The Platform-Aware Compilation Environment [1]

Status and Future Directions

June 13, 2012

[1] The Platform-Aware Compilation Environment project (PACE) is funded by the Defense Advanced Projects Research Agency (DARPA) through Air Force Research Laboratory (AFRL) Contract FA8650-09-C-7915 with Rice University. PACE is part of the Architecture-Aware Compilation Environment program (AACE).

Credits

The Platform-Aware Compiler Environment (PACE) project is an inter-institutional collaboration.

Organization	Location	Principal Contacts
Rice University (lead)	Houston, TX, USA	Keith D. Cooper, PI John Mellor-Crummey Erzsébet Merényi Krishna Palem Vivek Sarkar Linda Torczon
ET International	Newark, DE, USA	Rishi Khan
Ohio State University	Columbus, OH, USA	P. Sadayappan
Stanford University	Palo Alto, CA, USA	Sanjiva Lele
Texas Instruments, Inc.	Dallas, TX, USA	Reid Tatge

The PACE team includes a large number of additional colleagues and collaborators:

Laksono Adhianto,[1] Rajkishore Barik,[1] Heba Bevan,[1] Milind Chabbi, [1] Jean-Christophe Beyler,[2] Zoran Budimlić,[1] Michael Burke,[1] Vincent Cavé,[1] Lakshmi Chakrapani,[1] Phillipe Charles,[1] Jack Dennis,[2] Sebastien Donadio,[2] Mike Fagan,[1] Guohua Jin,[1] Paul Hahn,[1] Timothy Harvey,[1] Thomas Henretty,[3] Justin Hoelwinski,[3], Zhao Jishen,[1] Sam Kaplan,[2] Kirk Kelsey,[2] Mark Krentel,[1] Abid Malik,[1] Dung "Zung" Nguyen,[1] Rene Pečnik,[4] Louis-Noël Pouchet,[3] Atanas Rountev,[3] Jeffrey Sandoval,[1] Arnold Schwaighofer,[1] Jun Shirako,[1] Ray Simar,[1] Brian West,[1] Yonghong Yan,[1] Anna Youseffi,[1] Jisheng Zhao[1]

[1] Rice University
[2] ET International
[3] Ohio State University
[4] Stanford University
[5] Texas Instruments, Incorporated

Technical Contacts:	Keith D. Cooper	713-348-6013	keith@rice.edu
	Linda Torczon	713-348-5177	linda@rice.edu
	Vivek Sarkar	713-348-5304	vsarkar@rice.edu
Design Document Master:	Michael Burke	713-348-4476	mgb2@rice.edu
Administrative Contacts:	Penny Anderson	713-348-5186	anderson@rice.edu
	Lena Sifuentes	713-348-6325	lenas@rice.edu
Web Site:	http://pace.rice.edu		

Contents

Acronyms Used in This Document	
AACE	The DARPA Architecture-Aware Compilation Environment Program, which funds the PACE Project
PACE	The Platform-Aware Compilation Environment Project, one of four efforts that form AACE; this document describes the design of the PACE environment.
API	Application Programming Interface
AST	Abstract Syntax Tree
CFG	Control-Flow Graph
DARPA	Defense Advanced Research Projects Agency
`gcc`	Gnu Compiler Collection, a widely-used open-source compiler infrastructure
HIR	High-Level Intermediate Representation
ILP	Instruction-Level Parallelism
IR	Intermediate Representation
ISA	Instruction-Set Architecture
LLVM	An open-source compiler that is used as the code base for the PACE TAO
LLVM IR	The low-level, SSA-based IR used in LLVM and the TAO
ML	The PACE Machine Learning subproject and tools
OPENMP	A standard API for programming shared-memory parallel computers
PAO	The Platform-Aware Optimizer, a component of the PACE compiler
PAO→TAO	The translator from the SAGE III IR to the LLVM IR, a component of the PACE compiler; also called the Rose-to-LLVM translator
POSIX	An international standard API for operating system functionality
RC	The PACE Resource Characterization subproject and tools
RISC	Reduced Instruction-Set Computer
RTS	The PACE Runtime System subproject and tools
RPU	Refactored Program Unit
SAGE III IR	The IR used in Rose, an open source compiler that is the code base for the PACE PAO
SCoP	Static Control Part, a sequence of loop nests that is amenable to polyhedral transformations
SSA	Static Single-Assignment form
TAO	The PACE Target-Aware Optimizer, a component of the PACE compiler
TLB	Translation Lookaside Buffer, a structure in the memory hierarchy that caches information on virtual to physical page mapping

Chapter 1
Overview of the PACE System

The Platform-Aware Compilation Environment (PACE) is an ambitious attempt to construct a portable compiler that produces code capable of achieving high levels of performance on new architectures. The key strategies in PACE are the design and development of an optimizer and runtime system that are parameterized by system characteristics, the automatic measurement of those characteristics, the extensive use of measured performance data to help drive optimization, and the use of machine learning to improve the long-term effectiveness of the compiler and runtime system.

1.1 Introduction

The Platform-Aware Compilation Environment (PACE) project is developing tools and techniques to automate the process of retargeting an optimizing compiler to a new system. The basic approach is to recast code optimization so that both the individual optimizations and the overall optimization strategy are parameterized by target system characteristics, to automate the measurement of those characteristics, and to provide both immediate runtime support and longer term intelligent support (through machine learning) for the parameter-driven optimization.

The PACE project was part of a larger effort, the DARPA-sponsored Architecture-Aware Compiler Environment (AACE) program.[1] Because the DARPA-sponsored AACE program was cancelled, the PACE system was not completed under DARPA AACE funding as originally envisioned. Research on aspects of the PACE system continues under funding from a variety of other sources. The implementation status of the design described in this document, as of November 2011, is outlined in § 1.4.

1.1.1 Motivation

Over the last twenty years, the average time to develop a high-quality compiler for a new system has ranged between three and five years. Given the rapid evolution of modern computer systems, and the correspondingly short lifetimes of those systems, the result is that quality compilers appear for a new system only at the end of its useful lifetime, or later.

Several factors contribute to the lag time between appearance of a new computer system and the availability of high-quality compilation support for it. The compiler may need to deal with new features in the target system's instruction set architecture (ISA). Existing optimizations must be retargeted to the new system;[2] those optimizations may not expose the right set of parameters

Principal Contacts For This Chapter: Keith Cooper, keith@rice.edu

[1] The PACE project is funded by the Defense Advanced Projects Research Agency (DARPA) through Air Force Research Laboratory (AFRL) Contract FA8650-09-C-7915 with Rice University. The opinions and findings in this document do not necessarily reflect the views of either the United States Government or Rice University.

[2] Datta et al. showed that variations in target machine architecture necessitate different optimization strategies for stencil

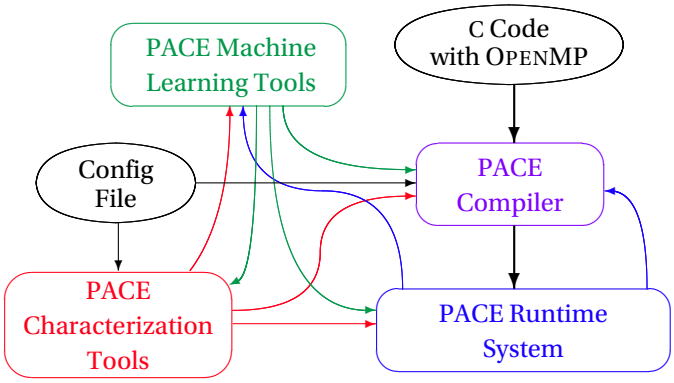


Figure 1.1: Major Components of the PACE *Systems*

to simplify retargeting. Finally, the new system may present system-level features that are not well addressed by existing optimizations, such as the DMA interfaces on the IBM CELL processor. In such cases, the retargeting effort may require invention and implementation of new transformations to address system-specific innovations.

The PACE system attacks the first two problems.

- The PACE compiler is built on Rose and LLVM. For code generation, the PACE compiler relies on either LLVM native backends or native C compilers for code generation—that is, to emit the appropriate assembly language code. Since both Rose and LLVM are capable of producing C from their intermediate representations, the PACE compiler can generate C code for the native C compiler when an LLVM native backend is not available for a particular architecture.

- PACE will include a suite of transformations that are parameterized by target-system characteristics, both hardware and software. These transformations will use specific, measured characteristics to model the target system and will reshape the code accordingly. These transformations will be retargeted by changing the values of the system characteristics that they use as parameters. The behavior of the compiler will change with the values of the system characteristics.

PACE does not address the final problem, inventing new optimizations for radical new features. It will, however, free the compiler writer to focus on new transformations that address new architectural features.

Thus, PACE transforms the problem of tuning the optimizer for a new system into the problem of deriving values for key system characteristics. PACE includes a set of portable tools that measure those characteristics. Thus to retarget the optimizer, an installer runs the characterization tools and installs the compiler.

Finally, because the values of some important characteristics cannot be determined accurately until runtime, PACE includes a runtime system that can adjust optimization parameters in compiler-generated code. The runtime system makes specific and precise measurements of runtime performance. It is capable of identifying rate-limiting resources by code region. It can report the results of these analyses to either the end user or to the other components in the PACE system.

computations [34]. Equally important, follow-on analysis showed that code tailored for any machine in their study performed poorly on any other machine [59].

Component	Chapter
PACE Resource Characterization Tools	
Microbenchmarks	2
Interface to other tools	2
PACE Compiler	
Compiler Overview	3
Platform-Aware Optimizer (PAO)	4
Polyhedral Framework	5
AST-based Transformations in the PAO	6
Rose-to-LLVM Translator	7
Target-Aware Optimizer (TAO)	8
PACE Runtime System	9
PACE Machine Learning Tools	10

Table 1.1: Document Organization

1.1.2 Document Roadmap

This chapter provides a survey of the structure and functionality of the PACE system, along with discussion of system-wide design decisions. Section 1.2 provides a description of the major software components of the PACE system, shown in Figure 1.1. The later chapters of this document describe those components in more detail. Table 1.1 shows how the remaining chapters of this document map into the software components of the PACE system.

1.2 Structure of the PACE System

The PACE system has three major components: the PACE Compiler, the PACE Runtime System, and the PACE Resource Characterization tools. The PACE system is designed to support a fourth component: a PACE Machine Learning tool. Figure 1.1 shows the major components of the PACE system.

- **The PACE Compiler** is an optimizing compiler that tailors application code for efficient execution on the target system. It accepts as input parallel programs written in C with OPENMP calls. It produces, as output, either a C program or native code for the target machine. In either case, the resulting program has been tailored to the system's measured characteristics.
- **The PACE Runtime System** provides support for program execution. It measures application performance and can report those results to both the user and other PACE tools. It can work in concert with the PACE Compiler to provide runtime tuning of specific optimization parameters, such as tile sizes for blocking.
- **The PACE Resource Characterization Tools** measure the performance-sensitive characteristics of the target system that are of interest to the PACE Compiler and the PACE Runtime System. The tools measure the resources available to a C program, which may differ from the documented limits of the underlying hardware.
- **A PACE Machine Learning Tool** could be included to perform offline analysis of application performance, using data from the runtime system, and of compiler behavior. The tool could develop recommendations for specific components of the compiler and the runtime system. The tool could also play a role in analyzing the impact of sharing on available resources.

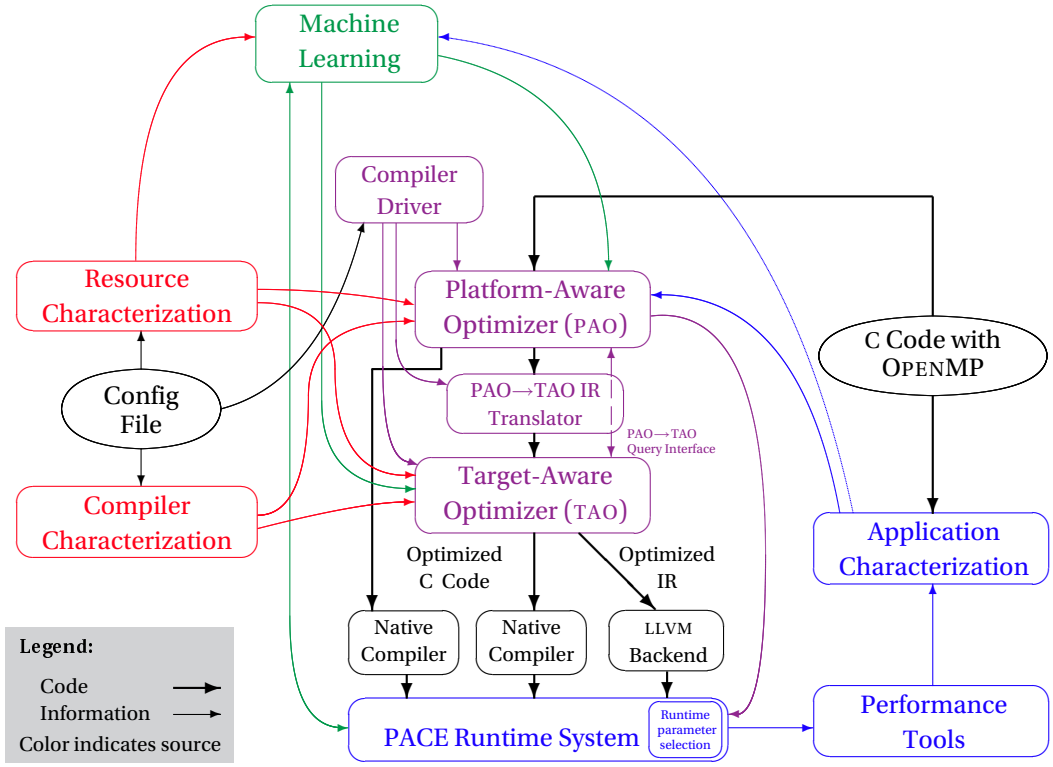


Figure 1.2: The PACE *System*

To configure an installation of the PACE system on a new computer, the system manager installs the software, produces a *configuration file*, and invokes the characterization tools. The characterization tools produce the data used by the other components in PACE to tailor the system and its behavior to the target system.

The configuration file contains base information about the target system and its software.

1.2.1 Information Flow in the PACE System

Figure 1.2 expands on Figure 1.1 to show the tools that make up the major PACE components and to show the interfaces between the various tools. Thick black lines represent the flow of code. Thin lines represent the flow of information other than code; they are colored to match the tool that generates the information. (Double-ended lines are colored to match one of the two tools they join.) The chapters that describe the individual components (see Table 1.1) provide detail on how each of these interfaces work.

1.2.1.1 The Compiler

To compile an application using the PACE tools, the programmer creates a directory that contains the source code for the application and any libraries that are to be optimized with it. If future versions of PACE were to include a machine learning tool, PACE could create, within the application directory, a directory to hold its work products that support machine learning and optimization (e.g., annotations, performance results, and records of prior compilations). This directory could become part of the PACE system's distributed repository.

To compile code, the programmer invokes the compiler driver in the application directory. The

compiler driver then sequences the individual components of the PACE Compiler to optimize the application and to produce executable code for it (see § 3.3). Depending on the application and its optimization plan (see § 1.2.2), the compiler driver may use the Platform-Aware Optimizer (PAO), the PAO→TAO IR Translator, the Target-Aware Optimizer (TAO), and the native compiler to create one of three distinct compilation paths.

- The compiler driver may follow the full compilation path, using all of the PACE tools to optimize the application and generate transformed C source code, which it then compiles with the native C compiler.
- If the native compiler has strong optimization capabilities, the compiler driver may follow a short compilation path, in which it relies on the native compiler to perform some of the optimization. This path uses a subset of the PACE Compiler components.
- If the target system is one for which the PACE Compiler provides backend support,[3] the compiler driver may use PACE Compiler components to optimize the code and to generate native code for the application.

In each of these scenarios, the compiler driver also invokes the linker to create the actual executable code. During compilation, the PAO may invoke the TAO to obtain low-level, detailed information about the expected performance of alternate code sequences (see § 8.3.5).

1.2.1.2 The Runtime System

The 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.[4]

When invoked, the RTS would interpose itself between the application and the operating system to intercept events such as program launch and termination, thread creation and destruction, signal handler setup, signal delivery, and loading and unloading of dynamic libraries. It would monitor the application's behavior using a variety of mechanisms (see § 9.3.1), and record the results.

The runtime system 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 that could provide the RTS harness for online feedback-directed optimization (see § 9.3.4).

1.2.1.3 The Characterization Tools

The PACE resource characterization (RC) tools are a stand-alone 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 RC 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 included in the PACE resource characterization software release are described in § 2.2.3 and Appendix A.

---

[3] Since the PACE Target-Aware Optimizer is built on top of the open-source LLVM system, this option exists on systems that have a native LLVM backend. LLVM already supports several backends.

[4] 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.

Measured versus Absolute Numbers In many cases, the RC 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, an application may not be able to rely on using the full capacity of the processor's level-two cache memory; it may, instead see a smaller available capacity. Many factors can cause the applications to see a smaller *effective* capacity. The L2 cache may be shared among multiple cores on the same processor; in that case, competition from activity on other cores may reduce the capacity available to the application. Some factors apply even in a unicore L2 cache. The L2 cache may contain the images of both the L1 data cache and the L1 code cache, reducing its effective size for data. The L2 cache may be mapped using physical addresses, which introduces a degree of randomness into the mapping and complicates any attempt to use the full cache capacity. The L2 cache may contain portions of the operating system's page table, locked into place for use by hardware lookup methods; again, this reduces the capacity available for application data. In the best of scenarios, the application probably sees smaller effective cache sizes than the hardware documents would suggest, at least at cache level two and higher.[5]

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 cache associativity with any certainty.

Methodology In general, the PACE RC tools provide one or more microbenchmarks to measure a given characteristic. A microbenchmark is a small code kernel designed to provoke a specific response, coupled with code to analyze the kernel's behavior. Typically, that response is a change in the time that the kernel requires to perform a fixed number of operations. Automatic analysis of the kernel's behavior can be complex; effects that a human can read easily from a graph can be difficult to isolate numerically.

The RC tools produce information that can be accessed through two distinct interfaces: one designed for the grading tools built by the AACE Task 2 teams and the other designed for internal use in the PACE system. The grading interface is a flat ASCII file in an XML schema designed by the Task 2 teams. The internal interface is a procedural interface that PACE tools can call to obtain individual values. The PACE RC tools also produce a human-readable report based on the XML schema.

1.2.1.4 Machine Learning Tool

The PACEdesign included a PACE Machine Learning (ML) tool that could augment specific decision making processes within the PACE system, through analysis of past experience and behavior. The goal would be to improve the behavior of the other PACE tools over time. 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.

In the PACE design, the ML tool would be tied closely to specific components in the PACE system,

[5] An interesting example occurs on the IBM Power 7 architecture. It supports a shared 32 MB level three cache. However, the cache is organized so that each of the eight cores has faster access to a 4 MB portion of the L3 cache and slower access to the remaining 28 MB. Our RC tools analyze this cache structure and classify it as a small (3 to 4 MB) L3 cache and a larger (20 to 25 MB) L4 cache. They detect the NUMA nature of the cache and describe it as two caches. From the code optimization perspective, the RC tools' description may be more useful than characterizing it as a single larger cache.

where they could provide additional input, in the form of directives, refined input parameters, or changes to optimization plans (see Figure 10.1 on page 86). The ML tool would draw its inputs from the other PACE tools, as shown in Figure 1.2. The ML tool would have its own private repository where it could store context, data, and results.

To facilitate offline learning, the PACE system could invoke the offline portions of the ML tools on a regular basis. Problems that are solved online could invoke the appropriate ML tools directly.

1.2.2 Storing Knowledge in a Distributed Fashion

The PACE system was designed to store its knowledge about an application with the application's source code. This strategy would allow information and annotations generated by the PACE tools to be stored in multiple locations. These locations would form a distributed, rather than a centralized, repository.

Consider, for example, the collection of information that governs how the PACE Compiler optimizes and translates an application. In a traditional compiler, that control information is encoded in a series of command-line flags to the compiler. While such flags are useful, their very form limits their ability to express complex control information. In the PACE system, each application would have an associated *optimization plan* that specifies how the compiler should optimize the code. The optimization plan would be a persistent document that specifies both the compilation path and the optimizations that the compiler should use. It might also include parameters to individual optimizations, suggested application orders for those optimizations, or commands to control the individual components.

Since each of the compiler components would consult the optimization plan, the various components could modify each other's behavior by making changes to the optimization plan. This simple mechanism would facilitate feedback-driven adaptive compilation, by allowing an adaptive controller to explore and evaluate the space of possible optimization strategies over multiple compile-execute cycles. It would also allow one phase of compilation to change the behavior of another. The next section describes the design for adaptation. Section 3.2.4 discusses the role of the optimization plan in more detail.

To ensure that all the PACE tools have easy access to the information that they need, the PACE Compiler could inject critical information into each executable that it produces. For example, it could record both the location of the application directory and its optimization plan in an initialized static data item in each executable. At runtime, the RTS could retrieve that information and record it directly with the performance data, to link the necessary information in a simple and explicit way. This scheme eliminates the need for the executable and the RTS to access a centralized knowledge base;[6] instead, the information that they need is encapsulated in the executable.

1.3 Adaptation in the PACE Compiler

Adaptation is a key strategy embodied in the PACE compiler. 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. Four different mechanisms can be used 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 1.2 and described in the following sections.

In combination, these four mechanisms can provide the compiler with the ability to adapt its behavior to the target system, the application, and the runtime situation. These mechanisms would

[6] A centralized knowledge base can create the situation where the user either cannot run an executable unless it has network access to the knowledge base or the user loses all feedback information from such runs. Neither is a good scenario.

	Characteristic 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	PAO, TAO	PAO, TAO	RTS	PAO, TAO
Persistence	Until next run of RC tools	Short-term	Records results for ML and PAO	Long-term

Table 1.2: Kinds of Adaptation in the PACE *Compiler*

allow the PACE system to be flexible in its pursuit of runtime performance. We anticipate that interactions between these mechanisms would produce complex optimization behavior.

1.3.1 Characteristic Driven Optimization

The concept of characterization-driven optimization forms the core of the PACE system. In the PACE Compiler, for example, the non-polyhedral loop optimizations can use the measured parameters of the memory hierarchy to choose tile sizes, while the tool that regenerates C source code can tailor the number of concurrently live values to the number of such values that the target system's compiler can maintain in registers. [7]

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 that the RC tools should measure. They implement the transformations that use the results from the RC tools. This process automatically adapts the transformation to the target system; it does not take into account any properties of the application or its data set.

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

1.3.2 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 PAO and TAO each consult the application's optimization plan before they transform the code (see § 3.2.4). Changes to the optimization plan cause changes in 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 cur-

[7] The polyhedral optimizations generate code that is parameterized by tile sizes; the mechanism that selects values for those parameters can use the results generated by the RC tools.

rent compilation of an application. In practice, one can explore 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. We anticipate that this mechanism would modify gross properties of optimization, such as the specific transformations applied and their relative order or the compilation path (full, short, or LLVM backend).

- Any phase of the compiler may contain an optimization pass that performs self-adaptation. For example, the non-polyhedral loop optimization in the PAO might consider several transformation strategies; to choose among them, it can generate each alternative version of the loop nest and invoke the PAO–TAO query mechanism to have the TAO estimate some aspects of performance. In a similar way, the TAO might consider multiple strategies for algebraic reassociation and choose between them based on an estimate of execution efficiency from the instruction scheduler.

- One phase of the compiler may change the optimization plan for another phase, based on the code that it generates. We envision this facility as serving 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 occurs within a single compilation.

  Alternatively, one phase might 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 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 are not learning techniques.[8]

1.3.3 Online Feedback-Driven Optimization

A third potential 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 other processors and interference from the runtime behavior of other applications can degrade actual performance.

To cope with such dynamic effects, PACE could include 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 RTS could use to perform the runtime tuning (see § 9.3.4). The RTS could use that information to find, at runtime, settings for the control parameters that produce good performance. The result would be an execution that 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 could recognize that interference from other threads and other applications can impact optimal tile size, and thus it could generate code that read tile dimensions from a designated place

[8] 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 [29].

in memory. In this latter scheme, the runtime system could use performance counter information, such as the L2 cache miss rate, to judge performance and vary the tile size accordingly.

The PACE RTS both defines and implements an API for online, feedback-driven optimization (see § 9.3.4). The API lets the compiler register tunable parameters and suggested initial values, and provides a runtime search routine (an adaptive controller) that the RTS could use to vary those parameters. The RTS could collect the data needed by the runtime search routine and ensure that it is invoked periodically to reconsider the parameter values.

Online feedback-directed optimization could produce 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 RTS could record the final parameter values along with its record of the the application's performance history. Other components in PACE could use these final parameter values as inputs to long-term learning.

1.3.4 Machine Learning

A potential 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 strategy for adaptation. A PACE ML tool could derive models that predict appropriate optimization decisions and parameters. We have identified several specific problems to attack with ML techniques (see § 10.2.2).

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; an ML tool has the opportunity to draw features from any other part of the environment—the RC tools, the compiler tools, and the RTS tools. 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.

The application of machine learning has the potential to create a process that will automatically improve the PACE system's behavior over time. Offline learning tools could 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 could use ML-derived models directly in its decision processes. As ML models mature, the compiler could replace some static decision processes and some short-term adaptive strategies with a simpler implementation that relies on predictions from ML-derived models.

1.4 Status

Components of the following tools are available in source or binary form:

- RC tool: A source release of the PACE RC tool is available on the PACE web site. The release includes code to produce the characteristics described in Table 2.2.

- PAO: A subset of the PAO design described in this document has been implemented thus far. It includes a complete polyhedral loop transformation framework, as well as non-polyhedral AST-based transformations for loop tiling and loop unrolling. The AST-based transformations include the use of cost information from TAO to guide the selection of loop unroll factors, as well as an interface to use array dependence analysis information from the polyhedral framework to aid in legality testing of transformations.

- Rose-to-LLVM translator: The translator has been fully implemented, as described in § 7.
- TAO: A subset of the TAO design described in this document has been implemented. LLVM passes for performing operator strength reduction, linear function test replacement, and register allocation (including rematerialization and biasing) are complete (§ 8.3.2). Source code for these passes is available under the LLVM license on the PACE web site. LLVM passes for generating short SIMD vector code (§ 8.3.3) and computing cost information for the PAO (§ 8.3.5) are complete. All of the completed optimization passes will be available in the final binary release of the of the PACE compiler, built from the source code in the Rice repository.
- RTS: The RTS and the compiler have not yet been integrated. The measurement infrastructure for PACE has been implemented in the context of the HPCToolkit performance tools [66]. HPCToolkit is capable of measuring and attributing costs for dynamic calling contexts, procedures, and loops. To date, we have not computed rate limiting factors for individual program contexts, though the PerfExpert team at the University of Texas at Austin has built such a capability on top of HPCToolkit; they introduce the concept of Local CPI for that purpose [20]. Work on automated runtime selection of parameters has been deferred.

The RC tool, polyhedral analysis in Rose, the Rose-to-LLVM translator, several TAO LLVM passes, and the RTS performance tools are also available in source from either Rose or other channels.

Other tools and underlying structures described in this document are still in the preliminary exploration and design stage, and have not been implemented. In particular, the ML tool and the structure to support adaptation over multiple compile cycles, such as the distributed repository and high-level optimization plan, are not implemented. In that the distributed repository has not been implemented, it has not been necessary to implement the application directory. Currently the compiler driver takes a list of files to compile, as `gcc` would do.

Chapter 2

Resource Characterization in the PACE System

Resource characterization plays a critical role in the PACE project's strategy for building an optimizing compiler that adapts itself and tunes itself to new systems. The PACE compiler and the PACE runtime system need access to measurements of a variety of performance-related characteristics of the target computing system. The goal of the PACE Resource Characterization subproject is to produce those measured values.

2.1 Introduction

The ability to derive system performance characteristics using portable tools lies at the heart of the AACE program's vision and the PACE project's strategy for implementing that vision. The Resource Characterization (RC) subproject of PACE is building tools, written in a portable style in the C language, to measure the specific performance characteristics that are of interest to the PACE compiler (both the PAO and the TAO) and the PACE runtime system (RTS).

2.1.1 Motivation

The PACE compiler and RTS rely on the values of a number of performance-related system parameters, or characteristics, to guide the optimization of an application program. The RC subproject is developing tools that produce those specific values in reliable, portable ways.

The design of the PACE compiler and RTS both limits and focuses the RC subproject. The PACE compiler is designed to be capable of generating native code for a limited set of target processors. It is also designed to be capable of generating a transformed program as a C program, a strategy that ensures portability across a broader set of architectures. This strategy also prevents the PACE compiler from applying some optimizations, such as instruction scheduling, to transformed programs that will be generated as C programs.

The RC subproject is focused on characteristics that the PACE compiler and the other PACE tools can effectively use in this scenario. (Table 2.2 provides a full list of characteristics that were measured in Phase 1 of the AACE program and that were included in the PACE RC software release.) As an example, consider the information needs of the PAO's non-polyhedral loop transformations. The transformations need to know the geometry of the cache hierarchy—that is, for each level of the hierarchy, the size, the associativity, and the granularity (line size or page size) of that level. The RC tools derive those numbers.

Why not obtain the numbers from reading the manufacturer's documentation? The AACE program depends on a strategy of deriving these characteristics rather than supplying them in a con-

Principal Contacts For This Chapter: Keith Cooper, keith@rice.edu

figuration file. This strategy is critical for several reasons.

