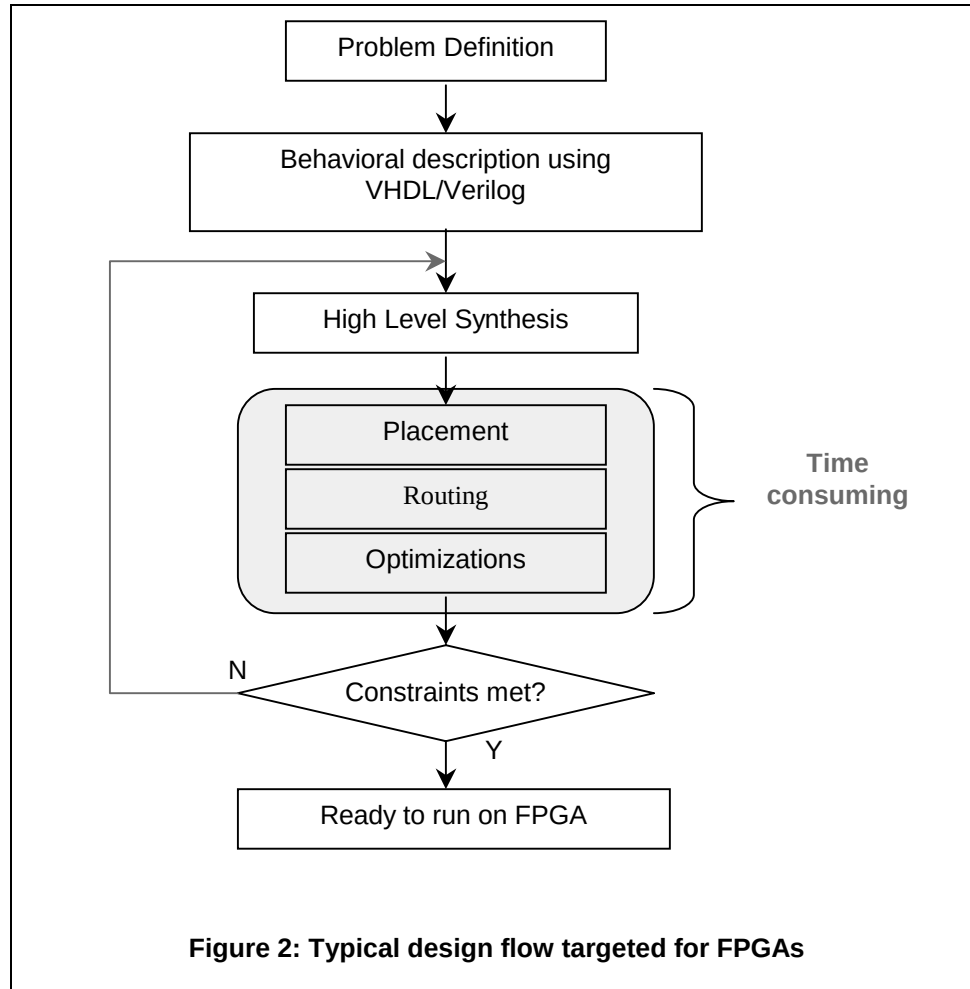
Starting from the SA-C source program, the resource and timing estimations for a design are available after the synthesis phase. Moreover, the compiler optimizations have a large potential impact in the resources used on the FPGA. For example, unrolling a large loop too many times might overflow the capacity of the FPGA. This means that prior to the synthesis and mapping, the compiler has no idea if the design can fit on the FPGA or not. Traditionally, the user himself is involved in making sure that the computation fits onto the FPGA. This might involve several iterations of compile/synthesize/place and route before the best fit and performance can be achieved. However, the design space can be exploited more efficiently if area estimations are available before the synthesis phase. In this paper we introduce a compile-time estimation approach in the SA-C compilation process. Our technique makes the resource estimations available before the DFG to VHDL translation. The estimation serves as a feedback to the compiler to aid in further optimizations. Experimental results show that our technique achieves an error rate of 2.5 % for small image-processing operators and 5.0 % for larger benchmarks, as compared to the synthesis and mapping. Our estimator takes 5 to 6 orders of magnitude less time to compute than the commercial synthesis tools.

The rest of the paper presents the motivation in section 2, followed by the details of the SA-C compiler and the dataflow graphs in section 3. Compile-time estimation approach is presented in section 4. Experimental results are presented in section 5. References to related work are given in section 6 and section 7 concludes and describes some future work.

2 MOTIVATION

In a typical design flow targeted for FPGAs, logic synthesis is performed on the behavioral description, followed by partitioning, placement and routing. Even though this cycle shortens the cycle for mapping the design on the FPGA under the given constraints, the mapping, placement and routing are very complex and time-consuming phases. As shown in Figure 2, these phases form the main bottleneck in the design process. Inability to meet the constraints leads to the modification of the design and repeating the entire cycle.

In practice, reporting accurate resource usage of the final design is very important. However, when the design process starts at a much higher level, estimation can play a vital role in guiding the performance of the final design. Estimations applied at higher levels can obviate the repeated use of time-consuming phases to examine if the design meets the required constraints or not. This motivates us to look at a compile-time area estimation approach in the SA-C compilation process. We apply the estimation technique even before the VHDL description is generated. The estimation serves as a feedback to aid in compiler optimizations. Section 4 describes this technique in complete detail.



3 SA-C COMPILER AND DATAFLOW GRAPHS

3.1 Unique Features of SA-C

SA-C is designed specifically to make it easier for the compiler to analyze the SA-C code and extract both fine-grained and coarse-grained parallelism. SA-C is an expression-oriented, single-assignment (functional) language, which can be translated into hardware descriptions. The SA-C syntax is closely related to C; however, there are significant differences as well. The differences are mostly due to its use as a hardware generation language. Unique features of the SA-C language are as follows:

- It provides a flexible *type* system, including signed and unsigned integers of any bit-width, as well as fixed-point numbers.
- True multi-dimensional arrays, with a specific size and shape.
- No pointers and no recursion, designed to prevent programmers from applying Von Neumann models² that do not map well onto FPGAs.

² Von Neumann model is the traditional computer architecture model, which consists of the input unit, output unit, ALU, control unit and the memory unit.

- Loop generators, which are usually used in place of the more traditional “loop index used as an array subscript” to perform operations on arrays.
- Reduction operators, which perform common operations on the data produced in loop bodies, such as array sum and histogram. This allows programmer access to efficient VHDL implementations of these operations.

A simple SA-C program is shown in Figure 3. This program accepts a 2-D array (named **Arr**) of 8-bit unsigned integers (i.e. of type **uint8**) as input. A window generator statement (**for window...**) extracts all 3x3 sub arrays from the image-array and sums the elements in each sub-array. A new array (named **r**) is formed such that each element is either the sum of the corresponding window or, if the sum is greater than 100, the sum minus 100. Upon compilation, an intermediate form of the program, called a dataflow graph (DFG) is produced.

```
uint8[:,:] main (uint8 Arr[:,:]) {
    uint8 r[:,:] =
        for window W[3,3] in Arr {
            uint8 s = array_sum(W);
            uint8 v = if(s>100) return (s-100)
                      else return (s);
        } return(array(v));
} return(r);
```

Figure 3: Example SA-C program

3.2 SA-C Compiler Optimizations

The SA-C compiler performs numerous optimizations before generating the DFG. Several traditional optimizations minimize the calculations required by the hardware; these include code motion, constant folding, array and constant value propagation and common subexpression elimination. Function inlining and loop unrolling increase parallelism and often enable other optimizations. Some of these optimizations (e.g. strip-mining and loop fusion) improve the performance of the design at the cost of area. Since the resource area is limited, the compiler needs to know the effect of applying optimizations on the resource usage of the final design.

The optimizations consist of:

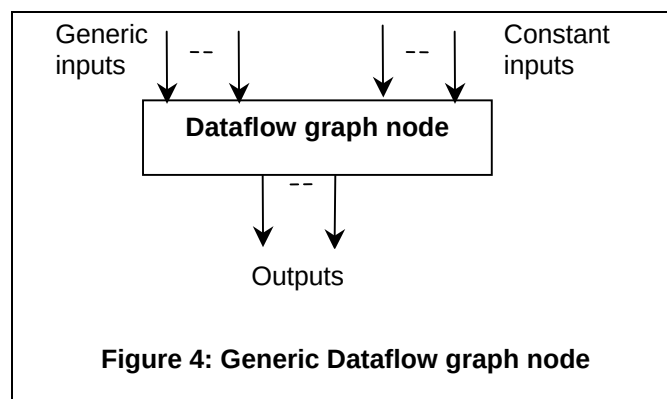
- *Constant Switch* takes a switch graph with a constant key value and replaces the graph with the case graph corresponding to the key.
- *Size Propagation* analyzes loops and arrays, propagating extents information through the graph in order to enable full loop unrolling.
- *Array Value Propagation* replaces an array reference with its corresponding expression when possible.
- *Constant Folding* replaces an operator with a value if the operator's inputs are constants.
- *Identities* applies various algebraic identities and other rules. E.g., $a * 1$ is replaced with a .
- *Strength reduction* replaces certain operators with simpler equivalents.
- *Dead Code Elimination* removes code if its values are not used in producing the output results.
- *Full Loop Unrolling* replaces a loop with multiple explicit loop bodies if the number of loop iterations is statically known.

