AFRL-RY-WP-TR-2012-0214

THE PLATFORM-AWARE COMPILATION ENVIRONMENT (PACE)

Keith D. Cooper, John Mellor-Crummey, Erzsébet Merényi, Krishna Palem, P. Sadayappan, Vivek Sarkar, and Linda Torczon

William Marsh Rice University

SEPTEMBER 2012
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

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

This report was cleared for public release by the Defense Advanced Research Projects Agency (DARPA) and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RY-WP-TR-2012-0214 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

//SIGNED//

AL SCARPELLI, Project Engineer
Integrated Circuits & Microsystems Branch
Aerospace Components Division

//SIGNED//

BRADLEY J. PAUL, Chief
Integrated Circuits & Microsystems Branch
Aerospace Components Division

//SIGNED//

BRADLEY CHRISTIANSEN, Lt Col, USAF
Deputy Division Chief
Aerospace Components Division
Sensors Directorate

The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

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

*Disseminated copies will show "//signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
September 2012	Final	18 March 2009 – 2 June 2012

4. TITLE AND SUBTITLE
PLATFORM-AWARE COMPILATION ENVIRONMENT (PACE)

5a. CONTRACT NUMBER
FA8650-09-C-7915

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)
Keith D. Cooper, John Mellor-Crummey, Erzsébet Merényi, Krishna Palem, P. Sadayappan, Vivek Sarkar, and Linda Torczon

5d. PROJECT NUMBER
3000

5e. TASK NUMBER
YD

5f. WORK UNIT NUMBER
Y0H3

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
William Marsh Rice University
Office of Sponsored Research MS-16
6100 Main ST
Houston TX 77005-1827

8. PERFORMING ORGANIZATION REPORT NUMBER
AFRL-RY-WP-TR-2012-0214

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory
Sensors Directorate
Wright-Patterson Air Force Base, OH 45433-7320
Air Force Materiel Command
United States Air Force

Defense Advanced Research Projects Agency/
Microsystems Technology Office (DARPA/MTO)
675 North Randolph Street
Arlington, VA 22203-2114

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RYDI

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RY-WP-TR-2012-0214

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
PAO Case Number: DARPA 19970; Clearance Date: 10 October 2012. Report contains color.

14. ABSTRACT
The PACE Project investigated the feasibility of using a combination of characterization-driven optimization, feedback-directed optimization, and automatic selection of transformations to retarget an optimizing compiler to new computer systems. This report provides an overview of the design, the implementation, and the ways in which the PACE components advanced the state of the art in compilation, in performance measurement and attribution, in portable measurement of system characteristics, and in machine learning.

15. SUBJECT TERMS
Computer performance, tool portability, code optimization, resource characterization, characterization-driven optimization, feedback-directed optimization, runtime performance measurement and attribution

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Al Scarpelli
Unclassified	Unclassified	Unclassified	SAR	52	19b. TELEPHONE NUMBER *(Include Area Code)* N/A

Table of Contents

ABSTRACT

The PACE Project investigated the feasibility of using a combination of characterization-driven optimization, feedback-directed optimization, and automatic selection of transformations to retarget an optimizing compiler to new computer systems. The effort produced the design for a complete compilation environment that embodied those concepts, along with a significant set of components for that environment. Work was completed as defined in the Modified Statement of Work, shown in Appendix A. This report provides an overview of the design, the implementation, and the ways in which the PACE components advanced the state of the art in compilation, in performance measurement and attribution, in portable measurement of system characteristics, and in machine learning.

FOREWARD

The Platform-Aware Compilation Environment (PACE) project was funded by the Defense Advanced Research Projects Agency (DARPA) through the Air Force Research Laboratory (AFRL) Contract FA8650-09-C-7915 with Rice University. Rice University was the lead institution on the PACE Project, with subcontracts at Ohio State University, Stanford University, ET International, and Texas Instruments.[1] PACE was part of the DARPA-sponsored Architecture-Aware Compiler Environment (AACE) program.

The PACE Project Team investigated the feasibility of using a combination of characterization-driven optimization, feedback-directed optimization, and automatic selection of transformations to retarget an optimizing compiler to new computer systems. The overall direction of PACE was set by the AACE program; the detailed approach was set by the investigators.

The PACE Project produced a substantial body of software (See Section 3.5). The software produced under the PACE Project is available for use by others, under appropriate licenses.

[1] For a variety of reasons, the Texas Instruments subcontract was never executed. However, Reid Tatge of Texas Instruments was an active participant in many of the PACE design discussions and meetings.

1. EXECUTIVE SUMMARY

The Platform-Aware Compilation Environment Project (PACE) explored the efficacy of using characterization-driven code optimization, feedback-directed code optimization, and automatic selection of transformations to simplify the task of porting optimizing compilers to new computer systems. PACE was funded as part of the Architecture-Aware Compiler Environment (AACE) Program, which explored portable techniques to measure the performance-critical characteristics of computer systems and ways to use the knowledge of those characteristics to retarget optimizing compilers.

This final report describes the fundamental questions that the investigators explored in the PACE Project and the results that they obtained. Section 2 describes the investigators' vision for PACE and the overall design of the PACE System. More detailed information on the vision and design can be found in the design document produced for the AACE Preliminary Design Review [1]. Section 3 describes the actual progress that we made in the implementation of PACE. A more detailed picture of the status of the PACE System can be found in the PACE Status and Futures document [2]. Section 4 discusses specific results in the various subprojects, with particular attention to how the PACE work advanced the state of the art.

The PACE Project produced a substantial body of software (See Section 3.5). The software produced under the PACE Project is available, along with appropriate licenses, at http://pace.rice.edu/ under "Software".

2. INTRODUCTION

The difficulty of achieving a desired level of performance is a major challenge in the design, procurement, and use of modern computer systems, from supercomputers to embedded controllers. Performance arises from characteristics of both the computer system and the application, and from the extent to which the application code can be mapped efficiently onto the system's hardware. To achieve a reasonably high fraction of available performance with existing tools, the programmer must understand critical properties of both the underlying computer system and the application code, or tools must automatically tailor the code for the underlying computer system, or both.

Application code must be translated from some human-usable "programming language", such as Fortran, C, C++, Java, Python, or Ruby, into a form where it executes on the targeted system. Unfortunately, the details of that translation determine, to a large extent, the performance that the application can achieve.

That translation is accomplished using a software system called a *compiler*. A compiler is just a computer program that translates one executable representation of a program into another. The compiler and its runtime support software determine how the algorithms and abstractions expressed in the source-language version of the application are implemented on the actual computer system when the application executes—or, more properly, when the compiled code for the application runs. Thus, the compiler is one of the principal determiners of application performance.

Experience over the last thirty years has shown that it typically takes a minimum of three to five years after the introduction of a new architectural feature before compilers can make consistent good use of the feature. Given the rapid obsolescence of computing systems, the practical implication of this delay is that systems are retired before their compilers can take full advantage of new features. As a result of this almost constant time lag, most applications run on systems where they achieve a small fraction of the available performance—say five to fifteen percent.

The overriding goal of the PACE Project was to develop language translation tools—specifically a compiler and its surrounding tools—that would simplify the task of achieving a reasonable fraction of available performance on new systems. The approach that we pursued in PACE was to design and develop a compiler based on a strategy for automatic retargeting: automatic tools to measure performance-critical parameters of the target system coupled to a compiler environment that relied on those measured performance parameters to tailor code optimization to the specific targeted system.

The remainder of this section describes the investigators' vision for PACE and the overall design of the PACE System. More detailed information on the vision and design can be found in the design document produced for the AACE Preliminary Design Review [1]. Section 3 describes the actual progress that we made in the implementation of PACE. A more detailed picture of the status of the PACE System can be found in the PACE Status and Futures document [2]. Section 4 discusses specific results in the various subprojects, with particular attention to how the PACE work advanced the state of the art.

In the design and implementation of the PACE System, we were constrained by two key requirements of the AACE Program.

- AACE environments were designed to be portable; in general, the tools should rely on the standard C language libraries plus POSIX interfaces.
- AACE environments should rely on a native C compiler to perform the final translation of application code for execution on the target system.

These two requirements had a strong influence on the design and implementation of the PACE tools. The PACE Project designed and partially implemented an AACE compiler environment.

The PACE environment, as envisioned in *PACE Preliminary Design Document* [1], and shown in Figure 1 has four major components: the PACE Compiler, the PACE Runtime System, the PACE Characterization Tools, and the PACE Machine Learning Tools.

- The PACE Compiler is responsible for translating application code so that it executes efficiently on the target system.
- The PACE Runtime System provides services for monitoring application performance and for runtime adaptation.
- The PACE Characterization Tools derive information about the performance-critical characteristics of the target system for use by the other PACE tools.
- The PACE Machine Learning Tools observe the behavior of the other tools and applications compiled through PACE, and attempt to encode that observed experience into knowledge that the compiler and runtime system can use to improve performance.

Thus, the four major themes of the PACE Project are:

- ***Measure the effective performance of the target computer system:*** The PACE Characterization tools measure specific characteristics of the target computer system. Characteristic values are derived by a combination of microbenchmarks and careful analysis.[2] The tools are written in a portable style in C. They rely on the capabilities of the POSIX standard libraries.
- ***Use measured characteristics to drive code optimization:*** The PACE Compiler contains optimizations that are parameterized around the characteristic values measured by the PACE Characterization Tools. Changing the characteristic information causes a change in optimization behavior, in effect, tuning the compiler's behavior to the target system.
- ***Observe, adjust, and adapt application behavior at runtime:*** The dynamic nature of high-performance multiprocessor computation suggests that runtime conditions will have a strong impact on performance. Because neither the programmer nor the compiler can easily predict or understand runtime conditions, the PACE Runtime System includes

[2] It is important to note that the AACE program specified that the AACE environments should rely on a native C compiler, either open source or vendor-supplied, to perform the final compilation step. Thus, any property of the target machine and its compiler that could not be discerned from an ANSI C program was not useful to the optimizer. If the microbenchmark could not expose the property, the PACE Compiler could not capitalize on it.

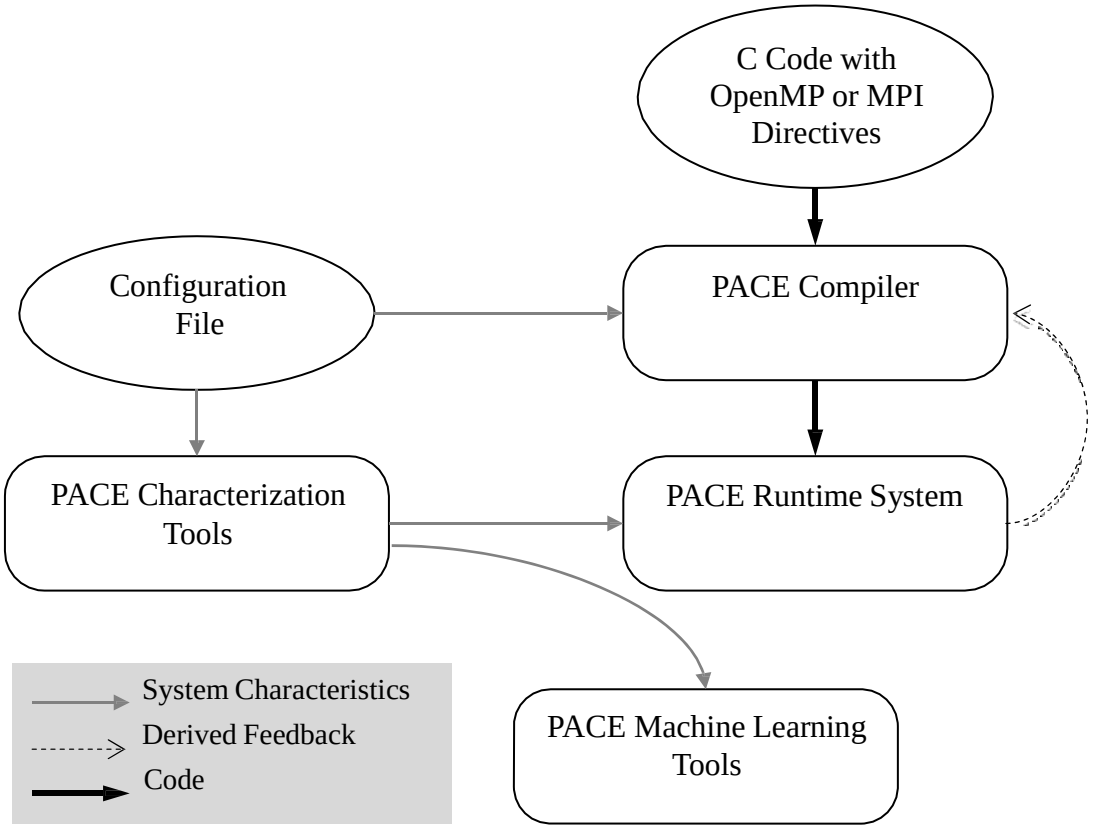

Figure 1: High-level Overview of the PACE System

facilities to measure and report runtime performance and to adjust application behavior in response to those measurements.