1. The compiler needs to understand the characteristics as they can be seen from a C source program. For example, the documentation on a multicore processor may list the level two data cache size as 512 kilobytes.[1] The amount of level two cache available to the program, however, will depend on a number of factors, such as the size of the page tables and whether or not they are locked into the level two cache, the number of processors sharing that level two cache, and the sharing relationship between the instruction and data cache hierarchies. In short, the number in the documentation would mislead the compiler into blocking for a larger cache than it can see.

2. The documentation, even from the largest manufacturers, is often incomplete or inaccurate. Documentation on the memory hierarchy focuses on the capacity of the largest level; it rarely describes the delay of a level one cache or TLB miss. Equally problematic, the documents provide inconsistent information; for example, one processor manual studied provides multiple conflicting latencies for the integer divide operation, none of which match the numbers that the carefully constructed PACE microbenchmark measures.

3. The characteristics themselves can be composite effects that result from the interaction of multiple factors. For example, the PAO might want to understand the rough cost of a function call for use in the decision algorithms that guide both inlining and outlining. The cost of a function call depends, however, on specific details of the target system's calling convention, the manner in which the native compiler generates code for the call, the number of parameters and their source-language types, and the presence or absence of optimizations for recursive calls and leaf-procedure calls in the native compiler. The amalgamation of all these factors makes it difficult, if not impossible, to derive reasonably accurate numbers from reading the manufacturer's manuals.

In addition, the PACE system is intended to adapt itself to both current and future architectures. From this perspective, the design of the RC system should minimize its reliance on idiosyncratic knowledge of current systems and current interfaces. The AACE program assumes that future systems will support the POSIX standard interfaces. Thus, the RC tools rely on POSIX for interfaces, such as a runtime clock for timing, and for information about operating system parameters, such as the page size in the virtual memory system.[2] They cannot, however, assume the presence of other runtime interfaces to provide the effective numbers for system characteristics that the PACE compiler and RTS need. Thus, the PACE project derives numbers for most of the characteristics currently used by the PACE compiler and RTS.

2.1.2 Approach

To measure the system characteristics needed by the PACE compiler and RTS, the RC project uses a series of *microbenchmarks*—small programs designed to expose specific characteristics. Each microbenchmark focuses on eliciting a specific characteristic from the system—from the cost of an integer addition through memory hierarchy characteristics. This approach produces a library of microbenchmark codes, along with a harness that installs and runs those codes.

The individual microbenchmarks produced by the PACE project include both a code designed to elicit the effect and a code that analyzes the results and reduces them to one or more characteristic values. Developing the PACE microbenchmarks was challenging. Designing a code to elicit

[1] Many manufacturers provide an interface that exposes model-dependent system parameters, such as the size and structure of the levels in the memory hierarchy. For example, Intel processors support its `cpuinfo` protocol. Unfortunately, such facilities vary widely in their syntax and the set of characteristics that they support. PACE cannot rely on their presence.

[2] Page size and line size are measurements where the effective size and the actual size are, in our experience, identical.

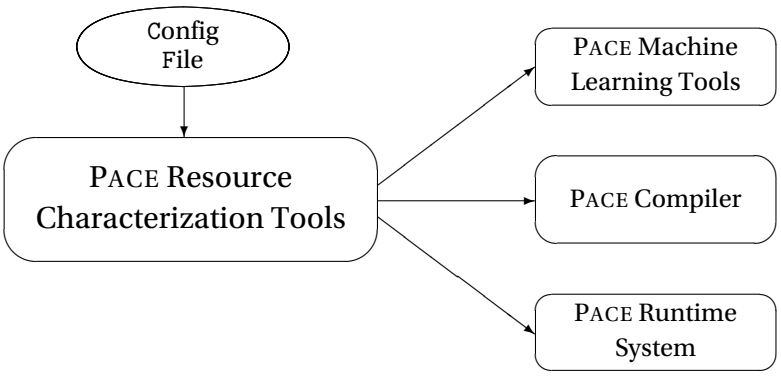


Figure 2.1: Interfaces to the PACE *Resource Characterization Tools*

the desired effect (and only that effect) required, in every case, multiple iterations of the design-implement-test cycle. In many cases, it required the invention of new measurement techniques. The analysis of results can be equally challenging. The experimental results that expose a given effect contain noise. They often expose interference from other effects. The data analysis problems are, in some cases, harder than the problem of exposing the effect.

The result of this effort is a library of microbenchmarks that both elicit system behavior and analyze it. Those codes, written in a portable style of C, rely on the POSIX interface to system resources and on a handful of common POSIX tools, such as the `make` utility. They provide the compiler with a sharp picture of the resources available on a new system.

2.2 Functionality

2.2.1 Interfaces

The primary task of the RC tools is to produce data used by the other major components of the PACE system: the PACE compiler, the PACE RTS, and the PACE Machine Learning tools (ML). As shown in Figure 2.1, the RC tools take as their primary input a system configuration file. The tools use the native C compiler, system calls supported in the POSIX standard, and some additional software tools, as specified in Table 2.1.

Item	Description
C compiler	Native compiler, as specified in the configuration file; must be able to produce an assembly-code listing
OPENMP library	Standard-conforming OPENMP library, with location and linker flags specified in configuration file
Utilities	Standard Linux commands, including autoconf, automake, awk, grep, make, sed, wc, and the bash shell

Table 2.1: Software Requirements for the PACE RC *Tools*

2.2.2 Inputs

The primary input to the RC tools is the configuration file for the target system. This file has a simple format of space-separated name/value pairs: one per line. The pairs are read in and then exported

Category	Name	Units	Page	Notes
Cache	Capacity / Size	Bytes	102	Effective size
	Line Size	Bytes	103	
	Associativity	Integer	104	Only reported for L1 cache Value of zero implies full associativity
	Latency	Cycles	105	Assumes int32 add takes one cycle
TLB	Capacity / Size	Bytes	106	Total footprint of TLB
	Page Size	Bytes		From Posix `sysconf()`
Operations	Ops in Flight	Integer	107	+, -, *, /, for int32, int64, float, double Maximum number of operations in progress by type
	Op Latencies	Cycles	108	+, -, *, /, for int32, int64, float, double Assume int32 add takes one cycle
System	Compute Bound Threads	Integer	109	Test must run standalone
	Memory Bound Threads	Integer	110	Test must run standalone
Compiler	Live Ranges	Integer	111	int32, int64, float, double Number of simultaneous live ranges that native compiler can maintain without spilling

Table 2.2: PACE *Characteristics Included in the* PACE *Resource Characterization Software Release*

as environment variables, so that the individual tools have access to them without the need for each one to know where the file is stored. The RC tools need this information to include at least:

1. The location, name, and invocation sequence for the native compiler and linker. The RC tools need syntax to invoke the native compiler, link against standard libraries, create an executable image, run that image, and connect disk files to input and output streams. (Under POSIX systems with the `bash` shell and the standard C libraries, much of this knowledge is standard across systems.)

2. A specific command set to compile and link for vectorization, if it is supported by the native compiler. This command set must be distinct from the default command set.

3. A set of optimization flags to use with the native compiler during resource characterization. These flags are provided by the system installer in the configuration file. These flags must include the options necessary to produce the appropriate behavior from each microbenchmark. In `gcc` terms, the flag `-O2` appears to be sufficient.

Value	Tool	Use
DCache Capacity DCache Line Size DCache Associativity	PAO	Tiling memory hierarchy [1]
TLB Capacity TLB Page Size	PAO	Tiling memory hierarchy [1]
Operations in Flight	TAO	Compute critical path length for PAO queries
	PAO, TAO	Estimate & adjust ILP
	TAO	Instruction scheduling [2]
Operation Latency	TAO	Algebraic reassociation of expressions
	TAO	Operator strength reduction
	TAO	Compute critical path lengths for PAO queries
	RC	Compute throughput
Compute-bound Threads Memory-bound Threads	PAO	Adjusting parallelism
Live Values	TAO	Answering PAO queries

[1] Both polyhedral transformations (see § 5) and AST-based transformations (see § 6)
[2] We may modify the scheduler in the native TAO backend, to use derived latencies as a way to improve portability. The TAO's query backend (see § 8.3.5) may also perform instruction scheduling to estimate execution costs.

Table 2.3: Optimizations That Can Use Measured Characteristics

4. Basic information on the target system including microprocessors and their components; number of cores; clock rate(s) of the processors; memory architecture on a processor; memory architecture on a node or system; number of chips (memory and processors) per node; interconnection of nodes; and composition of the processing system.

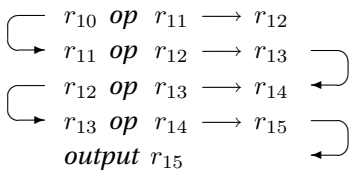
2.2.3 Output

The PACE RC tools produce, as output, a set of measured characteristic values. Those values are available in two forms: a human-readable report of the values measured, and an internal format used by the interface that the RC tools provide for the PACE compiler, RTS, and ML tools. § 2.2.1 provides more detail on these interfaces.

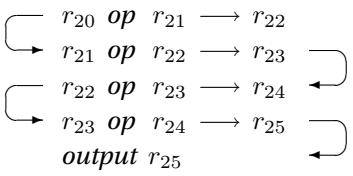
Table 2.2 shows the characteristic values measured by the PACE RC tools included in the PACE RC tools software release[3]. These characteristics range from broad measurements of target system performance, such as the number of compute-bound threads that the processor can sustain, through microarchitectural detail, such as the latency of an integer divide operation. Each characteristic can be used elsewhere in the PACE system. Table 2.3 gives a partial list of those potential uses.

Note that the PACE RC tools do not report units of time. Varying timer precision on different systems and the possibility of varying runtime clock speeds make it difficult for the tools to report time accurately. So, latencies are reported in terms of the ratio between the operation's latency and that of integer addition.

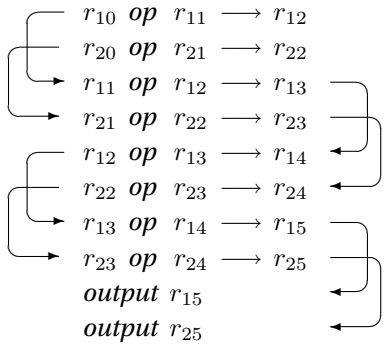
[3]This set of RC tools was also submitted for the end-of-phase trials in Phase 1 of the PACE project.



(a) Stream 1



(b) Stream 2



(c) The Two Streams Interlaced

Figure 2.2: Example of Combining Streams to Expose Parallelism

2.3 Method

Conceptually, each microbenchmark consists of two distinct pieces: a code designed to expose the characteristic and a code that analyzes the resulting data to derive a value for the characteristic. The complexity of these codes varies from characteristic to characteristic, but they largely rely on some variation of a *stream* code construct. A stream is a sequence of code that is designed to run exactly as specified; that is, code that cannot be altered (rearranged, simplified, *etc.*) by an optimizing compiler. Each specific characterization pass will have its own flavor of one or more streams, but the methodology is similar across all PACE applications:

1. Construct a stream and measure its behavior.
2. Modify the stream in a controlled fashion and measure the new stream's behavior.
3. Repeat Step 2 until a significant change in behavior is detected.

Consider two examples that show the use of streams and their variations. The first example is a micobenchmark that measures instruction parallelism, in essence determining the number of functional units of a particular type that the architecture has. This microbenchmark starts with a single stream of instructions as shown in Figure 2.2(a). The arrows in the figure represent data dependences that inhibit reordering by a compiler. The microbenchmark measures the time required to run the single stream and uses this measurement as the base case against which all other tests will be compared. Next, the microbenchmark adds a second, independent stream (Figure 2.2(b)), which is interlaced with the first stream as shown in Figure 2.2(c). This step results in a piece of code in which any pair of successive instructions has no data dependence, which should allow the compiler to schedule the instructions simultaneously if sufficient resources exist and the

compiler is sophisticated enough to allow it. The microbenchmark measures the runtime of the two-stream version and, if it is close to the runtime of the single-stream version, concludes that the architecture can run at least two streams in parallel. The microbenchmark continues adding streams in this manner while the runtime of successive tests is close to the base case measurement, constructing and measuring the runtime of a series of three independent instructions, then four, and so on. When the code produced by the microbenchmark saturates the architecture, larger streams will produce greater runtimes than the base case, and the microbenchmark halts and reports the observed limit.

The second example is a microbenchmark that uses a single stream to determine the number of available registers on an architecture. The goal is to control *register pressure*, the number of values that are active at any point in the program. As shown in Figure 2.3, the base-case stream is the same as shown in Figure 2.2(a). Also notice that, at each point in the program, there are only two values active. At each step, the microbenchmark increases the register pressure by pushing the use of each defined value further down the stream, which means that more values are active at each instruction. While there are sufficient registers to hold all the active values, the code generated by the compiler will not change much from test to test. When there are not enough available registers to hold all of the active values, the compiler will have to insert spill code, which can be detected by timing the execution of each stream. However, there is a simpler solution: While the compiler has sufficient registers to allocate, each compiled stream will have the same number of assembly-code instructions. When the compiler runs out of registers, the length of the code will increase due to the additional number of spill instructions added. Thus, the microbenchmark can measure the length of the assembly code for each stream to determine when the number of available registers on a particular architecture has been exceeded and use this information to determine the number of available registers on that architecture.

These two examples use streams differently, but the key characteristics of the stream approach include:

- The linear sequence of instructions must execute in a controlled fashion.
- As the stream changes from test to test, any significant change in behavior should be attributable to only one cause.
- The stream needs to be long enough to:
  - Give stable measurements. For example, when measuring runtimes, each execution should produce a runtime well above the granularity of the architecture's timer.
  - Cause a significant, obvious, detectable change in the measurements at the point of interest.

Technical details related to individual microbenchmarks listed in Table 2.2 can be found in Appendix A on the page indicated in the fourth column of Table 2.2.

2.3.1 Reporting Characteristic Values

The AACE Program testing teams defined an XML schema for reporting results to their testing programs. The PACE RC tools use the resultant hierarchical naming scheme for the values reported by the microbenchmarks. The microbenchmarks record their values in a flat text file. Values are recorded, one per line, as a pair:

name, *value*

Code	Live Values
$\vdots$	$\vdots$
$add\, r_{101}\, r_{102} => r_{103}$	r_{102}, r_{101}
$add\, r_{102}\, r_{103} => r_{104}$	r_{103}, r_{102}
$add\, r_{103}\, r_{104} => r_{105}$	r_{104}, r_{103}
$add\, r_{104}\, r_{105} => r_{106}$	r_{105}, r_{104}
$\vdots$	$\vdots$

Base-Case Stream with Register Pressure of Two

Code	Live Values
$\vdots$	$\vdots$
$add\, r_{100}\, r_{102} => r_{103}$	$r_{102}, r_{101}, r_{100}$
$add\, r_{101}\, r_{103} => r_{104}$	$r_{103}, r_{102}, r_{101}$
$add\, r_{102}\, r_{104} => r_{105}$	$r_{104}, r_{103}, r_{102}$
$add\, r_{103}\, r_{105} => r_{106}$	$r_{105}, r_{104}, r_{103}$
$\vdots$	$\vdots$

Stream with Register Pressure of Three

Code	Live Values
$\vdots$	$\vdots$
$add\, r_{102-N}\, r_{102} => r_{103}$	$r_{102}, r_{101}, \cdots, r_{102-N}$
$add\, r_{103-N}\, r_{103} => r_{104}$	$r_{103}, r_{102}, \cdots, r_{103-N}$
$add\, r_{104-N}\, r_{104} => r_{105}$	$r_{104}, r_{103}, \cdots, r_{104-N}$
$add\, r_{105-N}\, r_{105} => r_{106}$	$r_{105}, r_{104}, \cdots, r_{105-N}$
$\vdots$	$\vdots$

Stream with Register Pressure of N

Figure 2.3: Using Names to Control Register Pressure in a Stream

where *name* is a fully qualified name for the characteristic and *value* is the measured value. When all of the microbenchmark tools have been run, information from the resulting text file is used to produce a C interface in the form of an include file that describes the access functions into a database of these value pairs, and a linkable object file that holds the database itself. Information from the text file is also used to produce a human-readable report that can be found in the microbenchmarks directory.

2.3.1.1 Interface to Other PACE Tools

The database interface will consist of the following procedures:

Management Functions

`int rc_init()`	Initializes the RC interface. Returns 1 if successful or a negative number as an error code.
`void rc_final()`	Closes the RC interface and deallocates its data structures. Subsequent queries will fail.

Queries

`void *rc_query(char *s)`	`s` is a string that identifies the characteristic value. The call returns a structure that contains the measured value or an error code if there is some problem (*i.e.*, the value is unreliable, the query string does not match any entries in the database, *etc.*).

Chapter 3
An Overview of the PACE Compiler

The PACE compiler lies at the heart of the project's strategy to provide high-quality, characterization-driven optimization. The compiler uses a series of analyses and transformations to rewrite the input application in a way that provides better performance on the target system. The compiler supports feedback-driven optimization. It works with the RTS to implement runtime variation of optimization parameters. It has a mechanism to incorporate new optimization strategies derived by the ML tools.

3.1 Introduction

The PACE Compiler is a source-to-source optimizing compiler that tailors application code for efficient execution on a specific target system. It accepts as input parallel programs written in C with OPENMP calls. It produces, as output, a C program that has been tailored for efficient execution on the target system.

As shown in Figure 1.2, the PACE Compiler is as a series of tools that work together to create the optimized version of the application. Each of these tools is a complex system; each is discussed in its own chapter of the design document (see Table 1.1). This chapter serves as an introduction to the separate tools in the PACE Compiler and their interactions; § 3.3 discusses each of the components. Subsequent chapters provide more detail. This chapter also addresses the high-level, cross-cutting design issues that arise in the PACE Compiler.

3.2 Functionality

While the PACE Compiler is a collection of tools, it presents the end user with functionality that is similar to the functionality of a traditional compiler. The user invokes the PACE Compiler on an input application and the compiler produces executable code. To perform its translation and optimization, the PACE Compiler draws on resources provided by the rest of the PACE system, as shown in Figure 3.1. Because most of these interfaces are internal and most of the components are automatically controlled, the internal complexity of the PACE Compiler is largely hidden from the end user.

3.2.1 Input and Output

The PACE Compiler accepts, as input, an application written in C with calls to OPENMP libraries. The compiler assumes that the input program is a parallel program; while the compiler will discover some opportunities to exploit parallelism, detection of all available parallelism is not the focus of the PACE project.

Principal Contacts For This Chapter: Keith Cooper, keith@rice.edu, Vivek Sarkar, vsarkar@rice.edu, and Linda Torczon, linda@rice.edu

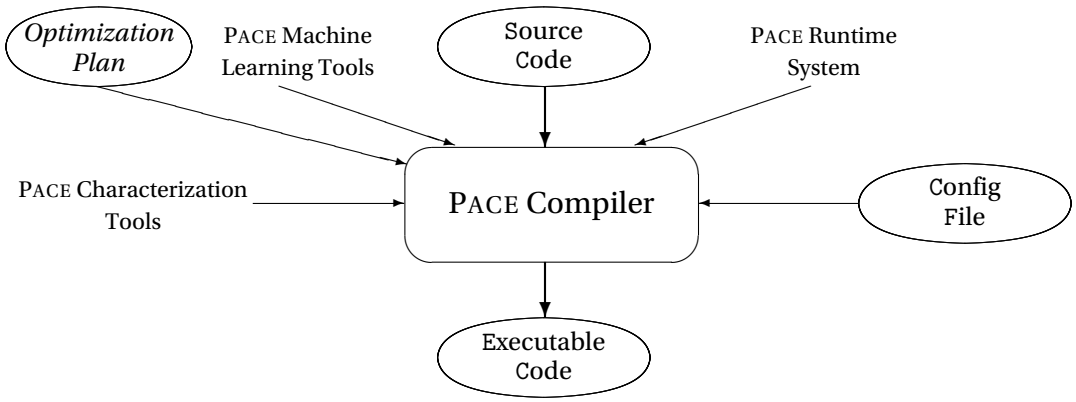


Figure 3.1: Interfaces to the PACE *Compiler*

The PACE Compiler produces, as output, an executable version of the input application, transformed to improve its performance on the target computer system. The compiler has several ways to generate executable code for the application. It can generate transformed C code and use the vendor-supplied native compiler to perform code generation. For some systems, it can use the LLVM backend to generate code (see § 3.4). The choice of code generators will depend on the quality of the native compiler and the availability of an LLVM backend for the system.

3.2.2 Interfaces

Figure 3.1 shows the primary interfaces between the PACE Compiler and the rest of the PACE system. The PACE Compiler uses information from several sources.

- **Characterization:** The PACE RC tools measure performance characteristics of the combined hardware/software stack of the target system. The PACE Compiler uses those characteristic values, both to drive optimizing transformations and to estimate the performance of alternative optimization strategies on the target system.
- **Machine Learning:** A PACE ML tool could provide suggestions to the compiler to guide the optimization process. Such a tool could communicate those suggestions by modifying the optimization plan for a given application or, perhaps, one of the default optimization plans.
- **Runtime System:** The PACE RTS could provide the compiler with measured performance data from application executions. This data would include detailed profile information. The Runtime System could pinpoint resource constraints that create performance bottlenecks.
- **Optimization Plan:** The PACE Compiler will coordinate its internal components, in part, by using an explicit optimization plan. The optimization plan is discussed in § 3.2.4.
- **Configuration File:** The configuration file is provided as part of system installation. It contains critical facts about the target system and its software configuration (see § 3.3.1).

The compiler could embed, in the executable code, information of use to the Runtime System, such as a concise representation of the optimization plan and data structures and calls to support runtime tuning of optimization parameters (see § 4.3.8 and 9.3.4). By embedding this information directly in the executable code, PACE could provide a simple solution to the storage of information needed for feedback-driven optimization and for the application of machine learning to the selection of optimization plans. This would avoid the need for a centralized store, like the central

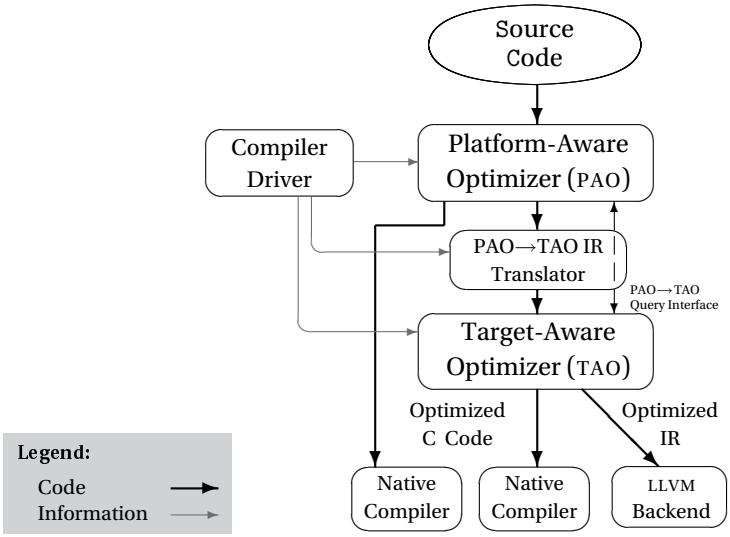


Figure 3.2: Structure of the PACE *Compiler*

repository in the $\mathcal{R}^n$ system of the mid-1980s. It would avoid the complication of creating a unique name for each compilation, recording those in some central repository, and ensuring that each execution can contact the central repository.

3.2.3 The Refactored Program Unit

The source code for an entire application can be refactored into *refactored program units* (RPUs) based on results from previous compilations, from previous executions, and from analysis in the current compilation. The refactoring has two primary goals:

1. To limit the size of any single compilation unit so that the rest of the compiler can do an effective job of optimization;[1] and

2. To group together procedures into program pieces that have similar performance profiles and rate-limiting resources so that they can be compiled using the same optimization plan.

The default RPU is a C compilation unit, i.e., a C main input file, along with elements on which the main file depends, defined in imported header files.

3.2.4 The Optimization Plan

To provide the flexibility that the PACE Compiler needs in order to respond to different applications and different target systems, the compiler needs a mechanism for expressing and recording optimization plans. The primary mechanism for changing the PACE Compiler's behavior is to suggest an alternate optimization plan. An *optimization plan* is a concrete representation of the transformations applied during a compilation. An optimization plan must specify, at least, the following items:

1. The location, i.e., the file system path, to the application directory;

[1] Evidence, over many years, suggests that code quality suffers as procedure sizes grow and as compilation unit sizes grow.

For each RPU:

2. The compilation path taken (e.g., full compilation path, short compilation path, or LLVM backend path);

3. The transformations that should be applied by the PAO and the TAO (see Figure 3.3), along with any parameters or commands that control those transformations;

4. The suggested order (if any) of those transformations;

5. Additional inputs, if any, to the PAO, PAO→TAO IR translator, TAO, and native compiler.

When the user invokes the compiler driver to prepare an application, the compiler driver provides a default optimization plan for the first compile. The executable is prepared using that default plan, the code is executed on user-provided representative data, and the RTS gathers performance statistics on that execution.

On the second compile, performance measurements from the RTS can be used to guide refactoring the code into RPUs; for example, context-sensitive profile information can play a role in decisions about both inline substitution and partitioning the code into RPUs.

The compiler driver then initializes the optimization plan for each RPU. It uses the rate-limiting resource information from each RPU to identify an initial plan for the RPU, drawn from a set of prefabricated plans. Possible rate-limiting resources include cache locality, ILP, multicore parallelism, and register pressure. Whenever the partition of the application into RPUs changes, the optimization plans for the RPU will be re-initialized.

The optimization plan guides the compiler components as they translate and transform the individual RPUs, in the second and subsequent compilations. The mechanism to change the behavior of those components is to change the optimization plan. In principle, any component of the compiler can change the optimization plan and affect the behavior of other components. In PACE, we will focus our attention on the kinds of changes described in § 1.3.2.

The definitive representation of an optimization plan resides in a file in the application directory. To simplify the handling of information in the PACE tools, the compiler will insert a concise representation of the optimization plan as data into the object code produced by any specific compilation.

The optimization plan as described in this section has not been implemented. As currently implemented, the user can specify the transformations to be applied by the PAO and the TAO through use of an optimization path option. There are four different options that can be specified. These are described §4.3.7.

3.3 Components of the PACE Compiler

The PACE Compiler has a number of major components, shown in Figure 3.2.

3.3.1 Compiler Driver

The compiler driver provides application programmers with their interface to the PACE Compiler. To compile a program, the programmer creates a directory that contains the source code for the application and for any libraries that are to be optimized with it. Next, the programmer invokes the compiler driver on that directory.

The compiler driver has the responsibility of managing the compilation process. It could create a location for the results of this compilation in a distributed repository. We will defer discussion of the first task until the other components have been described (see § 3.4).

The PACE Compiler could store its work products, such as annotations, the optimization plan, intermediate files, and analysis results, in the application directory. Within the application directory, the compiler driver could create a subdirectory for each compilation; it could pass the location of this directory to each of the other components that it invokes and ensure that the location is embedded in the final executable code where the RTS could find it. This subdirectory could become part of the distributed repository, containing the records of both compilation and execution for this application.

Application refactoring could change the number of implementations for any given procedure using either inline substitution or procedure cloning.[2] It could group together into an RPU procedures that have an affinity—either the same rate limiting resource as identified by the RTS or a frequent caller/callee relationship. It could pad, align, and reorder data structures.

3.3.2 Platform-Aware Optimizer

The compiler driver iterates through the RPUs. Each RPU serves as a compilation unit. The driver invokes the Platform-Aware Optimizer (PAO) for each RPU, passing the location of the application directory to it. The PAO applies analyses and transformations intended to tailor the application's code to platform-wide, or system-wide, concerns. Of particular concern are efficiency in the use of the memory hierarchy and the use of thread-level parallelism. The PAO operates on the code at a level of abstraction that is close to the original C source code.

The PAO could find the optimization plan for the application in the application directory. The PAO could modify the optimization plan executed by the PAO and TAO components. It could generate commands that instruct and constrain the TAO in its code transformations. PAO transformations could include loop tiling, loop interchange, unrolling of nested loops, and scalar replacement (see § 6). The PAO chooses transformation parameters, for example choosing unroll factors for each loop in a multidimensional loop nest, based on the results of querying the TAO through the PAO–TAO query interface (see § 4.3.5).

3.3.2.1 Polyhedral Analysis and Transformation Tools

The PAO includes a subsystem that uses polyhedral analysis and transformations to reorganize loop nests for efficient memory access (see § 5). The polyhedral transformations use parameters of the memory hierarchy, derived by the PACE RC tools, and the results of detailed polyhedral analysis to rewrite entire loop nests.

3.3.3 PAO→TAO IR Translator

Because the PAO and the TAO operate at different levels of abstraction, the PACE Compiler must translate the IR used in the PAO into the IR used in the TAO. The PAO uses the abstract syntax trees in the SAGE III IR. The TAO uses the low-level linear SSA code defined by LLVM.

The translator lowers the level of abstraction of PAO IR, converts it into SSA form, and rewrites it in TAO IR. Along the way, it must map analysis results and annotations created in the PAO into the TAO IR form of the code. The compiler driver invokes the translator for each RPU, and passes it any information that it needs.

3.3.4 Target-Aware Optimizer

The Target-Aware Optimizer (TAO) takes code in IR form that has been tailored by the PAO for the platform-wide performance characteristics and maps it onto the architectural resources of the individual processing elements. The TAO adjusts the code to reflect the specific measured capacities

[2] Specifically, it should clone procedures based on values from forward propagation of interprocedural constants in the last compilation. Out-of-date information may cause under-optimization; it will not cause a correctness problem [19].

of the individual processors. It also provides optimizations that may be missing in the native compiler, such as operator strength reduction, algebraic reassociation, or software pipelining.