- *Common Subexpression Elimination* eliminates redundant nodes if they compute the same value
- *Array Reference Elision* replaces two array references, if the first takes a slice and it feeds only the second, with a single reference.
- *Loop Fusion* can under certain circumstances, fuse two consecutive loops into one loop.
- *Window to Element Gen* converts window generators that take single element windows to element generators.
- *Push Array Casts* can, under certain conditions, push array casts to the places in which the array is being referenced.
- *Invariant Code motion* hoists invariant code out of loop bodies.
- *Function inlining* replaces a function call with the function body.
- *Lookup Tables* replaces function calls with references to a lookup table that holds pre-computed values.
- *N-dimensional strip-mining* strip-mines a loop by enclosing it in another loop that reads chunks of data with which to feed the inner loop.

The optimizations, some usual and some specific to SA-C interact each other. A sequence of a subset of optimizations form cyclic opts, which repeated in a sequence attain stability.

3.3 Dataflow Graphs

SA-C data flow graphs [5] are internal representations used by the SA-C compiler. The SA-C compiler translates the part of the program targeted to the reconfigurable system to dataflow graphs: a low-level, non-hierarchical and asynchronous program representation. As shown in Figure 4, DFGs can also be viewed as abstract hardware circuit diagrams without timing considerations. The functional elements of dataflow graph are nodes, each characterized by a node type, one or more input ports and one or more output ports. A dataflow graph node may fire when its firing rule is met and the act of firing consumes one token from each input, and the value of its output is the function of only these inputs.



Depending on the node type, the SA-C dataflow node functions can be classified into the following six categories.

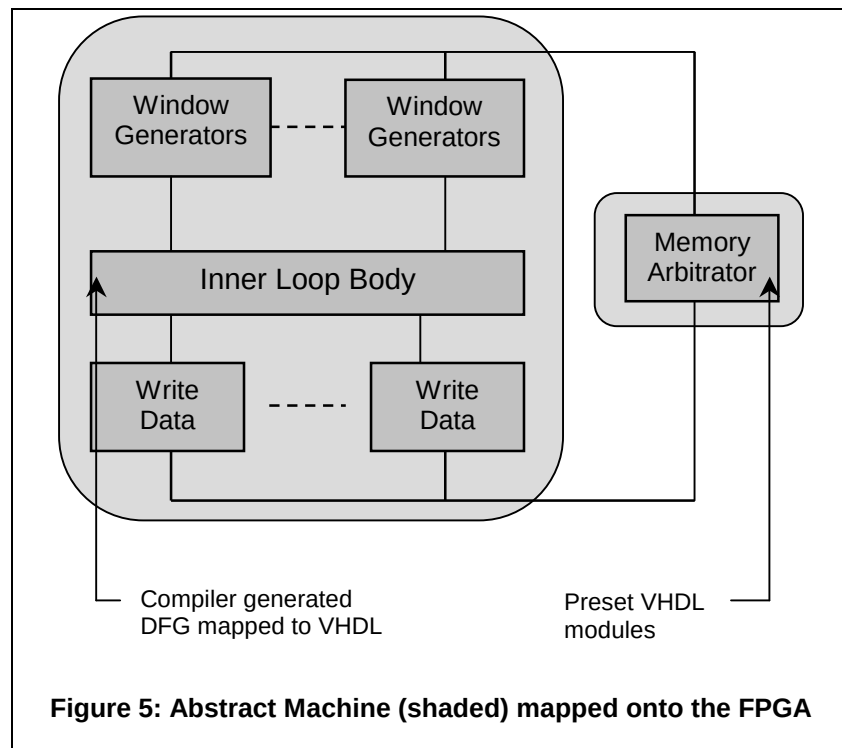
1. Arithmetic: perform common functions such as addition and logical operations
2. Bit: perform shift, sub-word selection and width changing operations
3. Selector: perform choosing one from the number of inputs
4. Generator: take token sequences and use them to specify output tokens
5. Reduction: take token sequences and reduce or store them
6. I/O: handle the interface between the dataflow graph and the outside world

Most of the SA-C DFG nodes can be implemented as combinational logic, except generator and reduction nodes, which implemented as sequential processes. These operations require multiple clock cycles, several state machines and coordination between the generators and the reduction nodes.

3.4 The Abstract Machine

Traditional processors provide a small set of instructions to the user to write applications that can be executed on the hardware. FPGAs on the other hand consist of an amorphous mass of configurable cells that can be interconnected in a large number of ways. To limit the number of ways of mapping a design onto the target FPGA, an *abstract machine* is defined. The *abstract machine* shown in

Figure 5 serves as a reasonable target for the compiler during the translation process. The compiler generates some of these modules at compile-time, while others are hand coded in VHDL.



The DFG to VHDL translator interfaces the appropriate signals between the various modules to build the complete configuration. The *abstract machine* describes interaction and the flow of data between the various modules. All the modules shown in

Figure 5 are mapped onto a single processing element on the FPGA (1PE model). The SA-C dataflow graph describes the Inner Loop Body (ILB). The *memory arbitrator* is part of the *glue code* for the system and its main role is to do the initialization, scheduling for memory access and handle resource contention. The *memory arbitrator* and the other interface modules are not captured by the DFG.

The ILB is purely combinational. The *window generator*, *write data* and *memory arbitrator* modules are sequential processes. Ideally, the resource usage for sequential circuits is calculated after applying scheduling, resource allocation algorithms. Since our emphasis is to do quick compile-time estimation, we only focus on using techniques of lower complexity. We pre-compute the LUT usage of the preset

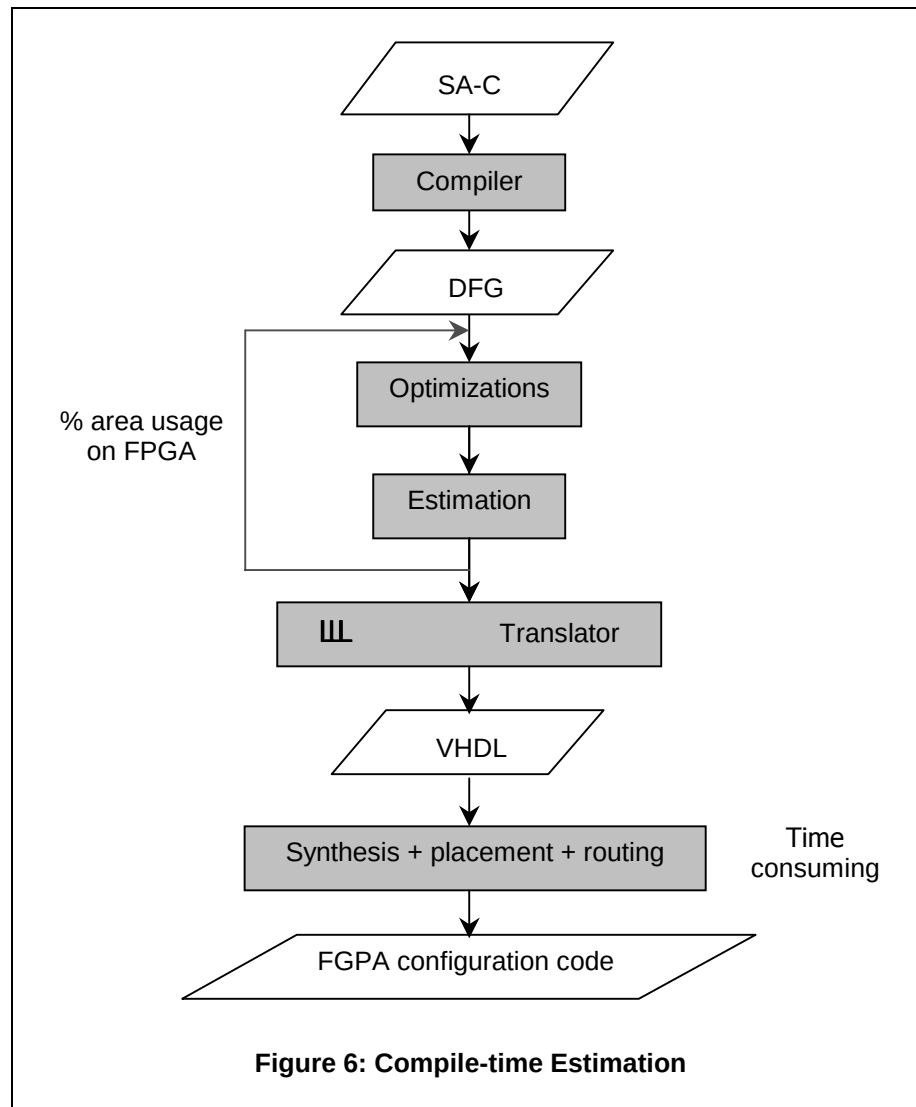
VHDL modules, because their design does not change for each SA-C program compilation. For all other modules we apply the estimation technique based on general formulae. Also, since the compiler optimizations mainly affect the ILB, we estimate the effect such optimizations on the size of the ILB.

Even though the benchmarks considered in this paper are compile for the machine model shown in Figure 5, the estimation technique can easily be adapted for a variety of machine models, as long as the SA-C DFG is used as the intermediate representation.