- ***Use offline learning to make long-term improvements in code quality:*** The interactions between system, software, and application characteristics and observed performance are too complex to solve with a single pre-defined optimization strategy. The PACE Machine Learning effort used machine learning techniques to capture these complex relationships, with the goal of using the knowledge to inform better optimization decisions.

The PACE Project was designed to instantiate these themes in a sophisticated, feedback-driven compilation environment.

When our work on the PACE Project began, our focus was on re-tuning an optimizing compiler for distinct computer systems. Over the course of our phase one activities, we became convinced that a second, equally powerful, rationale for characterization-driven optimization is the extreme divergence in performance characteristics among different implementations of the same instruction set architecture (ISA). For example, distinct implementations of the basic x86 ISA from a single vendor, either Intel or AMD, can differ in the number of functional units, in the relative cost of arithmetic and memory operations, in the size and organization of the various structures in the memory hierarchy, and in the number and kinds of processor cores provided by the implementation. Each of these details may change the performance of compiled code; a good compiler might generate radically different code for different models of the same ISA.

These model-specific differences are breaking down the "traditional" model of software development, in which the compiler can ignore model-specific differences and the application programmer can run the same compiled code on any "x86" implementation. If performance is an issue, compilers must pay attention to model-specific performance differences. Characterization-driven optimization, in the sense proposed in the PACE Project, is one rational way of dealing with the proliferation of implementation models. It offers the promise of model-specific optimization without a proliferation of different optimizing compilers.

The remainder of this section describes the PACE System in more detail. Specifically, Section 2.1 describes the individual components of the PACE System; Section 2.2 describes the different ways in which the PACE System adapts its behavior to the target system, application, and runtime context. Finally, Section 2.3 lays out the major research questions that the PACE Project tried to address. We return to those questions in Section 5.

2.1 Components of the PACE System

The PACE System has four major components: the PACE Compiler, the PACE Runtime System, the PACE Characterization Tools, and the PACE Machine Learning Tools.

2.1.1 The PACE Compiler

The PACE Compiler consists of four primary components, along with some infrastructure to coordinate the interactions between them. The PACE Compiler, shown in Figure 2 consists of a Compiler Driver that invokes and sequences the various parts of the compiler, an Application-Aware Partitioner (AAP) that performs some preliminary file-level and source-level transformations on the application's code, a Platform-Aware Optimizer (PAO) that performs high-level, or near-source, optimizations, and a Target-Aware Optimizer (TAO) that tries to shape the code so that it compiles well in the native compiler. A native C compiler for the target system performs the final translation into object code. On systems that the LLVM Backend supports, such as x86 systems, that backend can be used in place of a native C compiler.

To compile an application, the compiler driver first invokes the AAP, which examines the entire application and creates a set of refactored program units from the source code. The compiler driver then iterates over the program units, and invokes the PAO and the TAO on each of them. The gray lines in the figure represent these control relationships, as well as the information passed from the Compiler Driver to the other components. Among the parameters that the Compiler Driver passes to the TAO is a flag that indicates which backend it should invoke.

Compilation can take one of three paths through the compiler. Each path begins with the AAP and the PAO. The Compiler Driver can instruct the PAO to emit C code, which is then compiled with the native compiler—an attractive option for a capable native compiler. If the target machine has an LLVM backend, such as an x86 ISA machine, the Compiler Driver can direct the TAO to use the LLVM backend. On systems without an LLVM backend, the Compiler Driver can instruct the TAO to generate C code tailored for the native compiler, and then pass the C code to that native compiler.

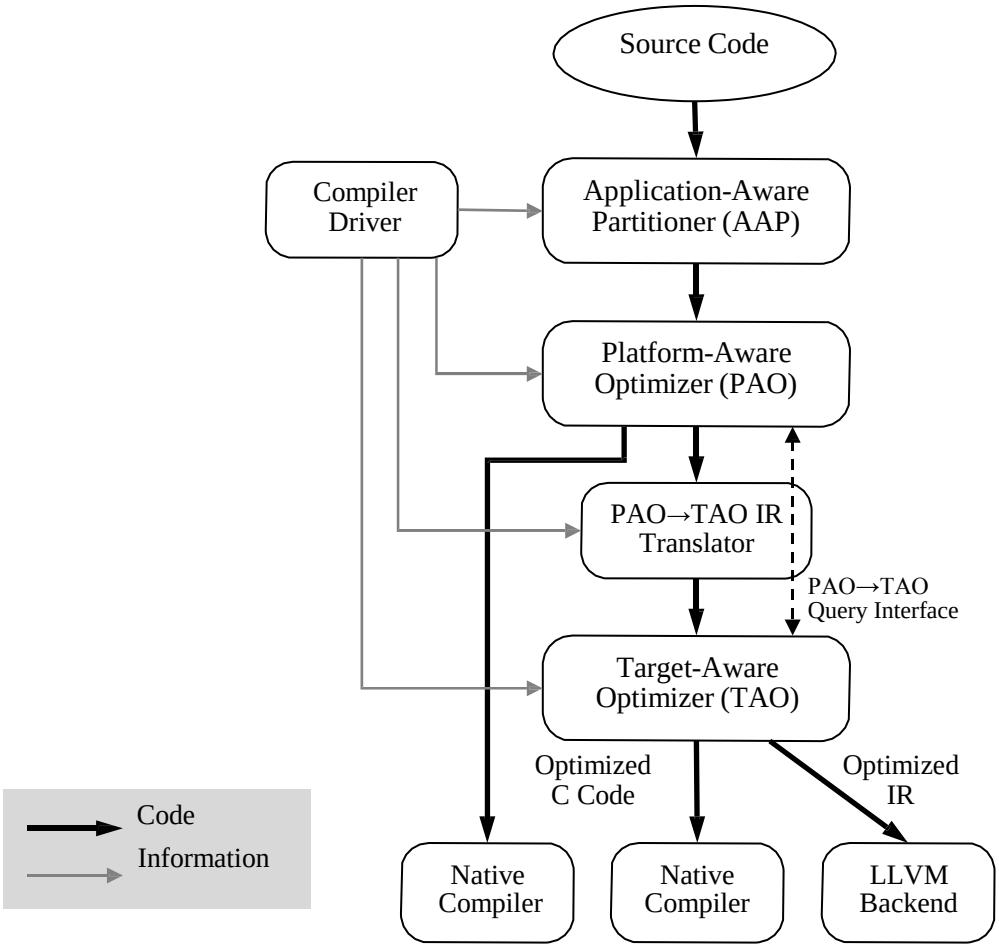

Figure 2: Structure of the PACE Compiler

Application-Aware Partitioner: The AAP is the first component invoked on an application. The AAP examines the source code for the entire application and rewrites it into refactored program units. The goals of the AAP are to (1) limit the size of any single compilation unit, and (2) group together procedures that have similar performance profiles and problems so that they can be optimized in the same way. Because compilation times rise non-linearly with the size of the compilation unit, the AAP's refactoring work should help to limit compilation times. Similarly, by grouping together procedures that need similar optimization strategies, the AAP should help to limit the time required to build a complete application.

As a practical matter, the AAP also ensures that the PACE Compiler has access to all of the source code for an application. It creates a private directory with a copy of the application. Within that directory, it creates a subdirectory for the current invocation of the compiler where it writes its refactored program units. The directory will also hold the work products and annotations of the other compiler components; it serves as the working directory for the current invocation of the compiler.

Platform-Aware Optimizer: The PAO reads in the C code for a refactored program unit and applies analyses and transformations to the code that are intended to tailor it based on platform-wide considerations. Of particular concern in the PAO are efficient use of the memory hierarchy and effective use of thread-level parallelism. The PAO includes a subsystem, PolyOpt, that uses polyhedral analysis and transformations to reorganize loop nests for efficient memory access. For loops that are not susceptible to polyhedral analysis, the PAO includes implementations of other loop-level transformations, such as tiling, interchange, unrolling, and scalar replacement.

The PAO uses data about system characteristics, such as the organization of the memory hierarchy and the effective amount of available parallelism, to drive its optimizations. The PACE Characterization Tools provide that data. The PAO also has access to performance measurements made by the PACE Runtime System during prior executions of the code that it can use to focus its attention on the hot paths in the code.

The PAO makes use of a novel feedback mechanism, the PAO→TAO query interface, to adapt its optimization decisions. For example, in choosing loop unroll factors, the PAO asks the TAO to compile a code fragment containing the prospective transformed loop; the TAO responds with information about the quality of the code that it can generate, including estimates of both operation counts and register spill operations in the final version of the loop. Using this feedback, the PAO can adjust the unroll factor to avoid generating code that would cause the TAO and the native compiler to insert excessive and expensive spill code.

As its final step, the PAO invokes the PAO→TAO IR translator to rewrite the abstract syntax trees used by the PAO into the low-level linear code used in the TAO. The translator also maps analysis results and annotations created in the PAO so that they refer correctly to the TAO form of the IR.

Target-Aware Optimizer: The TAO takes the transformed, annotated code produced by the PAO and maps it onto the architectural resources of the individual processing elements. The TAO also provides a strong level of scalar (or uniprocessor) optimization to compensate for potential weaknesses in the native compiler.

The TAO was designed to have three distinct back ends: a native backend for the x86 ISA, a backend that generates C code suitable for use with a vendor-supplied or open-source native C compiler, and the PAO→TAO query backend, which supplies feedback about generated code to the PAO.

2.1.2 The PACE Runtime System

The PACE Runtime System (RTS) provides performance monitoring and runtime parameter tuning. The PACE Compiler prepares an executable for the RTS by including the runtime hooks necessary to initialize the RTS, and by constructing a measurement script that sets environment variables and flags that control and direct the measurement system. The user invokes the executable through a measurement script.[3]

When invoked, the RTS interposes itself between the application and the operating system to intercept events such as program launch and termination, thread creation and destruction, signal

[3] It is possible to invoke a PACE-compiled executable without invoking the RTS. The preferred mechanism to achieve that goal is to invoke it through the measurement script, with the appropriate parameter settings to disable runtime performance monitoring.

handler setup, signal delivery, loading and unloading of dynamic libraries, and MPI initialization and finalization. It then launches the application, monitors its behavior using a variety of mechanisms, and records the results.

The RTS also provides an interface for runtime selection of optimization parameters. The compiler rewrites the code region into an optimized, parameterized form and builds the various data structures and support routines needed by the RTS harness for online feedback-directed optimization.

2.1.3 The PACE Characterization Tools

The PACE Characterization tools are a standalone package designed to measure the performance characteristics of a new system that are important to the rest of the PACE System, and to provide a simple consistent interface to that information for the other PACE tools. The tools consist of a collection of microbenchmark codes and scripts that invoke them. Each microbenchmark performs a series of measurements followed by an analysis of the measured data.

The Characterization tools are written in a portable style in the C programming language. They rely on entry points from the standard C libraries and the POSIX operating system interface. The specific characteristics that the tools measured in phase one of the AACE program are shown in Table 2.2 of the PACE Status and Futures document [2].

In many cases, the Characterization tools capture an *effective* number for the parameter, rather than the actual number provided by the underlying hardware. The effective quantity is, in general, defined as the amount of that resource available to a C program. For example, the effective number of floating-point registers available to a C program depends, in part, on the number of such registers provided by the underlying hardware. However, the compiler is almost always the limiting factor in this case. If the hardware provides 16 floating-point registers, but the compiler cannot allocate more than 14 of them to variables in the application code, then the effective number should be 14 registers.[4]

In some cases, a hardware characteristic may not be discernible from a C program. In those cases, the PACE Compiler cannot rely upon that characteristic in optimization, since the C code cannot control the behavior. Associativity in the memory hierarchy is a good example of this problem. If the L2 cache on a processor is physically mapped, the mapping between a source-level data structure, such as a large array, and its cache locations depends on the mapping of virtual memory pages to physical page frames, and the tools cannot measure the associativity of that cache level with any certainty.