The TAO can provide three distinct kinds of backend.

- On machines where the underlying LLVM compiler has a native backend, such as the X86 ISA, the TAO can generate assembly code for the target processor.
- A C backend for the TAO can generate a source language program in C. The C backend will adjust the code for the measured strengths and weaknesses of the native compiler.
- A query backend for the TAO answers queries from the PAO. This backend uses a generic ISA, with latencies and capacities established by measured system characteristics.

The TAO is invoked on a specific optimized RPU. One of its input parameters specifies which backend it should use in the current compilation step. Different paths through the PACE Compiler invoke the TAO. The compiler driver can invoke the TAO to produce either native assembly code, using an LLVM backend, or tailored C source code. The PAO invokes the TAO directly with the query backend to obtain answers to specific queries (see § 8.3.5).

The TAO can consult the optimization plan, stored in the application directory, to guide its actions. The specific actions taken by the TAO in a given invocation can depend heavily on (1) the choice of backend, specified by the tool that invokes it; (2) the context of prior optimization, recorded from the PAO, and prior compilations; and (3) results from the PACE RC tools.

3.4 Paths Through the PACE Compiler

The compiler driver can put together the components of the PACE Compiler in different ways. The thick lines in Figure 3.2 show the three major paths that code takes through the PACE Compiler.

Full Compilation Path The compiler driver can invoke, in sequence, the PAO, PAO→TAO IR translator, TAO, and native compiler. This path corresponds to the centerline of the figure. The compiler driver invokes all of the tools in the PACE Compiler and directs the TAO to generate tailored C source code as output.

Short Compilation Path If the target system has a strong native compiler, as determined by the RC tools, the compiler driver may bypass the TAO. (The compiler driver directs the PAO to emit C source code.) This sequence relies on the native compiler for target-specific optimization.

LLVM Backend Path If an LLVM backend is available on the target machine, the compiler driver can invoke a sequence that uses the LLVM backend to generate native code, bypassing the native compiler. In this scenario, it invokes, in sequence, the PAO, PAO→TAO IR translator, and TAO. The compiler driver tells the TAO to use the LLVM backend.

In any of these paths, the PAO can invoke the TAO through the PAO–TAO query interface.

3.5 Optimization in the PACE Compiler

To avoid redundant effort, the PACE Compiler should avoid implementing and applying the same optimization at multiple levels of abstraction, unless a clear technical rationale suggests otherwise. Thus the PAO and the TAO each have their own set of transformations; most of the transformations occur in just one tool.

Platform-Aware Optimizer	Target-Aware Optimizer
Dead code elimination[1]	Dead code elimination[1]
Inlining Outlining Procedure cloning	Superblock cloning Tail call elimination SW branch prediction
Interprocedural constant propagation Intraprocedural constant propagation	Local constant propagation Intraprocedural constant propagation[2]
Partial redundancy elimination Enhanced scalar replacement[3] Algebraic reassociation[3]	Partial redundancy elimination Enhanced scalar replacement[3] Algebraic reassociation[3]
Idiom recognition Synchronization reduction	Algebraic simplification[4] Operator strength reduction
If conversion Scalar expansion Scalar replacement[6]	Tree-height balancing Regeneration of SIMD loops[5]
PDG-based code motion	Lazy code motion
Polyhedral transformations Loop transformations[7]	Prefetch insertion Loop unrolling
Array padding Reorder structure nesting Array & loop linearization	

[1] Include multiple forms of "dead" code and multiple transformations.
[2] May be redundant in the TAO.
[3] Placement in PAO or TAO will depend on experimentation.
[4] Includes application of algebraic identities, simplification of predicate expressions, and peephole optimization.
[5] Generation of SIMD loops in C source form is tricky.
[6] Expanded to include pointer-based values.
[7] Will consider unroll, unswitch, fuse, distribute, permute, skew, align, reverse, tile, and shackling. Some combinations obviate need for others. Some overlap with polyhedral transformations.

Figure 3.3: Optimizations Considered for the PACE Compiler

Figure 3.3 shows the set of optimizations considered for implementation in the PACE Compiler, along with a division of those transformations between the PAO and the TAO.[3] This division was driven by the specific mission of each optimizer, by the level of abstraction at which that optimizer represents the application code, and by the kinds of analysis performed at that level of optimization. Overlap between their effects will make some of them redundant. Others may address effects that cannot be seen in the limited source-to-source context of the PACE Compiler.

The canonical counterexample to this separation of concerns is dead code elimination—specifically, elimination of useless code and elimination of unreachable code. Experience shows that routine application of these transformations at various times and levels of abstractions both reduces compile times and simplifies the implementation of other optimizations. Thus, in the table, they appear in both the PAO and the TAO.

[3] Because the DARPA-sponsored AACE program was cancelled, only a subset of the envisioned transformations have been implemented.

Some optimizations require collaboration among the various tools in the PACE Compiler. Consider, for example, vectorization. The PAO may identify a loop nest that is an appropriate candidate for vectorization—that is, (1) the loop nest can execute correctly in parallel; (2) the loop executes enough iterations to cover the overhead of vector startup; (3) the PACE RC tools have shown that such a loop can run profitably in vector form; and (4) the configuration file contains the necessary compiler options and commands to allow the PACE Compiler to generate code that will vectorize. When the PAO finds such a loop, it must encode the loop in a form where the remainder of the tools will actually produce vector code for the loop.

In the full compilation path, the PAO annotates the loop nest to indicate that it can run in vector mode. The PAO→TAO IR translator encodes the SIMD operations into the LLVM IR's vector operations. The TAO generates appropriate code, using either the C backend or the native backend. To ensure that the TAO can generate vector code for these operations, the PAO may need to disable specific TAO optimizations, such as software pipelining, block cloning, tree-height-balancing, and loop unrolling. To do so, the PAO would modify the optimization plan seen by the TAO. Finally, using the vectorization annotations for the vendor compiler provided in the system configuration file, the C backend could mark the vectorizable loops with appropriate annotations to inform the vendor compiler that they should be vectorized (e.g., IVDEP).

In the LLVM backend path, optimization proceeds as above, except that responsibility for generating vector operations lies in the LLVM backend. Again, the PAO may disable some TAO optimizations to prevent the TAO from introducing unneeded dependences that prevent vector execution.

In the short compilation path, the PAO annotates the loop nest, as in the full compilation path. In this case, however, the consumer of that annotation is the PAO pass that regenerates C code; it will mark the loop nest with appropriate annotations for the vendor compiler, as found in the system configuration file.

Appendix B provides more detail on our plans for handling vectorization in the PACE Compiler.

3.6 Software Base for the PACE Compiler

The PACE Compiler builds on existing open source infrastructure. The following table shows the systems used in each of the components of the PACE Compiler.

Infrastructure Used in the PACE Compiler	
PAO	EDG front end[†], Rose, Candl, Pluto, CLooG
TAO	LLVM

In the PAO and TAO, the actual PACE tools are built as extensions of the open source tools.

[†] Licensed software

Chapter 4

PACE Platform-Aware Optimizer Overview

4.1 Introduction

Figure 1.2 provides a high-level overview of the PACE Compiler design. This chapter provides an overview of the design for the Platform-Aware Optimizer (PAO). The PAO component performs transformations and optimizations on a *high-level*, or near-source, representation of the code, which we will refer to as a *high-level intermediate representation* (HIR). The HIR enables a comprehensive set of analyses and transformations on both code and data. Because the Rose system serves as the primary infrastructure on which the PAO is build, the SAGE III IR from Rose serves as the HIR for PAO. The motivation for having separate Platform-Aware and Target-Aware Optimizers is to support both transformations that must be performed at the near-source level and transformations that might be better performed at a much lower level of abstraction.

4.2 Functionality

The PAO takes as input *refactored program units* (RPUs). It generates as output transformed versions of the RPU. The transformed versions have been optimized for measured characteristics of the target platform, identified by the Resource Characterization (RC) component, as well as by the configuration file for the target platform. Figure 4.1 illustrates the ways that PAO interacts with other parts of the system.

4.2.1 Input

The primary input for a single invocation of the PAO is C source code for an RPU. Additional inputs (as shown in Figure 4.1) include compiler directives from the compiler driver, resource characteristics for the target platform, profile information with calling-context-based profile information for the source application, and TAO cost analysis feedback (Path 3 in Figure 4.2).

4.2.2 Output

As its output, the PAO produces a transformed HIR for the input RPU. The transformed code can be translated into either C source code or into the IR used in the Target-Aware Optimizer (TAO). This latter case uses the PAO→TAO IR translator, described in § 7; the translator is also used to translate from the SAGE III IR to the LLVM IR in the PAO–TAO query mechanism, as shown in Figure 4.2.

As described in § 3.3.1, the compiler driver invokes the PAO, passing the location of the application directory. The PAO consults the optimization plan for the application, and can modify its own

Principal Contacts For This Chapter: Vivek Sarkar, vsarkar@rice.edu

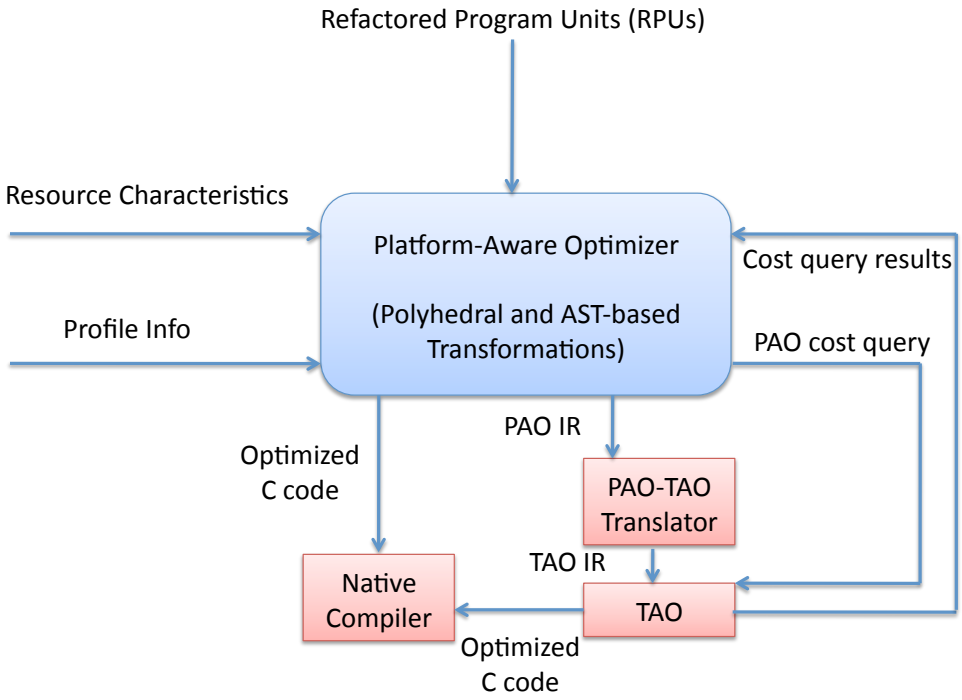


Figure 4.1: Platform-Aware Optimizer Interfaces

optimization plan to determine how it should process the RPU (see § 3.2.4). It can also modify the optimization plan for the TAO within a single compilation, to instruct and constrain the TAO in its code transformations.

The two primary compilation paths through the PAO are shown as Path 1 and Path 2 in Figure 4.2.

Path 1 of of Figure 4.2 shows how the PAO implements its part of both the full compilation path and the LLVM backend compilation path, as described in § 3.4. Along with optimized user code in SAGE III IR, the PAO produces auxiliary IR information, including profiling, aliasing, and dependence information. It may also amend the application's optimization plan, which determines the code transformations performed by the PAO and TAO for this RPU. Next, the compiler driver invokes the PAO→TAO IR translator to convert the SAGE III IR into the LLVM IR used in the TAO. The translator associates the auxiliary information from the SAGE III IR with the new LLVM IR for the RPU. Finally, the compiler driver invokes the TAO, which uses the LLVM IR, the auxiliary information associated with it, and the optimization plan; the TAO performs further optimization and produces executable code for the RPU.

Path 2 of Figure 4.2, shows the flow of information when the PAO queries the TAO for cost estimates to guide its high-level transformations. To perform a TAO query, the PAO constructs a synthetic function for a specific code region. The synthetic function contains a transformed version of the application's original code for which the PAO needs a cost estimate. The PAO uses the PAO→TAO IR translator to convert the synthetic function into LLVM IR and it invokes the TAO with a directive to use the query backend. Note that, on this path, the PAO directly invokes both the PAO→TAO IR translator and the TAO, rather than relying on the compiler driver to invoke those tools.

During compilation, the PAO may determine that certain parameters might benefit from runtime optimization. The PAO then prepares the inputs needed by the API for runtime feedback-

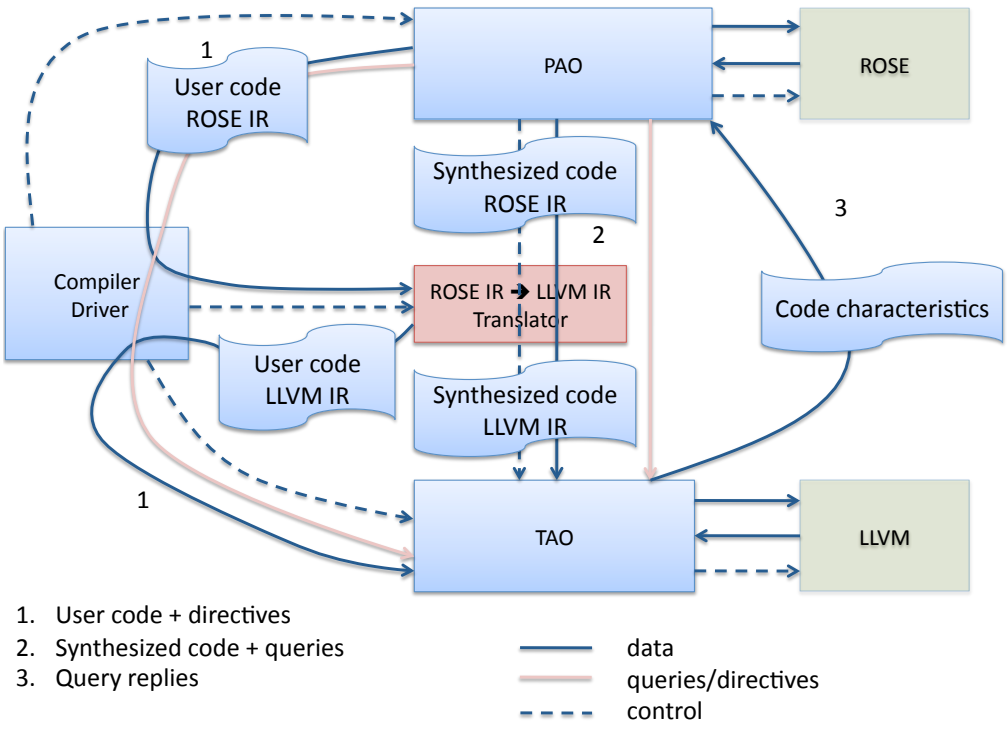


Figure 4.2: Overview of PAO-TAO Interfaces

directed parameter selection presented by the RTS (see § 9.3.4).

4.3 Method

In this section, we include design details for the PAO component of the PACE compiler. These details follow from a basic design decision to use the Edison Design Group (EDG) front end for translating C source code, and the Rose transformation system with the SAGE III IR for performing high level transformations.

After the SAGE III IR is created by the *front end* (§ 4.3.1), the passes described below in Sections 4.3.2 – 4.3.5 are repeated until no further transformation is performed (a fixed point is reached) or until a predefined maximum number of iterations is reached. At that point, a transcription phase produces either transformed C source code or LLVM IR bitcode (see § 4.3.6).

The PACE design enforces a strict separation of concerns among three aspects of performing each transformation: 1) *Legality analysis*, 2) *Cost analysis*, and 3) *IR update*. This design makes it possible to modify, replace, or consider multiple variants of any one aspect of a transformation without affecting the others. As described in § 5 and 6, there are two main modes of implementing transformations in the PAO. Section§ 5 summarizes the use of a polyhedral transformation framework that is capable of selecting a combination of transformations in a single unified step. Section§ 6 summarizes the classical transformations that the PAO can apply directly to the SAGE III IR. These two modes complement each other since neither one subsumes the other.

4.3.1 Front end

The PAO relies on the EDG front end to parse the input C source code and to translate the program to a SAGE III IR. The PACE Project will not modify the EDG front end; instead, we plan to include

pre-compiled binaries of the EDG front-end in the PACE distribution.

4.3.2 Program Analyses

Before the PAO begins to explore a wide space of transformations to optimize the input program, it needs to perform some *canonical program analyses* to broaden the applicability of transformations as much as possible. Examples of analyses that could be implemented include:

- Global Value Numbering identifies equivalent values.
- Constant Propagation identifies compile-time constants.
- Induction Variable Analysis identifies induction variables; it builds the two prior analyses.
- Unreachable Code Elimination identifies code that has no feasible execution path—that is, no path from procedure entry reaches the code.
- Dead Code Elimination identifies code that creates values that are never used.

The PAO includes an extensive framework for polyhedral analysis and transformation, described in § 5. That framework depends in critical ways on the fact that the code has already been subjected to the canonical analyses outlined above. In many cases, a loop nest as expressed in the input application may not a priori satisfy the constraints to be a Static Control Part (SCoP) in the polyhedral framework (§ 5.2.1). Often, the canonical analyses transform such loop nests into a form where they satisfy the constraints needed for application of the polyhedral techniques.

Though the canonical analyses that could be implemented in PAO all well understood in their scalar versions, their implementation in the PAO poses new research challenges for four reasons. First, the analyses in the PAO must handle both array and pointer accesses whereas previous implementations of the example analyses listed above have typically restricted their attention to scalar variables and their values. Second, the PAO must analyze code that is already in parallel form (OPENMP extensions to C) whereas prior work assumes sequential code. Third, the PAO could attempt to combine these analyses to the fullest extent possible. Prior work has shown that combining analyses can produce better results than computing them independently. Finally, the PAO could build analyses that can be incrementally updated after each transformation is performed, so as to reduce the overall cost of the analyses.

The polyhedral framework also plays a role in the process of identifying, transforming, and expressing loops that can execute in vector form. The polyhedral framework identifies such loops in a natural way. It computes data dependence information for the memory accesses in the loop and its surrounding code and identifies consecutive memory accesses. This information, created in the PAO's polyhedral analysis framework, is passed to the TAO as annotations to the IR by the PAO→TAO IR translator. The TAO uses the information to generate vector code. Appendix B provides a system-wide view of vectorization in the PACE Compiler.

4.3.3 Legality Analysis

A necessary precondition before a compiler can perform a code transformation is to check the *legality* of the transformation. Legality conditions for a number of well-known transformations have been summarized in past work e.g., [2, 69]. We summarize below the legality conditions for many of the transformations that could be performed by the PAO. While much of the research literature focuses on the data dependence tests for legality, it is also necessary to check control dependence and loop bound conditions to ensure correctness of loop transformations. The following list summarizes the legality conditions for the transformations described in § 6.

- Loop interchange: no resulting negative dependence vector, counted loop with no premature exit, loop-invariant or linear loop bounds with constant stride (or loop invariant-loop bounds with loop-invariant stride for loops being interchanged)
- Loop tiling: same conditions as loop interchange for all loops being permuted
- Unroll-and-jam: same data dependence conditions as loop tiling, but loop bounds must be invariant
- Loop reversal: no resulting negative dependence vector, counted loop with no premature exit, arbitrary loop bounds and stride
- Unimodular loop transformation (includes loop skewing): counted loop with no premature exit, no loop-carried control dependence, loop-invariant or linear loop bounds with constant stride
- Loop parallelization: no resulting negative dependence vector, counted loop with no premature exit, arbitrary loop bounds and stride
- Loop distribution: no control + data dependence cycle among statements being distributed into separate loops
- Loop fusion: no loop-independent dependence that prevents code motion to make fused loops adjacent, and no loop-carried fusion-preventing dependence
- Scalar replacement: no interfering aliased locations
- Constant replacement: variable must have propagated constant value on all paths
- Scalar renaming, scalar expansion, scalar privatization: no true dependences across renamed locations
- Unreachable code elimination: no feasible control path to code
- Useless (dead) code elimination: no uses of defs being eliminated
- Polyhedral transformation: input loop nest must form a "Static Control Part" (SCoP); see § 5.2.1 for more details

4.3.4 Cost Analysis: Memory Hierarchy

The other precondition that the compiler must satisfy before it performs some code transformation is to check the *profitability* of that transformation via *cost analysis*. Cost analysis will play a more central role in the PACE Compiler than in many earlier compilers, because the compiler has better knowledge of the performance characteristics of the target machine, as measured by the RC tools. One particular challenge is to perform effective and accurate memory cost analysis on an HIR such as the SAGE III IR.

Consider the lowest (level 1) levels of a cache and TLB. The job of memory cost analysis is to estimate the number of distinct cache lines and distinct pages accessed by a (hypothetical) *tile* of $t_1 \times \ldots \times t_h$ iterations, which we define as $DL_{total}(t_1, \ldots, t_h)$ and $DP_{total}(t_1, \ldots, t_h)$, respectively. Assume that the tile sizes are chosen so that DL_{total} and DP_{total} are small enough so that no collision and capacity misses occur within a tile i.e., $DL_{total}(t_1, \ldots, t_h) \leq$ effective cache size and $DP_{total}(t_1, \ldots, t_h) \leq$ effective TLB size.

An upper bound on the memory cost can then estimated be as follows:

$$COST_{total} \quad = \quad (\text{cache miss penalty}) \times DL_{total} + (\text{TLB miss penalty}) \times DP_{total}$$

Our objective is to minimize the memory cost per iteration, $COST_{total}/(t_1 \times \ldots \times t_h)$. This approach can be extended to multiple levels of the memory hierarchy.

An upper bound on the memory cost typically leads to selection of tile sizes that may be conservatively smaller than empirically observed optimal values. Therefore, we will also pursue a lower bound estimation of memory costs in the PACE project, based on the idea of establishing a lower bound ML, the minimum cache size needed to achieve any intra-tile reuse. In contrast to DL, the use of ML leads to tile sizes that may be larger than empirically observed optimal values. The availability of both bounds provides a limited space for empirical search and auto-tuning as opposed to a brute-force exhaustive search over all possible tile sizes.

4.3.5 Cost Analysis: PAO-TAO Query Interface

The HIR used in the PAO simplifies both high-level transformations and analysis of costs in the memory hierarchy. There are, however, other costs in execution that can only be understood at a lower level of abstraction. Examples include register pressure (in terms of both *MAXLIVE* and *spill-cost*, see § 8.3.5), instruction-level parallelism, simdization, critical-path length, and instruction-cache pressure. To address these costs, the PACE Compiler includes a PAO-TAO query mechanism that lets the TAO estimate various costs on specific code fragments. The PAO passes the particular cost estimates it needs to the TAO through a query data structure. The PAO uses the results of such queries to guide the selection of various parameters for the high-level transformations.

To perform a query, the PAO creates a synthetic function that encapsulates the transformed code from the region of interest and passes it, along with auxiliary information and a query, to the TAO.

There are three approaches for generating the synthetic function in the PAO:

1. Cloning of the entire function containing the specific user code region. This will include full intraprocedural context for the code region. This approach would yield the most precise query results, but there still will be some pollution of cost information by other code regions in the function. This is the first approach to be implemented in the PACE compiler.

2. Cloning of the user code region of interest along with its control and data slice from the function. This will include full intraprocedural context for the slice of the code region. This approach would produce more precise query results, but there will be some pollution of cost information by other code regions in function due to conservative slicing.

3. Cloning only of the user code region of interest as a separate function. This will require local variables from surrounding context to be passed as reference parameters. The advantage of this approach is that there will be minimal code duplication, the disadvantage is that it will lead to a less precise analysis of cost results due to the absence of the exact context for local variables.

The auxiliary information includes information on aliases, data structure alignment, and runtime profile information from the RTS. It passes this information to the TAO, along with a directive to use the query backend rather than a code-generating backend. For simplicity, each query includes a single function. If the PAO needs to submit multiple queries, they can be submitted separately as a sequence of independent single queries.

For each query, the query backend of the TAO (invoked in-core[1] with PAO) uses the optimized low-level code and values from the RC tools to produce cost estimates for the input synthetic function. The TAO records its estimates in the query data structure and passes it back (Path 3 on Figure 4.2) to the PAO.

[1]The TAO shares a process and an address space with the PAO to facilitate TAO access to PAO generated annotations and data structures. This arrangement should reduce the cost of making a PAO–TAO query.

There are three query interfaces implemented in the PAO:

1. Live Variable Query (MAXLIVE): the maximum number of live variables (integer variables and floating point variables);
2. Critical Path Query: the critical-path length;
3. Machine Code Query:
   (a) SPILLCOST: the number of stack load and store operations generated for the given synthetic function;
   (b) CODESIZE: the number of the machine instruction generated for the given synthetic function;
   (c) SIMD: the cost of SIMDizing the synthetic function.

Figure 4.3 gives the detail of PAO/TAO query process. The query starts from the cost driven transformation, which passes the input function's Sage IR code through Rose-to-LLVM translator to generate LLVM IR for TAO cost estimation. In TAO level, there are two types of cost estimation: architecture independent (MAXLIVE, critical path length) and architecture dependent (spilling cost, instruction size) (more detail is discussed in Section 8.3.5). Before the invocation of these cost estimation, the architecture independent/dependent transformations are applied respectively to get the optimized code for precise cost estimation.

To get cost value from TAO, the PAO and TAO share a common data structure named feedback value repository, the cost value retrieved from TAO is written into the feedback value repository. PAO can access the cost value through query interfaces that read value from feedback value repository. By interpreting the query results generated by TAO, PAO is able to choose optimal transformation parameters that benefit program performance. For example, the feedback driven loop unroll-and-jam uses TAO feedback to decide the best unroll factor during compilation time.

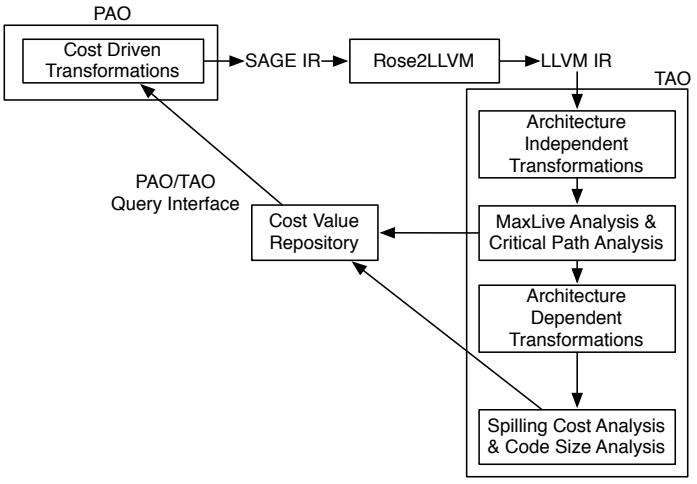


Figure 4.3: The Workflow of Cost Estimation and Query

4.3.6 Transcription

When the PAO has finalized its selection of transformations, the SAGE III IR is updated to finalize all transformations. At that point, a *transcription* phase uses the SAGE III IR to either generate source

code or LLVM IR. The compiler driver then uses the PAO→TAO IR translator to generate LLVM bitcode as input to the Target-Aware Optimizer (TAO).

4.3.7 The Optimization Plan

The PAO uses the application's optimization plan both to obtain guidance in its own operation and to affect the processing of code in other parts of the PACE Compiler. The initial optimization plan is generated by the compiler driver (see § 3.2.4). It may be modified by other components of the PACE system, including the PACE Machine Learning tools.

The PAO could modify the optimization plan within a compilation to guide the application of transformations in the TAO. For example, if analysis in the PAO and the PAO–TAO query interface shows that some loop nest has unsupportable demand for registers and that loop nest was produced by inline substitution, the PAO may direct that inlining be avoided in that function or that RPU. Similarly, on a loop that the PAO identifies as vectorizable, the PAO may direct the TAO not to apply transformations, such as tree-height balancing, that might change the code structure and prevent vector execution.

The optimization plan as described in this section has not been implemented. As currently implemented, the user can specify the transformations to be applied by the PAO and the TAO through use of an optimization path option. There are four different options that can be specified. One option invokes the non-polyhedral AST-based transformations for parametric loop tiling and the loop unroll and jam module. A second option invokes the polyhedral loop transformation module. A third option invokes the LLVM vectorizer (including the necessary analysis and transformations in the polyhedral module) and the unroll and jam module. A fourth option invokes polyhedral loop transformation, the LLVM vectorizer and the unroll and jam module. If no option is specified then a default compile path is applied: the input code just goes through the rose to LLVM translator.

4.3.8 PAO Parameters for Runtime System

The PAO might determine that certain parameters can most likely benefit from runtime optimization. During compilation, the PAO sets certain parameters and passes them to the RTS. The PAO could also present the RTS with a closure that contains an initial parameter tuple, a specification of the bounds of the parameter tuple space, a generator function for exploring the parameter tuple space, and a parameterized version of the user's function to invoke with the closure containing the parameter tuple and other state. Additional detail about the RTS interface for online feedback-directed parameter selection can be found elsewhere (§9.3.4).

To illustrate how the PAO could exploit RTS support for online feedback-directed parameter selection, consider the problem of Parametric Tiling described in § 5.3.6. Here, the PAO recognizes the need to block the code to improve performance in the memory hierarchy, but it cannot determine the optimal tile size at compile time. In this case, the PAO could prepare the loop nest for runtime tuning by constructing the inputs for the RTS online feedback-directed parameter selection API and rewriting the code to use that facility.