4 ESTIMATION APPROACH

4.1 Compile-time Estimation

The job of the SA-C compiler and synthesis tools is to improve the overall performance of the design in terms of area and/or timing under the given constraints. In order to exploit the parallelism available on the FPGAs, the SA-C compiler performs a large number of optimizations. However, the resources on the FPGA are limited and therefore the compiler needs to know the effect of the compiler optimizations on the resource usage of the design.



As shown in Figure 6, we apply the estimation model at an intermediate format used by the SA-C compiler and feed the estimation results back to the compiler to aid in further optimizations. Any modification to the design can be incorporated without performing the synthesis and mapping of the design. Our focus is to provide quick and relatively accurate resource usage estimation of a SA-C program, where resources are defined to be number of look-up tables (LUTs). Even a number of alterations of the design or the source program do not undermine the advantages of using this scheme. Thus, the SA-C compiler can make full use of the estimation feedback to translate the SA-C source program into an efficient design that can fit on the FPGA.

4.2 Estimation Methodology

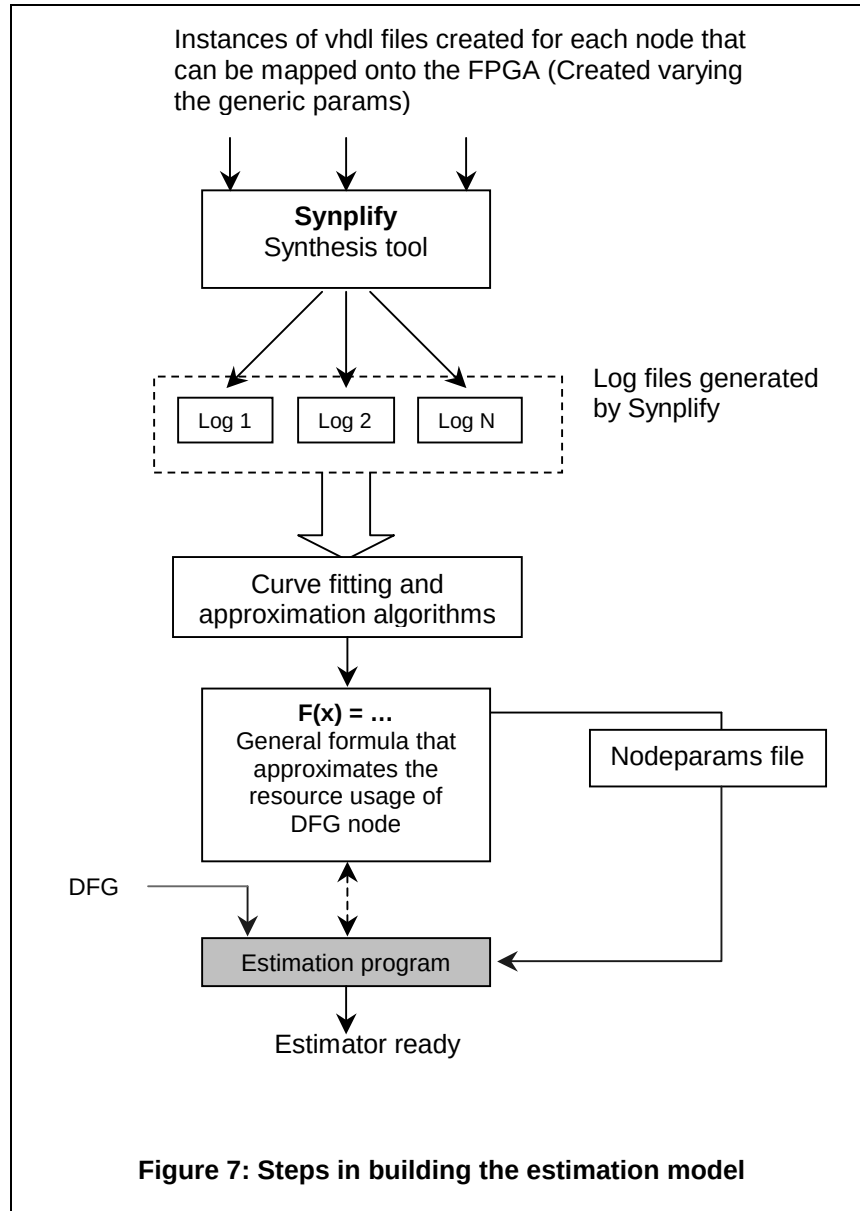
Since the DFG does not contain any low-level structural or timing information, our estimation tool does not incorporate scheduling, resource allocation and binding algorithms. Our method is to pre-compute a set of general formulae for all types of DFG nodes and at compile-time, use these formulae to estimate the resource usage of the SA-C program. The general formula and its coefficients are stored in a data file called *nodeparams*. Input to the estimator is the SA-C DFG and the *nodeparams* file. Using the information in the *nodeparams* file, the estimator calculates the resource usage of the dataflow graph. The complexity of the estimation algorithm is $O(n)$, where n is the number of nodes in the DFG.

Figure 7 shows the steps involved in building the estimation model. Initially, we vary the generic parameter of each type of DFG node and create the corresponding VHDL instances. In the next step, we synthesize these VHDL files and record the estimations reported by Synplify³. The general formula for each DFG node is obtained by doing a regression analysis⁴ on the estimation values reported by the synthesis tool. Simple linear regression as well as nonlinear regression is done using NLREG⁵ [16], which can virtually fit any function to a set of data values.

³ Synplify is a widely used logic synthesis tool developed by Synplicity Inc.

⁴ Regression analysis is a general mathematical technique used to fit a curve to the given data.

⁵ NLREG is an extremely powerful statistical regression analysis tool developed by Phillip H. Sherrod



At compile-time, we only execute the estimation program (shaded in Figure 7) to get the estimated area usage of the corresponding DFG.

4.3 Approximation formulae

The general formulae are categorized as follows. Parameter y gives the estimated LUT usage and is a function of bit-width (x) and/or num-of-vals (z). C , p_0 , p_1 , p_2 , c_0 , c_1 are the coefficients that are recorded in the *nodeparams* file.

- **Constant: $y = C$**

These nodes synthesize as signals and provide an interface with the outside world or memory. They consume a fixed amount of resource on the FPGA.

Table 1: Nodes using constant approximation

Name	Inputs	Outputs	Description
INPUT	1	1	Get a value from the input channel constant specified by I_0
OUTPUT	2	0	Take a value from token from I_0 and an output channel constant specified by I_1

- **Linear: $y = p_0 + p_1 * x$**

Most of the arithmetic nodes belong to this category. The resource usage is a linear function of the bit-width (x).

Table 2: Nodes using linear approximation

Name	Inputs	Description
UADD/IADD	2	Unsigned/signed addition of I_0 and I_1
USUB/ISUB	2	Unsigned/signed subtraction of I_0 and I_1
ULT/ILT	2	Unsigned/signed $<$ comparison of I_0 and I_1
ULE/ILE	2	Unsigned/signed $\leq$ comparison of I_0 and I_1
UGT/IGT	2	Unsigned/signed $>$ comparison of I_0 and I_1
UGE/IGE	2	Unsigned/signed $\geq$ comparison of I_0 and I_1
UEQ/IEQ	2	Unsigned/signed equality comparison of I_0 and I_1
UNE/INE	2	Unsigned/signed inequality comparison of I_0 and I_1
BIT-AND	2	Bit-wise AND of I_0 and I_1
BIT-OR	2	Bit-wise OR of I_0 and I_1
BIT-EOR	2	Bit-wise exclusive OR of I_0 and I_1

DFG nodes shown in Table 2 perform arithmetic and bit-manipulation operations. The arithmetic nodes implicitly treat their inputs as either signed or unsigned integers. Figure 8 shows the LUT usage of UADD node.

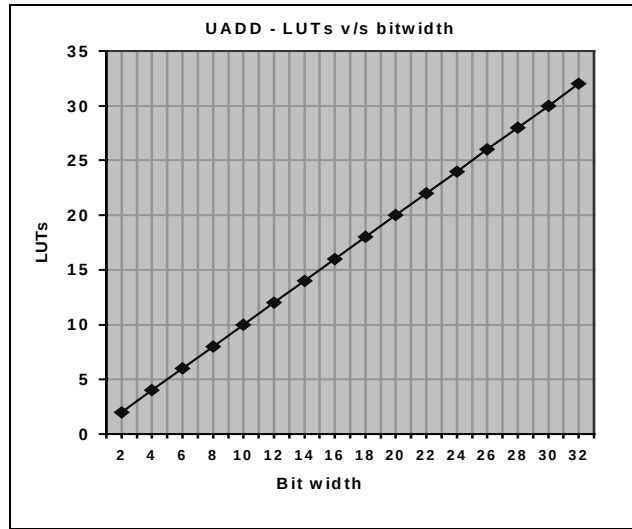


Figure 8: Linear approximation $y = p_0 + p_1 \cdot x$, where $p_0=0$ and $p_1=1$

- **Quadratic:** $y = p_0 + p_1 \cdot (x - p_2)^2$

The resource consumption of the nodes in this category is a quadratic function of the form $y = p_0 + p_1 \cdot (x - p_2)^2$, where x is the bit-width of the node.