[4] For example, some register allocators reserve a small number of registers to use for values that are used too infrequently to merit their own registers. If the compiler reserves two such registers, it reduces the effective number available to the application. The PACE Compiler, which will control the number of simultaneously live floating-point values, should not plan on using those last two registers. Similarly, sharing in the cache hierarchy (between instructions and data or between cores) can lead to effective cache sizes that are significantly smaller than the hardware cache size.

2.1.4 The PACE Machine Learning Effort

The PACE Machine Learning (ML) tools are intended to augment specific decision making processes within the PACE System, through analysis of past experience and behavior. A modern compilation environment, such as PACE, can produce reams of data about the application itself, the process used to compile it, and its behavior at runtime. Unfortunately, the application's runtime performance can depend in subtle ways on an unknown subset of that information, and neither humans nor algorithmic programs are particularly good at discerning those relationships.

The ML effort will develop tools that learn models for distinct decision problems that arise in the PACE Compiler and the PACE Runtime System. As those models emerge, the tools will use them directly to improve the quality of decisions. The models draw their inputs from the other PACE tools. They query the resource-characterization interface directly. They find the input data from the compiler and the runtime system where it is stored with the application. The ML tools have their own private repository where they can store their context, data, and results. As output, the models will produce directives, refined input parameters, or changes to optimization plans. In this way, the ML tools will improve the behavior of the other PACE tools.

To facilitate offline learning, the PACE System needs a mechanism that invokes the offline portions of the ML tools on a regular basis. The POSIX crontab facility could be used to schedule regular invocations of the offline PACE ML tools. Problems that are solved online would invoke the appropriate ML tools directly.

2.2 Adaptation in the PACE System

Adaptation is the key strategy embodied in the PACE System. The compiler changes its behavior in response to either internal or external feedback. The other PACE tools exist to provide data that guides adaptation or to facilitate that adaptation.

Adaptation in the PACE Compiler falls into two categories: short-term adaptation that tailors the behavior of one executable, and long-term learning that changes the behavior of the compiler. The PACE System includes four different mechanisms to achieve adaptation:

1. Characterization-driven adaptation,
2. Offline feedback-driven adaptation,
3. Online feedback-driven optimization, and
4. Long-term machine learning.

The mechanisms are summarized in Table 1 and described below.

In combination, these four mechanisms provide the compiler with the ability to adapt its behavior to the target system, the application, and the runtime situation. Taken together, 1 and 2 combine to let the PACE Compiler automatically select transformations. These mechanisms allow the PACE System to be flexible in its pursuit of runtime performance.

2.2.1 Characterization-Driven Optimization

The concept of characterization-driven optimization forms the core of the PACE Project. AACE compiler environments include tools that measure performance- critical characteristics of the target system and transformations that use those measured characteristics as an integral part of the optimization process. In the PACE Compiler, for example, the loop optimizations in the PAO use the measured parameters of the memory hierarchy to help choose tile sizes.

Table 1: Adaptation Mechanisms in the PACE Compiler

	Characterization Driven	**Offline Feedback-Driven**	**Online Feedback-Driven**	**Machine Learning**
Kind of Adaptation	Long-term learning	Short-term adaptation	Short-term adaptation	Long-term learning
Time Frame	Install time	Across compiles	Runtime	Across compiles
Affects	All applications	One application	One application	All applications
Adapts to	System	System Application	System Application Data	System Application PACE
Initiated by	RC tools	*various*	PAO	ML tools
Changes Behavior of	AAP, PAO, TAO	AAP, PAO, TAO	RTS	AAP, PAO, TAO
Persistence	Until next run of RC tools	Short-term	Records results for ML and PAO	Long-term

Characterization-driven adaptation is a simple form of long-term learning. It relies on algorithmic adaptation to pre-determined parameters. The compiler writers identify parameters and the PACE Characterization tools measure them. The compiler writers implement the transformations that use the results from the characterization tools. This strategy automatically adapts the transformation to the target system; it does not take into account any properties of the application or its data set.

In PACE, characterization-driven optimization performs its adaptation at installation time, when the PACE Characterization tools run. The adaptation can be repeated by re-running the characterization tools to generate a new target-system characterization. The results of this adaptation are persistent; they last until the PACE Characterization tools are re-run.

2.2.2 Offline Feedback-Driven Optimization

The second strategy for adaptation in the PACE Compiler is the use of offline feedback-driven optimization. This strategy produces a short-term adaptation. The actual mechanism for implementing feedback-directed optimization in PACE is simple. The Compiler Driver creates, for each application, a default optimization plan.[5] The AAP, PAO, and TAO each consult the application's optimization plan before they transform the code. Changes to the optimization plan change the behavior of these components. This design simplifies the implementation and operation of an adaptive compiler. It does not, however, provide a clear picture of how PACE will perform offline, feedback-driven adaptation.

In principle, any component in the PACE System can change the optimization plan for the current compilation of an application. We considered three strategies for controlling offline feedback-driven adaptation.

- The compiler driver may use an external adaptive controller to change the optimization plan across multiple compile-execute cycles. This mechanism can modify gross properties of optimization, such as the specific transformations applied and their relative order or the path through the compiler.
- Any phase of the compiler may contain an optimization pass that performs self-adaptation. For example, the implementation of unroll-and-jam in the PAO uses the PAO→TAO query mechanism to choose good unroll factors. It generates prospective versions of the loop and has the TAO estimate operation counts and register pressure. It adjusts the unroll factor to reduce operation counts and avoid register spilling.
- One phase of the compiler may change the optimization plan for another phase, based on the code that it generates. This capability addresses two different needs. It allows one phase to disable transformations that might reduce the impact of a transformation that it has applied. For example, the PAO might disable loop unrolling in the TAO to prevent the TAO from de-optimizing a carefully tiled loop nest. This adaptation would occur within a single compilation.

[5] The optimization plan is a persistent document that records compiler options for a specific application, including the path to be taken through the compiler, specific optimizations to be applied, and parameters that should be passed to those optimizations [1].

Alternatively, this capability allows one phase to provide feedback to another phase in the next compilation. For example, if the TAO discovers that the code needs many more registers than the target system (hardware + compiler) can supply, it might change the AAP's optimization plan to forbid inline substitution in the region. Similarly, it might tell the PAO to reduce its unroll factors.

While these offline feedback-driven adaptations can produce complex behavior and subtle adaptations, their primary impact is short term; they affect the current compilation (or, perhaps, the next one). They do not build predictive models for later use, so they cannot be considered learning techniques.[6]

2.2.3 Online Feedback-Driven Optimization

The third strategy for adaptation in the PACE System is the use of online feedback-driven optimization. Because the performance of optimized code can depend on the runtime state of the system on which it executes, even well planned and executed transformations may not produce the desired performance. Issues such as resource sharing with other cores and interference from the runtime behavior of other applications can degrade actual performance.

To cope with such dynamic effects, PACE includes a mechanism that lets the compiler set up a region of code for runtime tuning. The PAO establishes runtime parameters to control the aspects of the code that it wants the runtime to adjust. It generates a version of the code for that region that uses these control parameters to govern the code's behavior. Finally, it creates a package of information that the PACE Runtime needs to perform the runtime tuning. The PACE Runtime uses that information to find, at runtime, settings for the control parameters that produce good performance. The resulting application tunes itself to the actual runtime conditions.

As an example, consider blocking loops to improve locality in the memory hierarchy. The compiler could assume that it completely understood memory behavior and use fixed tile sizes. Alternatively, it can generate code that is parameterized around the tile sizes and use the PACE Runtime System's API for runtime adaptation to select good tile sizes at runtime.[7] The compiler's choice of bounds for the runtime search are, in turn, informed by values on effective cache sizes from the PACE Characterization Tools. The runtime search might use performance counter information, such as the L2 cache miss rate, from the Runtime System to judge performance.

Online feedback-directed optimization produces a short-term adaptation of the application's behavior to the runtime situation—the dynamic state of the system and the input data set. The technique, by itself, does not lead to any long-term change in the behavior of either the PACE System or the application. However, the PACE Runtime System records the final parameter

[6] In the ACME system, we coupled this kind of adaptation with a persistent memoization capability and randomized restart. The result was a longer-term search incrementalized across multiple compilation steps [20].

[7] The PACE Runtime System both defines and implements an API for online, feedback-driven optimization. The API lets the compiler register tunable parameters and suggested initial values, and provides a runtime search routine (an adaptive controller) that the PACE Runtime System can use to vary those parameters. The PACE Runtime will collect the data needed by the runtime search routine and ensure that it is invoked periodically to reconsider the parameter values.

values in its record of the application's performance history. Other components in PACE may use these final parameter values as inputs to long-term learning.

2.2.4 Machine Learning

The fourth strategy for adaptation in the PACE System is to apply machine learning techniques to discover relationships among target system characteristics, application characteristics, compiler optimization plans, and variations in the runtime environment. Machine learning is, by definition, a long-term adaptation strategy. Machine Learning tools will derive models that predict appropriate optimization decisions and parameters. In the PACE Project, we chose several specific problems to attack with Machine Learning techniques.

A central activity in the design of a machine-learning framework for each of these problems is the design of a *feature vector* for the problem—the set of facts that are input to the learned model. The PACE System provides an information rich environment in which to perform learning; the Machine Learning tools have the opportunity to draw features from any other part of the environment—the characterization tools, the compiler tools, and the runtime. The determination of what features are necessary to build good predictive models for various compiler optimizations is an open question and a significant research issue in PACE.

The goal of learning research in the PACE Project was to create a process that will automatically improve the PACE System's behavior over time. Offline learning tools examine records of source code properties, optimization plans, and runtime performance to derive data on optimization effectiveness, and to correlate source-code properties with effective strategies. This knowledge will inform later compilations and executions.

The PACE Compiler uses Machine Learning-derived models directly in its decision processes. The goal is to replace some static decision processes and some short-term adaptive strategies used by the compiler with a simpler implementation that relies on predictions from Machine Learning-derived models.

2.3 Major Questions in the PACE Project

The PACE Project focused on finding the answer to two fundamental questions.

1. Can we simplify the task of building and deploying high-quality optimizing compilers for new computer systems?
2. Can we reduce the effort required to achieve good application performance on those systems?

The PACE Project explored a specific approach to solving both of these problems: the use of characterization-driven optimization to re-target or re-purpose an optimizing compiler. In the context of PACE, we set out to answer the following questions.

3. Can a characterization-driven approach to compiler-based code optimization yield both portability and performance?
4. Can we measure performance-critical system characteristics in a portable way, with enough accuracy to use in a characterization-driven optimization system?
5. To what extent can we use feedback and adaptation — either runtime performance measurement or introspective insights from compiler components — to improve the results of characterization-driven optimization?
6. What kinds of runtime support are necessary to achieve reasonable application performance on modern computing systems?
7. Can machine-learning techniques predict optimization parameters as well as, or better than, measured performance characteristics?

The next two sections describe what we accomplished in the PACE Project. Section 3 focuses on the actual implementation and clarifies which parts of the design were implemented and which were not. Section 4 focuses on our results, with an emphasis on how the PACE work extended existing tools and advanced the state of the art. Section 5 concludes by revisiting these fundamental questions in light of our experience in the PACE Project.

3. METHODS, ASSUMPTIONS, AND PROCEDURES

In the PACE Project, our approach to resolving the fundamental questions was constructive—to design and build a retargetable optimizing compiler that achieved reasonably high levels of application performance by relying on characterization-driven optimization, feedback-directed adaptation, and automatic selection of transformations. This section focuses on the implementation effort.

The PACE System components were designed and implemented under a series of assumptions and constraints that were defined by the AACE BAA and by the AACE testing teams (University of California San Diego and University of Tennessee Knoxville) in consultation with the PACE and BAE AESOP development teams. The AACE testing teams presented the PACE team with a set of specifications for interfaces that we could assume were present on the target system. To a first approximation, the PACE implementation could assume the presence of C, C++, and Fortran compilers, a POSIX interface, OpenMP, and MPI. The detailed specifications should be found in the AACE testing teams' documentation.

3.1 The PACE Compiler

In the PACE Compiler, our phase one activity focused on: (1) selection of base infrastructure for the PAO and the TAO; (2) design of the PACE Compiler and its interfaces; (3) construction of the foundational infrastructure for the compiler; (4) implementation of the interfaces between the PACE Compiler and the other PACE tools; (5) implementation of the PAO-TAO interfaces, including the PAO→TAO IR Translator and the PAO→TAO query interface; and (6) implementation of a prototype of the AAP.