4.3.9 Guidance from Runtime System

If the application has already been profiled, the RTS could provide the PAO with high-level quantitative and qualitative guidance about runtime behavior. This information may include data on resource consumption, on execution time costs, and on specific inefficiencies in the code (see § 9.2.3). The PAO could use this information to determine where to focus its efforts and how to alter the optimization plan to improve overall optimization effectiveness.

Chapter 5

PolyOpt – The Polyhedral Optimization Framework

The Polyhedral Optimization (PolyOpt) subsystem of PAO (Platform Aware Optimizer) is developed to perform transformations such as fusion, distribution, interchange, skewing, shifting, tiling, etc. on affine loop nests. The polyhedral transformation approach is based on the Pluto system that has shown good promise for transformation of a number of affine computational kernels. PolyOpt is implemented as a Sage AST to Sage AST transformer integrated with Rose.

5.1 Introduction

The Polyhedral Optimization (PolyOpt) subsystem of PACE is a component of the PAO (Platform Aware Optimizer). It enables data dependence analysis and complex loop transformations such as fusion, distribution, interchange, skewing, shifting, tiling, etc. to be applied to affine loop nests in a program optimized by PACE. PolyOpt is integrated with Rose. It takes as input the Sage AST representation for the source code to be optimized by PolyOpt, identifies subtrees of the AST that represent affine computations, transforms those AST subtrees to a polyhedral representation, performs loop transformations using the polyhedral representation, and finally converts the polyhedral representation back to Sage AST. The Sage AST transformations performed by PolyOpt will be preceded and followed by other (non-polyhedral) transformations described in § 6. The polyhedral transformation approach implemented is based on the Tiling Hyperplane method [14, 15] that has shown great promise for transformation of a number of affine computational kernels.

5.1.1 Motivation

The polyhedral model [38] provides a powerful abstraction to reason about transformations on collections of loop nests by viewing a dynamic instance (iteration) of each assignment statement as an integer point in a well-defined space called the statement's *polyhedron*. With such a representation for each assignment statement and a precise characterization of inter- and intra-statement dependences, it is possible to reason about the correctness of complex loop transformations. With the conventional abstractions for data dependences used in most optimizing compilers (including `gcc` and vendor compilers), it is very difficult to effectively perform integrated model-driven optimization using key loop transformations such as permutation, skewing, tiling, unrolling, and fusion across multiple loop nests. One major challenge with AST-based loop transformation systems is the case of imperfectly nested loops; this is seamlessly handled in a polyhedral compiler

Principal Contacts For This Chapter: Atanas Rountev, `rountev@cse.ohio-state.edu`, and P. Sadayappan, `saday@cse.ohio-state.edu`

transformation framework.

5.1.2 Background

The input to a transformation system based on the polyhedral model is a region of code containing a sequence of loop nests. Variables that are invariant in the region (e.g., array sizes) are referred to as *parameters*. The main constraints imposed on the region of code are as follows (see § 5.2.1 for a complete list of constraints). Loop bounds are affine functions (i.e., $c_1 i_1 + \ldots + c_n i_n + c_{n+1}$; c_k are compile-time constants) of loop iteration variables and parameters; this includes imperfectly nested and non-rectangular loops. Array index expressions are also affine functions of iteration variables and parameters. Such program regions are typically the most computation-intensive components of scientific and engineering applications, and they appear often in important scientific applications [11].

A statement s surrounded by m loops is represented by an m-dimensional polyhedron[1] referred to as an iteration space polyhedron. The coordinates of a point in this polyhedron (referred to as an *iteration vector*) correspond to the values of the loop indices of the surrounding loops. The polyhedron can be defined by a system of affine inequalities derived from the bounds of the loops surrounding s; each point in the polyhedron corresponds to a run-time instance of s.

A significant advantage of using a polyhedral abstraction of statements and dependences is that compositions of loop transformations have a *uniform algebraic representation* that facilitates integrated global optimization of multi-statement programs. In particular, it is possible to represent arbitrary compositions of loop transformations in a compact and uniform manner, and to reason about their cumulative effects through well-defined algebraic cost functions. In contrast, with the traditional model of data dependence that is used in most optimizing compilers, it is very difficult to model the effect of a sequence of loop transformations. Previous work using unimodular transformations and iteration-reordering transformations (see for instance [6, 78, 70]) were limited to modeling the effect of sequences of iteration-reordering transformations. However, they could not accommodate transformations that changed a loop body such as distribution and fusion, or transformations on imperfect loop nests.

Further, global optimization across multiple statements is not easily accomplished (e.g., transformation of imperfectly nested loops is a challenge). Phase ordering effects as well as rigid and less powerful optimization strategies are all factors that make syntax-based transformations of loop nests less powerful than polyhedral-based ones for optimizing affine loop nests [41].

5.2 Functionality

The polyhedral transformation framework in PACE takes as input the Sage ASTs for all functions in an input program. In each AST, it identifies code fragments (i.e., AST subtrees) that can be targets of polyhedral analyses and optimizations. Each such fragment is referred to as a *Static Control Part* (SCoP). Each SCoP is analyzed and transformed; the result is a new subtree which is then inserted in the place of the original subtree in the function's AST. Polyhedral data dependence analysis is exposed to subsequent passes of the PACE compiler, through annotations of the Sage nodes with dependence information computed during the polyhedral analysis.

[1] A hyperplane is an $n-1$ dimensional affine subspace of an n-dimensional space. A half-space consists of all points of an n-dimensional space that lie on one side of a hyperplane (including the hyperplane); it can be represented by an affine inequality. A polyhedron is the intersection of finitely many half-spaces.

5.2.1 Static Control Part (SCoP) Code Fragments

A SCoP is an AST subtree with a particular structure which allows powerful polyhedral analyses and optimizations. A conceptual grammar for a SCoP can be defined as follows[2]

⟨*SCoP*⟩	`::=`	⟨*ElementList*⟩
⟨*ElementList*⟩	`::=`	⟨*Element*⟩ | ⟨*Element*⟩ ⟨*ElementList*⟩
⟨*Element*⟩	`::=`	⟨*Statement*⟩ | ⟨*Loop*⟩ | ⟨*If*⟩
⟨*Loop*⟩	`::=`	`for` ⟨*IteratorVariable*⟩ `=` ⟨*LowerBound*⟩ `,` ⟨*UpperBound*⟩ `{` ⟨*ElementList*⟩ `}`
⟨*If*⟩	`::=`	`if` ⟨*Expression*⟩ `comp_op` ⟨*Expression*⟩ `{` ⟨*ElementList*⟩ `}` `else` `{` ⟨*ElementList*⟩ `}`

Expressions and statements Each loop bound must be an affine expression $c_1 i_1 + \ldots + c_n i_n + c_{n+1}$ where c_k are compile-time constants. The two expressions compared in operator `comp_op` in an if-statement must also be affine. Inside an statement, each index expression e_k in an array access expression $a[e_1]\ldots[e_d]$ (where a is a d-dimensional array) must be an affine expression.[3]

For every statement, each expression which denotes a memory location must be a scalar variable x or an array access expression $a[e_1]\ldots[e_d]$. No pointer dereferences `*p` or accesses to structure fields `s.f` or `p->f` are supported. Conservatively, function calls are also disallowed in a statement. It is possible to relax this last constraint by allowing calls to side-effect-free functions, assuming that such function names are provided by external sources. Currently, function names matching the prototypes defined in `math.h` header can optionally be accepted by PolyOpt.

Iterators, parameters, and affine expressions All scalar variables that appear anywhere in the SCoP can be classified into three disjoint categories:

- Loop iterators; there must not be any reference to a loop iterator which is a write, beside the `for` loop increment
- Parameters: not iterators; there must not be any reference to a parameter which is a write
- Modified variables: all variables referenced in the scop that are not loop iterators nor parameters

Expressions in loop bounds, if-statements, and array access expressions must be affine forms of *the SCoP parameters and the iterators of surrounding loops in the SCoP*. Checking that an expression is of the form $c_1 i_1 + \ldots + c_n i_n + c_{n+1}$ (where c_k are compile-time constants) is not enough; variables i_k need to be SCoP parameters or iterators of surrounding SCoP loops.

5.2.2 SCoP Detection and Extraction of Polyhedra

A high-level diagram of framework components and their interactions is shown in Figure 5.1. The first component, described in this subsection, takes as input the Sage ASTs for all functions in the input program. Each function-level AST is analyzed to identify subtrees that satisfy the definition of SCoP described above. In addition to the subtree, the SCoP detection also identifies SCoP parameters and iterators. Once a proper subtree is identified, it is traversed to extract its *polyhedral representation*. This representation contains a set of *statements* (each one corresponding to ⟨*Statement*⟩ from the conceptual grammar), a set of parameter names, and a set of array names. This representation is the output of the SCoP Detection / Extraction stage.

[2]This is an abstract description of the structure of the code; an actual SCoP will, of course, respect the grammar of the C language

[3]The PACE implementation handles general affine expressions in C code — e.g., `3*i - i*13 + (-5)*j + (-(-4))`. In all such expressions, variables and constants must be of C integer types [21, §6.2.5].

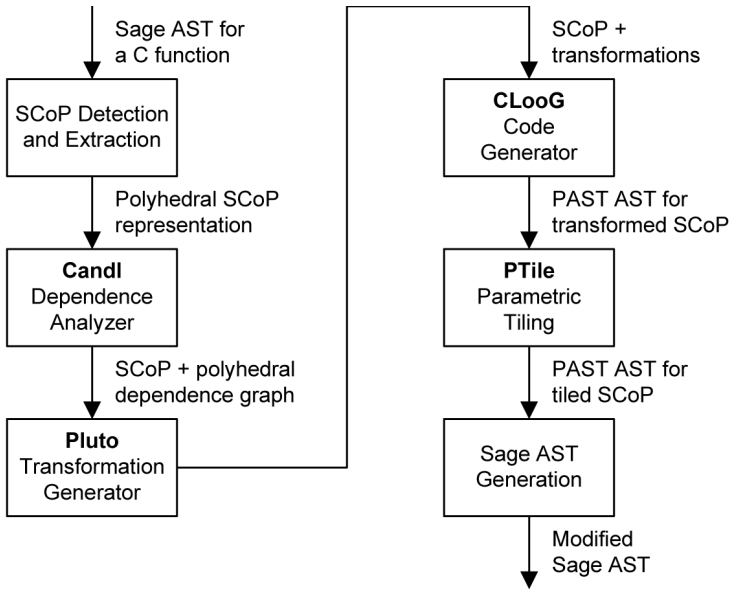


Figure 5.1: Overview of the polyhedral transformation framework.

In the polyhedral representation of a SCoP, each statement is associated with a set of iterator names, a matrix encoding the polyhedron that is the statement's iteration space, a matrix representing the array elements that are read by the statement, and a matrix representing the array elements that are written by the statement. Scalar variables are treated as a special type of array with a single element.

5.2.3 Polyhedral Dependence Analysis with Candl

A central concept of program optimization is to preserve the semantics of the original program through the optimization steps. Obviously, not all transformations, and hence not all affine schedules (i.e., orderings of statement instances), preserve the semantics for all programs. To compute a *legal* transformation, we resort to first extracting the *data dependences* expressed in a polyhedral representation [37]. This information is later used in two ways. First, the polyhedral optimizers uses data dependence information to constrain the schedules to ensure that they respect the semantics of the original program. Second, the generated Sage AST is annotated with relaxed data dependence information in the form of Dependence Distance Vectors (DDV) computed from the dependence polyhedra, such that other components of PACE can access this information. Typically AST-based transformations and vectorization passes use DDVs to retrieve which loops are permutable, parallel, etc.

The polyhedral dependence analysis stage takes as an input the polyhedral representation of a SCoP, and extends its content with data dependence information. Candl, the Chunky ANalyzer for Dependences in Loops, is an open-source tool for data dependence analysis of static control parts [22]. To capture all program dependences, Candl builds a set of dependence polyhedra, one for each pair of array references accessing the same array cell (scalars being a particular case of array), thus possibly building several dependence polyhedra per pair of statements. A point in a dependence polyhedron encodes a pair of iteration vectors for which the dependence may be occur. Given the polyhedral representation of the input program, Candl outputs the *polyhedral dependence graph*. It is a multi-graph with one node per statement, and an edge $e^{R \to S}$ labeled with a dependence polyhedron $\mathcal{D}_{R,S}$, for each dependence.

5.2.4 Pluto Transformation Generator

The polyhedral representation allows the expression of arbitrarily complex sequences of loop transformations. The downside of this expressiveness is the difficulty of selecting an efficient optimization that includes tiling together with fusion, distribution, interchange, skewing, permutation and shifting, and is still the subject of numerous recent research works [41, 61, 62, 63]. The Pluto transformation stage implements an effective approach to solving the optimization selection problem [15]. This stage takes as an input the polyhedral representation enriched with dependence information, and outputs a modified polyhedral representation where the original statement schedules have been replaced by those computed by Pluto.

Pluto is an automatic transformation selection tool that operates directly on the polyhedral representation [60]. It outputs *schedules* (combinations of transformations) to be later applied by the code generator. Pluto performs transformations for coarse-grained parallelism and locality simultaneously. The core transformation framework mainly works by finding affine transformations for efficient tiling and fusion, but is not limited to it. OPENMP-like parallel code for multicores can be automatically generated from sequential code. Outer, inner, or pipelined parallelization is achieved, besides register tiling and exposing inner parallel loops for subsequent vectorization (see Appendix B for details about code vectorization).

5.2.5 Polyhedral Code Generation with CLooG

Code generation is the final step of polyhedral optimization. This stage takes as an input the polyhedral representation of SCoP enriched with the schedules computed by Pluto, and outputs a code fragment in CLooG's internal syntactic representation, CLAST. The open-source CLooG code generator [9, 25] applies the transformations specified by the affine schedules, and generates a CLAST abstract syntax tree corresponding to the transformed code. This tree is then converted in a slightly more expressive syntax tree representation named PAST, for subsequent processing by PTile.

5.2.6 Parametric Tiling with PTile

Tiling is a crucial transformation for achieving high performance, especially with deep multi-level memory hierarchies. The tiling phase takes place inside the code generation stage, and subsequently processes the output of CLooG. It takes as an input a PAST tree being the result of the transformations embodied in the schedules, and produces a PAST tree being a parametrically tiled version of the input PAST tree. PAST supports non-affine expressions as generated by the parametric tiling algorithm [8], that the CLAST representation does not support.

Tiling is a well known technique for improving data locality and register reuse. It has received a lot of attention in the compiler community. However, the majority of work only addresses the tiling of perfectly nested loops. The few systems that can automatically generate tiled code for imperfectly nested loops require that tile sizes be compile-time constants. The PolyOpt system incorporates parametric tiling capability, where tile sizes do not have to be compile-time constants. Parametric tiled code is passed by the PAO to the RTS (as discussed in Sec. 4.3.8) to determine the best tile sizes for the target platform.

5.2.7 Translation to Sage ASTs

The final stage of PolyOpt consists in translating the PAST representation into a Sage AST, and reinserting this Sage AST in the program in place of the original Sage subtree for the SCoP. The result of the code generation of CLooG is converted from the CLooG IR to the PAST IR, which provides enough information to generate an equivalent Sage AST subtree. The resulting modified Sage AST is indistinguishable from the "normal" ASTs generated by Rose's front end, and can be used as input to other components of the PAO system.

5.3 Method

5.3.1 SCoP Detection and Extraction of Polyhedra

Given a Sage AST, a bottom-up traversal is used to identify AST subtrees that correspond to SCoPs. Since SCoPs cannot be nested, as soon as a node breaks the SCoP definition then none of its ancestor can be in a SCoP. During the traversal, when several sibling subtrees satisfy the SCoP definition, an attempt is made to create a larger SCoP encompassing these subtrees. At the end of this process, there may be several disjoint SCoP detected. Each one is independently subjected to the processing steps described in this section.

For each top-level element of the SCoP, a bottom-up traversal is performed for the Sage AST rooted at that node. During the traversal, information about upper/lower loop bounds is collected (represented as vectors that encode the affine constraints). Similarly, vectors encoding the read-/write accesses of array elements are constructed. When all children of a node have been traversed, their data is combined as necessary, based on the context of the node. When the root node of the subtree is reached, all necessary information for each statement appearing in the subtree has been collected.

5.3.2 Polyhedral Dependence Analysis with Candl

Data dependence representation Two executed statement instances are in a *dependence relation* if they access the same memory cell and at least one of these accesses is a write operation. For a program transformation to be correct, it is necessary to preserve the original execution order of such statement instances and thus to know precisely the instance pairs in the dependence relation. In the algebraic program representation described earlier, it is possible to characterize exactly the set of instances in dependence relation in a synthetic way.

Three conditions have to be satisfied to state that an instance $\vec{x}_R$ of a statement R depends on an instance $\vec{x}_S$ of a statement S. (1) They must refer to the same memory cell, which can be expressed by equating the subscript functions of a pair of references to the same array. (2) They must be actually executed, i.e. $\vec{x}_S$ and $\vec{x}_R$ have to belong to their corresponding iteration domains. (3) $\vec{x}_S$ is executed before $\vec{x}_R$ in the original program.

Each of these three conditions may be expressed using affine inequalities. As a result, exact sets of instances in dependence relation can be represented using affine inequality systems. The exact matrix construction of the affine constraints of the dependence polyhedron used in PolyOpt was formalized by Feautrier and Bastoul [37, 12].

```
for (i = 0; i <= n; i++) {
    s[i] = 0; // statement R
    for (j = 0; j <= n; j++)
        s[i] = s[i] + a[i][j] * x[j]; // statement S
 }
```

Figure 5.2: matvect *kernel*

For instance, if we consider the `matvect` kernel in Figure 5.2, dependence analysis gives two dependence relations: $\delta_{R,S}$ for instances of statement S depending on instances of statement R (e.g., R produces values used by S), and similarly, $\delta_{S,S}$.

For Figure 5.2, dependence relation $\delta_{R,S}$ does not mean that all instances of R and S are in dependence—that is, the dependence does not necessarily occur for all possible pairs of $\vec{x}_R$ and $\vec{x}_S$. Let $\vec{x}_R = (i_R)$ and $\vec{x}_S = (i_S, j_S)$. There is a dependence from R to S only when $i_R = i_S$. We can then define a *dependence polyhedron*, being a subset of the Cartesian product of the iteration

domains, containing all values of i_R, i_S and j_S for which the dependence exists. We can write this polyhedron in matrix representation: the first line represents the equality $i_R = i_S$, the next two encode the constraint that vector (i_R) must belong to the iteration domain of R and similarly, the last four state that vector (i_S, j_S) belongs to the iteration domain of S:

$$\mathcal{D}_{R,S} : \left[\begin{array}{rrrrr} 1 & -1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \end{array}\right] \cdot \begin{pmatrix} i_R \\ i_S \\ j_S \\ n \\ 1 \end{pmatrix} \begin{array}{l} = 0 \\ \hline \geq \vec{0} \end{array}$$

To capture all program dependences we build a set of dependence polyhedra, one for each pair of array references accessing the same array cell (scalars being a particular case of array), thus possibly building several dependence polyhedra per pair of statements. The *polyhedral dependence graph* is a multi-graph with one node per statement. An edge $e^{R \to S}$ is labeled with a dependence polyhedron $\mathcal{D}_{R,S}$, for all dependence polyhedra.

A dependence polyhedron is the most refined granularity to represent a dependence. However, for the cases where this precision is not needed it is easy to rebuild a more abstract and less detailed dependence information from the polyhedral dependence graph. For instance, one can generate a simple graph of dependent memory references, or rebuild the dependence distance vectors by extracting some properties of the dependence polyhedra.

Dependence analysis in Candl The `Candl` software was written by Bastoul and Pouchet. It implements the construction of the complete polyhedral dependence graph of a given static control part. The algorithm to compute all polyhedral dependences simply constructs the dependence polyhedron for each pairs of references to the same array, for all program statements. The polyhedron is then checked for emptiness. If it is empty then there is no dependence between the two considered references. Otherwise there is a (possibly self) dependence between the two references.

5.3.3 Pluto Transformation Generator

The *tiling hyperplane method* [14, 15] is a model-driven technique that seeks to optimize a SCoP through transformations encompassing complex compositions of multi-dimensional tiling, fusion, skewing, interchange, shifting, and peeling.

Representing optimizations A transformation in the polyhedral framework captures in a single step what may typically correspond to a sequence of numerous textbook loop transformations. It takes the form of a carefully crafted affine multidimensional schedule, together with iteration domain and/or array subscript transformations.

In the tiling hyperplane method, a given loop nest optimization is defined by a multidimensional affine schedule. Given a statement S, we use an affine form on the surrounding loop iterators $\vec{x}_S$. It can be written as

$$\Phi^S(\vec{x}_S) = C^S \begin{pmatrix} \vec{x_S} \\ 1 \end{pmatrix}$$

where C^S is a matrix of non-negative integer constants. The instance of S defined by iteration vector $\vec{x}_S$ is scheduled at multidimensional date $\Phi^S(\vec{x}_S)$. Multidimensional dates can be seen as logical clocks: the first dimension corresponds to days (most significant), next one is hours (less significant), the third to minutes, and so on. Note that every static control program has a multidimensional affine schedule [38], and that any loop transformation can be represented in the polyhedral representation [78].

Let ϕ_i^S be the i^{th} row of C_S. A row is an *affine hyperplane* on the iteration domain of S. For S with m_S surrounding loop iterators, let

$$\phi_i^S = [\, c_1^S \; c_2^S \; \ldots \; c_{m_S}^S \; c_0^S \,]$$

Here c_i^S are integer constants; c_0^S is the coefficient attached to the scalar dimension.

The tiling hyperplane method Intuitively, the tiling hyperplane method computes an affine multidimensional schedule [38] for the SCoP such that parallel loops are at the outer levels, and loops with dependences are pushed inside [14, 15], and at the same time, maximizing the number of dimensions that can be tiled. The method proceeds by computing the schedule level by level, from the outermost to the innermost. Specifically, affine hyperplanes with special properties are computed, one for each row of the scheduling matrix. Such specific hyperplanes are called tiling hyperplanes.

Computing valid tiling hyperplanes Let $\mathbf{S} = \{S_1, S_2, \ldots, S_n\}$ be the set of statements of the SCoP. Let $G = (V, E)$ be the data dependence graph for the original SCoP. G is a multi-graph with $V = \mathbf{S}$ and E being the set of data dependence edges. Notation $e^{S_i \rightarrow S_j} \in E$ denotes an edge from S_i to S_j, but we will often drop the superscript on e for readability. For every edge $e \in E$ from S_i to S_j, $\mathcal{D}_{S_i,S_j}$ is the corresponding dependence polyhedron.

Tiling along a set of dimensions is legal if it is legal to proceed in fixed block sizes along those dimensions. This requires dependences to not be backward along those dimensions [46, 65, 14]. $\{\phi_{S_1}, \phi_{S_2}, \ldots, \phi_{S_k}\}$ is a legal (statement-wise) tiling hyperplane if and only if the following holds true for each dependence edge $e^{S_i \rightarrow S_j} \in E$:

$$\phi_{S_j}\left(\vec{x}_{S_j}\right) - \phi_{S_i}\left(\vec{x}_{S_i}\right) \quad \geq \quad 0, \quad \left\langle \vec{x}_{S_i}, \vec{x}_{S_j} \right\rangle \in \mathcal{D}_{S_i,S_j} \tag{5.1}$$

Cost model for computing the tiling hyperplanes There are infinitely many hyperplanes that may satisfy the above criterion. An approach that has proved to be simple, practical, and powerful has been to find those directions that have the shortest dependence components along them [14]. For polyhedral code, the distance between dependent iterations can always be bounded by an affine function of the SCoP parameters (represented as a p-dimensional vector $\vec{n}$).

$$\begin{aligned} \forall \quad & \left\langle \vec{x}_{S_i}, \vec{x}_{S_j} \right\rangle \in \mathcal{D}_{S_i,S_j}, \\ & \delta_e\left(\vec{x}_{S_i}, \vec{x}_{S_j}\right) = \phi_{S_j}\left(\vec{x}_{S_j}\right) - \phi_{S_i}\left(\vec{x}_{S_i}\right) \\ \forall \quad & \left\langle \vec{x}_{S_i}, \vec{x}_{S_j} \right\rangle \in \mathcal{D}_{S_i,S_j}, \forall e \in E, \vec{u} \in \mathbb{N}^p, w \in \mathbb{N}, \\ & \mathbf{u}.\vec{n} + w - \delta_e\left(\vec{x}_{S_i}, \vec{x}_{S_j}\right) \geq 0 \end{aligned} \tag{5.2}$$

The legality and bounding function constraints from (5.1) and (5.2) respectively are cast into a formulation involving only the coefficients of ϕ's and those of the bounding function by application of the Farkas Lemma [38]. Coordinates of the bounding function are then used as the minimization objective to obtain the unknown ϕ's coefficients.

$$\text{minimize}_{\prec} \; (\mathbf{u}, w, \ldots, c_i, \ldots) \tag{5.3}$$

This cost function is geared towards maximal fusion. This allows to minimize communication and maximize locality on the given set of statements. The resulting transformation is a complex composition of multidimensional loop fusion, distribution, interchange, skewing, shifting and peeling. Finally, multidimensional tiling can be applied on all permutable bands.

Enabling vectorization Due to the nature of the optimization algorithm, even within a local tile (L1) that is executed sequentially, the intra-tile loops that are actually parallel do not end up being

outer in the tile: this goes against vectorization of the transformed source for which we rely on the native compiler. Also, the polyhedral tiled code is often complex for a compiler to further analyze and say, permute and vectorize. Hence, as part of a post-process in the transformation framework, a parallel loop is moved innermost within a tile, and annotations are used to mark the loop as vectorizable (see Appendix B for details about code vectorization). Similar reordering is possible to improve spatial locality that is not considered by our cost function due to the latter being fully dependence-driven. Note that the tile shapes or the schedule in the tile space is not altered by such post-processing.

5.3.4 Polyhedral Code Generation with CLooG

The code generation stage generates a *scanning code* of the iteration domains of each statement with the lexicographic order imposed by the schedule. Statement instances that share the same date are typically executed under the same loop, resulting in loop fusion. Scanning code is typically an intermediate, AST-based representation that can be translated to an imperative language such as C or FORTRAN.

For many years this stage has been considered to be one one of the major bottlenecks of polyhedral optimization, due to the lack of scalability of the code generation algorithms. Eventually the problem was addressed by the work of Bastoul [9, 10] who proposed an extended version of Quilleré's algorithm [64] that significantly outperformed previously implemented techniques such as the ones by Kelly et al. in the Omega framework [48] or by Griebl in the Loopo framework [42]. The only constraints imposed by the CLooG code generator are (1) to represent iteration domains with a union of polyhedra, and (2) to represent scheduling functions as affine forms of the iteration domain dimensions. This general setting removes previous limitations such as schedule invertibility [5].

Code generation time is a function of the number of statement domains and the depth of the loop structure to be generated. Polyhedral tiling can be performed directly with CLooG when using constant (i.e., scalar) tile sizes, by extending the dimensionality of the iteration domains. This approach significantly increases code generation time, because of the higher number of domain dimensions. In PolyOpt this problem is avoided by the parametric tiling approach: the domain dimension is not extended before using CLooG, but instead after the polyhedral code generation. Preliminary results demonstrating the strong scalability gain for code generation time can be found in [44].

The CLooG code generator is unanimously considered the state-of-the-art polyhedral code generator, as it implements the latest and most scalable algorithm for code generation [9]. Given the polyhedral representation of the SCoP together with the schedules computed by Pluto, it outputs a CLooG AST in an internal representation referred to as CLAST. This representation is then translated into an extended version of CLAST called PAST, that supports non-affine loop bounds (as parametric tiling can generate such non-affine bounds). The constructed PAST AST is then translated back into a Sage AST.

5.3.5 Translation to Sage ASTs

The translation of a PAST AST to a Sage AST is based on a bottom-up traversal of the PAST IR, which involves

- re-mapping of control structures and expressions
- mapping back to symbols that existed in the original program (e.g., names of arrays and parameters)
- introduction of new symbols in symbol tables (e.g., new iterators)

- rewriting of array index expressions and loop bounds in terms of the new iterators

5.3.6 Parametric Tiling with PTile

Tiling is a key transformation in optimizing for parallelism and data locality. Tiling for locality involves grouping points in an iteration space into smaller blocks (tiles) allowing reuse in multiple directions when the block fits in a faster level of the memory hierarchy (registers, L1, or L2 cache). Tiling for coarse-grained parallelism partitions the iteration space into tiles that may be executed concurrently on different processors with a reduced frequency and volume of inter-processor communication: a tile is atomically executed on a processor with communication required only before and after execution. The first effective approach for tiling of imperfectly nested affine loops was developed in the Pluto polyhedral transformation framework [15]. However, Pluto can only generate tiled code where the tile sizes are fixed at compile-time. In the PACE infrastructure, tile sizes can be adapted at run-time, as a function of the machine parameters and the problem size to be used. It is unpractical to re-compile the program for each of its execution, and it is highly desirable to be able to specify the tile sizes as run-time parameters in the code instead of compile-time parameters. This is *parametric tiling* code generation [44, 8].

Overview of the method In the case of a program with single statement, the loop structure is a perfect loop nest. Generating aligned tiled code involves syntactic processing of the loop bounds in addition to generating the tile loops. The tile loops are generated as perfectly nested loops that enumerate the tiles as tile numbers in the tile space. Figure 5.3 illustrates an example for generating aligned tiled code for a single statement program.