As shown in Figure 9, multiplication is an expensive operation and consumes more than 1000 LUTs for operands that are more than 32-bit wide.

Table 3: Nodes using quadratic approximation

Name	Inputs	Description
UMUL	2	Unsigned multiplication of I_0 and I_1
IMUL	2	Signed multiplication of I_0 and I_1

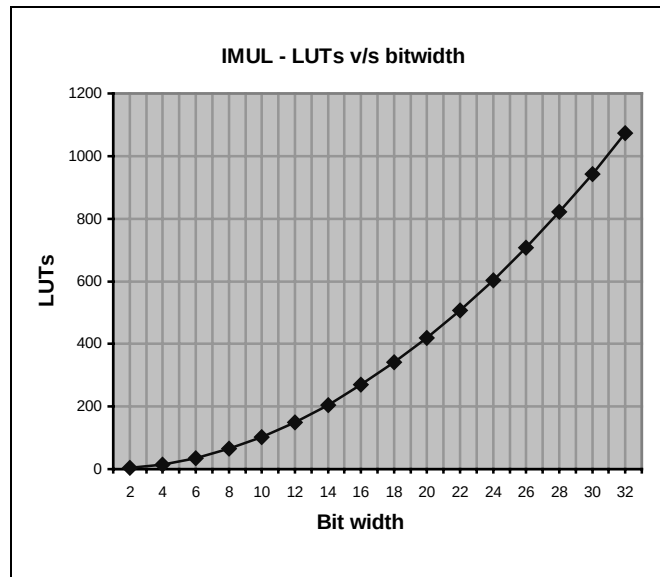


Figure 9: Quadratic approximation $y = p_0 + p_1 * (x - p_2)^2$, where $p_0 = -2.509$, $p_1 = .041$, $p_2 = -0.125$

- **BiProduct: $y = (z-p_0) * (x-p_1) + p_2$**

The resource usage of nodes in this category depends on the bit-width and the num-of-vals parameters. The LUT usage is calculated using $y = (z-p_0) * (x-p_1) + p_2$, where z and x are the num-of-vals and the bit-width respectively. Table 4 lists the multi-input arithmetic nodes that use biproduct approximation. These nodes allow an arbitrary number of input values, each with an associated Boolean mask value.

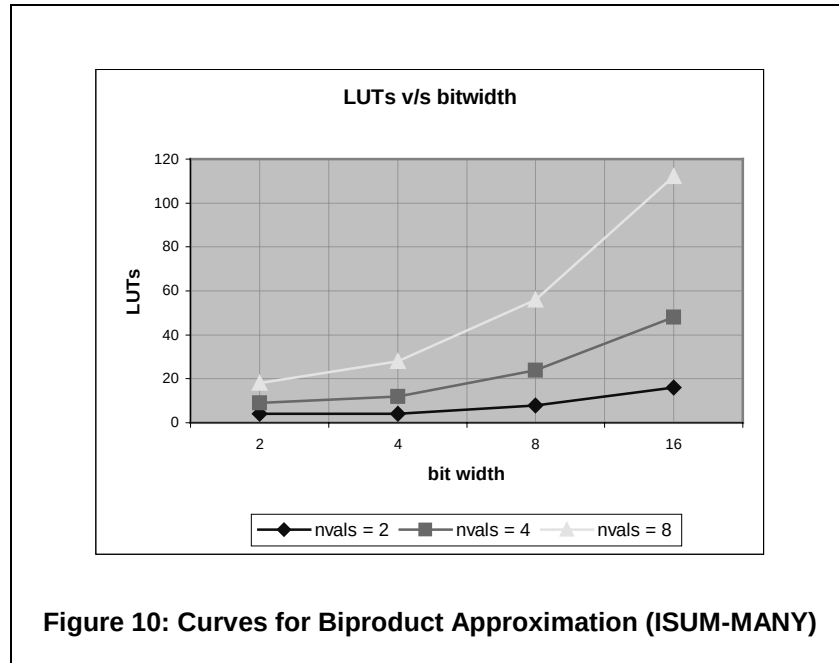
Table 4: Nodes using biproduct approximation

Name	Inputs	Description
ISUM-MANY	2	Sum the unsigned input values; each input pair represents a value and a Boolean mask; the arithmetic is performed at the output port's bit width
USUM-MANY	2	Sum the signed input values; each input pair represents a value and a boolean mask; arithmetic is done at the output port's bit width

Table 5: Biproduct approximation for ISUM-MANY

Num-of-vals (z)	Bit-width (x)	LUTs
2	2	4
2	4	4
2	8	8
2	16	16
4	2	9
4	4	12
4	8	24
4	16	48
8	2	18
8	4	28
8	8	56
8	16	112

Table 5 shows the resource usage for the ISUM-MANY node when the number of input values is varied from 2 to 8 and the bit-width is varied from 2 to 16. Figure 10 shows the family of curves plotted for biproduct approximation.



- **MultiLinear2: $y = c_0 + (c_1 * x/2) (z/2 - 1)$**

The resource usage of these nodes depends on the two generic parameters bit-width (x) and num-of-vals (z). Table 6 lists the multi-input logic operator nodes that use multilinear2 approximation. These nodes allow an arbitrary number of input values, each with an associated Boolean mask value.

Table 6: Nodes using multilinear2 approximation

Name	Inputs	Description
AND-MANY	var	'and' the input values; each input pair represents a value and a boolean mask
OR-MANY	var	'or' the input values; each input pair represents a value and a boolean mask

Table 7: Multilinear2 approximation for AND-MANY

Num-of-vals (z)	Bit-width (x)	LUTs
2	2	4
2	4	4
2	8	8
2	16	16
4	2	6
4	4	12
4	8	24
4	16	48
8	2	10
8	4	20
8	8	40
8	16	80

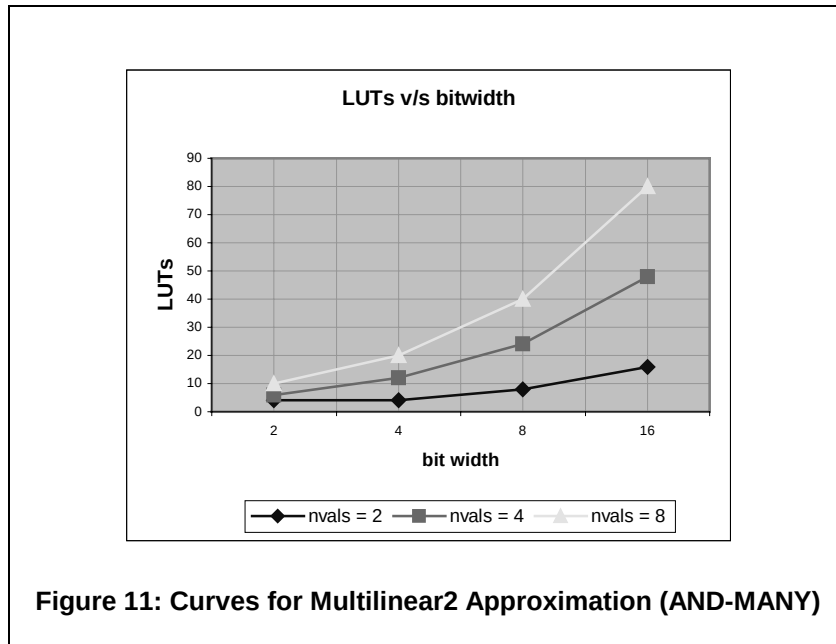


Table 8 shows the LUT usage of AND_MANY node, when the number of inputs is varied from 2, 4 to 8. Figure 11 shows the LUT usage curves for AND-MANY node.

4.4 Estimation Heuristics

The synthesis process used in commercial synthesis tools is divided into two steps. In the first step, the synthesis tool identifies all the high level structures and converts them into a technology independent representation called the *netlist*. The second phase, known as technology mapping [6] or library binding makes use of the architecture dependent constraints, resource library and the *netlist* to map the design onto the target hardware. Most synthesis tools do timing and resource optimization, wherein they try to make the design as small (in terms of area) and as fast (in terms of timing) as possible. These tasks are very complex, because they entail solution of intractable problems. Our approach does not address the solution to such problems. Our emphasis is to provide quick but relatively accurate estimation of the resource area and leave the complexity of the mapping to the synthesis phase. However, an estimator that does not account for the optimized logic would be in a large error with respect to the synthesis and mapping. Hence, we introduce heuristics in our estimation. Our heuristics are based on *structural patterns*, which are patterns that are frequently optimized during synthesis.