In phase two, we began to implement the transformations in the PAO and TAO, as well as the analyses that support them. Under the modified Statement of Work (see Appendix A), we implemented a limited set of transformations in the PAO (polyhedral transformations, parametric loop tiling, and unroll-and-jam) with a limited set of supporting analyses (array dependence analyses), and in the TAO (operator strength reduction, linear function-test replacement, graph-coloring register allocation, and improved vectorization). The remainder of the implementation is potential future work under other funding.

The PAO is implemented in the ROSE compiler infrastructure, while the TAO is implemented in the LLVM compiler infrastructure. These choices affected the design of individual components, as well as our distribution strategies for the resulting software. Both ROSE and LLVM have their own distinct open-source licenses. Components developed with the ROSE infrastructure are distributed under the ROSE license, while those developed with the LLVM infrastructure are distributed under the LLVM license.

3.1.1 The Application-Aware Partitioner

The long-term goal for the Application-Aware Partitioner was to manage the source code for an entire application and to refactor that source code in ways that reduce overall compilation times. In phase one, we built and tested a preliminary version of the AAP, with the necessary infrastructure for refactoring. The prototype AAP used a clustering heuristic along with profile data derived from the Runtime System to make refactoring decisions. It also performed

interprocedural array padding. Our plan for phase two was to use feedback from the PACE Runtime System and other components in the compiler to direct the refactoring process. Under the modified Statement of Work, we discontinued work on the AAP.

3.1.2 The Platform-Aware Optimizer

The primary goal for the Platform-Aware Optimizer is to prepare the application code for execution on the target system, with particular attention to efficient use of the memory hierarchy, mapping the application well onto inter-core parallelism, and exposing enough intra-core parallelism to achieve a high rate of utilization on each core. The PAO emphasizes the key PACE themes: characterization-driven optimization, feedback-directed optimization, and automatic selection of transformations for the specific application and target.

A large part of the PAO implementation effort was devoted to putting in place the infrastructure and interfaces to support the PACE vision: characterization-driven optimization, feedback-driven improvement, and automatic selection of transformations based on properties of the specific target system and application. The remainder of the effort went into implementation of specific analyses and transformations. To that end, we:

1. Modified the ROSE compiler infrastructure to simplify the process of building and installing those components needed for the PACE infrastructure and to improve portability across multiple systems.
2. Built the various interfaces between the PAO and other PACE System components. These interfaces include the PAO→TAO query interface, an interface to read data produced by the PACE Characterization tools, and an initial interface to read execution profile data produced by the PACE Runtime System. (This latter interface was also used by the AAP.)
3. Implemented PolyOpt, a subsystem that performs polyhedral analyses and transformations on affine loops. We extended the polyhedral techniques to cover some non-affine loops, as well. PolyOpt performs data dependence analysis, data alignment analysis, loop fusion, loop distribution, loop interchange, loop skewing, loop shifting, and loop tiling [3]. PolyOpt is integrated with the ROSE compiler infrastructure.

   PolyOpt operates as a subsystem of the PAO. (See the description under PAO Optimization Strategy below.) It takes as input a source code fragment in Sage AST form. It identifies subtrees of the AST that represent affine computations, transforms those subtrees to a polyhedral representation, analyzes and transforms the code in its polyhedral form, and finally converts the polyhedral form back into Sage AST. Analysis results from PolyOpt are mapped back into the Sage AST; dependence polyhedra are translated into conventional distance vectors. These annotations are then persistent throughout the PAO and are carried through the TAO by the PAO→TAO IR Translator.

   From a practical perspective, PolyOpt is the first fully automatic, end-to-end system for polyhedral optimization available in the ROSE compiler infrastructure. The development of PolyOpt included development or improvements to a number of packages, including the Candl package for data dependence analysis, the Pluto transformation set, and the CLooG package for polyhedral code generation, along with development of the PTile parametric tiling software and the code in ScopLib and PAST that supports polyhedral

intermediate representations. The PolyOpt code is currently in use by several other projects.

4. Implemented a set of high-level loop transformations that operate directly on the Sage AST. These transformations include parametric loop tiling and unroll-and-jam. These transformations rely on dependence analysis results from the polyhedral dependence analyzer, translated from representation as polyhedra to representation as classical dependence distance vectors.
5. Implemented support in the PAO for the TAO vectorization pass. The PAO performs a legality analysis for vectorization. It annotates loops where vectorization is legal with information on data dependences and data alignment.
6. Implemented a parametric tiling transformation that separates the specification of tile sizes from the actual transformations. The code produced by this transformation is suitable for either runtime selection of tile sizes [1] or for choice of tile sizes by an external model, as in [4].
7. Implemented a demonstration system that used the PAO→TAO query mechanism to select loop unroll factors based on low-level code quality concerns from the register allocator and scheduler.

PAO Optimization Strategy: The PAO contains both polyhedral analyses and transformations, and traditional non-polyhedral transformations. It uses each kind of transformation to its best advantage. PolyOpt is applied to those code regions that have a suitable form. Code regions that are not suitable for PolyOpt are subjected to traditional techniques. To accomplish this, the PAO takes the following steps with each function or procedure.

1. Identify "Static Control Parts" (SCoPs) in each function. There are multiple constraints that restrict which loop nests are eligible for inclusion in a SCoP.
2. Invoke the PolyOpt component separately for each SCoP. The PolyOpt component performs a number of loop transformations on each SCoP including fusion, distribution, interchange, skewing, permutation, shifting and tiling, in addition to identifying vectorizable loops that are marked as such and passed to the TAO for vectorization.
3. Invoke polyhedral dependence analysis on all non-SCoP loop nests.
4. Perform parametric loop tiling on non-SCoP loop nests (using RC information), when legal to do so.
5. Perform unroll-and-jam on innermost loops nests (using RC information and TAO cost feedback), when legal to do so.

Strategy for Automatic Selection of Transformations: To build a compiler where we could experiment with automatic selection of transformations, we established a clear separation of concerns among *legality analysis*, *profitability analysis*, and *program transformation*. This separation makes it far easier to reason about the selection and sequencing of operations. In practice, it limits the phase-order restrictions between passes in the optimizer.

To facilitate automatic selection, we decomposed the selection of high-level optimization problems into separate categories by effect—that is, we consider optimization of memory hierarchy behavior, of inter-core parallelism, and of intra-core parallelism as separate problems.

The formulation of each optimization problem is, then, based on quantitative cost models that are built on derived system characteristics and on application characteristics that include context-sensitive profiles. Multiple transformations may be used to optimize a single class of hardware resources (e.g., loop interchange, tiling, and fusion may all be used in tandem to improve memory hierarchy locality), and a single transformation may be employed multiple times for different resource optimizations (e.g., the use of loop unrolling to improve both register locality and instruction-level parallelism).

3.1.3 The PAO→TAO IR Translator

The PAO→TAO IR Translator takes an abstract syntax tree in ROSE's Sage AST format and translates it into a valid program in the low-level, linear, static single-assignment (SSA) form IR used by LLVM. It assumes that the input AST represents a valid C program. This tool allows direct and efficient communication between tools built on the ROSE infrastructure and tools built on the LLVM infrastructure. It is a vital link between the PAO and TAO that ensures the faithful transmission of both code and derived annotations.

The alternative process, pretty printing the Sage AST into C code and then parsing the C code into LLVM IR is problematic. The PAO develops knowledge about the input program that may have no expression in C. Examples include data dependence and data alignment information. A given loop nest might have multiple legal translations into C, some of which lose information. For example, the Sage-to-C translation might convert array-style code to pointer-style code. (In a naïve compilation, the pointer code may, in fact, be faster.) The pointer-style code is likely to be inferior for further optimization because of the difficulty of analyzing pointer-based references.

The PAO→TAO IR Translator is distributed through the ROSE Project's web site. A link to the distribution can be found on the PACE web site.

3.1.4 The Target-Aware Optimizer

The Target-Aware Optimizer has several specific goals. It should include optimizations that specifically target the code produced by the PAO, such as the output from polyhedral analysis and transformation and the output from vectorization activities. The TAO should generate C code that carefully matches the capabilities and assumptions of the native C compiler. The TAO should provide a testing ground for low-level, characterization-driven, scalar optimizations. Finally, the TAO should provide direct feedback to the PAO on the impact that PAO optimization decisions have on the effectiveness of low-level optimization.

As groundwork for our implementation activity, we first completed a study on LLVM's optimization effectiveness. The study used the Nullstone and SPEC benchmarks, along with some smaller microbenchmarks, to assess the quality of LLVM's output code.[8] The study identified several areas where improvements to LLVM would have an impact. To this end, we built a series of enhancements to the LLVM 3.0 compiler infrastructure.

1. *Operator Strength Reduction and Linear Function Test Replacement:* The PAO transformations, both in PolyOpt and in the non-polyhedral optimizer, produce loop nests that contain many similar and related array references. Operator Strength Reduction

[8] Specific provisions in the NULLSTONE license prohibit us from publishing the results of those tests. We can, however, discuss in general terms the weaknesses that the tests revealed.

directly attacks the cost of array access by simplifying the code for array address computations. Operator Strength Reduction can significantly reduce the overhead of computations in array intensive loops.

LLVM's original strength reduction algorithm was limited in its scope and effectiveness. We implemented the more general Cooper-Simpson-Vick algorithm that performs strength reduction and linear function test replacement [5] [6]. This pass can run inside a standard LLVM 3.0 implementation.

2. *Short SIMD Vector Code Generation*: LLVM had no provision to make systematic use of short vector instructions, such as the Intel SSE operations. Vectorization was an explicit requirement of the AACE program. The TAO team worked with the PAO team to implement a vectorization pass for LLVM, based on the dynamic programming approach to vector code generation [7]. The pass relies on analyses in the PAO's PolyOpt package for information on legality and on results from the PACE Characterization Tools for profitability analysis (see Section 8.3.3 and Appendix B of *PACE: Status and Future Directions* [2]). The PAO analysis results are translated, first from polyhedral form to conventional dependence vectors, and then from referring to the Sage AST to referring to constructs in the LLVM IR.

   Because it relies on information from other parts of the PACE Environment, this pass can only run inside a complete PACE installation; it will not work in a standard LLVM implementation.

3. *Register Allocator*: To improve LLVM's ability to handle the complex loop nests produced by the PolyOpt tools, we implemented a Chaitin-Briggs graph-coloring register allocator [8] that includes rematerialization [9]. Improved register allocation was of particular interest to the PACE Project because the complex loop nests produced by the PAO's transformations tend to have high demand for registers.

   LLVM includes several previous, but less powerful register allocators. The Chaitin-Briggs implementation can run inside a standard LLVM 3.0 implementation.

4. *PAO-TAO Feedback:* To provide answers in the PAO→TAO query mechanism, the TAO team created a synthetic back end that can provide information about the quality of code compiled for a specific code fragment. To use this facility, the PAO wraps the code fragment in a function, annotates it with auxiliary information that includes alias information and array dependence information, and sends that package to the TAO. The TAO optimizes the code and produces estimates of several performance parameters, including demand for registers, the amount of spill code required, the critical path length, and the cost of SIMD vectorization. Details can be found in [2], Section 8.3.5.

   Because it relies on information from other parts of the PACE Environment, this pass can only run inside a complete PACE installation; it will not work in a standard LLVM implementation.

Our study of LLVM optimization effectiveness revealed some shortcomings that we did not, because of limited time and resources, address. Foremost among these is the lack of systematic code motion in LLVM. (Many of the best practice algorithms are not well suited to SSA form.) We considered several strategies and settled on an implementation of the Cytron, Lowry, and Zadeck algorithm [10]. We began an implementation of this algorithm, but ran into significant

problems transforming the LLVM IR. In particular, the implementation needs to replicate large amounts of code, which creates code that is not in correct SSA form. The process of creating this non-SSA-form code and then repairing it so that it is correct SSA-form code made it difficult to design, difficult to implement, and difficult to reason about the results. In the end, we concluded that implementing this particular transformation in LLVM was untenable. Were we to make another attempt at adding code motion to LLVM, we would probably start from one of the algorithms that has been specifically formulated for SSA form.

3.2 The PACE Runtime System

The design for the PACE Runtime System included four capabilities: measurement of performance of generated code, attribution of performance measurements back to the source program for feedback-directed optimization, analysis of rate-limiting factors for different code regions to provide guidance for offline tuning, and online tuning of program performance via parameter selection. Work on the PACE Runtime System began late in the project and work on the last two aspects was not completed. As a result, the principal results of work on the PACE Runtime System are support for online performance measurement and support for attributing performance measurements to support feedback-directed optimization.