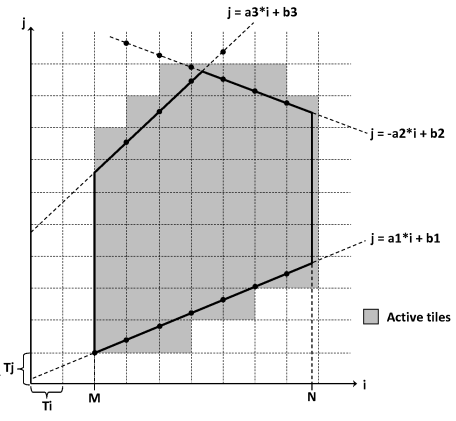
In the case of a program with multiple statements, the loop structure is an imperfectly nested loop. Generating aligned tiled code in this case involves additional processing to generate perfectly nested tile loops. The convex hull of the union of the domains of all statements is found and used to generate the loop structure of the tile loops [8].

```
for (i=M;i<=N;i++)
 for (j=b1+a1*i;j<=min(b2-a2*i,b3+a3*i);j++)
   S(i,j);
```

(a) Original loop structure

```
/* Intertile loops it,jt */
for (it=floor(M/Ti);it<=floor(N/Ti);it++)
 for (jt=floor((a1*(it*Ti)+b1)/Tj);
      jt<=floor((min(b2-a2*(it*Ti),
                     a3*(it*Ti+Ti-1)+b3))/Tj);
      jt++)
  /* Intratile loops i,j */
  for (i=max(M,it*Ti);i<=min(N,it*Ti+Ti-1);i++
   for (j=max(a1*i+b1,jt*Tj);
        j<=min(min(b2-a2*i,b3+a3*i),jt*Tj+Tj-1);
        j++)
     S(i,j);
```

(b) Tiled loop structure



(c) Tiled iteration space (shown with region of active tiles)

Figure 5.3: Parametric tiling of a single statement domain in PTile

Generation of parallel tiled code When the tile sizes are parametric, it is problematic to generate parallel code using the polyhedral framework since nonlinear expressions arise in the specification of constraints and objective functions. Hence for an arbitrary parametric tiled code, it is non-trivial to extract parallelism. PolyOpt implements the method from Baskaran et al. [8] to address this problem.

After tiling as depicted above, if any of the tiling loops is parallel (i.e. has no loop carried dependences), coarse-grained parallel tiled execution is directly possible. However, even if none of the tiling loops is parallel, wavefront parallelism is always feasible among the tiles. But instead of viewing wavefront-parallel tile execution as involving a unimodular transformation from one n-dimensional space (nesting order $t_1, t_2, \ldots, t_n$ of sequential tiled execution) to another n-dimensional space (nesting order $w, t_1, t_2, \ldots, t_{n-1}$), it is viewed in terms of a sparse $n+1$ dimensional space with nesting order $w, t_1, t_2, \ldots, t_n$. While this might seem very wasteful, by optimizing the scanning of this higher dimensional space, parameterized parallel tiled execution is achieved with negligible overhead of scanning empty tiles. The primary problem of generating loop bounds for the outermost w loop via Symbolic Fourier Motzkin elimination is eliminated by generating the lowest and highest numbered wavefronts in the untiled form of the loops and then generating bounds for the lowest and highest numbered tiled wavefront loop. No explicit "skewing" of the tile space is done; the $t_1, t_2, \ldots, t_n$ loops are executed in original lexicographic order but constrained to include only those tiles that actually belong in the current tile wavefront w. The $n+1$ dimensional loop nest $w, t_1, t_2, \ldots, t_n$ is optimized by addition of constraints derived from the wavefront inequalities.

Chapter 6

AST-based Transformations in the Platform-Aware Optimizer

6.1 Introduction and Motivation

This chapter summarizes the design of AST-based transformations in the Platform-Aware Optimizer (PAO). As discussed in § 4, each of these transformations could be followed by incremental re-analysis. These transformations complement the polyhedral transformation framework described in § 5, both by performing transformations on regions of code that are ineligible for polyhedral transformation (non-SCoP regions) and by performing transformations that are not included in the polyhedral framework (such as data transformations, idiom recognition, scalar replacement, and loop-invariant redundancy elimination). AST-based transformations contribute to the overall goal of the PAO to automate selection of an appropriate set of high level transformations for a given platform as viewed through the lens of the platform-specific resource characteristics derived by the PACE RC tools.

Most of the non-polyhedral transformations in the PAO are extensions of classical high-level loop and data transformations introduced in previous work [79, 2, 68]. However, there has been relatively little attention paid in past work to the question of which transformations should be automatically selected for optimizing performance, especially for the wide set of transformations available in the PAO's arsenal. Automatic selection of transformations is a critical issue for the PACE Compiler because the developers of the PACE Compiler do not know a priori the performance characteristics of the target platform. In PACE, the compiler must adapt to new platforms using measured values for performance-critical system characteristics, where a vendor compiler can be carefully tailored to match a known processor release.

An important aspect of high level transformations in the PAO that distinguishes them from many lower-level transformations in the TAO is that most high level transformations are reversible and, if improperly selected, can degrade performance just as effectively as they improve performance. For example, loop interchange can improve the cache locality of a loop nest with a poor loop ordering, but it can also degrade the performance of a well-tuned loop nest. In contrast, while the performance improvement obtained by traditional lower-level optimizations (e.g., operator strength reduction) can vary depending on the source program and target platform, such optimizations typically do not significantly degrade performance.

Our overall approach to address this issue is to leverage the separation of concerns mentioned in § 4 among Legality Analysis, Profitability Analysis, and IR Transformation, and to use a a quantitative approach to profitability analysis. The problem of selecting high level transformations is decomposed into different optimization problems that address the utilization of different classes

Principal Contacts For This Chapter: Vivek Sarkar, vsarkar@rice.edu

of hardware resources (e.g., memory hierarchy, inter-core parallelism, intra-core parallelism). The formulations of the optimization problems are based on quantitative cost models, which are built on measured characteristics of the target system and application characteristics that include measured 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).

6.2 Functionality

As described in § 4.2, the PAO takes as input *refactored program units* (RPUs), and generates as output transformed versions of each RPU using a combination of polyhedral and AST-based transformations.

6.2.1 Input

The primary input for a single invocation of the AST-based transformer is the HIR (SAGE III IR) for an RPU. Additional inputs (as shown in Figure 4.1) include compiler directives from the optimization plan, resource characteristics for the target platform, profile information with calling-context-based profile information for the source application, and TAO cost analysis feedback (Path 3 in Figure 4.2).

6.2.2 Output

As its output, the AST-based transformer produces a transformed HIR for the input RPU. The transformed code can be translated into either C source code or into the IR used in the Target-Aware Optimizer (TAO). This latter case uses the PAO→TAO IR translator, described in § 7; the translator is also used in the PAO–TAO query mechanism, as shown in Figure 4.2.

6.3 Method

The proposed structure of AST-based transformations in the PAO for a single function is as follows. Though not listed explicitly, the incremental program reanalysis described in § 6.3.7 is assumed to be performed after each transformation listed below. The transformations described below could be performed on all functions within the RPU, starting with entry functions (functions called from other RPU's), and transitively traversing the call graph within the RPU. Transformations that have already been implemented will be explicitly noted as such.

1. Perform *function inlining and path duplication* within an RPU. This step is driven by context-sensitive and path-sensitive execution profiles obtained by the PACE Runtime System.

2. Perform *canonical program analyses*. As indicated in § 4, these analyses include Global Value Numbering, Constant Propagation, and Induction Variable Analysis. This analysis information could be updated incrementally, whenever a transformation is performed by a later step.

3. Perform *preprocessing transformations*. The purpose of this step is to increase opportunities for subsequent polyhedral and non-polyhedral transformations. It could start with a clean-up phase that includes Unreachable Code Elimination, Dead Code Elimination, and to separate SCoP-compliant and non-SCoP-compliant statements into separate loop nests as far as possible. (SCoP stands for "Static Control Part", and represents a loop nest that is amenable to polyhedral transformations. See § 5.2.1.) It could also attempt to maximize the number of enclosing perfectly nested loops for each statement.

4. Identify SCoPs in each function, and invoke the PolyOpt component separately for each SCoP. This identification and invocation have been implemented. As described in § 5, the PolyOpt component has already been implemented and 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 TAO for vectorization described in Appendix B.

5. Perform the following steps for each maximal non-SCoP loop nest in the IR [1]

   (a) Pattern-driven Idiom Recognition — if the loop nest is found to match a known library kernel (or can be adapted to make the match), then replace the loop nest with the appropriate library call. This transformation can be applied even to the SCoP-compliant loop nests, and it may even lead to a better code than polyhedral transformations. We could evaluate both alternatives. More details are given in § 6.3.1.

   (b) Loop Privatization — create private per-iteration copies of scalar and array variables, when legal to do so.

   (c) Locality optimization — use the measured characteristics of the target machine's memory hierarchy (from the PACE RC tools) to select a set of interchange, tiling and fusion transformations to optimize locality (with support from other iteration-reordering loop transformations as needed, such as loop reversal and loop skewing). Cost-driven parametric tiling and fusion transformations have already been implemented for this step, and are applicable to non-SCoP loops.

   (d) Parallelization of outermost loop nests — if the loop nest does not already have explicit OPENMP parallelism, use OPENMP to automatically create parallel loops at the outermost level, with loop coalescing for efficiency.

   (e) Unrolling of innermost loop nests — use iterative feedback from the TAO (guided by measured processor characteristics) to select unroll factors for each innermost loop nest. This transformation can be applied even to the loops produced by the PolyOpt component. This technique has been implemented and details are provided in § 6.3.5.

   (f) Scalar replacement — perform loop-carried and loop-independent scalar replacement of array and pointer accesses within the loop nest. More details are provided in § 6.3.6

   (g) Commit all transformations and perform incremental reanalysis.

6. Return the updated SAGE III IR to the compiler driver so that it can invoke the later steps of compilation, including the PAO→TAO IR translator. This has (of necessity) been implemented.

6.3.1 Pattern-driven Idiom Recognition

In some cases, a computational kernel in the input application may be implemented by a platform-specific library such as BLAS call. If so, it is usually beneficial to replace the user-written kernel by call to the platform-specific library. However, in addition to recognizing opportunities for this transformation, it is important to factor in the cost of *adaptation* (e.g., additional copies).

For example, consider the code fragment in Figure 6.1. On one platform, the PAO might select a combination of tiling, interchange, unrolling, and scalar replacement as usual. Tile sizes are initialized using analytical cost model and updated by runtime, while the unroll factors are proposed by cost model and refined by feedback from TAO.

[1] Transformations in the list will be applied to the non-SCoP loop nest. The PolyOpt framework applies some of these same transformations, such as loop tiling, to the SCoP loop nests.

```
for(i = 0; i < n; i++){
   for (j = 0; j < n; j++){
      a[i,j] = 0;
      for (k = 0; k < n; k++){
         a[i,j] = a[i,j] + b[j,k] * c[k,i];
      }
   }
}
```

Figure 6.1: Matrix multiplication and transpose

However, on a different platform, the PAO might recognize that the computation above can be implemented with two library calls (matrix multiply and transpose) that are available in optimized form on that platform. The PAO could still explore transformations as in the previous case (using system characterization values for this particular platform), but it may conclude that the cost of using library routines will be lower than the compiler-optimized version for values of n greater than some threshold.

6.3.2 AST-based Loop Tiling

Loop tiling is a critical optimization for effectively using the memory hierarchy on the target machine. As described in Section 6.3, the AST-based transformation framework in PAO could apply parametric loop tiling to the non-SCoP loop nests that is not eligible for the PolyOpt framework. The AST-based tiling is applied to the loop nests that satisfy the following conditions.

- The loops are innermost perfectly nested.
- There is no conditional branch to jump outside the loop body.
- Loop indexes have invariant iteration space.
- Data dependence vectors don't contain negative elements (the loop nest is fully permutable)
- Each loop index carries temporal and/or spatial locality on the accessed arrays.

Conditions 1 to 4 are evaluated by classical compiler analyses. As described in Section 4.3.4, the AST-based transformation framework employs the DL model to compute the memory cost and estimate data locality carried by each loop index (condition 5).

```
for(i2 = 0; i2 < n; i2 += B){
   for(j2 = 0; j2 < n; j2 += B){
      for(k = 0; k < n; k++){
         for(i1 = i2; i1 < min(i2 + B - 1, n); i1++){
            for(j1 = j2; j2 < min(j2 + B - 1, n); j1++){
               a[i1,j1] = a[i1,j1] + b[j1,k] * c[k,i1];
            }
         }
      }
   }
}
```

Figure 6.2: Matrix multiplication and transpose, tiled with a BxB tile size

Figure 6.2 shows the example from Figure 6.1, tiled with a BxB tile size across the i and j dimensions, as might be done to prepare for online tuning by the RTS. Note that the initialization statement is split as a different loop nest so as to enable loop tiling.

6.3.3 Selection of Tile Size

In order to maximize cache usage, PAO will have to select the right combination of the tile size, unroll factor, and loops to interchange. Tile size will naturally depend on the measured values for cache size and associativity of the target platform from the PACE RC tools. PAO will employ two analytical models to optimize tile size of parameterized tiling loop nests generated by both PolyOpt and AST-based frameworks, 1) *DL model*: an existing conservative model, based on the data footprint of a tile, which ignores intra-tile cache block replacement, and 2) *ML model*: an aggressive new model that assumes optimal cache block replacement within a tile. These two models are used to determine the initial tile size and theoretical lower and upper bounds for the optimal tile size. The initial tile size could, in turn, be tuned at runtime using the online feedback-directed parameter selection facility of the PACE RTS (§ 9.3.4), where the search space of runtime tile size tuning is bounded by the above theoretical boundaries. The code for the parameterized version of the loop could be packaged for runtime tuning using the approach described in § 4.3.8.

6.3.3.1 DL Model

The DL (Distinct Lines) model was designed to estimate the number of distinct cache lines accessed in a loop-nest [39, 68]. Consider a reference to an m-dimensional array variable (called A, say), enclosed in n perfectly nested loops with index variables $i_1, \cdots, i_n$:
$A[f_m(i_1, \cdots, i_n)] \cdots [f_1(i_1, \cdots, i_n)]$ where $f_j(i_1, \cdots, i_n)$ is an affine function. An exact analysis to compute DL is only performed for array references in which all coefficients are compile-time constants (affine). An upper bound for the number of distinct lines accessed by a single array reference [39] with one-dimensional subscript expression $f(i_1, \cdots, i_n)$ is
$DL(f) \leq \min\left(\frac{(f^{hi}-f^{lo})}{g}+1, \left\lceil\frac{(f^{hi}-f^{lo})}{L}\right\rceil+1\right)$,
where g is the greatest common divisor of the coefficients of the enclosing loop indices in f, and L is the cache line size in units of array element size. f^{hi} and f^{lo} are the maximum and minimum values taken by subscript expression f across the entire loop nest. For the special case when $L = 1$, $DL(f) = (f^{hi} - f^{lo})/g + 1$ becomes an estimate of the number of distinct accesses made by the array reference. In practice, the relative error of this estimation is small when, as is usually the case, the size of the $(f^{hi} - f^{lo})$ range is much larger than the size of the individual coefficient of f. For a multidimensional array reference, $A\,(f_1, \cdots, f_m)$, the upper bound estimate given in [39] is as follows:
$DL(f_1, \cdots, f_m) = DL(f_1) \times \prod_{j=2}^{m}\left(\frac{(f_j^{hi}-f_j^{lo})}{g_j}+1\right)$.
It was also shown in past work how this model can be extended to account for multiple array accesses in a loop nest [39, 68].

This bound provides a reasonable estimate when the stride of second dimension is larger than L. These DL definitions for an entire loop nest are also applicable to a tile, whose loop boundaries are expressed with tile sizes. In such a case, the DL definition is a symbolic function of tile sizes $t_1, \cdots, t_n$ denoted by $DL(t_1, \cdots, t_n)$ [68].

The DL definition is also applicable to any level of cache or TLB by selecting its cache line size or page size as L. Unfortunately, the DL model ignores possible replacement of cache lines and therefore provides conservative over-estimation for the number of cache lines needed.

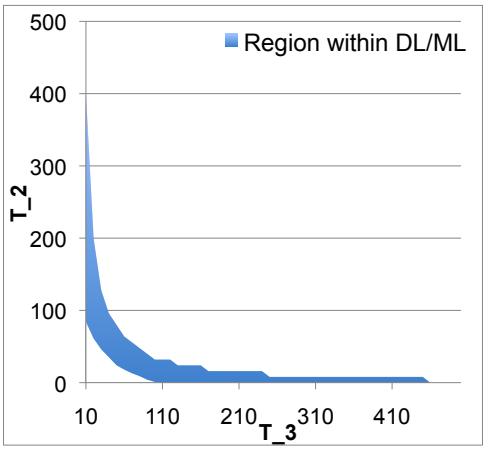


Figure 6.3: Search Space for Matrix Multiplication for $T_1 = 30$

6.3.3.2 ML Model

ML (Minimum working set Lines), which is based on the cache capacity required for a tile when intra-tile reuse patterns are taken into account, is a new analytical cost model introduced in the PACE project. The essential idea behind the ML model is to develop an estimate of the minimal cache capacity needed to execute a tile without incurring any capacity misses, if the pattern of intra-tile data reuse is optimally exploited as described below. Consider a memory access trace of the execution of a single tile, run through an idealized simulation of a fully associative cache. The cache is idealized in that it has unbounded size and an optimal replacement policy where a line in cache is marked for replacement as soon as the last reference to data on that line has been issued (through an oracle that can scan the tile's entire future references). Before each memory access, the simulator fetches the desired line into the idealized cache if needed. After each memory access, the simulator evicts the cache line if it is the last access (according to the oracle). ML corresponds to the maximum number of lines (high water mark) held in this idealized cache during execution of the entire trace (tile).

The PAO will compute ML for a tile by first constructing a special sub-tile based on analysis of reuse characteristics and then computing the DL value for that sub-tile. Although we mainly discuss cache capacity in this section, the idea and definition is directly applicable to TLBs by replacing the cache line size by the page size.

6.3.3.3 Bounding Search Space and Selecting Initial Tile Size

DL and ML models give the theoretical lower and upper bounds for the tile size search space. ML is used for optimistic cache and TLB capacity constraints for intra-tile data reuse and gives the upper boundaries for estimated tile sizes. In contrast, DL is used for conservative constraints, and gives the lower boundaries. These lower and upper boundaries drastically reduce the search space. Furthermore, DL model gives the average memory cost for given tile sizes, and a tile size with minimum memory cost is selected as the initial tile size for the runtime auto-tuning in the PACE RTS.

In PAO, ML and DL are represented as the functions of tile sizes T_1, T_2, ..., T_n for n-th nested tiling loops. CS_1 represents the number of cache lines or TLB entries at level-1 cache or TLB memory. All tile sizes within the lower boundaries due to DL and upper boundaries due to ML satisfy the following constraints.

$DL(T_1, T_2, \cdots, T_n) \geq CS_1$

```
for(i2 = 0; i2 < n; i2 += B){
   for(j2 = 0; j2 < n; j2 += B){
      for(k = 0; k < n; k++){
         for(j1 = j2; j1 < min(j2 + B - 1, n); j1++){
            for(i1 = i2; i1 < min(i2 + B - 1, n); i2++){
               a[i1,j1] = a[i1,j1] + b[j1,k] * c[k,i1];
            }
         }
      }
   }
}
```

Figure 6.4: Matrix multiplication and transpose, tiled with a BxB tile size, with i1 and j1 loops interchanged

$ML(T_2, T_3, \cdots, T_n) \leq CS_1$

We have two bounded regions according to cache and TLB. In our approach, we consider the union of both regions as candidates for optimal tile sizes, and give higher search priority to the intersection of both regions.

For example, DL and ML boundary constraints for a single-level tiling example of Matrix Multiplication is calculated as follows. We assume the experimental platform has 32 Kb L1 cache with 64 Byte line size (total 512 lines), and program size *N* = 3000 with array element size 8 Bytes.

$DL = T_1 \left\lceil \frac{T_2}{8} \right\rceil + T_1 \left\lceil \frac{T_3}{8} \right\rceil + T_3 \left\lceil \frac{T_2}{8} \right\rceil \geq 512$

$ML = 1 + \left\lceil \frac{T_3}{8} \right\rceil + T_3 \left\lceil \frac{T_2}{8} \right\rceil \leq 512$

Figure 6.3 shows the bounded search space for (T_2, T_3) when T_1 is 30. These regions bounded by DL/ML constraints are much smaller than the original 2-D search space 3000^2.

6.3.4 Loop Interchange

Loop interchange is another important compiler transformation that can significantly improve the performance through improving locality and increasing the effect of loop tiling described above.

For example, in the tiled matrix multiplication and transpose example on Figure 6.2, the elements of a tile are accessed in a row-major order, while the arrays are stored in column-major order in Fortran. If the whole tile fits in cache and the arrays are lined up properly to avoid conflict misses, then the code on figure 6.2 should perform equally well regardless of the order of the `i1` and `j1` loops. If not, the performance can be improved by interchanging the two inner loops, as shown on Figure 6.4.

6.3.5 Unrolling of Nested Loops

Loop unrolling can significantly improve code performance by reducing the loop iteration overhead and increasing the size of the loop body, making further optimizations of the loop body more effective. Loop unroll-and-jam can improve the efficiency of pipelined functional unit. However, excessive loop unrolling can create additional register pressure, which can have detrimental effect on performance if the register allocator is forced to spill some values to memory.

In PACE compilation framework, a cost driven mechanism is employed to perform multi-level loop unroll-and-jam with the most optimal unroll-and-jam factors identified by compile time cost estimation module. The definition of multi-level loop unroll-and-jam is: given a n-level loop nest[2], apply loop unrolling for the innermost loop (level 1) and apply $n-1$ times loop unroll-and-jam

[2]The level of innermost loop is marked as 1, and outermost loop is marked as n.

from the level 2 loop to outermost loop (level n) respectively. The unroll-and-jam factor is a tuple: (f_0, f_1, ... f_n), each element is the unroll factor for the corresponding loop level, e.g. f_i is the unroll factor for the i level loop in the given loop nest.

6.3.5.1 Cost Driven Loop Unroll-and-Jam

The basic workflow of cost driven loop unroll-and-jam contains such steps:

1. Identify the loop nest that can be transformed by unroll-and-jam;
2. setup the search space;
3. for each configuration in the search space:
   (a) perform unroll-and-jam on the target loop nest and generate synthetic function;
   (b) perform TAO query to get the estimated cost of the synthetic function;
   (c) check if current synthetic function is the optimal one that has minimal cost.
4. return the optimal synthetic function.

Step 1 is the legality check. As it may not be able to apply unroll-and-jam among the whole loop nest regarding data dependency, the legality check should identify the highest loop level for applying unroll-and-jam. The loop level here is the level of loop nest starts from the innermost loop. The legality of unroll-and-jam is determined by the data dependency, so the selection of loop level is based on checking data dependence vectors (DDV). In PACE compiler, the PolyOpt module is used to perform the dependence vector analysis and produce DDV set for a given loop nest.

Before analyzing the DDV, there are 3 constraints for the target loop nest.

1. The loop nest is canonical for-loop nest, if not, only the innermost loop can be unrolled;
2. The loop body does not contain `continue` statement, if not, this loop nest can not be unrolled;
3. The loop body does not contain `break` and `return` statement, if not, only the innermost loop can be unrolled;

The definition of canonical loop follows such rules: 1) the loop nest should be perfect loop nest; 2) the stride values of the loop nest must be either constant value or invariance of the loop nest; 3) the iterator changing operator must be either increment or decrement; 4) the iteration boundary must be either constant value or invariance of the loop nest; 5) boundary comparison operator must fall in there 4 types: `greater`, `greater equal`, `less` or `less equal`.

If these 3 constraints are satisfied, next step is to go through DDVs and identify the highest loop level L that can be applied with unroll-and-jam. By applying data dependence analysis on a given loop nest, we get a list of DDVs: ddv_0, ddv_1, .. ddv_n. For each element ddv_i, get the highest loop level l_i base on the constraint that the dependence distance of all of the loops whose loop level is less equal than l_i is not $>$ (i.e.positive value). After the list l_0, l_1, ... l_n was built, L is identified by $min(l_0, l_1, ... l_n)$.

The step 2 is to build the search space for identifying the optimal unroll-and-jam factor regarding the TAO cost estimation. Given a n level loop nest that is selected for unroll-and-jam, the number of possible configuration (i.e. unroll-and-jam factor) is $(B_{upper} - B_{lower})^n$. B_{lower} is the lower bound of unrolling factor for each loop level and B_{upper} is the upper bound [3].

[3]The B_{upper} and B_{lower} are configurable parameters that are retrieved from PACE compiler input, and both of them should be larger than 0.

```
for (i = 0; i < 100; i ++)
    for (j = 0; j < 100; j ++)
      for (k = 0; k < 100; k ++) {
            A[i][j][k] = A[i+2][j][k] + 0.1;
      }
```

Figure 6.5: Loop Nest example for Unroll-and-Jam

The last step is applying multi-level loop unroll-and-jam for each unroll-and-jam factor and selecting the optimal solution based on TAO cost estimation. The mechanism of multi-level loop unroll-and-jam has been mentioned in previous paragraphs. Here gives an example, Figure 6.5 is the input loop nest, the dependence distance is $(>, =, =)$ and the level 1 and 2 can be applied with unroll-and-jam. Figure 6.6 gives the transformed code by applying factor $(1, 2, 2)$.

To query the cost value from TAO cost estimation module, the PAO generate synthetic function for each unroll-and-jam factors and pass through the PAO-TAO query interface (described in § 4.3.5). TAO's answers to PAO's queries (§ 8.3.5) will be then analyzed by PAO to select the best unroll-and-jam factor that produces the minimal cost. In current implementation, the CODESIZE is used as the major metric for evaluation.

6.3.5.2 Pruning the Search Space

To reduce compile-time overheads, the PAO could prune this search space using an analytical cost model to compute the infeasible unroll configurations based on the register pressure of the unrolled loop and the measured number of registers available on the target platform. PAO could only evaluate the feasible unroll configurations.

Figure 6.7 shows an example of a search space for unroll configurations for the middle and outermost loops in a triple nested loop from Figure 6.1, on a hypothetical platform with 16 registers. Instead of searching the whole space of 380 unroll configurations, PAO could only evaluate 44 feasible unroll configurations.

6.3.6 Scalar Replacement

Scalar replacement is another classical optimization that could have a large impact on the performance of the code generated by the PAO. Scalar replacement reduces memory access, by rewriting the code so that the compiler can store a reused value in a register instead of in memory. Unfortunately, this rewrite can both increase register pressure and reduce available parallelism. Thus, the PAO will need to strike the right balance between the potential for improvement and the potential for degradation by choosing carefully those array elements to be rewritten as scalar variables. This choice must work well within the tile size and unroll factors that the PAO has selected for the loop, as discussed earlier.

Figure 6.8 shows the code from Figure 6.1 where the array element `a[i,j]` has been replaced with a scalar sum.

6.3.7 Incremental Reanalysis

Incremental reanalysis in the PAO could be supported by maintaining auxiliary structures for program regions. The partitioning of the input program into regions can be tailored to optimizations of interest. A common partitioning is to place each loop in a separate region (as in the Loop Structure Tree [68]) but other partitions are possible.

```
  for (i = 0; i < 100; i++) {
    int _j_fringe_1 = (100 % 2 == 0 ? 0 : 2);
    for (j = 0; j <= 99 - _j_fringe_1; j += 2) {
      int _k_fringe_3 = (100 % 2 == 0 ? 0 : 2);
      for (k = 0; k <= 99 - _k_fringe_3; k += 2) {
        A[i][j][k] = A[i + 2][j][k] + 0.1;
        A[i][j + 1][k + 0] = A[i + 2][j + 1][k + 0] + 0.1;
        A[i][j + 0][k + 1] = A[i + 2][j + 0][k + 1] + 0.1;
        A[i][j + 1][k + 1] = A[i + 2][j + 1][k + 1] + 0.1;
      }
      for (; k <= 99; k += 1) {
        A[i][j][k] = A[i + 2][j])[k] + 0.1;
        A[i][j + 1][k] = A[i + 2][j + 1][k] + 0.1;
      }
    }
    for (; j <= 99; j += 1) {
      int _k_fringe_2 = (100 % 2 == 0 ? 0 : 2);
      for (k = 0; k <= 99 - _k_fringe_2; k += 2) {
        A[i][j][k] = A[i + 2][j][k] + 0.1;
        A[i])[j][k + 1] = A[i + 2][j][k + 1] + 0.1;
      }
      for (; k <= 99; k += 1) {
        A[i])[j][k] = A[i + 2][j][k] + 0.1;
      }
    }
  }
}
```

Figure 6.6: Loop Nest example after applying Unroll-and-Jam with factor (1, 2, 2)

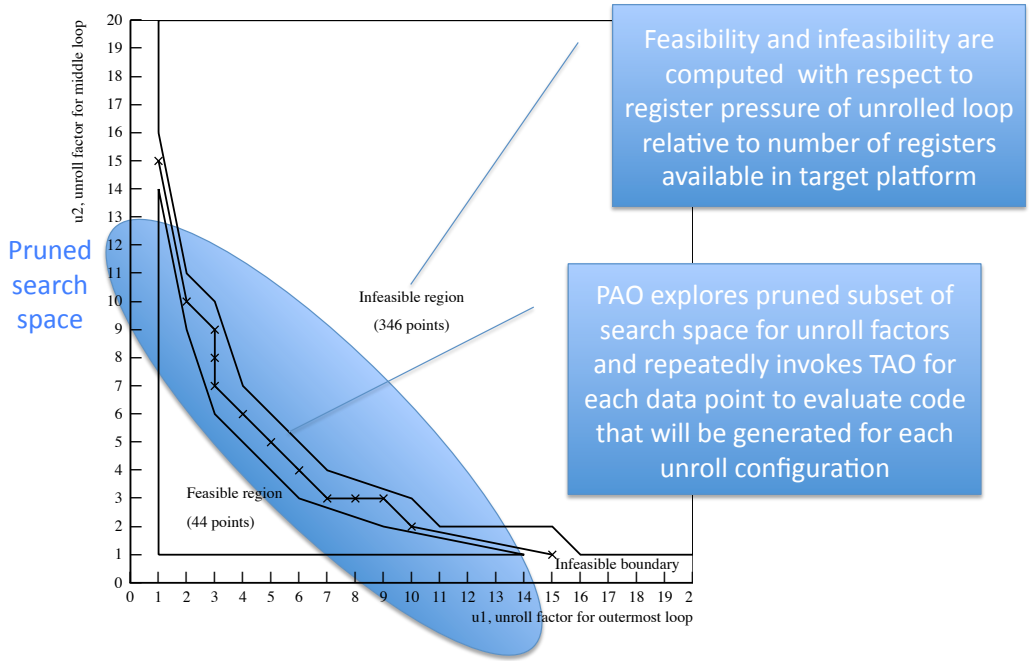


Figure 6.7: Search Space for Loop Unrolling Configurations

```
for (i = 0; i < n; i++){
   for (j = 0; j < n; j++){
      sum = 0;
      for (k = 0; k < n; k++){
         sum = sum + b[j,k] * c[k,i];
      }
      a[i,j] = sum;
   }
}
```

Figure 6.8: Matrix multiplication and transpose with scalar replacement of the a[i,j] element

Chapter 7
The Rose to LLVM Translator

The Platform-Aware Optimizer is implemented on top of the Rose infrastructure, while the Target-Aware Optimizer is implemented on top of the LLVM infrastructure. Thus, PACE needs a translator from the SAGE III IR used in the PAO to the LLVM IR used in the TAO. This chapter describes PACE-cc, the tool that implements this translation.