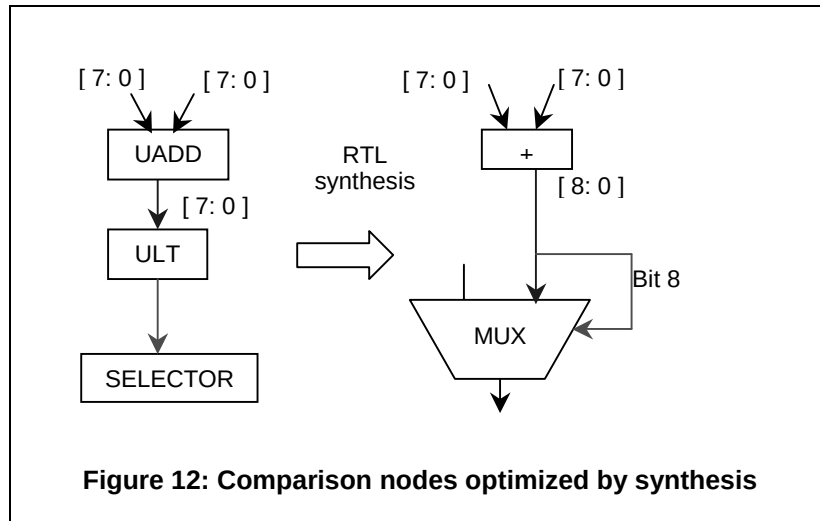
In our compile-time estimation algorithm, we check for the existence of *structural patterns* in the DFG. If a pattern is found, we apply the estimation heuristics to the pattern, instead of the individual nodes that form the pattern. This involves one pass over the SA-C dataflow graph and does not increase the complexity of the algorithm. We present some of these patterns in the following paragraphs.

- **Multiplication by constants**

When there is a multiplication operation performed in the SA-C source program, the compiler generates a UMUL node in the corresponding DFG. If the multiplication is by a constant that is power of 2, then the synthesis tool optimizes it as a shift operation. This implies that the gate level description for the UMUL node will no longer exist in the netlist. To account for this optimization, we identify the pattern at the DFG itself and incorporate the optimization by avoiding the estimation of UMUL node.

- **Comparison nodes**

Here we are mainly considering the ULT (Integer Less Than) and UGT (Integer Greater Than) nodes. As shown in Figure 12, the output of an adder is compared to check if it is less than or equal to 255 (1111111_2) and this result is used to select the right value in the SELECTOR node. Such patterns are easily optimized away during synthesis. The output of the adder is directly fed (most significant bit) to the multiplexer. To account for this optimization, we identify the pattern at the DFG itself and incorporate the optimization by avoiding the estimation of ULT node.



5 EXPERIMENTAL RESULTS

There are three sets of experiments that we wish to present in this paper. In the first set of experiments we apply our estimation method on the common image processing (IP) operators supported in SA-C. The second set of experiments address larger image processing benchmarks. In the third set we will discuss the impact of SA-C compiler optimizations on the resource usage. All benchmarks used in this paper are compiled for Annapolis Micro System Inc's WildStar board [7]. This board uses the Xilinx Virtex (XCV1000) FPGAs [8, 9], each with 1 million system gates per chip. There are 27648 4-input LUTs in each FPGA.

We show that our compile-time estimation technique works, because the estimation formulae and algorithms are low in complexity and the error rate is within acceptable limits for the SA-C compiler.

5.1 Image-processing Operators

Image processing operators [10] are common IP operations supported in SA-C. Table 8 compares our resource estimations with those obtained after synthesis and mapping.

The average execution time for the estimator is only 1 millisecond. Also, the results are relatively accurate and serve as a good estimate for the IP operators. Weighted error is as calculated as absolute error between the sum of all the Synplify estimations and the sum of Cameron estimation.

Table 8: Results for IP operators

Benchmark	Synplify estimation (# of LUTs)	Cameron estimation (# of LUTs)	% Error
AddD	1124	1150	2.31
AddM	879	917	4.32
Convolution	1783	1860	4.32
Convolution5	3235	3269	1.05
Convolutionsm	2609	2616	0.27
Dilation	1365	1451	6.30
Erosion	1373	1451	5.68
Gaussianfilter	1117	1125	0.71
Laplacefilter3x3	1198	1247	4.09
Max	821	832	1.34
Maxfilter	1407	1436	2.06
Min	867	832	4.04
Minfilter	1230	1200	2.44
Mp4	1150	1154	0.35
MultiplyM	920	958	4.13
MultiplyD	1171	1182	0.94
Prewitt	1083	1020	5.82
Prewittmag	1815	1792	1.27
Reduce	1013	1016	0.30
Robertsmag	1441	1414	1.87
Sobelmag	1943	1921	1.13
Sqrt	984	944	4.07
SubtractD	1121	1123	0.18
SubtractM	865	901	4.16
Threshold	911	903	0.88
Average Error: 2.56 %		Weighted Average Error: 0.86 %	

5.2 Image-processing Benchmarks

Larger benchmarks are written using simple image processing operators. Using two or more simple routines in a sequence gives a greater opportunity to do SA-C compiler optimizations. Since the inner loop bodies of the individual routines are transformed and restructured due to the optimizations, the resource estimation of such larger routines is an interesting task.

- *Open* is defined as *dilation* followed by *erosion* (*Close* operator is just the reverse). The *open* operator is thus a suitable candidate for loop fusion.
- *Wavelet* is a common image compression algorithm. The version implemented in SA-C is based on the Cohen-Daubechies-Feauveau wavelet algorithm [11].
- The *tridiagonal* code solves the matrix equation $[A].[X] = [B]$. The basic idea is to use the tridiagonal matrix $A[8,8]$ and the vector $X[8]$ as input, calculate B and then solve $[A].[X] = [B]$ iteratively to obtain $[X]$.

Table 9: Benchmark results

Benchmark	Synplify estimation (# of LUTs)	Cameron estimation (# of LUTs)	% Error
Open	4500	4780	6.22
Close	4500	4780	6.22
Wavelet	2172	2065	4.93
Tridiagonal	7476	7188	3.85
Average Error: 5.30 %		Weighted Average Error: 0.88 %	

Open is actually the combination of *erosion* + *dilation*, but when the two loops are fused, it runs 2 times faster than the individual routines at the expense of more area. In *wavelet* and *tridiagonal*, the loops are fully unrolled to exploit more parallelism.

Table 10: Comparison of estimation time

Benchmark	Synthesis time	Cameron estimation time
Open	3.4 mins	1 millisec
Close	3.4 mins	1 millisec
Wavelet	8.3 mins	1 millisec
Tridiagonal	10.0 mins	1 millisec
Average	6.2 mins	1 millisec

As shown in Table 10, the synthesis tool takes 6.2 minutes on an average to do the synthesis and mapping. Our estimator takes only 1 millisec to do the estimation. Thus, our estimator takes 5 to 6 orders of magnitude less time to compute than the commercial synthesis tools.

5.3 Effect of SA-C Compiler Optimizations

Example 1

In the first example, we examine the effect of strip-mining on the resource usage of the design. We use the *convolution* routine shown in Figure 13, which does a 3x3 convolution of an *image* with *kernel*. We do compile-time estimation of this SA-C code for the default case (no strip-mining) and then strip-mine the loop for windows of sizes [4,3], [5,3], [6,3], [7,3], [8,3] and [20,3] (indicated using a PRAGMA⁶).

⁶ PRAGMAs in SA-C source code allow control of individual optimizations


```

uint20[:,:] main (uint8 image[:,:], uint8 kernel[:,:])
{
    uint20 res[:,:] =
        // PRAGMA (strip-mine(4,3))
        for window win[3,3] in image
        { uint20 val =
            for elem1 in win dot elem2 in kernel
            return(sum((uint20)elem1*elem2));
        } return(array(val));
    } return (res);
}

```

Figure 13: Convolution

Table 11: Strip-mining Results

Convolution	Synplify estimate (# of LUTs)	Cameron estimate (# of LUTs)	% Error	% LUT used on Virtex XCV1000
No strip-mining	1783	1949	9.3	7.04
Strip-mine(4,3)	2609	2794	7.09	9.43
Strip-mine(5,3)	3458	3516	1.67	12.71
Strip-mine(6,3)	4269	4134	3.16	14.95
Strip-mine(7,3)	5122	4980	2.77	18.01
Strip-mine(8,3)	6709	5686	4.5	20.56
Strip-mine(20,3)	15883	14727	7.2	53.26
Average Error: 5.09 %		Weighted Average Error: 5.26 %		

Table 11 shows the results of strip-mining the *convolution* example. Loop strip-mining when followed by full loop unrolling produces the effect of multidimensional partial loop unrolling. In our example, strip-mining creates just one loop with lesser iterations than before. This increases the parallelism at the cost of area. Figure 14 shows the total resource usage of the SA-C routine on the Xilinx XCV1000 FPGA. The SA-C compiler can make use of this estimation to regulate the size of window used in strip-mining the inner loop body. If the compiler has already estimated that a different part of the program requires more than 50 % of the total LUTs, then strip-mining with a (20,3) window is not feasible.