Support for Performance Measurement: The PACE Runtime System was designed to measure program performance for compiler-generated code, assess performance and scalability bottlenecks, and attribute performance characteristics to the program source code. Attribution to source code leverages compiler-generated mappings, binary analysis of the generated code, and unwinding of the dynamic call stack at runtime. This enables the performance measurement subsystem to support attribution of performance at multiple levels of granularity, including attribution to full calling dynamic contexts, procedures, inlined functions, and loop nests, as well as individual source code statements.

Performance measurement, analysis, and attribution capabilities for the PACE Runtime System were integrated into Rice University's HPCToolkit performance tools—a standalone toolkit for application performance analysis. New capabilities developed for PACE included expanding the set of architectures supported by HPCToolkit. For this purpose, a new version of the libunwind interface was integrated into HPCToolkit to add the capability to provide support for call stack unwinding on the ARM and Itanium processor families, both of which were represented in the set of AACE test systems. The libunwind support complements existing stack unwinding capabilities already present in HPCToolkit for MIPS, PowerPC, and x86, which leverage custom binary analyzers. Additionally, as part of work on PACE, we extended the unwinding capabilities for PowerPC to support 64-bit Power architectures, which include the Power7 and Blue Gene/Q processors.

An important aspect of the PACE Runtime System was the ability to measure and identify performance bottlenecks in parallel code. A key challenge for modern processors is to make efficient use of thread-level parallelism on multicore architectures. As the PACE Project began, quad-core processors were standard. Today, AMD's Interlagos processors have 16 cores; the emerging Blue Gene/Q processor has 16 cores, each with 4-way hardware multithreading; and the forthcoming Intel MIC chip will have over 50 cores, each with 4-way hardware multithreading. Support for quantifying performance losses due to insufficient threaded parallelism and attributing losses to program regions is essential guidance needed for either manual or automatic tuning.

As part of PACE, we refined ideas for diagnosing performance bottlenecks in multithreaded programs by pinpointing program regions with insufficient parallelism. We extended our measurement system to maintain information about instantaneous thread idleness and refined a technique that we call "blame shifting" to accumulate and attribute measurements of idleness and work to program regions – shifting blame from symptoms of performance losses due to idleness (i.e., spin waiting for work or for access to a critical section guarded by a lock) to their

underlying causes – insufficient parallelism and resource contention for locks. As the PACE Project was winding down, we explored the utility of these ideas in the context of OpenMP codes, as well as programs that use heterogeneous cores in systems with both CPU and GPU components. The fundamental ideas and implementation to support blame shifting for multithreaded programs is complete.

Support for Feedback-Directed Optimization: A second aspect of the PACE Runtime System is to attribute performance measurements back to an abstract syntax tree representation of the program source code to support feedback-directed optimization. To provide this capability, measurement data collected by the PACE Runtime System is parsed and attributed directly to the Sage AST representation of the ROSE compiler infrastructure, which serves as a substrate for the PACE Compiler.

Prior to the PACE Project, the ROSE compiler infrastructure contained some support for importing performance measurements from HPCToolkit. This support was developed several years ago by members of the ROSE team at LLNL and not maintained since. As a result, it was several years out of date and unable to digest call stack profiles generated by the current version of HPCToolkit. As part of PACE, we updated the support in ROSE to import HPCToolkit's new call stack profiles and put in place regression tests to verify this functionality.

Identifying Rate Limiting Factors: An important goal of the PACE Runtime System was to develop support for identifying the rate limiting factors associated with various code regions. To address this issue, we collaborated with Jim Browne (University of Texas) and Martin Burtcher (Texas State University) to develop support for this capability based on performance measurements collected by the HPCToolkit measurement subsystem. The PerfExpert tool developed by Browne and Burtcher uses measurements from hardware performance counters in conjunction with machine characteristics (e.g., pipeline latencies, cache latencies, TLB miss latency, etc.) to calculate LCPI – local cycles per instruction – for individual loop nests. The performance losses associated with each factor (e.g., L1 cache misses, L2 cache misses, L3 cache misses, TLB misses, floating point pipeline stalls, mispredicted branches, etc.) is quantified by taking the product of each kind of hardware counter event measured with its associated delay. Based on this information, PerfExpert qualitatively assesses performance, rating it from poor to excellent. Rate limiting factors are identified by assessing the magnitude of the losses associated with each cause of performance loss (e.g., cache misses, pipeline stalls, mispredicted branches, etc.). Based on the type of loss and the structure of the code (e.g., single loop, nested loops) identified by HPCToolkit's binary analysis, PerfExpert identifies some potential code transformations that might improve performance. As future work, it would be interesting to integrate the LCPI assessment of rate limiting factors and transformation recommendations by PerfExpert into the PACE Compiler.

As part of phase two, our plan for PACE was to also use information collected by the PACE Runtime System to guide the Application Aware Partitioner (AAP) to direct program refactoring. While measurement data collected by the PACE Runtime System was read and attributed to the internal program representation of the AAP, under the modified Statement of Work, we discontinued the AAP and our work on measurement-directed refactoring was abandoned.

Additionally, phase two plans for PACE included building a harness in the PACE Runtime System for runtime tuning by online parameter selection (e.g., tile size selection). Under the Revised Statement of Work, we abandoned work on this capability.

3.3 The PACE Characterization Tools

The goal of the PACE Characterization Tools subproject was to develop a set of standalone, portable tools that could reliably measure the system characteristics needed by the other PACE tools. Thus, the project had three key thrusts: leading the effort to identify those performance-critical system parameters; developing microbenchmarks to expose those parameters; and packaging the tool set in a way that let the AACE Testing Teams evaluate the work and let the other PACE tools use the results. See Chapter 2 and Appendix A of *PACE: Status and Future Directions* [2] for a detailed discussion of the design and implementation of these tools.

3.3.1 Identifying Characteristics

The Characterization Tools team worked with the PACE Compiler and Runtime teams, as well as the other AACE teams, to identify a set of system parameters that would affect optimization and tuning. The AACE testing teams also helped to identify parameters and to standardize definitions across the AACE program.

As a general rule, the Characterization Tools only measure a characteristic if it can be used in the source-to-source optimization provided by the PACE Compiler (or in the Runtime System). This rule eliminates some parameters from consideration and changes the definitions of others. For example, the tools measure the number of simultaneous live values that the native compiler can maintain, rather than the number of hardware registers available on the target processor. In a compiler that has a native backend, the number of registers is an important number. In a system that relies on the native compiler to perform the final translation step, the capabilities of that compiler matter more than the capacities of the underlying hardware.

While the tools should, in principle, measure any property of the target system and its software environment that might impact optimization, practical limits imposed by the project's schedule, its budget, and the plans of the other tools limited the number of characteristics that we measured. Tables 2.2 in *PACE: Status and Future Directions* [2] summarize the characteristics measured by the PACE tools. As a design goal, we only wanted to measure parameters that would be used by some component in the PACE System. Table 2.3 in *PACE: Status and Future Directions* [2] shows potential uses of each characteristic that the tools measure.

3.3.2 Microbenchmarks and Analyses

The PACE Characterization Tools consist of a set of microbenchmarks. Each microbenchmark consists of code designed to expose the characteristic behavior and analysis that distills a single number from the measured system behavior. *PACE: Status and Future Directions* [2] describes the various microbenchmarks in Chapter 2 and Appendix A. Two examples will make clear the wide differences in approach that are required for different characteristics.

The cache capacity microbenchmark measures the amount of time required to access thoroughly an array of a given footprint. By gathering data on successively larger footprints, the microbenchmark builds up a curve that describes memory latency as a function of footprint. It repeats each test many times and retains the minimum time at each footprint. This strategy lets the microbenchmark sample a large number of virtual-address to physical-address mappings. The automatic analysis smooths that curve and fits a step function to it. The step function indicates the number of levels of cache memory in the system, the effective latency of each level of cache,

and the effective capacity of each of those cache levels, defined as the footprint beyond which latency begins to rise.

By contrast, the test for live ranges—the number of simultaneously live values that a C program can have before register spilling degrades performance—does not need to execute the compiled code. The microbenchmark compiles a series of carefully designed codes and compares the size of the compiled code. Each version of the code contains a long sequence of additions. The codes differ in the pattern of names used in the additions; the patterns produce a different number of simultaneously live values. To discover the characteristic value, the analysis finds the first version where the code size jumps—it jumps because the compiler had to insert loads and stores to spill and reload live values. That version is one live range above the compiler's effective limit.

Both parts of the microbenchmark—the code that demonstrates the effect and the code that analyzes the results—require careful design and testing across multiple ISAs, multiple chip models, multiple system configurations, and multiple compilers. It is difficult to achieve the twin goals of accuracy and portability in a single code.

Our goal for the PACE Project was to achieve a 75% measured accuracy for the measured parameters by the end of phase one and then raise that accuracy to 90% by the end of phase two. The AACE testing teams measured the accuracy of the PACE Characterization Tools on a set of systems whose properties were unknown to the PACE team. Both testing teams found that the PACE tools met their standards for accuracy—the 75% accuracy listed as our goal in Section 3.3.1.[9] Most of the individual tests achieved accuracy better than 90%, in our internal trials and in the end-of-phase tests. We had one microbenchmark that had accuracy problems; those problems are easily correctable. Table 2.2 on page 16 in [2] shows the characteristics used in those tests along with the page numbers in Appendix A [2] where additional information is provided for each of the characteristics.

For phase two, our plan was to focus on better metrics for shared resources under controlled loads on other cores, to measure effective latencies for a variety of thread-related primitives, to measure several more compiler-related characteristics, and to investigate the impact of "processor throttling" behavior on Intel Nehalem and Westmere processors. Table 2.3 on page 18 in the PACE Preliminary Design Document [1] details most of these characteristics.

We achieved partial results on many of these characteristics. Under the Revised Scope of Work, we shifted the remaining characterization resources from exploration of new characteristics to ensuring that the phase one software was distributed in a useful and accessible form.

On the issue of characterization under load, our initial results were disappointing. We saw large variances in the measured values for effective cache size, with constant-sized loads on the other cores. We surmise that the variances are, in fact, real, and that they arise due to the impact of virtual-to-physical address mapping on all of the cores, coupled with the fact that the shared caches are physically mapped. We have ideas on how to test this hypothesis, but it will require the use of superpages—a distinctly non-portable feature not included in the POSIX API.

On the issue of processor throttling, we found that it was difficult to write a microbenchmark that consistently triggered the behavior. Either our systems have adequate cooling for the processor

[9] Each of the testing teams had its own systems and scoring metrics. Their testing and scoring methodology is described, presumably, in their final reports. The AACE program established a passing score of 75% accuracy in phase one.

chips, or the effect almost never occurs. In either case, our inability to trigger the behavior suggests that compiled code is not particularly likely to encounter this problem.

3.3.3 Packaging the Characterization Tools

The PACE Characterization Tools were designed to produce output for three different end users: the AACE testing teams, the other PACE tools, and interested individuals. The interface for the AACE testing tools specified a tight set of requirements so that the PACE Characterization Tools could fit into an automated testing harness. Under the Revised Scope of Work, we stopped support for that interface and re-packaged the tools as a standalone system that produces results in two forms: a human-readable file and an automatic interface used by the other PACE tools.

The distributed software unpacks into a directory. The user invokes it by running a “makefile”. The tools write their results into an ASCII text file. The PACE tools interface reads that file to obtain data.

3.4 The PACE Machine Learning Effort

The PACE Machine Learning (ML) team identified four problems to attack as part of the PACE effort [1]:

1. Finding the optimal tile size to maximize performance of a loop nest
2. Selecting compiler flag settings for good performance of an application program, given a fixed optimization sequence
3. Predicting an application program's performance based on characteristics of the program, the target system, and the compiler
4. Finding a sequence of compiler optimizations for an application program that yields performance improvements over some baseline sequence

We prioritized these problems, taking into account the overall project schedule and the availability of other PACE components. Problems one and three were attacked first, with plans to tackle problem two next and problem four last.

Consider problem one. To train a model so that it can predict good tile sizes, we need extensive data. The ML team took a single benchmark code—a tiled FFT—and measured execution times on one machine configuration at the complete set of reasonable tile sizes. The PIs proposed that we consider a predicted execution time as a "good" prediction if it is within five percent or less of the actual measured time.