7.1 Introduction

Figure 1.2 provides a high-level overview of the PACE system design. This chapter focuses on the design of PACE-cc, a translator from the SAGE III IR to the LLVM IR. The SAGE III IR is an abstract syntax tree (AST) representation produced by the Rose compiler and used by the Platform-Aware Optimizer for transformations. The Target-Aware Optimizer operates on the LLVM IR, a linear code in static single-assignment form (SSA).

7.1.1 Motivation

The Platform-Aware Optimizer is implemented on top of the Rose infrastructure, while the Target-Aware Optimizer is implemented on top of the LLVM infrastructure. Thus, PACE needs a translator from the Sage III IR used in Rose to LLVM's IR. PACE-cc is the Sage→LLVM translator used to generate LLVM's bitcode, which is fed as an input to the Target-Aware Optimizer.

A critical aspect of the PAO/TAO interaction is the PAO's use of the TAO as an oracle for feedback on the performance of potential code transformations. The PAO produces Sage IR for synthetic functions, which represent transformed versions of selected user code fragments for which the PAO needs cost estimates (see 4.3.5). Here too a translation to LLVM's IR is needed by the TAO. The PACE-cc translator also implements this translation (Path 2 in Figure 4.2).

7.2 Functionality

7.2.1 Input

The translator is invoked in two distinct situations: as part of the full compilation path or the LLVM backend path (see § 3.4), and as part of a PAO-to-TAO query. In the first case, illustrated by Path 1 in Figure 4.2, the compiler driver invokes the translator, after the driver has invoked the PAO and before it invokes the TAO. The input to the translator along Path 1 is an AST in SAGE III IR with auxiliary information, and compiler directives passed by the compiler driver. These directives include optimization directives, some of which are generated by the PAO and instruct and constrain the TAO in its code transformations (see 4.2.2). The auxiliary information includes profile data, and information about aliases and dependences. To aid in vectorization, the auxiliary information

Principal Contacts For This Chapter: Philippe Charles, pgc1@rice.edu, and Michael Burke, mbg2@rice.edu

may include alignment information and sets of memory accesses (bundles). See Appendix B for a detailed description of the auxiliary information needed for vectorization.

In the second case, illustrated by Path 2 in Figure 4.2, the PAO invokes the translator and provides it with an AST in SAGE III IR form for the synthetic function that it has created for the query. Auxiliary information accompanies the AST, as in the first case. On this path, the PAO invokes the translator and the TAO.

7.2.2 Output

PACE-cc produces as output, along Path 1, the LLVM IR that corresponds to the input AST, along with LLVM metadata that provides links to the SAGE III IR auxiliary information described above. In that the PAO and the TAO will share a single address space, PACE-cc will give the TAO access to the SAGE III IR auxiliary information by constructing a global table of pointers to it and passing table indices to the TAO by means of the LLVM metadata facility.

PACE-cc produces as output, along Path 2, the LLVM IR that corresponds to the input AST for the synthetic function, along with LLVM metadata that provides links to the SAGE III IR auxiliary information described above. Once again, the communication between the PAO and the TAO will be facilitated by constructing a global table of pointers to the SAGE III IR auxiliary information and passing table indices to the TAO by means of the LLVM metadata facility.

7.3 Method

To perform a translation, PACE-cc makes two passes over the SAGE III IR with Pre/Post-order visitor patterns provided by Rose.

In the first pass, the translator generates attributes, associated with AST nodes, as part of the analysis necessary for mapping C constructs into LLVM constructs. Attributes are added to the AST to process global declarations and constants; map the C types into corresponding LLVM types; process local declarations; generate temporaries and labels.

In the second pass, LLVM code is generated. Each RPU is mapped into an LLVM module. First, global variables are processed, followed by aggregate types and function headers. Finally, code is generated for each function body in turn.

Due to incomplete (and in some cases, incorrect) semantic processing in the Rose compiler or semantic differences between C and LLVM, additional semantic analyses must be performed in PACE-cc. LLVM, unlike C, is strongly typed. All these semantic issues are resolved in the first pass of the translator using the SAGE III IR persistent attribute mechanism, without transforming the AST.

For example, instead of supporting a type for Boolean values, C uses the integer type to represent them. Boolean values often occur in a SAGE III IR representation, for example, as the result of an intermediate comparison operation. In the SAGE III IR, these Boolean values are represented as integers. LLVM has a bit type to represent Boolean values. The translator has to extend the SAGE III IR AST (with attributes) to include the proper casting between integer and bit values.

The Rose compiler's semantic processing of pointer subtraction is incorrect. The subtraction of two pointers yields a pointer value instead of an integer value. The translator corrects this error with the persistent attribute mechanism. Other issues of type include:

- The sizeof operator, whose value is not always correctly computed in the Rose compiler and not provided at all for structure types.
- Structure storage mapping.
- Integer (integral) promotion/demotion is not performed for *op*= operation on integer types.

For a given RPU input file, the SAGE III IR AST constructed by Rose is a complete representation of the file after preprocessing expansion. To avoid code bloat, including code duplication, we do

not generate code for extraneous program fragments that are imported from header files by the C preprocessor but are not relevant to the file being translated.

Thus, translation requires more than a simple pass over the AST. However, the SAGE III IR supports two traversal modes, both of which use the Pre/Post-order visitor pattern. Using these two traversals, PACE-cc can achieve the desired effect. We start with a complete traversal of the main input files. A function *traverseInputFiles(SgProject *)* traverses only AST components whose definition originated from a main (.c) input source file. While processing the elements in the main input files, we record the external elements, defined in imported header files, on which they depend. A function *traverse(SgNode *)* is given an arbitrary starting node in the AST and will traverse the subtree rooted at the node in question. After traversal of the main input files, we traverse the recorded external elements, and record the imported elements on which they depend. This process continues until there are no remaining imported elements.

Hence, a pass over the SAGE III IR AST consists of an initial call to *traverseInputFiles(SgProject *)* to process the elements in each main input file, followed by invocations to *traverse(SgNode *)* to import the needed elements defined in imported header files. These are the elements that the main file depends on, directly or indirectly.

To further avoid traversing duplicate AST representations emitted by the Rose compiler for certain features, we add a facility for short-circuiting a traversal at a given node during a visit.

7.4 Example

Consider the following C program:

```
int add(int x, int y) { return x + y; }

int main(int argc, char *argv[]) {
    int x = 5,
        y = 6,
        z = add(x, y);
    printf("z = %i\n",  z);
}
```

This program consists of a main program and a local function `add`. In addition to some basic declarations and initializations, the main program contains a call to `add` and accesses two global entities: the external function `printf` and the string constant `"z = %i\n"`. PACE-cc begins the translation of this C file with the following LLVM declarations for the global entities:

```
@"\01LC0" = internal constant [8 x i8] c"z = %i\0A\00"
declare i32 @printf(...)
```

The LLVM code generated by PACE-cc for the add function is:

```
define i32 @add(i32 %x, i32 %y) nounwind {
.entry:
  %.retval = alloca i32, align 4  ; storage for return value
  %x.addr = alloca i32, align 4   ; parameter x
  %y.addr = alloca i32, align 4   ; parameter y
  store i32 %x, i32* %x.addr      ; store value of parm x in mutable copy of x
  store i32 %y, i32* %y.addr      ; store value of parm y in mutable copy of y
  %.tmp0 = load i32* %x.addr      ; load x
  %.tmp1 = load i32* %y.addr      ; load y
  %.add2 = add i32 %.tmp0, %.tmp1 ; x + y
  store i32 %.add2, i32* %.retval ; store (x + y) in the return value
  %.tmp8 = load i32* %.retval     ; load return value
  ret i32 %.tmp8                  ; return the return value
}
```

The LLVM code generated by PACE-cc for the main function is:

```
define i32 @main(i32 %argc, i8** %argv) nounwind {
.entry:
  %.retval = alloca i32, align 4     ; storage for return value
  %argc.addr = alloca i32, align 4   ; parameter argc
  %argv.addr = alloca i8**, align 4  ; parameter argv
  %x = alloca i32, align 4           ; int x
  %y = alloca i32, align 4           ; int y
  %z = alloca i32, align 4           ; int z
  store i32 %argc, i32* %argc.addr   ; store value of argc in mutable copy of argc
  store i8** %argv, i8*** %argv.addr ; store value of argv in mutable copy of argv
  store i32 5, i32* %x               ; initialize x to 5
  store i32 6, i32* %y               ; initialize y to 6
  %.tmp3 = load i32* %x              ; load x
  %.tmp4 = load i32* %y              ; load y
  %.call5 = call i32 (i32, i32)*  @add(i32 %.tmp3, i32 %.tmp4) ; add(x, y)
  store i32 %.call5, i32* %z         ; z = add(x, y)
  %.tmp6 = load i32* %z              ; load z
  %.call7 = call i32 (...)*          ; call printf(..., z)
            @printf(i8 * getelementptr ([8 x i8]* @"\01LC0", i32 0, i32 0), i32 %.tmp6)
  store i32 0, i32* %.retval         ; store return value of 0
  %.tmp9 = load i32* %.retval        ; load return value;
  ret i32 %.tmp9                     ; return the return value
}
```

Note that the code generated for these two functions has a similar structure: a header statement similar to the C header statement; a variable declaration to hold the return value of the function (if needed); declarations of mutable local variables for the formal parameters (if any); declarations for user-declared local variables (if any); code generated to initialize the local variables associated with the parameters (if any); initialization code generated for user-defined local variables (if any); code generated for each executable statement in the body of the function.

Chapter 8

The PACE Target-Aware Optimizer

The Target-Aware Optimizer (TAO) is a major component of the PACE compiler. The TAO has two primary functions: performing target-aware optimization and providing feedback to the Platform-Aware Optimizer (PAO) . When performing target-aware optimization, the TAO optimizes the input code to better match the microarchitectural details of the target system's processors, as revealed by the PACE system's resource-characterization tools. When providing feedback to the PAO, the TAO returns information about how a specific segment of code will translate onto the target processor.

8.1 Introduction

The PACE compiler includes two major optimization tools: the Platform-Aware Optimizer (PAO) and the Target-Aware Optimizer (TAO). Figure 1.2 describes the relationships between these tools as well as the relationships between the TAO and other parts of the PACE system, such as the Resource Characterization tool (RC), the Runtime System (RTS), and the Machine Learning tool (ML). This chapter describes the functionality and design of the TAO, along with its interfaces to the rest of the tools in the PACE system. The TAO builds upon the open-source LLVM compilation system.

8.1.1 Motivation

Target-aware optimization sits between the PAO and the underlying hardware system on both the full compilation path and the LLVM backend compilation path. The TAO generates versions of the PAO-transformed application source code tailored to individual processors. To accomplish this task, the TAO must consider performance at a near-assembly level of abstraction, perform resource-aware optimization, and then either map the results of that optimization back into source code for vendor compilers or invoke an LLVM backend. Key aspects of the TAO include:

Resource-specific Tailoring The TAO uses knowledge from resource characterization to tailor the code for specific targets. For example, the RC might discover the relative costs of a variety of operations, including addition, multiplication, division, load, store, and multiply-add. The TAO can use that information when performing optimizations that make decisions based on the relative costs of operations.

Novel Optimizations The TAO provides a location for inserting new optimizations into the toolchain without modifying vendor compilers that are invoked on the full compilation path. For example, the tree-height restructuring pass that we completed reorders chains of arithmetic operations to expose additional ILP [32]. Inserting this optimization into the TAO made it uniformly available across all PACE supported targets through both the full compilation path and the LLVM

Principal Contacts For This Chapter: Linda Torczon, linda@rice.edu

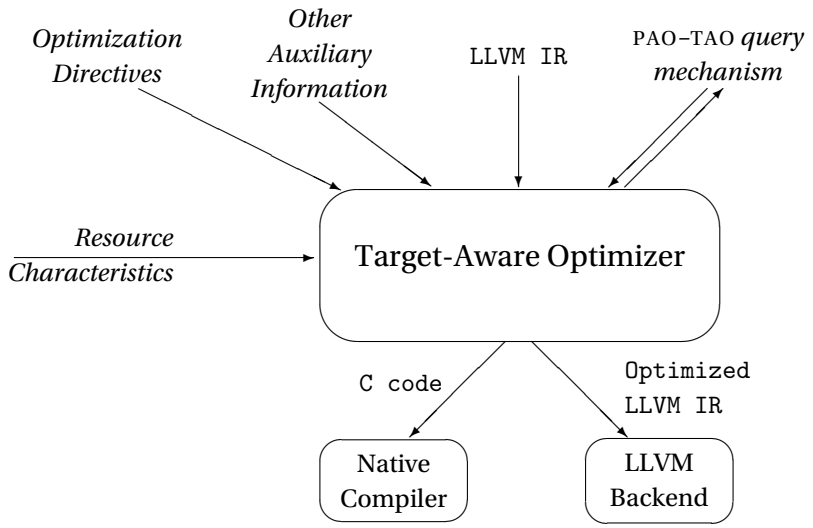


Figure 8.1: Target-Aware Optimizer Interfaces

backend compilation path.

Evaluation for the PAO The TAO lowers the code to a near-assembly level, where the mapping between specific code segments and the target hardware (as seen through the vendor compiler) is more visible. Thus, the TAO has a clearer picture of the match or mismatch between decisions made in the PAO and the target hardware. For example, the TAO can provide direct feedback to the PAO on register pressure or available ILP based on either introspective measurement of optimization effectiveness or evaluation of a model. That feedback should improve the PAO's ability to tailor its transformations to the target hardware.

8.2 Functionality

The TAO takes as input an optimized, annotated code fragment represented in the LLVM intermediate representation (LLVM IR), which is produced by the PAO and the ROSE-to-LLVM translator; characterization information and configuration information provided by the PACE RC; optimization directives provided by the PAO and the ML; and queries from the PAO. The TAO operates in two distinct modes: it is invoked by the compiler driver to produce optimized code from the output of the PAO, and it is invoked by the PAO as an oracle to obtain information about various properties of transformed code fragments produced by the PAO. As shown in Figure 3.2 and described in the "Target-Aware Optimizer" section on page 27, the TAO supports three distinct execution paths: an LLVM IR to assembly code translation on machines where the underlying LLVM compiler has a native backend, an LLVM IR to C translation, and a PAO query and response path.

8.2.1 Interfaces

Figure 8.1 shows the interfaces supported by the TAO. The TAO takes, as its primary input, a code fragment represented in the LLVM intermediate representation (LLVM IR). Auxiliary information may be tied to that LLVM IR fragment, including analysis results from the PAO and runtime performance information from the RTS. When invoked by the compiler driver, the TAO receives as input: the LLVM IR, metadata associated with the IR, and optimization directives produced by the PAO

and/or the ML components. In this mode it operates as a compiler and produces, as its primary output, a translated version of the LLVM IR code fragment. The code fragment can be expressed in either C code[1], which can be then compiled using a native C compiler, or as optimized LLVM IR for an LLVM backend. When invoked by the PAO, the TAO operates as an oracle and produces, as its primary output, a data structure containing responses to PAO queries. More detailed descriptions of the PAO–TAO query interface can be found in § 4.3.5 and § 8.3.5.

The TAO consumes resource characterization information produced by the PACE RC. It uses resource characteristics, defined as performance-related properties of the target system, to change the behavior of optimizations that it applies to the code being compiled. The TAO also relies on information from the configuration file for the target platform, which is provided by the system installer. (See § 3.2.2 for details.) The interface for information provided by the RC is described in § 2.3.1.

8.3 Method

The following sections describe the PACE approach to aspects of the TAO design and implementation: optimization in LLVM, vectorization, examples of implemented optimizations, selecting optimization sequences, and producing answers to PAO queries.

8.3.1 Optimization in LLVM

When invoked by the compiler driver, the TAO is presented with user code expressed as one or more LLVM IR procedures; when invoked by the PAO, the TAO is presented with a fragment of encapsulated synthetic code expressed in the LLVM IR. Under both scenarios, the TAO will apply a sequence of optimizations to the IR form of the code; the sequence may consist of both existing optimizations from the LLVM framework and new optimization passes developed for the LLVM/TAO framework as part of PACE. The specific optimizations (and possibly their order) is dictated by concrete optimization directives contained in the optimization plan; the TAO takes these optimization directives as one of its inputs. When invoked by the compiler driver, the TAO either generates native code using an LLVM backend or generates C code. When invoked by the PAO, the TAO returns optimization information to the PAO.

Optimization Directives When the PAO and/or the ML components provide optimization directives, the TAO uses those optimization directives to determine the sequence of LLVM optimizations to use (§ 4.3.7). If no optimization directives are provided, the TAO will use a generic optimization plan as its default. In extreme cases, the end user might provide distinct optimization directives to exert direct control over the TAO's behavior.

The TAO bases its processing of the application's code on the content of the optimization directives. Optimization directives may express both specific sequences of optimization, as in *perform algebraic reassociation followed by operator strength reduction,* and high-level goals, such as *optimize to minimize code and data space.* Directives might come from the ML as the distillation of prior experience or from the PAO as a result of the PAO's analysis of the code.

Thus, the optimization directives play a critical role in adapting the compiler's behavior to a specific application and, to a lesser extent, to a specific target system.[2] For this scheme to work, PACE must correlate information about specific optimization plans, applications, and their resulting performance. To simplify this process, the TAO will embed a concrete representation of its optimization plan in the code that it produces. For more detail, see § 3.2.2.

[1]To produce C code, the TAO uses the C backend interface in LLVM (/lib/Target/CBackend), which was produced by the LLVM open source development team.

[2]Most of the application-independent target adaptation should occur as a result of resource characterization and the use of characterization-driven optimizations.

Processor Characteristics	
Operations in Flight	Computation of ILP for feedback to PAO, as well as input to the query backend for scheduling
Operation Latencies	Algebraic reassociation, operator strength reduction, as well as input to the query backend for scheduling
Memory System Related Characteristics	
I-Cache Size	Comparison against code size for feedback to loop unrolling
Live values	Input to the query backend for register allocation and scheduling.

Figure 8.2: PACE Characteristics Used by the TAO

Transformations The open-source LLVM compiler system already includes a large set of optimization passes that implement a substantial set of transformations. The TAO both builds on the existing LLVM code base and uses pre-existing passes from LLVM.

1. Some LLVM passes are used without change. For example, LLVM includes a set of passes that eliminate "dead" code or data. While unified algorithms are available that would reduce the internal complexity of dead code elimination in LLVM, the existing passes are functional and effective. Since neither target-system characteristics nor application characteristics factor into dead code and data elimination, PACE uses those passes without modification.

2. Some LLVM passes use data produced by other components in the PACE system. PACE produces three major kinds of information that are of interest to the TAO: characterization information produced by the PACE RC tool, auxiliary information passed into the TAO from the PAO, and optimization directives as described earlier in this section.

   Using Characterization Data: Figure 8.2 shows characteristics measured by the RC that the TAO uses. Figure 8.2 also lists some of the applications for that data in the TAO's transformations.

   Using IR *Auxiliary Information:* In PACE, the TAO always runs after the PAO. This enforced order means that the TAO can rely on results from analyses performed in the PAO that are relevant to the TAO. The PAO passes analysis results to the TAO as auxiliary information to the LLVM IR; the ROSE-to-LLVM translator maps the auxiliary information to the LLVM IR while translating the ROSE IR to LLVM IR. This auxiliary information may include aliasing information, dependence information, and runtime profile information (derived by the RTS and mapped onto the code by the PAO). See § 4.2.2 for a description of the auxiliary information produced by the PAO and § 7.2.2 for a description of the mapping process.

3. Some LLVM passes may be modified to improve their effectiveness. We have completed preliminary studies of the effectiveness of optimization in LLVM with the goal of identifying weaknesses in specific optimization passes in the existing LLVM code base and modifying the transformations to address those measured weaknesses, whether they are implementation issues or algorithmic issues. In addition, the construction of the query backend (see § 8.3.5) may necessitate extensive modification to the register allocator and instruction scheduler.

   To understand optimization effectiveness in LLVM, we used the NULLSTONE compiler benchmark suite[3] to compare LLVM's performance against other compilers, such as `gcc` and `icc`.

[3] NULLSTONE is a registered trademark of the Nullstone Corporation. The NULLSTONE compiler performance suite is a

The NULLSTONE analysis identified several weak spots. We have also studied the performance of a variety of other benchmarks as part of our testing process.

4. We have implemented transformations in LLVM. The implemented transformations target either specific opportunities identified by our analysis of LLVM's effectiveness, or opportunities created by the transformations used in the PAO.[4]

   This activity involves a combination of implementing known algorithms from the literature and inventing new algorithms, with a focus on finding effective solutions to the underlying performance problems. We have built a tree-height restructuring pass that reorders chains of arithmetic operations to expose additional ILP [32]. We have also implemented operator strength reduction and linear function test replacement passes as well as a vectorization pass. (See § 8.3.2 and § 8.3.3 respectively for details.) Implementations of additional optimizations are in progress.

8.3.2 Examples of Implemented Optimizations

Brief descriptions of selected PACE LLVM transformations follow.

Operator Strength Reduction Strength reduction is a well known compiler optimization dating back to the very first production compilers. The basic premise is that time expensive instructions are replaced by those of lesser cost. This optimization can be seen in the address calculation instructions for an array within a loop. This address calculation can contain a multiplication. The strength reduction optimization will replace the multiplication with an addition using a new loop induction variable. Thus, the expensive multiplication will get replaced by an addition.

Operator strength reduction (OSR) [31] was developed as a replacement for the traditional strength reduction optimization [1], which was both hard to understand and hard to implement. OSR discovers induction variables in the program by finding loops in the Static Single Assignment (SSA) [33] graph. OSR next finds uses of the induction variable that participate in address calculations, creates a new induction variable, and uses the new induction variable in the address calculation, potentially replacing a multiplication with an addition.

The OSR optimization was added to LLVM as a stand alone optimization, which can be conditionally run per each LLVM invocation [77].

Linear Function Test Replacement After the OSR optimization has completed, the original induction variables might only have a single use in the loop ending test. An additional optimization, Linear Function Test Replacement (LFTR) [31], can be performed which replaces the loop test with a test using an induction variable that strength reduction added. Once the loop test is replaced, the code that initializes and increments the original induction variable becomes useless and can be deleted.

The LFTR optimization was included within the stand alone LLVM OSR optimization [77].

Register Allocation The PACE register allocator, an implementation of the Chaitin-Briggs graph coloring register allocator [16, 18], is based upon prior work at Rice University that resulted in an implementation for LLVM version 1.3 [27]. Subsequent effort by the PACE team produced an LLVM version 2.7 register allocator that was part of the generic LLVM code generator, which supported the PAO-TAO query interface. A later PACE effort on register allocation implemented a fully functional

proprietary product that we are using as part of our design process to assess compiler strengths and weaknesses. We intend to use the suite as part of our internal regression testing, as well. The NULLSTONE code is not part of PACE nor is it required to install or use the final system.

[4] In practice, many optimization algorithms contain implicit assumptions about the properties of the code being compiled. The PAO transformations create code that contains optimization opportunities that appear rarely, if ever, in code written by humans.

Chaitin-Briggs graph coloring register allocator for the LLVM x86-64 code generator. This work built on the PACE LLVM version 2.7 register allocator and targeted LLVM version 3.0. The spill code cost and spill code insertion functionality was rewritten to reflect more closely Briggs's original work. Rematerialization functionality was also added [17]. Currently, LLVM 3.0 with this register allocator can cleanly compile and execute the LLVM test suite.

8.3.3 Vectorization

When the PAO invokes the TAO for generating short SIMD vector code, the PAO passes the TAO the LLVM IR of a function, data dependence information, alignment information, and *bundle* information, which describes memory accesses to consecutive memory locations, as hints. The LLVM IR of the function contains metadata that indicates the innermost loop body that needs to be vectorized in the TAO. This innermost loop body is made amenable to vectorization in the PAO by the Polyopt subsystem using loop transformations in the polyhedral framework. Polyopt also provides data dependence information, alignment information, and information to build bundles. After Polyopt has processed the loop, the PAO unrolls the loop adding bundle and alignment annotations using the information obtained from Polyopt.

The TAO performs vectorization before any other standard compiler transformation is applied. It builds a dependence graph for each basic block of the annotated innermost loop nest using the dependence information from the PAO. If there is no dependence information available the TAO uses LLVM's alias analysis information to add memory dependence edges to the graph. After building the graph, the TAO performs dynamic-programming-based vector code generation using the bundle and alignment information to generate correct memory accesses. The dynamic programming selects between the usage of vector and scalar instructions using a cost-based model to determine optimal vector code for the input LLVM IR.

The details of how the PAO invokes the TAO and the vectorization algorithm are provided in Appendix B.

8.3.4 Selecting Optimization Sequences

Choosing which optimizations to apply is a critical part of the design process for any optimizing compiler. In PACE, the selection of specific transformations has several components. First, the PAO and the TAO address different concerns; this separation of concerns leads to some division in the set of transformations that the various tools will implement. (See § 3.5 for the division of transformations between the PAO and the TAO.) Second, the PAO and ML may suggest specific optimization directives for a given compilation. Third, the RTS will provide the compiler with information about application performance that can inform and guide optimization decisions in the TAO.

External Guidance The TAO accepts external guidance on optimization in the form of optimization directives. The TAO is responsible for the optimization plan mechanism and the implementation, but not for the generation of optimization directives. Directives may be generated by the PAO and/or the ML. In an extreme case, an end user might create a custom optimization plan to precisely control the process. In the absence of optimization directives from the PAO, the ML, or the user, the TAO relies on a default optimization plan.

The TAO may also receive external guidance in the form of performance information from the RTS, passed into the TAO from the PAO as auxiliary information to the LLVM IR form of the code. Performance information can influence optimization, ranging from decisions about path frequencies and code motion through the placement of advisory prefetch operations.

8.3.5 Producing Answers to PAO Queries

When the PAO invokes the TAO as an oracle, the PAO passes the TAO a synthetic code fragment encapsulated in a function; standard PAO–TAO auxiliary information, including profile and alias information; and a query data structure requesting particular information. The synthetic function will consist of a code region that contains a single loop nest, a loop body, or an entire function (see § 4.3.5).

The TAO produces, as its primary output, an updated query data structure (feedback value repository) containing metric information on the synthetic code that it would have compiled. Examples of PAO queries include requests for an estimate of register pressure or critical-path length in a code region. Details related to the types of queries that the PAO will generate can be found in § 4.3.5. Details related to TAO responses are included below.

When responding to queries from the PAO, the TAO can provide additional feedback to the PAO on the effectiveness of the transformed code produced by the PAO. For example, If the TAO finds that register pressure is too high in a loop, it can inform the PAO so that the PAO will know to transform the code in a way that reduces the demand for registers.

The prototype version of the PAO/TAO query interface computes the following low-cost estimates for each synthetic function:

1. MAXLIVE to estimate register pressure,
2. SPILLCOST of the generated code,
3. critical-path (CP) length to estimate the amount of instruction-level parallelism,
4. machine code size of the generated code, and
5. cost of SIMDizing the synthetic function.

Register Pressure and Spill Cost The MAXLIVE information is combined with the RC characteristics for the register file of the underlying architecture to determine the spillcost of the generated code using a simple register allocation algorithm such as linear scan or graph coloring. Another approach to estimating SPILLCOST is to invoke the machine dependent transformation passes and retrieve the number of spill operations after the register allocation phase.