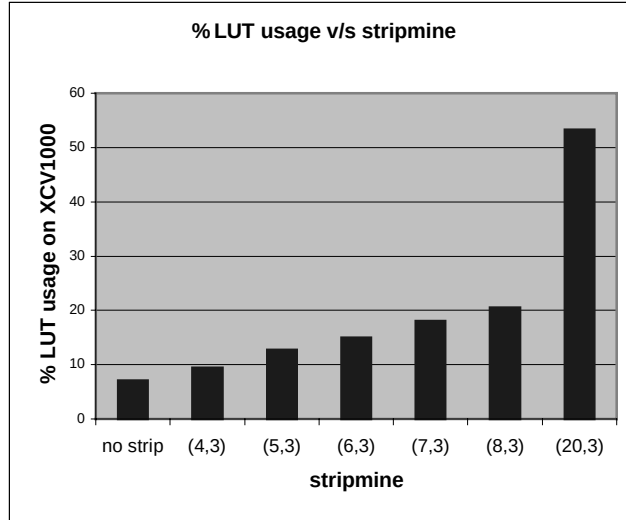


Figure 14: Total LUT usage for Convolution (Strip-mine results)

Example 2

In this example we present the effect of both, loop fusion and strip-mining on the *prewitt + threshold*. We examine the estimation reported by our tool for three cases: a) independent loops b) loop fusion c) strip-mine and loop fusion, for the SA-C code shown in Figure 15. When no loop fusion is applied, there are two loops running on the reconfigurable board, one of them being activated multiple times.

```
// Prewitt
int8 V[3,3] = { {-1, -1, -1}, { 0, 0, 0}, { 1, 1, 1} };
int8 H[3,3] = { {-1, 0, 1}, {-1, 0, 1}, {-1, 0, 1} };

uint8 R[:,:] = for window W[3,3] in Image {
    int8 iph, int8 ipv =
        for h in H dot w in W dot v in V
            return(sum(h*w), sum(v*w));

    uint8 mag = sqrt(iph*iph + ipv*ipv);
} return( array(mag) );

// Threshold
uint8 T[:,:] = for pix in R{
    uint8 t = pix>127 ? 255 : 0;
} return(array(t));
```

Figure 15: Prewitt and Threshold (two independent loops)

The performance of many systems is often limited by the time required to move data to the processing units. The fusion of producer-consumer loops is often helpful, since it reduces traffic and may eliminate intermediate data structures. Figure 16 shows the two loops after they are unrolled and fused.

```
// Loops fused
uint8 T[:,:] = for window W[3,3] in Image {
    int8 iph = (W[0,2]+W[1,2]+W[2,2]) - (W[0,0]+W[1,0]+W[2,0]);
    int8 ipv = (W[2,0]+W[2,1]+W[2,2]) - (W[0,0]+W[0,1]+W[0,2]);
    uint8 mag = sqrt(iph*iph + ipv*ipv);
    uint8 t = mag>127 ? 255 : 0;
} return( array(t) );
```

Figure 16: Prewitt + Threshold (loops fused)

Here there is only one loop running on the reconfigurable board and the loop is activated once. Loop fusion sometimes does redundant computation. If FPGA space is plentiful, then this is not a problem since the computation is done in parallel. But if space is scarce, then other optimizations have to be applied to remove the redundancies.

Loop strip-mining when followed by full loop unrolling produces the effect of multidimensional partial loop unrolling. As shown in Figure 17, strip-mining the loop in Figure 16 wraps the loop inside a new loop with a 4x3 window generator.

```
// Loop strip-mining
uint8 T[:,:] = for window W[4,3] in Image step(2,1) {
    int8 iph1 = (W[0,2]+W[1,2]+W[2,2]) - (W[0,0]+W[1,0]+W[2,0]);
    int8 ipv1 = (W[2,0]+W[2,1]+W[2,2]) - (W[0,0]+W[0,1]+W[0,2]);
    uint8 mag1 = sqrt(iph1*iph1 + ipv1*ipv1);
    uint8 t1 = mag1>127 ? 255 : 0;
    int8 iph2 = (W[1,2]+W[2,2]+W[3,2]) - (W[1,0]+W[2,0]+W[3,0]);
    int8 ipv2 = (W[3,0]+W[3,1]+W[3,2]) - (W[1,0]+W[1,1]+W[1,2]);
    uint8 mag2 = sqrt(iph2*iph2 + ipv2*ipv2);
    uint8 t2 = mag2>127 ? 255 : 0;
    uint8 t[2,1] = {{t1},{t2}};
} return( tile(t) );
```

Figure 17: Prewitt and Threshold (loops strip-mined)

Strip-mining results in just one loop running on the reconfigurable board and with only half the iterations as before. Thus, new loops are created during optimization and the resource usage is affected depending on the window size used for strip-mining

Table 12: Prewitt +Threshold area estimation

Type of Optimization on Prewitt +Threshold	Estimated LUT usage	% area on XCV1000
Independent Loops	1644	1.28
Loops fusion (default)	1063	0.83
Loop fusion + Strip-mine(4,3)	1280	1.00

Table 12 shows the estimation results. When the two loops exist independently, 1644 LUTs are used. Fusing the 2 loops (1063 LUTs) gives a better performance as well as reduces the LUT usage. If the compiler decides to strip-mine and fuse the first loop, it would cost 183 LUTs more than the default case. If the area is limited, then the default optimization is sufficient; else the loop fusion and strip-mining case gives better performance at the cost of more area. This example is just a small piece of code that might appear in a larger application (e.g. *tridiagonal* algorithm) and in such cases compile-time estimations becomes more crucial. Large SA-C programs might contain hundreds of loops and a number of combinations of optimizations can be performed on them. Depending on the availability of area on the target FPGA, the SA-C compiler can make a decision to apply the appropriate optimizations.

6 PREVIOUS WORK

Kurdahi and Xu [12, 13] have developed a strategy for accurate prediction of quality metrics for FPGA based designs. Given a netlist description, the tool estimates the area of the design in terms of the configurable logic blocks. The FPGA timing estimator uses the placement information and the area approximation to estimate the timing. The average estimation error in delay in this scheme is about 5.3 %, while the worst-case error is 13.2 %. Moreover, the experiment takes 1 to 2 orders of magnitude less time to compute than the synthesis tools.

The idea of lower bound area estimation from behavioral descriptions for multiplexer-based and bus-based architectures is explored in [14]. The behavioral description is expressed in the form of dataflow graph and the total performance and clock period expressed in real time are given as constraints. Given all this information the tool estimates the lower bounds on the number of functional units of each type, the number of registers and the number of buses.

A power estimation approach for SRAM-based FPGAs is discussed in [15]. The authors infer that if finite state machines dominate the structure, then the power estimation approach can be successfully applied. If the structure has more combinational logic, then algorithm fails.

Our estimations approach concentrates only on the LUT estimation on the Xilinx Virtex FPGA and targets at a much higher level than the netlist representation. Also, the algorithms used are not as complex as the ones used in the commercial synthesis tools.

7 CONCLUSION

In this paper we have presented a compile-time area estimation approach for LUT-based FPGAs. The estimation model is mainly developed to aid the SA-C compiler optimizations that affect the resource usage of a SA-C program. We have successfully demonstrated our estimation technique on a variety of individual benchmarks and as a useful tool to aid compiler optimizations. Experimental results show that our technique achieves an error rate of 2.5 % for small image-processing operators and 5.0 % for larger benchmarks. The worst-case error is 10.32 %. The estimation time is in the order of 1 millisecond as compared to 6.2 minutes for a synthesis tool. Thus, our estimator takes 5 to 6 orders of magnitude

less time to compute than the commercial synthesis tools. Even though our tool is specific for the Xilinx Virtex (XCV1000) FPGA, it can be easily modified to work for a variety of Xilinx FPGAs.

8 REFERENCES

- [1] A. DeHon, "The Density Advantage of Reconfigurable Computing," *IEEE Computer*, vol. 33, pp. 41-49, 2000.
- [2] R. Rinker, M. Carter, A. Patel, M. Chawathe, C. Ross, J. Hammes, W. Najjar, and A. P. W. Böhm, "An Automated Process for Compiling Dataflow Graphs into Hardware," *IEEE Trans. on VLSI*, vol. 9(1), 2001.
- [3] "The Cameron Project", Colorado State University, www.cs.colostate.edu/cameron, 1998.
- [4] M. Chawathe, M. Carter, C. Ross, R. Rinker, A. Patel, and W. A. Najjar, "Dataflow Graph to VHDL Translation," Computer Science Department, Colorado State University 2000.
- [5] J. P. Hammes, R. E. Rinker, D. M. McClure, A. P. W. Böhm, and W. A. Najjar, "The SA-C Compiler Dataflow Description", Colorado State University., www.cs.colostate.edu/cameron, 2001.
- [6] R. J. Francis, "A Tutorial on Logic Synthesis for Lookup Table-Based FPGAs," IEEE/ACM International Conference on Computer-aided Design, Santa Clara, CA USA, 1992.
- [7] "WildStar Reference Manual", Annapolis Micro Systems Inc., www.annapmicro.com, 2000.
- [8] Xilinx, "Virtex 2.5 V Field Programmable Gate Array," in *Product Specification*, 2000.
- [9] "Virtex 2.5 V Field Programmable Gate Array", Xilinx, Corp., <http://www.xilinx.com/partinfo/ds003-2.pdf>, 2000.
- [10] A. P. W. Böhm, B. Draper, W. Najjar, J. Hammes, R. Rinker, M. Chawathe, and C. Ross, "One-Step Compilation of Image Processing Algorithms to FPGAs," IEEE Symposium on Field-Configurable Custom Machines (FCCM 2000), Rohnert Park, CA, 2001.
- [11] A. Cohen, I. Daubechies, and J. C. Feauveau, "Biorthogonal Bases of Compactly Supported Wavelets," *Comm. on Pure and Applied Mathematics*, vol. XLV, pp. 485-560, 1992.
- [12] M. Xu and F. J. Kurdahi, "Accurate Prediction of Quality Metrics for Logic Level Designs Targeted towards Lookup Table Based FPGAs," *IEEE Trans. on VLSI Systems*, 1996.
- [13] S. Y. Ohm, F. J. Kurdahi, and N. Dutt, "Comprehensive Lower Bound Estimation from Behavioral Descriptions," IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD), San Jose, CA, 1994.
- [14] S. Y. Ohm, F. J. Kurdahi, N. Dutt, and M. Xu, "A Comprehensive Estimation Technique for High-Level Synthesis," *Int. Symp. on System Synthesis (ISSS)*, 1995.
- [15] K. Weis, C. Oetker, I. Katchan, T. Steckstor, and W. Rosenstiel, "Power Estimation Approach for SRAM-based FPGAs," FPGA '2000, Monterey, CA, 2000.
- [16] Phillip H. Sherrod, "NLREG - Nonlinear Regression Analysis Program", <http://www.sandh.com/sherrod/nlreg.htm>