To obtain accurate and consistent data, all runs were made using the RCacheSim simulator, rather than relying on native execution. This strategy eliminated any impact of other system loads from the data for an individual run. It also decoupled progress on the ML tasks from progress in the PACE Compiler and PACE Runtime System. The data from all these runs was then validated—to ensure completeness and to verify that the execution times made sense.[10]

The ML effort used artificial neural networks with self-organizing maps to build models. These models were compared against linear least squares regression, cubic least squares regression, and a conventional multi-layer perceptron using back propagation [1].

Next, multiple rounds of learning experiments were performed to determine the sampling resolution of the training data in the parameter space that was needed for accuracy in the predicted tile sizes.

1. Predict the performance of the application on a hardware configuration based on data from the same application and same hardware with different input data.

   In these tests, our neural network models predicted execution times based on a given tile size. The accuracy of the four models is as follows:

   - Our neural network made good predictions for 88.7% of the trials.

[10] The individual timing runs were run on a large cluster at Rice. Because of the sheer number of individual executions and the vagaries of the batch queuing system, some individual runs were terminated early and produced misleading results; others failed for ancillary reasons. Our validation process discovered these problems and allowed us to repeat the suspicious runs and replace the questionable data with correct data.

- Linear least squares regression made good predictions for 7 to 8.5% of the trials.
- Cubic least squares regression made good predictions for 12.8% of the trials.
- Back propagation made good predictions for 59.6 to 65.6% of the trials.

Based on the predicted execution times, the neural network model would predict tile sizes close to the optimal tile sizes, as would the back propagation model. The least squares models did not predict well enough to produce good tile sizes.

2. Predict the performance of the same application with different input data on other hardware configurations. In this context, a hardware configuration consists of a set of sizes for the L1, L2, and L3 caches, as well as the TLB. Again our neural network model did quite well. It made good predictions for 95.8 to 97.4% of the cross-machine predictions.

   The increase in accuracy for the cross-machine trials over the single-machine trials is partly due to the fact that the predictions were for another configuration[11] and partly due to improvements in learning—refinement of the models during the single-machine experiments.

In summary, we achieved roughly 95% success at predicting execution time as a function of configuration, input data, and tile size—a good result for both problems one and three.

When the project ended, we were working on cross-program predictions—using data from one application to predict the performance of another application, both on the same configuration and on other configurations. Unfortunately, we ran out of time before completing those experiments.

[11] We would expect prediction accuracy to vary somewhat between architectural configurations.

3.5 Software

The PACE Project produced a set of software artifacts. All are available from the "SOFTWARE" link on the PACE web site (http://pace.rice.edu). Some of them have been re-integrated into the distributions for the underlying systems, in particular ROSE and HPCToolkit.

PACE Compiler:

- **AAP:** The code for the AAP is built on top of the open-source Xerces XML parser and the standard GNU utilities Bison and Flex. The code is available from the PACE web site under the Rice open-source license.
- **PAO:** The PAO is built on top of the ROSE Compiler Infrastructure, an open-source system designed to facilitate the implementation of compilers and source-to-source translators. The code for the PAO is included in the PACE Compiler distribution. Most of the code for the PAO falls under the ROSE open-source license. Any code not covered by the ROSE licenses falls under the Rice open-source license.
    - The PolyOpt tools are also available from the ROSE web site.
    - The PAO→TAO IR Translator is also available from the ROSE web site, where it is described as a translator from the ROSE Sage AST to LLVM IR.
- **TAO:** The released components of the TAO are built on top of the LLVM compiler infrastructure. They are available from the PACE web site under the LLVM 3.0 license.
    - Most of these components are included in the PACE Compiler distribution.
    - The operator strength reduction/linear function test replacement pass and the Chaitin-Briggs graph-coloring register allocator are available as standalone distributions from the PACE web site. Both passes are usable in LLVM without any other PACE software.

PACE Runtime System:

- **Performance Measurement:** The HPCToolkit performance tools, including enhancements developed to support the goals of the PACE Project, are available as a standalone distribution. HPCToolkit leverages a large collection of third-party open source software including GNU Binutils, the Apache Xerces XML parser, libDWARF—a library for reading compiler-generated debugging information, libelf—a library for reading and manipulating application executables in ELF format, libunwind—a library that provides an API to determine the dynamic call chain of an executing program, symtabAPI—a library for reading and manipulating symbol table information for executables, libxml2—an XML C parser used by symtabAPI, OpenAnalysis—a library that supports representation independent analysis of programs, and the Apache Xerces XML library, which HPCToolkit uses to read and validate XML. Code for the HPCToolkit performance tools is available separately under a simple open-source license.
- **Support for Feedback-Directed Optimization:** Support for parsing and attributing measurement data collected by the PACE Runtime System's measurement subsystem (HPCToolkit) to the Sage AST of the ROSE compiler infrastructure was integrated

directly into ROSE. This code will be available as part of the standard distribution of ROSE provided by Lawrence Livermore National Laboratory.[12]

PACE Characterization Tools

- The PACE Characterization Tools are a standalone package. The package can be downloaded from the PACE web site. The software is available under the Rice open-source license.

PACE Machine Learning Tools

- The ML project built on software developed by Dr. Erzsébet Merényi in earlier projects, under funding from the National Aeronautics and Space Administration (NASA). The software was original applied to analysis of hyperspectral remote sensing images, a problem with a high-dimension feature space and complexly structured data. The software modules are accessible for licensed use; contact Dr. Merényi for details (erzsebet@rice.edu). A link is included on the software page of the PACE web site.

[12] As of this writing, the code for reading and attributing HPCToolkit performance measurements is undergoing extensive multi-platform regression testing, which is required to commit code to the master software repository for the ROSE project at Lawrence Livermore National Laboratory.

4. RESULTS AND DISCUSSION

4.1 The PACE Compiler

The primary goal for the PACE Compiler was to optimize an input program for a wide range of hardware characteristics in a given platform, as identified by the PACE Characterization Tools, and a representative set of program inputs, represented in the form of execution profile information from the PACE Runtime System. We recognized early in the process that achieving this goal would require tight integration between high-level transformations and low-level transformations. This observation motivates many aspects of the PACE Compiler's design, including the specification of separate platform-aware and target-aware optimizers and the query interface between them [1].

Many of the individual transformations proposed for the PACE Compiler have already appeared in past work. Their application in the PACE Compiler is novel for two reasons. First, the PACE implementation modified these transformations to use measured characteristics of the target hardware/software system. Second, the PACE Compiler design and implementation focused on the question of automatic selection of transformations and combinations of transformations. Together, these two strategies simplify retargeting the PACE Compiler for a new platform and re-tuning it for application or system properties. These strategies are critical for the PACE Compiler because we cannot know, a priori, the target platforms for the compiler. This situation differs from that of a vendor-produced compiler, where compiler releases are tightly matched to specific processor releases.

Novel Contributions:

- The use of characterization information to drive high-level transformations is novel. In prior art, platform constants are usually hard-coded in compiler files. PACE built a compiler environment in which platform constants were obtained by automatic tests that measured the resource characteristics of a target system through the lens of the native platform compiler. In [4], we observed that the use of these measured characteristics could lead to more effective code optimization than the use of publicly advertised platform constants.
- PolyOpt includes approximation techniques that extend polyhedral analysis and transformation techniques to certain non-affine computations. These extensions increase the set of codes that the PAO can optimize with PolyOpt.
- PolyOpt includes novel algorithms that generate parametrically tiled code—loop nests where the tile sizes are parameters rather than fixed constants [11] [12] [13] [4]. The resulting code makes it easy for a runtime mechanism to adapt tile sizes to compensate for the actual runtime conditions [13] or for a tool external to the compiler, such as a model generated by machine learning techniques, to provide tile sizes.
- Working with parametrically tiled loops, we developed a new approach to static estimation of tile sizes that significantly reduces the size of the search space. It uses the compiler to generate analytical lower and upper bounds on tile sizes that should be considered for optimality. Further, the combination of these lower bounds and the values measured by the PACE Characterization tools leads directly to a set of tile sizes that can be automatically selected by the compiler, either for use as default tile sizes or for use as a starting point for

searching for optimal tile sizes. We are unaware of any prior art that follows this approach of compiler-generated analytical bounds on tile sizes.

- PolyOpt's implementation was designed to accommodate iterative compilation [14], allowing the combination of empirical search and model-driven optimization. As part of this effort, we developed an advanced theoretical characterization and exploration of a large space of possible polyhedral optimizations [15]. PolyOpt's flexibility opened the way for experiments that used machine-learning techniques to address the problem of selecting polyhedral optimizations. The space of possible combinations of transformations is enormous, making systematic search prohibitively expensive and causing most work in iterative compilation to rely on either manually constructed heuristics or inefficient automatic techniques such as genetic algorithms. To improve on search quality without paying the price of genetic algorithms, we applied machine-learning techniques to build models that predict the performance of transformed code from a characterization of the program's dynamic behavior [16]. This work holds the promise of making good selections of polyhedral transformations for a region of code in an efficient way

- The PolyOpt team also considered the problem of selecting tile sizes for arbitrary programs by sampling in the full space of tile sizes. We developed a technique that builds a performance predictor for a specific program, using statistical machine learning to train an artificial neural network to predict the distribution of execution times. This technique improves significantly over the variability of random search [12].

- The use of dependence information from polyhedral analysis to guide non-polyhedral transformations is, in practice, novel. In current practice, polyhedral tools are strictly segregated from non-polyhedral parts of the optimizer. While the polyhedral frameworks are quite precise, they are only applicable to a restricted set of code structures. Non-polyhedral frameworks are more robust with respect to code structures, but usually employ lower-precision analysis techniques.

  In the PAO, we showed how to convert PolyOpt's dependence polyhedra into the dependence vectors that are more commonly used in non-polyhedral frameworks, even for code structures that do not match the traditional requirements of polyhedral frameworks. We are unaware of any prior work that has taken this approach. This approach produced improvements in the dependence analysis performed in the non-polyhedral optimizer in ROSE, which were used in the PACE Compiler to check data dependence legality for tiling and unrolling.

- The PAO→TAO query mechanism allowed us to build an unroll-and-jam transformation that used feedback from the low-level compiler to pick better unroll factors that would be otherwise possible. Prior implementations of unroll-and-jam have had problems selecting good unroll factors. Too large a factor leads to problems in code generation that erase many of the benefits of unroll-and-jam. Too small a factor leaves additional performance unrealized. The practical difficulty that arises in building a feedback mechanism to guide unroll-factor selection is that the dependence analysis necessary for legality analysis on unroll-and-jam can be performed more effectively as a near-source level of abstraction (in the PAO) while the detailed cost estimates necessary for profitability analysis can be performed more effectively in a near-assembly level of abstraction (in the TAO).

To address this challenge, we leveraged the unique PAO→TAO query mechanism to provide the PAO with fine-grained cost-estimates on actual code sequences that result from various unroll factors. To obtain an estimate, the PAO creates a synthetic function for a specific unroll configuration, and then passes it to the TAO via the PAO→TAO IR translator. The TAO then optimizes the code and estimates the cost of the synthetic function along multiple dimensions (register spills, instruction counts), reporting the cost back to the PAO. The PAO repeats this process for multiple unroll configurations, enabling the PAO to converge on a set of optimized unroll factors. To our knowledge, this is the first time that anyone has built a system that uses a feedback cycle between high-level and low-level optimizers. We are unaware of any prior systems that use this kind of internal, introspective, cross-optimization feedback in a systematic way.

- The vectorization pass that we built for LLVM is a new capability—both a new capability for LLVM, which had no systematic vectorization facility, and a new idea in that it used dependence analysis and alias results from the high-level optimizer (from the polyhedral optimizer, translated into the non-polyhedral framework and then translated by the PAO→TAO IR Translator into LLVM IR) in low-level vectorization. The AACE Program specifically highlighted vectorization as a key capability; this cross-tool implementation directly addresses that specific issue.

The implementations of Operator Strength Reduction and the Chaitin-Briggs register allocator did not create significant new algorithmic results. They do, however, improve the code optimization capabilities of LLVM, a major open-source compiler infrastructure.

4.2 The PACE Runtime

The PACE Runtime measurement subsystem, integrated into the HPCToolkit performance tools, enables us to measure and attribute program performance characteristics on multicore systems to a wide array of processor architectures. This software is in wide use for measurement and analysis of application performance bottlenecks.