Critical-Path Length The critical-path (CP) length estimate is an indication of the amount of instruction-level parallelism available in the synthetic function. For straight-line and acyclic code regions in the synthetic function, CP is determined using a dependency graph of the LLVM IR instructions. Each instruction is associated with a cost available from the RC. A simple depth-first traversal of the dependency graph yields the critical-path length estimate of the synthetic function. For code regions with multiple control flow paths, we can use either of the following approaches: (1) compute critical path length for each control flow path and weight them based on profile information or (2) use a control-dependence-based critical path length estimate. Approach (1) needs accurate profile information to limit the combinatorial explosion of the number of control flow paths.[5]

Machine Code Size The machine code size is the number of machine instructions generated for the synthetic function. This is an architecture dependent metric and retrieved after the invocation of machine dependent transformations in TAO, including instruction scheduling and register allocation. The PAO cost driven transformations can use this metric as a precise guide for estimating the performance of the transformed code. For example, the cost driven loop unroll-and-jam module uses the machine code size to evaluate and select the best factor for unrolling (see § 6.3.5).

[5] We are still deciding on the technique to use for dealing with cyclic code regions and software pipelining.

Cost of SIMDization SIMDization is an important optimization performed in the PAO. The PAO would like to know the cost of current high-level SIMDization performed for the synthetic function and, if possible, would like to get feedback on any improved SIMDization using code shaping and SIMD code selection. This analysis in the TAO requires cost estimates for various vector instructions and the length of the vector unit from the RC. Our approach in the TAO is to build the data dependence graph and perform a vector code selection algorithm based on these costs.

The above described estimates are computed in an efficient manner in terms of time and space. The results are accumulated in a shared data structure and fed back to PAO when the TAO is invoked in-core for a synthetic function.

In future versions of the TAO, more architecture dependent metrics may be introduced. For example, if the TAO scheduler detects that there is too little ILP in the PAO-provided synthetic code fragment, the TAO will inform the PAO that there is insufficient ILP when it responds to the PAO's original query. If the TAO identifies a loop that would benefit from software pipelining, it will inform the PAO; the PAO may remove control flow to support the decision to software pipeline the loop.

Chapter 9

The PACE Runtime System

The principal role of the PACE Runtime System (RTS) is to gather performance measurements of a program execution to support compile-time feedback-directed optimization and online selection of parameters, such as tile sizes and scheduling policies. At a minimum, the RTS uses an interval timer to measure time consumed in various parts of a program. By identifying costly regions in a program, the RTS can direct the PACE Compiler where to focus optimization. If hardware performance counters are available, RTS uses them to gather additional information about resource consumption and inefficiency; such information provides detailed insight into opportunities for improving performance on a target platform. This information can help the PACE Compiler identify appropriate optimizations needed to improve performance.

9.1 Introduction

The purpose of the PACE Runtime System (RTS) is to measure the performance of program executions with three aims: to help identify important program regions worthy of intensive optimization, to provide data to support feedback directed optimization, and to provide a harness that supports measurement-driven online parameter selection. Here, we describe the functionality and design of RTS, along with its interfaces to other components in the PACE system. The performance monitoring infrastructure of RTS builds upon Rice's HPCTOOLKIT performance tools [66]—open-source software for measurement and analysis of application performance.

9.1.1 Motivation

With each generation, microprocessor-based computer systems have become increasingly sophisticated with the aim of delivering higher performance. With this sophistication comes behavioral complexity. Today, nodes in microprocessor-based systems are typically equipped with one or more multicore microprocessors. Individual processor cores support additional levels of parallelism typically including pipelined execution of multiple instructions, short vector operations, and simultaneous multithreading. In addition, microprocessors rely on deep multi-level memory hierarchies for reducing latency and improving data bandwidth to processor cores. At the same time, sharing at various levels in the memory hierarchy makes the behavior of that hierarchy less predictable at compile time.

As the complexity of microprocessor-based systems has increased, it has become harder for applications to achieve a significant fraction of peak performance. Attaining high performance requires careful management of resources at all levels. To date, the rapidly increasing complexity of microprocessor-based systems has outstripped the capability of compilers to map applications onto them effectively. In addition, the memory subsystems in microprocessor-based sys-

Principal Contacts For This Chapter: John Mellor-Crummey, johnmc@rice.edu

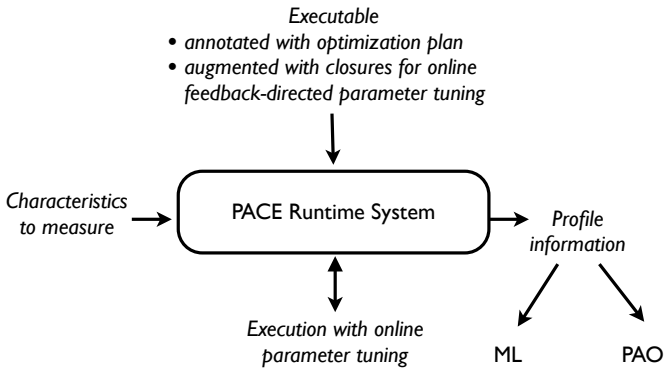


Figure 9.1: PACE *Runtime System inputs and outputs.*

tems are ill suited to data-intensive computations that voraciously consume data without significant spatial or temporal locality. Achieving high performance with data-intensive applications on microprocessor-based systems is particularly difficult and often requires careful tailoring of an application to reduce the impedance mismatch between the application's needs and the target platform's capabilities.

To improve the ability of the PACE Compiler to map applications onto modern microprocessor-based systems, the PACE RTS will collect detailed performance measurements of program executions to determine both where optimization is needed and what problems are the most important targets for optimization. With detailed insight into an application's performance shortcomings, the PACE Compiler will be better equipped to select and employ optimizations that address them.

9.2 Functionality

Figure 1.2 shows the major components of the PACE system and the interfaces between them; in that figure, the RTS components appear in blue. Figure 9.1 shows the inputs and outputs of the PACE RTS. The RTS will provide support for guiding online and offline optimization. This support comes in several forms:

- Runtime monitoring of metrics that can be measured using timers and/or hardware performance counters.
- Attribution of metrics to static and dynamic program contexts.
- A framework for providing performance profile information to (1) the AAP IS GONE to support application partitioning, and (2) the machine learning tools and the PAO to support offline feedback-directed optimization.
- A framework for runtime parameter selection based on measured metrics.

The measurement subsystem of the RTS monitors the performance of an executable in machine-code form. There are two ways in which the measurement subsystem can be used: it can be statically linked into an executable at program build time, or for dynamically-linked executables, it can be pre-loaded into the application's address space at launch time. In either case, when the program is launched, the measurement subsystem is initialized, environment variables set by a measurement script are read to determine what to monitor, and then execution begins with monitoring enabled.

9.2.1 Interfaces

There are several interfaces between the RTS and the rest of the PACE system.

- An RTS measurement script will drive application characterization by the measurement subsystem. The script will repeatedly execute an application to survey performance metrics that will be used to guide compilation.
- The RTS measurement subsystem will interpose itself between the application and the operating system on the target platform to intercept program launch and termination, creation and destruction of threads and processes, setup of signal handlers, signal delivery, loading and unloading of dynamic libraries, and MPI initialization/finalization.
- A profiler associated with the measurement subsystem could analyze binary measurement data recorded by the measurement system and produce call tree profiles in XML form that will be read by the PAO.
- A performance analyzer associated with the PACE RTS could digest performance profile data in XML format and provide an XML file that contains high-level quantitative and qualitative guidance to the Platform-Aware Optimizer about resource consumption, costs, and inefficiencies.

9.2.2 Input

The components of the PACE RTS receive several kinds of inputs from other parts of the PACE system.

Measurement script. An RTS measurement script will drive application characterization by repeatedly executing an application under control of the measurement subsystem to survey performance metrics that will be used to guide compilation. Inputs to the measurement script are an application in machine code form and a specification of a test input for the program (arguments, input files, etc.). What characteristics to measure will be derived from the hardware counters available on the target platform. If a characteristic is to be measured using asynchronous sampling, the RTS will choose an appropriate period for sampling the characteristic. The compiler driver provides a default measurement script in the application directory.

Runtime feedback-directed optimizer. During compilation, the Platform-Aware Optimizer (PAO) could determine that certain parameters may benefit from runtime optimization (Section 4.3.8). The PAO could present the RTS with a closure that contains an initial parameter tuple, a specification of the bounds of the parameter tuple space, a generator function for exploring the parameter tuple space, and a parameterized version of the user's function to invoke with the closure containing the parameter tuple and other state. The selection of tile sizes for parametrically tiled code is an example.

9.2.3 Output

The RTS measurement subsystem will produce a raw profile XML document that will associate static and/or dynamic contexts (which can include call paths, procedures, loops, and line numbers in source files) annotated with measured values of performance metrics, including call counts. It stores the raw profile document in an appropriate subdirectory of the application directory. The RTS performance analysis subsystem will augment the raw profile XML document with derived metrics that provide high-level quantitative and qualitative guidance about resource consumption, costs, and inefficiencies.

The RTS performance analysis subsystem will register the name of the executable, the time of a run, and the location of the performance information produced by the RTS with the PACE Compiler and the PACE Machine Learning tools using callbacks provided by each of these subsystems. RTS performance profiles can be used by the and PAO (§ 4.3.9) to support feedback-directed changes to the application's optimization to improve memory hierarchy utilization by adjusting data layouts (e.g. adding inter-variable or intra-variable padding; transposing arrays) and adjusting the code shape as necessary.

9.3 Methods

9.3.1 Measurement

The PACE Runtime System must accurately measure and attribute the performance of fully optimized applications. It is important to have an accurate measurement approach that simultaneously exposes low-level execution details while avoiding systematic measurement error, either through large overheads or through systematic dilation of execution. For this reason, the PACE RTS will build upon Rice's HPCTOOLKIT performance tools [66] as the basis of its measurement subsystem. The measurement subsystem will record profiles in a collection of files in a compact binary form that associates metric values with the static and/or dynamic contexts (identified by machine-code addresses) where the metrics were measured. No support is needed from the PAO or the TAO to support this profiling. Below, we outline the methods used for measuring application performance.

Asynchronous sampling. HPCTOOLKIT primarily uses asynchronous sampling rather than instrumentation to measure performance. Asynchronous sampling uses a recurring event trigger to send signals to the program being profiled. When the event trigger occurs, a signal is sent to the program. A signal handler then records the context where the sample occurred. The recurring nature of the event trigger means that the program counter is sampled many times, resulting in a histogram of program contexts. Asynchronous sampling can measure and attribute detailed performance information at a fine grain accurately as long as (1) code segments are executed repeatedly, (2) the execution is sufficiently long to collect a large number of samples, and (3) the sampling frequency is uncorrelated with a thread's behavior. Under these conditions, the distribution of samples is expected to approximate the true distribution of the costs that the event triggers are intended to measure.

Event triggers. Different kinds of event triggers measure different aspects of program performance. Event triggers can be either asynchronous or synchronous. Asynchronous triggers are not initiated by direct program action. HPCTOOLKIT initiates asynchronous samples using either an interval timer or hardware performance counter events. Hardware performance counters enable HPCTOOLKIT to statistically profile events such as cache misses and issue-stall cycles. Synchronous triggers, on the other hand, are generated via direct program action. One example of an interesting event for synchronous profiling is lock acquisition; one can measure the time per call to look for lock contention.

Call path profiling. Experience has shown that comprehensive performance analysis of modern modular software requires information about the full *calling context* in which costs are incurred. The calling context for a sample event is the set of procedure frames active on the call stack at the time the event trigger fires. We refer to the process of monitoring an execution to record the calling contexts in which event triggers fire as *call path profiling*.

When synchronous or asynchronous events occur, the measurement subsystem records the full calling context for each event. A calling context is a list of instruction pointers, one for each

procedure frame active at the time the event occurred. The first instruction pointer in the list is the program address at which the event occurred. The rest of the list contains the return address for each active procedure frame. Rather than storing the call path independently for each sample event, we represent all of the call paths for events as a calling context tree (CCT) [4]. In a calling context tree, the path from the root of the tree to a node corresponds to a distinct call path observed during execution; a count at each node in the tree indicates the number of times that the path to that node was sampled.

Exposing calling patterns. Besides knowing the full calling context for each sample event, it is useful to know how many unique calls are represented by the samples recorded in a calling context tree. This information enables a developer interpreting a profile to determine whether a procedure in which many samples were taken was doing a lot of work in a few calls or a little work in each of many calls. This knowledge in turn determines where optimizations should be sought: in a function itself or its call chain. To collect edge frequency counts, we increment an edge traversal count as the program returns from each stack frame active when a sample event occurred. We do this by having the trampoline function increment a "return count" for the procedure frame marked by the sentinel as it returns. A detailed description of this strategy can be found in our prior work [40].

Coping with fully optimized binaries. Collecting a call path profile requires capturing the calling context for each sample event. To capture the calling context for a sample event, the measurement must be able to unwind the call stack at *any* point in a program's execution. Obtaining the return address for a procedure frame that does not use a frame pointer is challenging since the frame may dynamically grow (space is reserved for the caller's registers and local variables; the frame is extended with calls to `alloca`; arguments to called procedures are pushed) and shrink (space for the aforementioned purposes is deallocated) as the procedure executes. To cope with this situation, we developed a fast, on-the-fly binary analyzer that examines a routine's machine instructions and computes how to unwind a stack frame for the procedure [73]. For each address in the routine, there must be a recipe for how to unwind the call stack. Different recipes may be needed for different intervals of addresses within the routine. Each interval ends in an instruction that changes the state of the routine's stack frame. Each recipe describes (1) where to find the current frame's return address, (2) how to recover the value of the stack pointer for the caller's frame, and (3) how to recover the value that the base pointer register had in the caller's frame. Once we compute unwind recipes for all intervals in a routine, we memorize them for later reuse.

To apply our binary analysis to compute unwind recipes, we must know where each routine starts and ends. When working with applications, one often encounters partially stripped libraries or executables that are missing information about function boundaries. To address this problem, we developed a binary analyzer that infers routine boundaries by noting instructions that are reached by call instructions or instructions following unconditional control transfers (jumps and returns) that are not reachable by conditional control flow.

HPCTOOLKIT 's use of binary analysis for call stack unwinding has proven to be very effective, even for fully optimized code [73]. At present, HPCTOOLKIT provides binary analysis for stack unwinding on the x86_64, Power, and MIPS architectures. On architectures for which HPCTOOLKIT lacks a binary analyzer for call stack unwinding, where available we will use `libunwind` [58], a multi-platform unwinder that uses information recorded by compilers to unwind the call stack. `libunwind` currently supports ARM, IA64, x86, x86_64, MIPS, and PowerPC architectures.

Flat profiling. On some platforms, support for call stack unwinding might not be available. On these platforms, the measurement subsystem could use simpler profiling strategy and collect only program counter histograms without any information about calling context. This form of profiling is referred to as *flat profiling*. Even such simple profiling can quantitatively associate costs with

program regions, which can serve to guide a compiler as to where optimization is most important.

Maintaining control over parallel applications. To manage profiling of an executable, HPCTOOLKIT intercepts certain process control routines including those used to coordinate thread-/process creation and destruction, signal handling, and dynamic loading. To support measurement of unmodified, dynamically linked, optimized application binaries, HPCTOOLKIT uses the library preloading feature of modern dynamic loaders to preload a profiling library as an application is launched. With library preloading, process control routines defined by HPCTOOLKIT are called instead of their default implementations. For statically linked executables, HPCTOOLKIT provides a script that arranges to intercept process control routines at link time by using linker wrapping—a strategy supported by modern linkers.

Handling dynamic loading. Modern operating systems such as Linux enable programs to load and unload shared libraries at run time, a process known as *dynamic loading*. Dynamic loading presents the possibility that multiple functions may be mapped to the same address at different times during a program's execution. During execution, the measurement subsystem could ensure that all measurements are attributed to the proper routine in such cases by dividing an execution into intervals during which no two load modules map to overlapping regions of the address space.

9.3.2 Profile Analysis

For measurements to be useful, they must be correlated with important source code abstractions. Profiles collected by the measurement subsystem will be digested by `hpcprof`, a tool that will correlates measured metrics with static and dynamic contexts at the source code level. `hpcprof` produces a profile XML document that associates static and/or dynamic contexts (which can include call chains, procedures, loops, and line numbers in source files) annotated with measured metric values. Here, we briefly outline the methods used by `hpcprof` to correlate profile data with static and dynamic application contexts.

Correlating performance metrics with optimized code Measurements are made with reference to instruction addresses in executables and shared libraries; it is necessary to map measurements back to the program source for them to be of much use. To associate sample-based performance measurements with the static structure of fully optimized binaries, we need a mapping between object code and its associated source code structure. HPCTOOLKIT's `hpcstruct` constructs this mapping using binary analysis; we call this process *recovering program structure*.

`hpcstruct` focuses its efforts on recovering procedures and loop nests, the most important elements of source code structure. To recover program structure, `hpcstruct` parses a load module's machine instructions, reconstructs a control flow graph, combines line map information with interval analysis on the control flow graph in a way that enables it to identify transformations to procedures such as inlining and account for transformations to loops [73].[1]

Several benefits naturally accrue from this approach. First, HPCTOOLKIT can expose the structure of and assign metrics to what is actually executed, *even if source code is unavailable*. For example, `hpcstruct`'s program structure naturally reveals transformations such as loop fusion and scalarized loops implementing Fortran 90 array notation. Similarly, it exposes calls to compiler support routines and wait loops in communication libraries of which one would otherwise be unaware. `hpcstruct`'s function discovery heuristics expose distinct logical procedures within stripped binaries.

[1] Without line map information, `hpcstruct` can still identify procedures and loops, but is not able to account for inlining or loop transformations.

Identifying scalability bottlenecks in parallel programs By using differential analysis of call path profiles collected by the measurement subsystem, the RTS will pinpoint and quantify scalability bottlenecks in parallel programs [26, 75]. Using a technique we call *blame shifting*, one can attribute precise measures of lock contention, parallel idleness, and parallel overhead in multithreaded programs [72, 74]. Combining call path profiles with program structure information, HPCTOOLKIT can quantify these losses and attribute them to the full calling context in which they occur.

9.3.3 Analyzing Measurements to Guide Feedback-directed Optimization

Identifying performance problems, rate-limiting resources, and opportunities for tuning often requires synthesizing performance metrics from two or more hardware performance counters. In general, our plan is to calculate and attribute *wasted cycles* associated with various features in a program.

We can measure or estimate exposed memory latency from hardware performance counters. Using instruction-based sampling support in AMD Opterons [35], one can measure the memory latency observed by an instruction directly. On systems that support only event-based sampling, we plan to estimate memory latency by multiplying numbers of cache misses at each level by their measured latency. When hardware counters permit, we plan to estimate exposed memory latency by combining measurements of total latency with measures of memory parallelism made with other hardware counters. We plan to measure and attribute the cost of pipeline stalls due to integer operations, floating point operations, and mispredicted branches. We will estimate total delay due to mispredicted branches in each context by multiplying the number of mispredicted branches by the delay each one causes. We will also compute instruction balance measures that will show the ratios of memory accesses, integer operations, branches, and floating point operations.

These metrics will highlight opportunities for improving efficiency that can be targeted by feedback-directed optimization in the PACE Platform-Aware Optimizer.

9.3.4 Runtime Feedback-directed Parameter Selection

The RTS could provide a harness to be used for runtime feedback-directed parameter selection. This harness could be used to select parameter settings for tilings and select among code variants. As input to this harness, the PACE PAO would provide a closure (§4.3.8) that would include the following information:

- A function that represents a parameterized region of application code. This code takes as input the closure.
- A parameter tuple that represents the current parameter setting. Initially, this tuple will contain the PAO's best estimate of the optimal parameter settings.
- The bounds of the parameter tuple space that needs to be searched.
- A generator function that takes as inputs (1) the current parameter tuple, (2) a map from parameter tuples to a vector of metrics that represent observed performance, and (3) the bounds of the parameter tuple space. The generator function will return the next parameter tuple, which may be the same as the current parameter tuple.
- A set of performance metrics that will be used to assess the goodness of a particular parameterization of a code region. Metrics may include time and perhaps hardware performance counter measures.
- Inputs other than the parameter tuple needed by the region of parameterized code.

- A flag that indicating whether or not this is the first use of this closure.
- A map between parameter tuples and runtime performance metrics. This map may be initially empty, or it may be partially filled in with information from the knowledge base.

The RTS could provide a harness for online feedback-directed optimization that uses this closure in the following way. If this is not the first invocation of the harness, the generator function would be invoked with the current parameter tuple and a map from tuples to a vector of measured metrics. The generator function would determine the next parameter tuple to try if the current parameter tuple is not satisfactory. The harness would arrange to measure the performance metrics specified. The harness would then call the parameterized application code using the current parameter tuple. The measured performance metrics for this tuple would be added to a map of tuples to metric vectors.

We could code a standard library of generator functions. Some generator functions could be as simple as an exhaustive search of the parameter space. Others could perform a sophisticated exploration of the parameter space using algorithms such as direct search, hill climbing, or other optimization techniques. The bound information from PAO, e.g., theoretical lower and upper boundaries according to DL/ML model in Section 6.3.3, also limits the parameter search space. In our design, the nature of the generator functions and the representation for a parameter tuple would be of no consequence to the RTS harness, which would merely need to be able to invoke the provided components in the aforementioned manner. For that reason, we would use the same harness to perform runtime feedback-directed optimization for a multiplicity of purposes, including selection of tiling and scheduling parameters.

Results of the online feedback-directed optimization can be recorded in the application directory, where they would be accessible by a PACE Machine Learning tool to help improve both the initial parameter tuple and the parameter spaces suggested by the PAO, and accessible by the compiler to improve its subsequent optimizations of the same code.

The measurement subsystem can collect performance profiles for timer and hardware performance counter events. `hpcprof` digests profiles from the measurement subsystem and assembles them into a profile XML document.

Chapter 10
Machine Learning in PACE

10.1 Introduction - Machine Learning for Compiler Optimization

10.1.1 Motivation

The Machine Learning component of the PACE project developed research infrastructure to support the central objective of the PACE project: to provide portable performance across a wide range of new and old systems, and to reduce the time required to produce high-quality compilers for new computer systems. Because the AACE program was cancelled, the PACE system was not completed under DARPA AACE funding as originally envisioned, and the ML tool has not been implemented, except for isolated learning engines.

Consider a problem in the PACE context: Given a program, a target system and a compiler, predict a good compiler configuration, such as a list of compiler flag settings which yields fast execution for the program. We shall refer to this problem as the "flag-setting problem". The selection of optimizations is part of the PACE compiler optimization plan; in particular, the generation of optimization directives (§ 3.2.4). The selection of optimizations that yields fast execution (optimum performance, in general) depends on the characteristics of the target system, the characteristics of the program being compiled, and the characteristics of the compiler. The relationship between the flag settings and the performance can be viewed as a relationship among points in a multidimensional space, spanned by the variables which characterize the program being compiled, the target system, the compiler flag settings and the performance.

To address this problem, a human designer uses past experience by remembering and applying a list of compiler flag settings used for similar programs encountered before; or by constructing a good list of settings based on trial runs of the program of interest. Thus the success of the designer depends on the ability to remember past experience, on the ability to distill, abstract, and generalize knowledge from past experience, and on the ability to spot patterns in the complex multidimensional space of non-linear interactions. This, in itself, is a formidable task. Furthermore, all this experience and knowledge might become irrelevant if the target system changes, and it would involve massive effort to re-acquire the relevant knowledge to be able to use the compiler effectively in a new target system. This is the central problem that the PACE project seeks to remedy. To remedy this problem, automation is needed to effectively and efficiently characterize the platform interactions: the interactions between programs, target systems, and compilers and use this characterization to optimize these interactions.

Machine learning aims to develop models of such complex relationships by learning from available data (past experience or from controlled experiments). The learned models facilitate discovery of complex patterns and recognition of patterns of known characteristics, in huge, unorganized high-dimensional parameter spaces, thereby making optimization tasks tractable and aiding in in-

Principal Contacts For This Chapter: Erzsébet Merényi, erzsebet@rice.edu, and Krishna V. Palem, palem@rice.edu

telligent decision making.

The machine learning group of the PACE effort is concerned with developing techniques to learn from the complex multidimensional data spaces that characterize the often non-linear interactions between programs, target system, and compiler optimizations. The result of the learning—the knowledge, captured in learned models of relevant optimization scenarios—can then be deployed and used in a variety of PACE related tasks such as compile-time program optimization (for speed, for memory usage, etc.), or for resource characterization. Moreover, with certain machine learning techniques, the models deployed after initial satisfactory *off-line* training could learn continuously in a run-time environment. This not only enables their use as oracles but allows ongoing improvement of their knowledge based on run-time feedback about optimization success.

10.1.2 Prior Work

Machine learning for compiler optimization is a relatively new area, with much unexplored territory. The following is a summary of what has been accomplished as of November 2011. Some demonstrable but not dramatic performance improvements have been accomplished. This leaves significant opportunities for further advances in this area. Prior work can roughly be divided into two categories, machine learning for optimization and machine learning to characterize platform interactions.

10.1.2.1 Machine learning for compiler optimization

Stephenson et al. use genetic programming (genetic algorithms applied specifically to programs) to determine priority functions used in compiler optimizations [71]. Priority functions are used extensively in compiler optimization heuristics. For example, in instruction scheduling algorithms, priority functions are used to assign priorities to instructions which in turn determine the instruction schedule (in general, the order of resource allocation.) When compared to hand-tuned priority functions used in the Trimaran compiler, a program-specific priority function for hyperblock formation yields an average improvement of about 25% in running time for the SpecInt, SpecFP, and Mediabench benchmark suites. A program agnostic priority function yields about 9% improvement on the average. Further discussion on the applicability of genetic algorithms to compiler optimization can be found in § 10.3.3.2. Cavazos et al. have used logistic regression, which is a technique to compute statistical correlation, to determine method-specific optimization settings [24] in Jikes RVM for a set of benchmarks drawn from SPECjvm, SPECjbb and DaCapo suites. The authors report improvements in execution time ranging from an average of 4% over $-O0$ optimization level with a corresponding improvement of 5% in total running time (the sum of the program execution time and the JVM), to a 29% (and 0%) improvement over $-O2$.

A similar approach has been used for the SPEC 95 FP, SPEC 2000 FP and INT, Polyhedron 2005, and MiBench benchmarks in the EKOPath compiler [23]. Average improvement of 17% in running time over all benchmarks over -Ofast setting (the highest optimization setting in the EKOPath compiler) has been reported. Agakov et al. construct Markov models to predict the effectiveness of optimizations and use this to inform an iterative search to determine good optimization sequences. This approach yields about 33% improvement in running time on a TI processor, after 5 rounds of searching whereas random search yields only about 32% improvement even after 50 rounds of searching.

10.1.2.2 Machine learning to characterize platform interactions

Cooper et al. and Almagor et al. [28, 3] characterized the space of compiler optimizations and its impact on the performance. The authors report that randomly evaluating 4 neighbors (the 4 most similar sequences) of a given optimization sequence yields more than 75% probability of finding a

better optimization sequence. Furthermore, 13% of local minima are within 2% of the best possible performance and about 80% of local minima are between 2% and 2.6% of the best possible performance, making descent algorithms with random restarts an ideal candidate to search for good optimization sequences. Joshi et al. attempt to use target system independent metrics to group similar programs from a benchmark suite [47]. The aim is to determine a representative subset of programs.

The reader is referred to "Survey of Machine Learning for Compilers" by the PACE machine learning group in the Rice PACE repository for a more thorough survey and comments on the strengths and weaknesses of each of these works.

10.1.2.3 The need for further development

This surveyed body of work demonstrates that machine learning can be successfully used to specialize compilers to new architectures (by tuning priority functions, for example). Though performance improvements have been reported, the effectiveness of machine learning itself has not been documented in most cases. For example, in the context of compiler optimization [23], it is not clear whether performance improvements arise from good decisions made by effective learning or from choosing randomly from a list of pre-filtered flag settings known to yield good performance. Joshi et al. achieve poor results in platform characterization. For example, representative programs (as determined by their technique) have an average cache miss rate which is about 40% more than the average cache miss rate of the entire benchmark suite. Thus further development is needed to (1) separate and quantify the effectiveness of the learning process itself and (2) to adopt more sophisticated machine learning techniques with the aim of effecting more dramatic performance increase in compiler optimizations.

10.2 Functionality

10.2.1 What Machine Learning Will Accomplish

Machine learning will be used to effectively and efficiently characterize the complex interaction between program characteristics, target system characteristics and compiler characteristics. This will be useful for solving problems encountered in several PACE tasks. As shown in Figure 10.1, which is an annotated overview of the PACE system presented in Figure 1.2 (§ 1.2.1), machine learning (ML) engines (marked ML1, ML2, ML3, ML4 and MLxx in rectangular boxes) are envisioned to help with the tasks in the Platform Aware optimizer (PAO), the Target Aware Optimizer (TAO) and the Run Time System (RTS). These engines correspond to the four PACE tasks identified in § 10.2.2 as likely to benefit from machine learning. From the point of view of the Run Time System the relevant ML engines will supplement the generator function (described in § 9.2.2) to help with the tasks of the RTS such as online feedback-directed parameter selection (described in § 9.3.4).