CAMERON PROJECT: FINAL REPORT

Appendix O: Khoral Research Inc. Subcontract

A COMPLETE DEVELOPMENT ENVIRONMENT FOR IMAGE PROCESSING APPLICATIONS ON ADAPTIVE COMPUTING SYSTEMS

A. Christopher Moorman, and Donald M. Cates, Jr.

Khoral Research Inc.
6200 Uptown Blvd. NE Suite 200
Albuquerque, NM 87110, USA

ABSTRACT

Adaptive computing has always been a topic of great interest, providing a means to automatically map an application to specific hardware. The hardware may be configured to a specific application, thereby providing optimal performance. However, the largest benefit is that this configuration may be performed optimally during program execution. Unfortunately, adaptive computing is a relatively new area of research, therefore imposing serious complications when developing applications for adaptive hardware. This problem limits ACS development to hardware experts, prohibiting application specialists, such as image processing experts, from utilizing these systems. This paper will discuss a development environment that can bring the world of ACS application development to the IP expert with minimal hardware knowledge.

1. INTRODUCTION

While significant research has been done trying to utilize Adaptive Computing Systems (ACS), it has generally been approached as being a strict hardware problem. The number of tools to support application development on adaptive systems is minimal, thereby limiting their acceptance within the technical community. Until recently, the ability to develop applied solutions such as image processing applications on these systems was considered to be impractical, and rarely considered due to the limited resources available to support the development process for these systems.

Adaptive computing systems, commonly referred to as re-configurable systems, are generally made of integrated circuits called Field Programmable Devices. In particular, Field Programmable Gate Arrays (FPGAs) have been used extensively for military and commercial applications. They have been important particularly in research applications, because they offer low start-up costs and allow for design changes to be made easily. Such design changes would not be possible with fixed hardware systems. The standard FPGA consists of a block of logic devices spread out over a large mesh of connections to each element [5]. These logic elements are then surrounded by a series of I/O devices that connect to the chips interface pins. The advantage of these systems is that the user now has a chip that is capable of being programmed to perform the functionality of choice. The user has the ability to specify which connections are to be made and what functions are to be performed by each logic element. One problem is that these FPGAs are programmed via a series of configuration codes, which is currently done through

hardware descriptive languages such as VHDL [1]. These languages work well to program control logic for complex logic devices, but, they become extremely complex when developing applied applications such as image processing. This means that an image processing expert would need to become a hardware expert to take advantage of these re-configurable systems without a friendlier development environment.

To bring these systems within reach of applications programmers, the development environment has to handle any hardware issues. In particular, for image processing applications, a image processing development environment would be needed. A major tool for image processing applications today is a visual programming environment. These visual environments allow users to construct complex applications via the connection of basic operations. This level of programming removes any lower abstraction layers, including machine level programming, allowing the IP designer to focus on the specific application. One such development environment is Khoros. While Khoros provides a visual programming environment, it also provides CASE (Computer Aided Software Engineering) tools to support the complete development process.

It should be obvious to the reader that visual programming environments will not entirely bridge the gap between this advanced hardware and true image processing applications. Additional steps must be taken. One such step is the advent of a new language called SA-C (Single Assignment C). While SA-C is actually a subset of C and not really a new language, it should be viewed as a tool to achieve the final goal, that of bringing re-configurable hardware to the IP developer. SA-C will provide a means of linking the FPGA hardware to the visual programming environment provided by Khoros, as will be shown later in this paper.

At this point, it is important to note why Image Processing has been chosen as the specific application area. In order to show this, a short explanation of some image processing issues is in order. Image processing applications generally operate on large images, requiring extremely fast throughput. Sometimes, even real-time throughput is required. Adaptive systems are capable of achieving this type of performance. In addition, their architecture is such that a user can extract parallelisms from the specific application. This allows a user to exploit the inherent parallelisms present in these windowing-type operations. Adaptive hardware systems also allow users to optimize performance by rearranging specific image operators to minimize data flow. The visual programming environment and performance monitoring could be used to achieve a reduction in

data flow and achieve enhanced performance from the application level, thereby, hiding any hardware details from the IP expert.

This paper discusses the details of Khoros, the development adaptive computing environment, SA-C, the language required to map an application to the FPGA hardware, and the overall design process to support adaptive computing systems.

2. Khoros Development Environment

Khoros is a complete software integration and development environment, providing development tools, a visual programming environment to support execution, as well as extensive libraries to support image and signal processing. These development tools support multi-language applications in an object-oriented fashion, allowing users to organize their applications into toolboxes. The cross-platform support, including numerous workstations, embedded platforms, and the HPC (High Performance Computing) arena, allow users to migrate their existing applications with minimal effort.

Khoros has been used for a wide variety of applications, ranging from Automatic Target recognition (ATR) and hyperspectral/multispectral imaging to medical and biological imaging. Khoros provides a significant number of utilities to the programmer, removing several abstraction layers and supporting many layers of programming. The developer has the option of programming at whichever level they prefer. An example is the ability to develop parallel algorithms within Advanced Khoros with minimal knowledge of MPI (Message Interface passing). Khoros provides data model support, allowing a user to work with input as objects, removing all file handling and memory management from the programming task. It also provides streaming models that may be used if performance is of a concern. The significant number of utilities within Advanced Khoros is only part of the complete development environment. The execution and development tools provide complete integration, maintenance and development support. The next sections discuss what each of the tools brings to the development process[4].

2.1 CASE Development Tools

The development tools provided within Khoros give the user the ability to develop, manage, and maintain complex software applications. The complete development suite consists of tools called Craftsman, Composer, and Guise.

Craftsman brings maintenance and management functionality to the software application, providing functions such as object creation, deletion, re-naming, and attribute modification, to name a few. Composer provides advanced functions to a specific software object, allowing users to edit source, generate code, compile the existing application and modify the GUI (Graphical User Interface). Guise, which allows the user to create and modify the applications GUI, provides a graphical representation of the UIS (User Interface Specification) under development. These tools are capable of supporting numerous languages and object types within Advanced Khoros, allowing a user to develop applications for even new languages, such as SA-C, or any user-created languages. The developer need only add the source

necessary to provide the needed functionality. The tools will handle the remaining tasks.

With this environment, the user now has the ability to develop, maintain, and manage these SA-C algorithms for the FPGA hardware. The only remaining concern is the method of execution provided by the visual programming environment.

2.2 Visual Programming Environment

Cantata provides a visual programming environment to the developer's applications. This visual environment provides a user with the ability to interconnect a series of basic routines, or primitives, which could be used to develop larger and more complex applications. By providing the visual interface, the developer has the ability to rearrange an application based on performance or functionality or even hardware limitations. An example of Cantata can be found in figure 1.

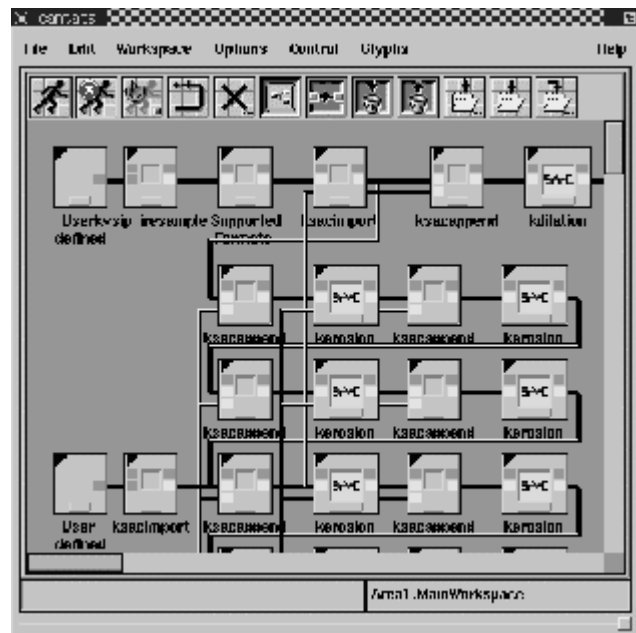


Figure 1. An sample of a workspace in the visual programming environment, Cantata, provided within Advanced Khoros.

In addition to providing a means of visual optimization for applications, visual programming also abstracts lower programming levels from a user, allowing applied experts to develop context specific applications with little or no programming knowledge. Image processing experts can develop complex image enhancement and recognition applications without writing a single line of code.