Preliminary tests of the blame shifting approach for pinpointing bottlenecks in thread-level parallelism enabled us to pinpoint and diagnose performance losses in an algebraic multigrid solver (AMG2006) on a multicore CPU and in a UHPC unstructured shock hydrodynamics code (LULESH) on a hybrid CPU+GPU system. Work on OpenMP is continuing under separate support from the DOE Center for Scalable Application Development Software to relate program measurement data back to a source-code level view from an implementation-level view. Reconstructing a source-level view requires integrating different runtime views of the main program and worker threads. Work to generalize our prototype for these capabilities to support an arbitrary OpenMP runtime system is ongoing. We expect that an eventual product of this work will be a new tools API for the OpenMP standard. Work with the OpenMP standards committee to define a standard application-programming interface for tools is ongoing as of this writing. A preliminary manuscript describing support for using blame shifting to analyze performance of OpenMP was written; at this writing, it is currently under revision.

The capabilities for identifying rate-limiting factors based on HPCToolkit's measurements using hardware performance counters have proven popular and the PerfExpert tool developed at UT and Texas State for this purpose has attracted significant interest. We aim to continue collaboration with our external collaborators on PerfExpert and extend this work to emerging manycore systems, including the Intel MIC.

4.3 The PACE Characterization Tools

The PACE Characterization Tools differ somewhat from the other subprojects, in that the first release of the tools underwent rigorous testing by the AACE testing teams. As part of the transition from phase one to phase two, we provided the testing teams with a release of these tools, in a format designed to plug directly into their internal testing harness. The AACE testing teams ran the PACE Characterization Tools on a set of systems that were unknown to us and measured the accuracy of the results. The PACE Characterization Tools passed the phase one trials, where the minimal passing score was set to 75% accuracy, under metrics defined by each of the AACE testing teams. Most of the individual microbenchmarks from PACE scored better than 90% accuracy.

The PACE Characterization Tools represent a significant advance over the prior state-of-the art in two respects. First, the tools provide greater coverage, both in terms of the number of characteristics measured and the number of systems across which the tools were demonstrated, than prior systems. Second, the automated analysis tools embedded in the microbenchmarks go beyond what has been applied in other projects.

To focus on the memory-hierarchy benchmarks, the best-known and most widely cited work is the X-Ray system developed by Pingali et al. [17]. The PACE Tools measure more memory parameters than the X-Ray tools. The X-Ray tools do not measure TLB effects. The X-Ray tools rely on superpages—a distinctly non-portable feature—to measure associativity and capacity in physically mapped caches, which include most of the upper level caches on modern microprocessors. The PACE Tools reliably find capacities in physically mapped caches using only portable (e.g., POSIX) features. Like X-Ray, the PACE approach would need to use superpages to discover associativity in physically mapped caches.

Two major advances in the Characterization Tools work are the robust analysis developed for the memory-hierarchy measurements, and the clear separation and isolation of effects.

Thresholds and Heuristics: Based on a close reading of prior papers and, in some cases, the code for these prior systems, it is clear that their analysis techniques relied on some combination of heuristics and thresholds—such as “a jump of ten percent in latency signals a new level in the memory hierarchy”. Our experience in testing the PACE Tools across more than twenty systems showed that such heuristics were unreliable. The ratio of operation latencies, such as a level-one cache hit to a level-two cache hit, change significantly across different systems and different generations of hardware. This effect makes any particular threshold suspect; in fact, for every threshold that we tried in the cache capacity benchmark, we found a system where that threshold did not work.

In contrast, the PACE Tools use an analytical technique based on smoothing the data and building a histogram to determine the number of levels present in the hierarchy, either cache or TLB; prior systems take the number of levels as a given. Then, they go back to the original data and fit it with a step function of the appropriate number of levels, which represents the ideal and expected response curve. The analysis discovers the number of levels in the cache from the latency, and then fits a model to the data.

These techniques can produce surprising results. For example, the PACE Tools characterize the level-three cache of a Power7 processor as comprising a small level-three cache and a much larger and slower level-four cache. Careful reading of the Power7 literature shows that the level

three cache is managed as a core-local adaptive victim cache, coupled with a shared cache. The level-three cache is implemented as a distributed cache with a non-uniform access time. Thus, the model suggested by the PACE Tools is a surprisingly accurate portrayal of a very complex cache structure.

Isolation of Effects: We found that modern memory hierarchies are sufficiently complex that the effects of different memory structures obscure each other. To combat this confusion, the PACE Characterization Tools carefully isolate each kind of behavior.

For example, the level-two cache and the TLB on an Intel Nehalem chip both cover 256 KB of RAM. For a variety of reasons, cache latency begins to rise at about 228 KB, so a combined test, such as the one that X-Ray uses (even with superpages) will produce a muddled picture of behavior between 200 KB and 300 KB. By carefully isolating cache behavior from TLB behavior, the PACE Tools obtain a clear picture of each, with the result that they discover cache latency rising at 228 KB and TLB latency jumping at 256 KB.

A similar confusion arises in some of the AMD x86 chips, where the parameters of the cache system cause false positives in the TLB test. As detailed in [18] and [19], we developed a simple extension to the testing methodology that exposes such false positives.

Sharing: The major shortcoming of the implemented PACE Characterization Tools is that they do not address well the effects of sharing in a multicore environment. Clearly, load from other executing programs can affect the availability of resources, such as thread-level parallelism, cache capacity in a shared cache level, or communications bandwidth. Before characterization-driven optimization can reach its full potential, we need methods to measure effective capacities under varying system loads.

Two major issues arise in making such measurements. First, we need a way to quantify and control system load. That problem was clearly beyond the scope of the PACE Project. Second, the microbenchmarks need an interface that allows them to understand and control the placement of specific threads. The POSIX API does not provide clean and reliable mechanisms to describe thread placement and to control it.

Using thread affinity interfaces that are not included in the POSIX APIs, we were able to build microbenchmarks that derived an accurate picture of sharing within the cache hierarchy of a multicore processor. The tool could discover which levels of cache were dedicated to a single core and which were shared among cores. It could discern how many cores shared a specific level of cache. The tool, however, could not be built within the stricture of the POSIX API.

4.4 The PACE Machine Learning Effort

The PACE Machine Learning effort produced models that demonstrate a high-degree of accuracy in predicting execution times, under limited circumstances. The accuracy of the cross-machine predictions (same application, different memory system configuration) is quite good—approximately 95% of runs were predicted within 5% accuracy. We are unaware of any other studies that produced similar accuracy. Unfortunately, most of the literature on predicting tile sizes and predicting execution times as a technique to guide compiler-based optimization has focused on measuring changes in optimization effectiveness, rather than reporting on the effectiveness of the learning regime.

The next step in that research, predicting execution times for an application from the performance of another application, should be pursued. Successful models for that problem could be extremely useful as drivers for optimization decisions: either as predictors of good strategies or as evaluators in a feedback-directed optimization scheme.

4.5 Students Supported by the PACE Effort

The PACE Project provided full or partial support for the following graduate students:

1. Raj Barik (Rice)
2. Thomas Barr (Rice)
3. Brian Bue (Rice)
4. Milind Chabbi (Rice)
5. Varadharajan Chandran (OSU)
6. Sanjay Chatterjee (Rice)
7. Karthiyayini Chinnaswamy (OSU)
8. Jason Eckhardt (Rice)
9. Nanzin Fauzia (OSU)
10. Ashwin Gururaghavendran (OSU)
11. Thomas Henretty (OSU)
12. Justin Holewinski (OSU)
13. Kishor Kommmanaboina (OSU)
14. Kirthi Muntimadugu (OSU)
15. Mohanish Narayan (OSU)
16. Scott Novich (Rice)
17. Patrick O'Driscoll (Rice)
18. Jeffrey Sandoval (Rice)
19. Kamal Sharma (Rice)
20. Sanket Tavarageri (OSU)
21. Anna Youssefi (Rice)
22. Lily Zhang (Rice)
23. Ryan Zhang (Rice)

5. CONCLUSIONS

The PACE Project designed a sophisticated compilation environment that would rely on characterization-driven optimization, feedback-directed optimization, and automatic selection of transformations to provide high-quality code optimization in a tool that was easy to retarget to new computer systems [1]. The PACE Project produced a significant body of software, described in Section 3.5. Many of these components can be used independent of the PACE System.

- Both PolyOpt and the PAO→TAO IR Translator (also known as the ROSE-to-LLVM IR Translator) are available as part of the ROSE compiler infrastructure.
- Some of the components of the PACE Runtime System are included in the ROSE infrastructure and the HPCToolkit system.
- The operator strength reduction and register allocation components developed for LLVM plug directly into in a standard LLVM installation

All of these components are available under "Software" at http://pace.rice.edu.

As we discussed in Section 2.3, the PACE Project was an attempt to address a number of questions about the design of compilation environments and tools.

1. *Can we simplify the task of building and deploying high-quality optimizing compilers for new computer systems?*

   The PACE System showed that we can design sophisticated environments that employ strategies which should automate large parts of retargeting and retuning an optimizing compiler. The PACE implementation effort did not reach the stage of proving the efficacy of that design. We believe, however, that the prospects for characterization-driven optimization, feedback-directed optimization, and automatic selection of transformations are strong.

2. *Can we reduce the effort required to achieve good application performance on those systems?*

   If we take current practice in high-performance computing as a baseline, then it is clear that we can simplify performance tuning and improve compiler effectiveness, both of which should decrease the effort required to achieve good application performance.

   PACE, however, did not reach the stage of providing experimental results that can quantify the improvement in performance attainable with the kinds of techniques it embodied; equally important, it did not conduct human studies to quantify changes in productivity.

3. *Can a characterization-driven approach to compiler-based code optimization yield both portability and performance?*

   At this point, we know of a number of significant optimizations—that is, optimizations that play a key role in determining performance on modern computer systems—that can be formulated in a parameterized, characterization-driven way. The PACE work on parametric loop tiling is a good example; it combines parametric code generation with good static bounds (derived from both system characterization and performance metrics from prior runs) and runtime techniques to adjust for dynamic conditions. The PACE

work on feedback-directed unroll-and-jam relies on measurements of the effective number of live ranges that the native compiler could support. These techniques use characterization results as fundamental inputs to their decision-making processes.

Another large opportunity for characterization-driven optimization lies in the compiler's back end. Much work has been done on retargetable instruction selection—porting the compiler to a new instruction set architecture. Less algorithmic work and less practical implementation work has focused on retargeting the other hard problems in code generation: register allocation and instruction scheduling. Both of these problems appear amenable to the characterization-driven approach.

4. *Can we measure performance-critical system characteristics in a portable way, with enough accuracy to use in a characterization-driven optimization system?*

   The PACE Characterization Tools demonstrate that we can measure many system characteristics in a portable and reasonably precise way. To be sure, the AACE restriction to POSIX APIs for portability limited the scope of what PACE could measure. Without superpages, we cannot measure the associativity of physically mapped caches. Without support for thread affinity, we can accurately measure the behavior of some shared resources. However, interfaces for these kinds of facilities are still evolving. We expect that standards will emerge to make these measurements possible.

   The lesson of the PACE Characterization effort is that, with appropriate care and creativity, we can accurately measure many performance-critical parameters of complex systems—particularly, parameters that matter to an optimizer.

5. *To what extent can we use feedback and adaptation — either runtime performance measurement or introspective insights from compiler components — to improve the results of characterization-driven optimization?*

   The PACE implementation effort did not reach the stage where we could measure improvements from runtime performance feedback. PACE did demonstrate, for what we believe is the first time, that introspective feedback from one compiler component to another could be used to drive optimization decisions in another, and that this process produced better performance than a static decision procedure.

   The PAO→TAO Query mechanism was used to guide selection of unroll factors; the feedback provided by the TAO allowed the PAO to avoid generating code that caused excessive register spilling or increased operation counts. A publication describing this experiment is in preparation. The publication will be posted on the PACE Web site (http://pace.rice.edu) under "Publications".

6. *What kinds of runtime support are necessary to achieve reasonable application performance on modern computing systems?*

   The PACE Project provided partial answers to this question. Clearly, before we can achieve reasonable application performance, we must understand application performance. Much of the PACE Runtime System effort was directed at improving our ability to understand runtime performance and to provide that information to the programmer or to the compiler. The PACE System did not reach the maturity to provide experimental measurements of how much benefit accrued from using that knowledge in optimization.

The PACE experience also suggests that optimization for modern computing environments must have a component that can adapt application behavior to runtime conditions. The PACE design included an API environments adaptation [1]; the PACE Compiler developed parametric loop tiling to target that API. This technique is a promising area for future work.

7. *Can machine-learning techniques predict optimization parameters as well as, or better than, measured performance characteristics?*

   The PACE Machine Learning effort demonstrated the ability to generate models capable of high-fidelity predictions of program performance. Those predictions can, in turn, be used to discover good optimization parameters, such as tile sizes for parametric loop tiling. More work remains to show that these ideas generalize to other problems.

Summary: The PACE Project suggests that the characterization-driven optimization can produce an optimizing compiler that delivers high-quality code optimization and automates some of the effort of retargeting and retuning the compiler. Unfortunately, the PACE Project ended before the tools reached the level of maturity where we could quantify those results.

6. REFERENCES

[1] Keith D. Cooper et al., "The Platform-Aware Compilation Environment: Preliminary Design Document," Computer Science Department, Rice University, Houston, TX, USA, Technical Report TR10-13, September 15, 2010. [Online at http://pace.rice.edu].

[2] Keith D. Cooper et al., "The Platform-Aware Compilation Environment: Status and Future Directions," Computer Science Department, Rice University, Houston, Technical Report TR12-04, June 13, 2012. [Online at http://pace.rice.edu].

[3] Ohio State University. (2011) PolyOpt/C: A Polyhedral Optimizer for the ROSE Compiler. [Online at http://hpcrl.cse.ohio-state.edu/wiki/index.php/PolyOpt/C].

[4] Jun Shirako et al., "Analytical Bounds for Optimal Tile Size Selection," in *21st International Conference on Compiler Construction (CC 2012)*, Tallinn, Estonia, March 24 - April 1, 2012.

[5] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick, "Operator Strength Reduction," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 23, no. 5, pp. 603 - 625, September 2011.

[6] Brian N. West, "Adding Operator Strength Reduction to LLVM," Computer Science Department, Rice University, Houston, TX, Technical Report CS TR11-03, October 20, 2011.

[7] Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar, "Efficient Selection of Vector Instructions Using Dynamic Programming," in *MICRO-43: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Atlanta, GA, December 4–8, 2010, pp. 201-212.

[8] Preston Briggs, Keith D. Cooper, and Linda Torczon, "Improvements to Graph Coloring Register Allocation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 428-455, May 1994.

[9] Preston Briggs, Keith D. Cooper, and Linda Torczon, "Rematerialization," in *PLDI '92: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 15-19, 1992, pp. 311-321.

[10] Ron Cytron, Andy Lowry, and Frank K. Zadeck, "Code Motion of Control Structures in High-Level Languages," in *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Prorgamming Languages (POPL '86)*, New York, pp. 70--85.

[11] Sanket Tavarageri et al., "Parametric Tiling of Affine Loop Nests," in *15th Workshop on Compilers for Parallel Computing (CPC '10)*, Vienna, Austria, July 2010.

[12] Mohammed Rahman, Louis-Noël Pouchet, and P. Sadayapan, "Neural Network Assisted Tile Size Selection," in *5th International Workshop on Automatic Performance Tuning (iWAPT '10)*, Berkeley, CA, June 2010.

[13] Sanket Tavarageri, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayapan, "Dynamic Selection of Tile Sizes," in *18th IEEE International Conference on High Performance Computing (HiPC '11)*, Bangalore, India, December 2011.

[14] Louis-Noël Pouchet et al., "Combined Iterative and Model-driven Optimization in an Automatic Parallelization Framework," in *Supercomputing 2010 (SC10)*, New Orleans, LA, November 13–19, 2010.

[15] Louis-Noël Pouchet et al., "Loop Transformations: Convexity, Pruning and Optimization," in *38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '11)*, Austin, TX, January 2011, pp. 549-562.

[16] Eunjung Park, Louis-Noël Pouchet, John Cavazos, Albert Cohen, and P. Sadayappan, "Predictive Modeling in a Polyhedral Optimization Space," in *9th IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*, Chamonix, France, April 2011, pp. 119-129.

[17] Kamen Yotov, Keshav Pingali, and Paul Stodghill, "X-Ray: A Tool for Automatic Measurement of Hardware Parameters," in *QEST '05: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, Washington, DC, September 19-22, 2005, pp. 168-177.

[18] Keith Cooper and Jeffrey Sandoval, "Portable Techniques to Find Effective Memory Hierarchy Parameters," Computer Science Department, Rice University, Technical Report CS TR11-06, 2011.

[19] Jeffrey A Sandoval, "Foundations for Automatic, Adaptable Computation," Department of Computer Science, Rice University, Houston, TX, Ph.D. Thesis 2011.

[20] Keith D. Cooper et al., "ACME: Adaptive Compilation Made Efficient," in *Proceedings of the 2005 ACM SIGPLAN Conference on Languages Compilers and Tools for Embedded Systems (LCTES 05)*, New York, 2005, pp. 69-77.

[21] Louis-Noël Pouchet, John Cavazos, Albert Cohen, and P. Sadayappan, "Predictive Modeling in a Polyhedral Optimization Space ," in *International Symposium on Code Generation and Optimization (CGO '11)* , Chamonix, France, April 2011.

[22] Milind Chabbi and John Mellor-Crummey, "DeadSpy: A Tool to Pinpoint Program Inefficiencies ," in *International Symposium on Code Generation and Optimization (CGO '12)*, San Jose, California, April 2012.

[23] Karthiyayini Chinnaswamy, "Compile Time Extraction And Instrumentation of Affine Program Kernels," Computer Science and Engineering Department, Ohio State University, Master's Thesis, 2010.

[24] Ashwin Gururaghavendran, "Applying Polyhedral Transformation to Fortran Programs," Computer Science and Engineering Department, Ohio State University, Master's Thesis, 2011.

[25] Mohanish Narayan, "PolyOpt/Fortran: A Polyhedral Optimizer for Fortran Programs," Computer Science and Engineering Department, Ohio State University, Master's Thesis, 2012.

[26] Ohio State University. (2011) PolyBench/C: A Benchmarking Suite for Polyhedral Compilers. [Online at http://polybench.sourceforge.net].

GLOSSARY OF ACRONYMS

Acronym	Definition
AACE	Architecture-Aware Compiler Environment Program, a DARPA/AFRL sponsored research program that include the PACE Project
AAP	Application-Aware Partitioner Air
AFRL	Force Research Laboratory
API	Application programming interface
AST	Abstract Syntax Tree
DARPA	Defense Advanced Projects Research Agency
ISA	Instruction-set Architecture
ML	Machine Learning
PACE	Platform-Aware Compilation Environment
PAO	Platform-Aware Optimizer
PolyOpt	Polyhedral Optimizer
POSIX	Portable Operating System Interface, specified by the ISO/IEC 9945 standard.
RTS	Runtime System
Sage	The internal representation inside the ROSE compiler infrastructure
SCoP	Static Control Part
SSA	Static Single Assignment form
TAO	Target-Aware Optimizer

APPENDIX A: MODIFIED STATEMENT OF WORK FOR THE PACE PROJECT

Platform Aware Compilation Environment
Proposed Settlement Work
— March 23, 2011—

Overview

This document outlines, at a high level, the tasks on which we propose spending the remaining dollars allocated for the PACE Project. The various tasks that we propose comprise a subset of the work envisioned under the original statement of work. Most are explicitly discussed in the PACE Phase 1 design document.

At a high level, our goal is to complete those aspects of the PACE Project that make sense within the reduced funding. In particular, we want to bring the various infrastructure projects begun under PACE to the point where the code can be distributed, and to produce several demonstrations of key pieces of PACE technology. In particular, we want to demonstrate (1) the efficacy of the characterization-driven approach to optimization and (2) the effectiveness of machine learning applied to one or more of the PACE machine-learning problems.

All of the activities that we plan are consistent with our plans for the PACE System, as expressed in the original proposal, in the statement of work, and in the Preliminary Design Document developed for the Phase 1 Preliminary Design Review. For each major piece of the PACE System, we have identified a set of specific tasks that we believe to be achievable.

In principle, we are deleting work that was included to satisfy the needs of the AACE Task 2 teams or work that would exceed our available funding. Within the funding envelope, we have identified individual tasks that we believe should have high priority, either because they demonstrate AACE program objectives or they advance the state of our knowledge.

Relationship to the Rough Order of Magnitude Estimate

The tasks described in this document relate directly to the "Tasks" shown in the Rough Order of Magnitude estimate of our efforts. We have not allocated funds to some of the subprojects that are, essentially, finished (e.g., the AAP and the IR Translator, both part of the PACE COMPILER).

TASK 1: PACE Resource Characterization

Much of the focus of the Phase 1 activity in PACE was on resource characterization (RC). With the remaining funds, we will:

1. Convert the Phase 1 software release into a standalone tool and make it available under Rice University's standard open source license on the PACE web site. We will produce a standalone tool that builds both the RC file needed by the PACE Compiler and a human-readable report.
2. Modify the compiler's interface to RC information to work with the results produced by the standalone tool from item 1.

3. Develop a microbenchmark to measure the effective capacity of a shared multicore cache under uniform load from all its cores. The resource characterization tools will be used in the compiler group's demonstration of characterization-driven optimization.

TASK 2: PACE Compiler

The PACE Compiler has three key components: the application-aware partitioner (AAP), the platform-aware optimizer (PAO), and the target-aware optimizer (TAO), along with a tool that translates from the PAO's intermediate representation to that of the TAO (hereafter called the Rose->LLVM IR Translator).

AAP Subproject

We consider the AAP a completed subproject. The Phase 1 prototype has enough functionality to allow us to continue work on the PAO and TAO. We will cease development work on the AAP. We will continue to use the prototype AAP in our work with the PAO and the TAO. We will distribute the source code for the existing AAP through the PACE web site.

PAO Subproject

The PAO is based on the Rose system, an open-source infrastructure for analyzing and transforming programs in a near-source level intermediate representation. With the remaining funding, we will:

1. Continue to work on the PAO optimizations that we have in progress, specifically polyhedral analysis and transformations, non-polyhedral loop transformations, and supporting analyses and transformations.
2. Explore incremental re-computation of program analysis results to facilitate exploration of different transformation sequences.
3. Release the code developed for the PAO under the open-source license inherited from Rose. Our preference will be to move this code into the Rose distribution. If that proves difficult, we will distribute it through the PACE web site.

TAO Subproject

The TAO is based on the LLVM system, an open-source compiler infrastructure. The TAO group has been working to improve optimization in LLVM, particularly optimization for codes similar to the ones produced by the polyhedral analysis and transformation tools in the PAO. With the remaining funds for this subproject, we will:

1. Complete the implementations of LLVM transformations that are currently underway, and undertake implementation of transformations for LLVM that are already in design.
2. Continue to modify existing LLVM passes in support of adaptation and the machine learning experiments.
3. Suspend all work aimed at using LLVM's facility for generating C code. The LLVM community has deprecated the current LLVM-to-C backend. LLVM implementers elsewhere are building a new C backend. For the remainder of the PACE Project, we will conserve resources and focus on using LLVM's native backends.

We will release the code developed in the TAO project in open-source form under the license inherited from the LLVM project. We will pursue moving our passes into the LLVM

standard distribution. PACE-developed code for LLVM will also be available on the PACE web site.

Rose->LLVM IR Translator

We will continue to perform maintenance work on the existing translator, in support of the work in the PAO and the TAO. The Rose group has expressed interest in distributing this code; it will also be available from the PACE web site.

TASK 3: PACE Runtime System

The PACE Runtime System is designed to provide runtime support for compiler-generated mappings, runtime adaptation of compiler-generated code, and binary analysis of programs, along with software that measures runtime performance, attributes runtime costs to source code lines, and pinpoints performance and scalability bottlenecks.

With the funds remaining in this portion of the project, we will

1. Continue to work on performance measurement and attribution to provide higher-level metrics and information.
2. Extend the current support to the ARM processor.
3. Provide runtime support to help the compiler pinpoint performance and scalability bottlenecks.

We will continue improvements as long as funds permit.

Code developed for the runtime system will be distributed under the same open source license as the underlying HPCToolkit system. It will be available from the HPCToolkit web site, hpctoolkit.org, with a link from the PACE web site.

TASK 4: PACE Machine Learning

The PACE Machine Learning group will continue work on development, training and testing of offline learning engines for the problems identified as ML1, ML2, and ML3 in the PACE Preliminary Design Document, funds permitting.

With the funds remaining in this portion of the project, we will:

1. Produce a trained engine for several variants of the ML1 problem, as well as engines for ML2 and/or ML3, as time and funding permit;
2. Produce software that allows more general use of the ML algorithms developed under PACE.

We will distribute software developed for these demonstrations from the PACE web site, under Rice's standard open-source license.