Additional Benefits to a visual programming environment:

- Abstraction of low-level programming.
- Abstraction of hardware programming
- A means for visual modification and optimization.

Cantata also hides any hardware and execution concerns from the developer while providing multi-architecture support and distributed computing capabilities.

An ATR workspace using SA-C routines is displayed in figure 1. The execution of these routines is currently on the local-host. The ability to execute SA-C objects on the FPGA hardware is still under development. The user will ultimately have the option of executing these routines locally as well as on the embedded systems. This capability allows the image processing expert to utilize existing SA-C library routines within Cantata, develop the IP application, and execute these routines on the FPGA without knowledge of the hardware.

Profiling and benchmarking capability will also be added to Cantata, providing visual performance feedback. This will allow users to modify the hardware mapping process or to identify any bottlenecks within the application. Additional information on this topic can be found in Section 4.

3. SA-C Single Assignment C

Single Assignment C (SA-C), was developed to minimize the problems associated with the hardware mapping procedure. While it is true that SA-C is a subset of C, it is important to note that it has been enhanced to truly support IP applications. The functionality removed was due to hardware mapping issues. The next sections discuss why SA-C was created and how it has been enhanced to support IP applications.

3.1 Language Overview

To help understand why SA-C was created, a list of objectives is shown below [2]:

- Efficient Expression of IP Constructs.
- Tailored to ACS Platforms.
- Ease of use.

These three objectives, taken together, express the point that SA-C was developed to bring the IP and Adaptive Computing communities together.

The ability to support multidimensional arrays is crucial in IP applications. The ability to access sub-structures within large images is also important for window-based and template-based procedures. SA-C addresses both of these issues while providing a significant number of statistical operations over multiple regions of the images.

As mentioned earlier, there are a significant number of problems that occur when mapping software to FPGA hardware. To eliminate some of these problems, certain features of the C language were removed from SA-C. For example in SA-C, only single assignment is allowed, and no pointer or recursion is supported. In addition, SA-C was developed to support the notion of Data Flow Graphs (DFG's), which are used for hardware mapping.

Finally, SA-C had to be easy to use before any real acceptance could be realized. Therefore SA-C follows the C syntax. In addition, it can be called from C and C++, and with the integration of SA-C into Khoros, a complete development environment has been created.

It is important to stress that SA-C is not an inferior language. In fact, quite the opposite is true. Reassignment, pointers, and recursion are lost, but true IP support is gained. C provides

minimal support of multiple dimensional arrays by allowing arrays of arrays requiring a series of mallocs, with no real data access methods. Users are forced to control data access, boundary conditions and overlapping. SA-C hides all this from the user, allowing the user to access images as windows at any focus within the complete image. Since SA-C controls the data access methods, it is capable of exploiting parallelisms that may be present for a specific application.

Finally, C does not provide multiple-bit precision, which SA-C does. Multiple-bit precision is crucial for optimized FPGA applications. Multiple-bit precision supports proper resource allocation by utilizing the necessary hardware to support minimal bit precision.

3.2 Image Processing Features

As mentioned earlier, image processing applications are generally window-oriented or template-oriented. Many of the operations such as convolution and basic filtering are kernel based. This type of manipulation is data intensive, requiring a significant number of data retrievals. SA-C handles all of the data access and allocation, allowing users to manipulate images using stencils, windows, and slices within an image. This capability allows for complex routines like convolution to be implemented with only a few lines of code. Looping constructs have been enhanced to support specific data acquisition procedures, significantly reducing the amount of code needed for data access.

Another important feature required within image processing is the generation of image statistics. A significant number of statistical methods have been incorporated within SA-C. Some of these include scalar reductions like min, max, median, and mean, as well as advanced measures like histogram statistics. SA-C supports statistical operations on a complete image, or on portions or regions within the image. This functionality actually gives SA-C the appearance of a higher level language, allowing users to specify operations with mere key words rather than lengthy function calls or code segments.

4. Complete Development Procedure

The details of some items from the development environment have been presented, and to some, it may be apparent that the notion of hardware mapping has not been addressed at this point. The integration of SA-C and Khoros provide the complete environment to the standard applications developer. These tools provide a means of developing complex image processing applications with SA-C, and executing them locally from within Cantata. In addition, if pre-compiled versions exist for the FPGA hardware, the user can execute these rather than the local versions within Cantata.

The intention is to provide a user with a complete development system, which means the IP applications should be executable locally, executable on the FPGAs using pre-compiled versions of the software, and executable on FPGAs using no pre-compiled versions. The first two methods of development and execution are specifically designed for the image processing expert. They require little or no hardware knowledge. The final step in the development process is the development of these pre-compiled

libraries. While hardware knowledge is required for this step, it is merely from an optimization standpoint rather than a development standpoint. The complete development process is shown in figure 2 shown below[2].

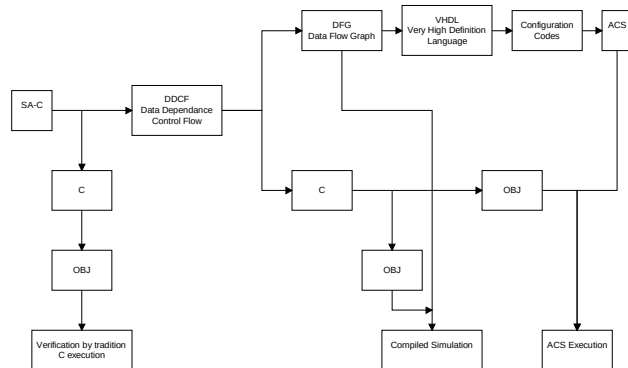


Figure 2. ACS Design Flow Diagram.

The initial branch of the SA-C block is what is achieved by the integration of SA-C with Khoros. This provides the user with the ability to execute on a local host. The SA-C source is compiled to a standard C version, which is compatible with Khoros, thereby providing portability across multiple platforms.

The remainder of the design process provides optimization and hardware mapping to the ACS platform. It is believed that the optimization and mapping can be efficiently done through the use of Data Flow Graphs (DFGs) and Data Dependence and Control Flow Graphs (DDCFs).

The DDCF graph provides an efficient means to make optimizations to the source. Some of these would include loop fusion, unrolling, and reordering. In addition to optimizations, it provides an optimized source to the DFG.

The lower branch immediately after the DDCF in figure 2 provides a means of optimizing the SA-C application. The tools will be modified to support the visual modification of this DDCF. The optimized code can then be compiled down to the local hardware or the ACS. This capability is extremely important in evaluating the performance achieved after optimization. A visual representation of the performance measures and benchmarking will also be added to Cantata, providing additional performance feedback.

The top branch of figure 2 is representative of the complete development process. The previous two methods mentioned were developed to bring the adaptive computing systems within reach and to reduce prototyping and application development time. This last step is representative of the ground up development process.

After the DDCF graph, the process flows into the DFG. This step provides an additional level of optimization from a different perspective. These are actually a flatter version of the DDCF. This is the point where machine-specific transformations occur.

It should be apparent that some sort of hardware language would be required to get to the FPGA level. The design process will now have several options at this point, the utilization of pre-compiled VHDL and macros, or the generation of completely new VHDL source. The latter opinion would add significant compilation time to the development process but would allow for efficient VHDL to be generated for the users specialized image processing applications.

5. SUMMARY

The goal of this project is to provide a complete design environment for the image processing expert which could be used to develop applications for adaptive computing systems. The integration of the SA-C language with Khoros provides such an environment. Khoros brings the tools needed to develop, manage and maintain software objects, while providing a means of execution via a visual programming environment. The SA-C language provides advanced functionality specifically designed for image manipulation and image processing applications. When these are combined with pre-compiled SA-C libraries for a specific FPGA hardware, IP experts can develop complex IP applications with no hardware knowledge. The execution of these applications is supported within Cantata locally or on the FPGA hardware. This part of the design process will significantly reduce prototype and development time of IP applications on the FPGA, as well as making it possible for IP designers to utilize this advanced hardware.

The development of these SA-C libraries and routines for the FPGA is accomplished via the DFG and DDCF graphs. Optimized VHDL is generated from these graphs, which is then directly mapped to the hardware by using the FPGA vendor-supplied software. This portion of the development process is somewhat time consuming, and requires additional compilation time. However, the time should be reduced due to the generation of optimized VHDL. Both methods combined, provided a complete IP development solution while reducing development time and improving performance of generated applications.

6. REFERENCES

- [1] D. Perry. *VHDL*. McGraw-Hill, 1993.
- [2] Najjar W., Bohm W., Draper B., and Beveridge R. "Optimized Compilation of Visual Programs for Image Processing on Adaptive Computing Systems". *Cameron Project Kickoff Presentation*, Washington D.C., USA, 1998, pages 3-4.
- [3] Bohm W. "The Sassy Language - Version 0.6". Colorado State University web page: <http://www.cs.colostate.edu/bohm/>, 1998.
- [4] Khoral Research's development research web page: <http://www.khoral.com>, 1998.
- [5] Hammes J., Draper B., and Bohm W. "Sassy: A language and optimizing compiler for image processing of reconfigurable computing systems". Colorado State University: <http://www.cs.colostate.edu/bohm/>, 1998.
- [6] Microsoft Research's Speech Technology Group web page: <http://www.research.microsoft.com/research/srg/>.