These ML engines will be provided data about the target system characteristics ①, program characteristics ② and compiler characteristics ③, by the subsystems where these circled numbers are indicated. Thus each of the circled numbers correspond to an arrow from the corresponding PACE subsystem to the ML subsystem

Machine learning is data driven therefore, the availability of *known instances* is essential. For example, revisiting the flag setting problem, machine learning can be used to learn the relationship between the program being compiled, the target system, the compiler flag settings and the performance from known instances. Briefly, as shown in Figure 10.2, a mapping $Y = f(X)$ exists from elements of an input space X (the program, compiler and target system characteristics) to elements of an output space Y (the compiler flag settings), where f is unknown. The role of machine learning is to construct a model based on known instances (known input-output pairs or *labeled training data*), which approximates the mapping f as well as possible, based on the quality of the

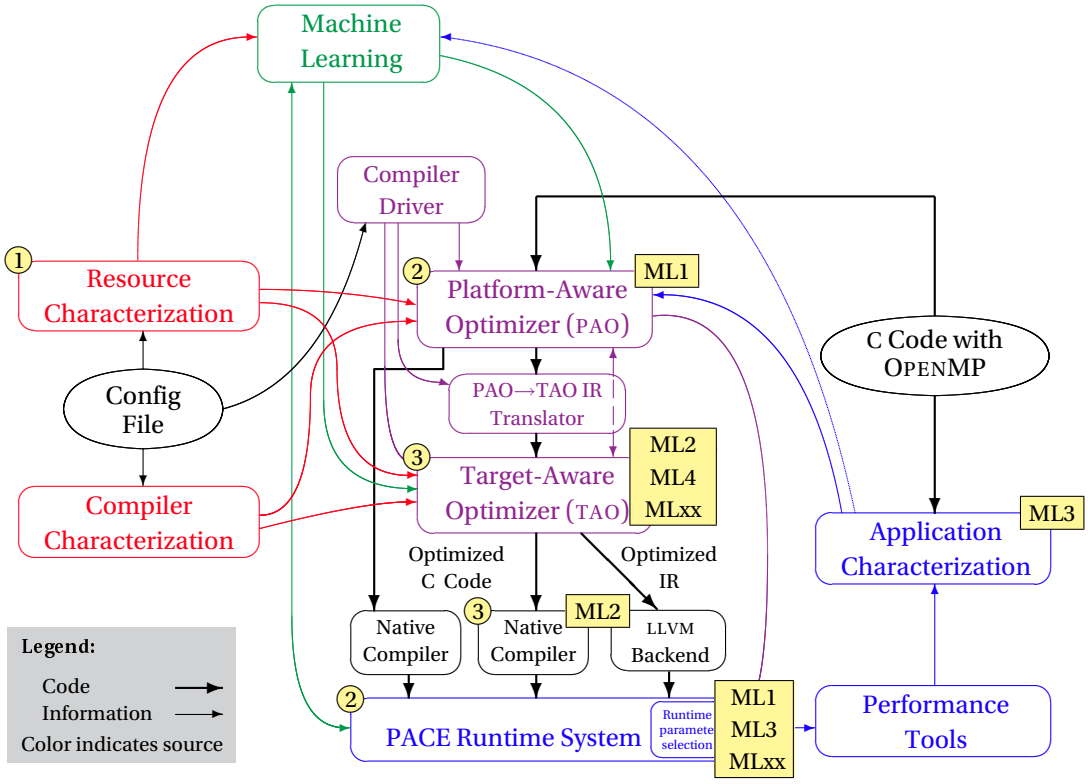


Figure 10.1: An overview of the PACE *system*

training instances. In the context of supervised learning, assuming the availability of a set X^{labeled} of known input-output pairs, elements from a subset $X^{\text{labeled}}_{\text{training}} \subset X^{\text{labeled}}$ are used by the machine learning system to construct a model by adjusting model parameters so that a good approximation of the actual mapping f is learned. A good learning process results in good *generalization* of the learned model, i.e., the model will make good predictions for patterns which were not part of the training set $X^{\text{labeled}}_{\text{training}}$. The learned model is then used for predicting a list of compiler flag settings for good performance for new programs that will be encountered by the PACE system.

10.2.2 Optimization Tasks Identified for Machine Learning

Four tasks have been identified as likely candidates to benefit from machine learning. The corresponding envisioned machine learning engines are indicated in Figure 10.1 in rectangular boxes labeled ML1 through ML4. In the context of compiler optimization, we use the term "good performance" to mean performance, in terms of execution time, code size or some other metric, which is reasonably close to the optimal performance or is a dramatic improvement over the baseline (unoptimized) performance.

1. Determination of tile size to optimize performance of a nested loop (ML1 in Figure 10.1)
2. Determination of compiler flag settings for good performance of a program (ML2 in Figure 10.1)
3. Prediction of program performance based on program characteristics (ML3 in Figure 10.1)
4. Determination of a good sequence of compiler optimizations for good performance of a program (ML4 in Figure 10.1)

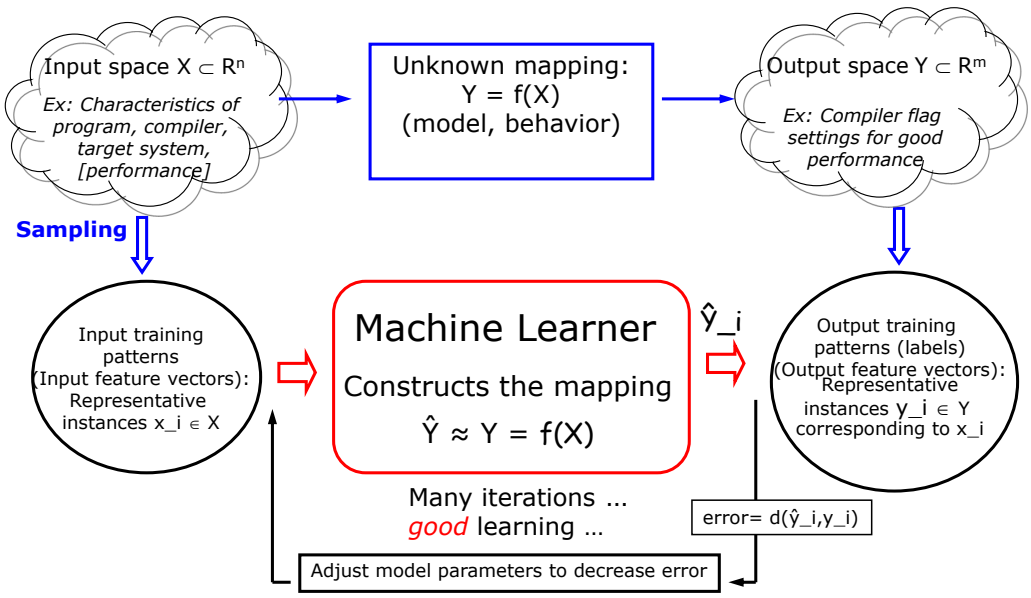


Figure 10.2: Schematics of supervised machine learning

For each of these tasks, the input variables (input features that make up the input feature vectors) will include descriptors of the target system, descriptors of the program, and the compiler, while output variables (output features) may be program performance indicators, compiler flag settings, or optimization sequences, as dictated by the particular ML task. The input and output variables can and will vary across different versions of models - typically models of progressive levels of complexity - for a task. For example, an initial, simple version of ML Task 1 (prediction of good tile sizes) may be designed for a single hardware platform in order to establish data need and baseline success without the complication of multiple platforms. In this case system characteristics need not be described in the input feature vector since they would be the same for all inputs. Once a simple model is shown to make acceptable predictions, we can proceed to set up a more complex model by including system characteristics such as cache sizes, line sizes, associativity, etc., in the input feature vector. The previously studied simple model will also help estimate data need for training a more complex model. Another reason for varying input and output features through different models for the same ML Task is to test the descriptive power of different sets of variables which may characterize the same properties. (For example, both the number of cache misses and the number of stall cycles can characterize the same aspect of the memory subsystem performance.) Selection of variables is guided by the accumulated experience of compiler experts, both within and outside the PACE teams, and may require separate models to work with non-overlapping sets of variables recommended by different expert groups. For these reasons, in this design document we are providing sets of typical variables that will likely be used, in various combinations, throughout a number of models that we will develop for each of the ML1 - ML4 Tasks. The specific set of variables for each model will be decided at the time a particular model is considered, and will often depend on the outcome of experiments with a previous model. We are including one specific feature set, as an example, for our first concrete model for Task ML1, at the end of § 10.2.2.1. Working lists of relevant variables, determined by PACE team members as well as adopted from literature, are maintained in the PACE Owl space in PACE Resources/Machine Learning/Data_Source/Variables_for_ML.xlsx file and will be revised as we accumulate experience.

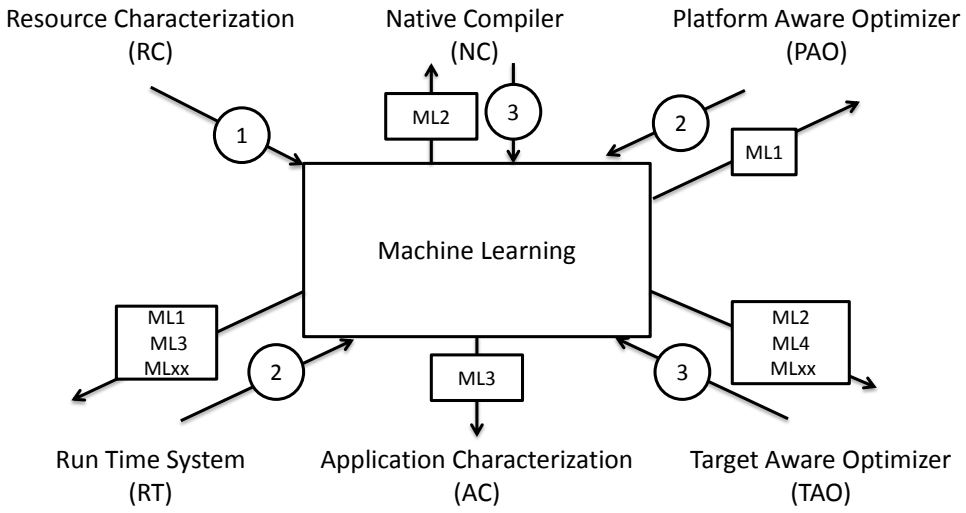


Figure 10.3: A machine learning centric view of the PACE *system*

Variables which capture the relevant target system characteristics will be obtained from the resource characterization (RC) subsystem of the PACE system. The target system characteristics, indicated as ① in Figures 10.1 and 10.3, for which measurement methodologies have been built so far are listed in § 2.2.3. Program characteristics indicated as ② in Figures 10.1 and 10.3 will be obtained from the PAO subsystem and the RTS. Compiler characteristics, indicated as ③ in Figures 10.1 and 10.3, will be obtained from the TAO subsystem and the Native Compiler (NC).

On a more general level we should point out that selection of appropriate input and output variables that describe causes and consequences needs to be done in two contexts. The first is a determination and listing of variables that potentially carry important information for the given problem. Such variables must be provided by compiler experts based on their understanding of how to represent relevant properties of programs, target systems, etc., and on their experience with program optimization. ML experiments should start with using as complete subsets of these expert-recommended variables as possible, and as appropriate for models of progressively increasing levels of complexity.

Once satisfactory prediction performance is achieved with ML for a given (version of a) task, we have a baseline of how well a model can perform when using all possible expert-recommended variables (pertinent to the given version of a task). The number of these variables, however, can be very large even if we do not count alternative variables for the description of similar causes (e.g., program properties). This is the second context in which variable selection should be considered, now as a subselection from the set of variables that were used to obtain the baseline results. Obviously, elimination of variables must not result in decline of prediction quality. Deselection of variables can be done with various dimensionality reduction approaches. Dimensionality reduction approaches that involve a transformation of the feature space, such as Principle Components Analysis (PCA), make it difficult or impossible to relate the transformed variables to the known meaningful quantities described by the original variables. Therefore, approaches that can assess the relative importances of the dimensions (variables) in the original feature space are much more advantageous. Approaches for the assessment of the relative importances of variables can be divided into two groups also from another point of view. The majority of available techniques make a determination with no regard to a known analysis objective. For example, in view of a known classification goal the important variables may be vastly different from those determined without

taking this goal into account. PCA, for example, would eliminate data based on statistical significance (as derived from the magnitude of the eigenvalues). However, this may eliminate the information needed to separate small classes, or to separate classes with slight differences, meaningful for the given problem. Furthermore, linear techniques, and techniques that use low-order statistics only, may miss relevant variations in the data. For all these reasons, non-linear techniques that can also take classification goals into account should be preferred for the determination of relevant variables, in the case of complicated data such as we have in PACE. One recent technique is *relevance learning*, published originally as GRLVQ (Generalized Relevance Learning Vector Quantization, [43]) and further developed specifically for high-dimensional data (GRLVQ Improved, [51]). These are two of very few available methods that jointly optimize a classification goal and the relevance weighting of the variables (see overview in [51]). GRLVQ(I) are non-linear machine learning approaches. We now describe the four machine learning tasks in greater detail.

10.2.2.1 Determine tile size to maximize performance of a nested loop

Given a nested loop in a program, the tile size that minimizes the average cost of memory access for data accesses from the loop, yields the best possible performance for the loop. Thus tile sizes that yield good performance can be determined by predicting the average cost of memory access corresponding to several instances of tile sizes and selecting a good instance. The selection of good tile sizes by the machine learning engine illustrated in Figures 10.1 and 10.3 and marked "ML1", would be helpful in program optimization tasks in the PAO as well as in the RTS where run time decisions on tile sizes in parametrized tiled code could be performed.

The average cost of memory access is the result of complex interaction between the memory hierarchy of the target system and the loop that uses a specific tile size. To illustrate key variables and their complex interaction, we use a vastly simplified example, but we emphasize that machine learning can be used for much more complex cases. (In fact, the whole point of machine learning is to be able to derive, from known examples, such complicated models of input / output relationships that cannot be given by closed formulae or easy-to-describe rules)

Consider a nested loop in Code Fragment A that accesses elements from a matrix of size $N \times N$.

```
A.1. For i = 1 to N
A.2.  For k = 1 to M
A.3.    For j = 1 to N
A.4.       = Matrix[i,j]
A.5.    End For
A.6.  End For
A.7. End For
```

Code Fragment A: Untiled Loop Nest

This loop can be made efficient if the elements Matrix[i,j] accessed in line A.4 can be cached and reused. However, traversing the entire length of the matrix before reusing the elements of a row might be inefficient, since the entire row of the matrix might not fit into the cache. Hence, one method of making the code efficient could be to transform the code to process the matrix "tile by tile" such that each tile fits into the cache and is completely processed before moving to the next tile. The corresponding code might look like this:

```
B.1. For Length = 0 to N/tile_length - 1
B.2.  For Breadth = 0 to N/tile_breadth - 1
B.3.   For k = 1 to M
B.4.    For i = Length * tile_length + 1 to Length * tile_length + tile_length
B.5.     For j = Breadth * tile_breadth + 1 to Breadth*tile_breadth + tile_breadth
B.6.       = Matrix[i,j]
B.7.     End For
B.8.    End For
B.9.   End For
B.10. End For
B.11.End For
```

Code Fragment B: Parametrically Tiled Loop Nest

In the equivalent code in Code Fragment B, the iteration space of the loop is divided into "tiles". Lines B.1 and B.2 loop over the first tile, second tile, third tile, ..., T^{th} tile. Lines B.4 and B.5 visit the points in the iteration space of a given tile. A tile size that is too small would lead to poor performance, since the loop body may not benefit from prefetching. A tile which accesses a piece of the matrix that is too big to fit into the cache, may cause misses in the cache adding memory overhead to the loop code.

- Target system characteristics (for each level of the memory hierarchy) such as
  1. The size of the cache, *L1 cache size* for the L1 cache
  2. The size of a cache line, *L1 line size* for the L1 cache
  3. The associativity of the cache, *L1 associativity* for the L1 cache
  4. The replacement policy of the cache, *L1 replacement* for the L1 cache
- Program characteristics such as
  6. The size of array(s) along each dimension, $size_{i,j}$ for the j^{th} dimension of array$_i$
  7. The index expression for each dimension j of the array$_i$, $expr_{i,j}$
  8. The loop iteration range of loop$_i$, $range_i$
  9. The size of padding for each dimension j of the array$_i$, $padding_{i,j}$
  10. The number of loop nesting levels, *n-nesting*
- Compiler characteristics such as
  6. For every loop level i the tile size, $tile\ size_i$
  7. Row or Column major layout, *layout*

Given these variables as input and corresponding execution time (as proxy for cost of memory access) for known instances in parameter regions which do not yield poor performances in an obvious manner, the machine learning system will build (learn) models that characterize this complex interaction. Thus, the learned model can be used for rapid search through the parameter space of (reasonable and not obviously disadvantageous) tile sizes to predict the execution time without having to run the program.

It may seem surprising that we predict execution time corresponding to an input tile size and post process rather than the intuitive approach of predicting good tile sizes directly. This is because several tiles might yield the same execution time and therefore the mapping from execution time to tile size, which is a one-to-many mapping would be difficult, if not impossible to learn in a supervised learning framework (as depicted in Figure 10.2). In contrast the many to one mapping of tile sizes to execution time can be learned. We describe this design decision again in § 10.3.1 and the more general philosophy.

Finally, we give a concrete example of a specific version of the model for ML1 task along with the specific variables we use in the training of that model. This model is the first and simplest

version of the ML1 task where the target system is kept constant and therefore we do not need variables to characterize the target system. To describe program characteristics in this case we chose tile size (*tile size*$_i$) the number of accesses and misses in the first and second levels of the data cache respectively (L1CDA, L1DCM, L2DCA, L2DCM), the number of accesses and misses in the TLB (TLBDA, TLBDM), and the number of vector instructions which have been executed (VECINS) as elements of the input feature vector to predict execution time. The use of execution time as proxy is based on expert opinion that the execution time is a linear function of the average cost of memory access. Likewise, ignoring the effects of vectorization, instruction-level parallelism, out of order execution etc. is based on expert opinion that these aspects do not affect the process of learning the mapping between tile sizes and execution time. Based on the understanding of the descriptive power of the variables included in this simple model, more variables may be considered in a subsequent more complex model. Concretely, the subsequent model we plan will include variables which characterize the effectiveness of hardware prefetch strategy. We think that this will improve the accuracy of predictions and will help generalize our model across loop bodies and across target systems. The added variables would be the average number of memory references (in a single iteration of the innermost loop) that could and could not be prefetched by the target system (n_PF, n_NPF). The reason for developing our model in an incremental fashion is to separate and understand the various aspects of the interaction between the program and the target system as well as to get a good grasp on the amount of training data required for good model building.

10.2.2.2 Determine selection of compiler flag settings for good performance of a program

Typically a compiler has several flags which turn optimizations on or off, set parameters for various optimizations, and so forth. For example, the flag -finline-functions-called-once, requests the `gcc` compiler to inline all functions which are called only once. Given a program, a target system and a compiler, one problem is to determine a list of flag settings which produces compiled code with good performance. In the PACE context, the setting of such flags and parameters is part of the generation of optimization directives and parameters for the optimization plan (§ 3.2.4).

The number of choices given k flags is typically exponential in k. The metric of the quality of the compiled code could be the execution time of the code or the size of the compiled code. In PACE, such flags are passed from the PAO to the TAO as directives (4.2.2). The machine learning engine marked as "ML2" in the Figures 10.1 and 10.3 will assist the PAO in selecting flags for good application performance.

The complexity of this problem arises from the fact that typical compilers have tens to hundreds of flags with an ever larger number of combinations of these flag settings. Furthermore, the effectiveness of specific optimizations depends on the interaction between the characteristics of the program, the target machine and other optimizations performed by the compiler. For example, function inlining may be beneficial, harmful or have no impact on the performance depending on

1. The effect on the instruction cache
2. The effect on the register pressure
3. The effect on other optimizations like constant propagation, common sub expression elimination etc.

Thus the optimal list of compiler flag settings is influenced by

- Target system characteristics such as
    1. The characteristics of the memory hierarchy of the target system described above
    2. The size of each type of register file, for example, *int reg size* for the integer register file size, *float reg size* for the floating point register file size and so on

3. The number of each type of functional unit, *FP mul num* for the number of floating point multipliers, for example
4. The length of the pipeline, *pipeline length*
5. The penalty for branch misprediction, *miss predict penalty* in number of cycles
6. $\cdots$

- Program characteristics such as

6. The dynamic instruction ratio for each type of instruction i, *dynamic inst ratio*$_i$
7. The static instruction ratio for each type of instruction i, *static inst ratio*$_i$
8. The ratio of backward branches to total number of branches, *forward branch ratio*
9. The average rate of branch mispredictions, *branch mispredict ratio*
10. $\cdots$

- Compiler characteristics such as

6. Callee vs. caller saved registers, *calling convention*
7. $\cdots$

By learning from known instances of the mapping between the list of variables which correspond to the characteristics enumerated above and the list of desired flag settings, the desired list of flag settings for a new program will be determined by machine learning. The desired list of flag setting is that list which achieves performance reasonably close to the optimal performance of the compiled code.

10.2.2.3 Predict program performance based on program characteristics

Consider the following scenario where there are two target systems A and B whose characteristics are known. For a set S of programs, the execution characteristics are known for each of the programs in S on the target system A. For a subset $S' \subset S$ of programs, the execution characteristics are known for the execution on the target system B. By learning from the execution characteristics of all programs on A and the execution characteristics of some of the programs on B, the machine learning system will be used to predict the performance of a program $P \in S \backslash S'$ when P is executed on the target system B. This engine, ML3 in Figures 10.1 and 10.3 will aid the application characterization task of the PACE system where predicted application performance (and performance of parts of applications such as procedures and loop bodies) serve as an indicator of application bottlenecks. This engine will also aid the RTS system where predicted application performance can serve as a basis for decisions regarding where and when to apply run time optimizations.

10.2.2.4 Determine a good sequence of compiler optimizations for good performance of a program

In typical compilers, not only can optimizations be turned on or off, the *order* in which various optimizations are applied and the number of times they are applied can be controlled as well. For example, optimizations such as dead code removal, common sub-expression elimination, constant propagation and inlining may be performed in an arbitrary order for an arbitrary number of times. Thus one frequently encountered problem is to determine the *sequence* of compiler optimizations to perform to yield good performance, where each optimization may be applied zero or more times. In the PACE context, this problem corresponds to item 5 in the optimization plan (§ 3.2.4).

We distinguish between the task described in § 10.2.2.2 (ML2) and the task described here (ML4) as follows: In ML2, the task is to determine a selection of flag settings with no implied order of optimizations while in ML4 the problem is to determine a sequence of optimizations which

yields good performance. These sequences can be of arbitrary length with possible repetition of optimizations. The corresponding learning engine is marked as "ML4" in Figures 10.1 and 10.3. Of the four tasks that have been identified in this section, this task is the least defined and least understood due to issues elaborated in § 10.3.2. Consequently the accomplishment of this task carries higher uncertainty than that of the other tasks.

The issues involved in effective learning, different machine learning approaches and the challenges associated with applying machine learning in the PACE context are discussed in the next section.

10.3 Methodology

10.3.1 Abstraction of PACE Problems For Machine Learning

We developed a framework for expressing compiler optimization problems as machine learning tasks. This is illustrated by the schematics in Figure 10.4 for the specific problem of tile size determination, described under § 10.2.2.1. The input and output feature spaces, shown for the general case in Figure 10.2, are determined by what feature(s) we want to learn from what other features. This is explained below through the specific example of determination of optimum tile size for a given loop.

The optimal tile size depends on several characteristics of the target system, the program, and the compiler, such as the number of distinct references made to the elements of the matrix and the spatial relationship of these references and their interaction with target system characteristics (as discussed in § 5.3.6).

Thus the input feature space which describes the multi-dimensional space of these variables could include the variables listed on page 90. The performance of a loop body with a particular tile size may be quantified using the *average cost of memory access* in cycles for each of the memory access in the loop body. In this case, this is the (single) dependent variable (a single output feature) that we want to be able to predict. This set of input and output variables span the multidimensional space which is the *feature space* for this task. Vectors in this feature space are called feature vectors and instances of known corresponding input - output feature vectors form the input-output pairs which will be used to train a supervised machine learning algorithm (as shown in Figure 10.2).

The total execution time of a loop body is a linear function of the average cost of memory access in most circumstances known to us. Therefore, we can use the execution time as a proxy in ML predictions. Specifically, we assume that the following factors do not distort the linear relationship[1]

1. Instruction-level parallelism
2. Out of order execution
3. Branch prediction accuracy
4. Compiler optimizations such as constant propagation and strength reduction
5. Other target system artifacts such as accuracy of prefetching

We note that it would seem more intuitive for this particular example to predict the tile size (use tile size as the output feature) and include the execution time in the inputs, but the execution time (and average cost of memory access of the loop body) has a one-to-many mapping to tile sizes, which is hard if not impossible to learn. The many-to-one mapping from tile sizes to the execution time (and the average cost of memory access of the loop body) can be learned. From predicted performances the favorable set of tile sizes can be filtered quickly by simple post-processing.

[1] Factors such as vectorization will have an impact on execution time, though the impact will most likely be the same across different tile sizes. This ensures that the linear relationship between the average cost of memory access and the total execution time across tile sizes is not distorted. There will be corner cases, such as one of the dimension of the tile size being 1, where vectorization might have a dramatically less effect on performance but we ignore or filter out such corner cases.

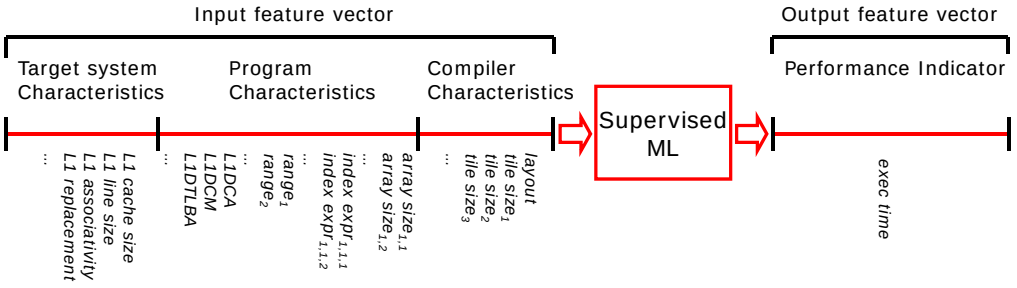


Figure 10.4: Machine learning schematics for the tile size optimization problem

The design of a feature space is a significant effort. It may require multiple phases beyond the initial abstraction exercise. For example, it should be ensured that variables relevant to the problem are captured in the input feature vector, else machine learning (and any learning) will be ineffective. Given a feature space, instances of the feature vector for which the target output features are known - *labeled samples* - should be generated or acquired. These constitute the training data from which the machine learner learns a model of the relationship between target system characteristics, program characteristics, compiler characteristics, and performance. The relevant features may not always be known in advance. If the learned model performs poorly one reason can be that some important feature has not been taken into account, which may warrant revision of the feature space, which in turn will necessitate a repeat of the learning experiments.

10.3.2 Challenges From a Machine Learning Point Of View

Compiler optimization involves a large number of variables in the input space (several dozens at least), and often also in the output space (dozens to over a hundred compiler flags, for example). The variety of complex interactions among the features results in a large number of patterns of behavior each of which requires a different optimization sequence to increase performance. This creates a learning task to map multi-variate inputs to multi-variate outputs, both potentially high-dimensional, and to delineate many classes (or precisely distinguish many degrees of some quantity such as execution time or tile size). The number of machine learning paradigms capable of dealing with such complexity of learning, is limited, and even the capable ones may not have been demonstrated on quite as ambitious tasks as those envisioned in PACE. Our experience from prior work [56, 76, 67, 53, 80, 81] with excellent machine learning performance on data that represent some of these or similar challenges (in a different application domain) will be utilized in this project. An additional challenge is that the variables in the PACE feature spaces are often mixed (disparate) types. This makes it hard to express or assess their relative importance, which in turn brings in issues of scaling and measures, both important for the success of machine learning. We are bringing considerable experience to PACE on this subject as well (e.g., [51] and references therein).

The specific ML technique for a particular ML task will depend on the nature of the task (regression, classification, clustering), the required resolution / precision of the prediction, the expected complexity of the mapping from input to output feature space, the dimensionality of the feature space, the amount and quality of training data available, the prediction capabilities of the given ML technique, and the computational expense.

Both supervised and unsupervised learning schemes will be used: supervised learning for regression (function approximation, prediction of continuous variables), or for classification, and unsupervised learning for clustering. Candidate learning approaches are discussed in some detail

under §10.3.3.

10.3.2.1 The impact of training data on machine learning

For learning complicated relationships among features, typically a large number of labeled patterns is needed for training, which may not exist or may be hard to acquire. A careful design of training data is critical, in any case, to ensure sufficient number of labeled samples and appropriate coverage and distribution over the problem space, for adequate representation. The availability and the time needed to generate training data is also an important aspect to be considered.

To test the performance of the learned model *test samples* are used. Test samples are labeled samples which are known to the model developers but not used for the training of the model, and which are set aside for the evaluation of the model's performance on data that the model has not learned from. The extent to which a learned model can make good predictions for unseen samples (samples outside the training set) is called the *generalization capability*. Producing models with good generalization capability is the main objective of machine learning. Sampling theories prescribe the number of test samples necessary for statistically significant assessment of the generalization capability. The requisite number can be very high for problems involving many classes and high-dimensional feature vectors.

The quality of the training data is also important. Noise and mislabeling are frequent adverse effects. Noisy data may require more samples to learn well, especially where classes are close in the feature space. Incorrect labeling can confuse the learner and decrease its performance. Careful evaluation of the noise situation, and verification of truth labels is imperative, and may take a few iterations, since the combined effect of noise and class proximities are usually not known; and incorrect labeling sometimes is only discovered from the learning itself. The above may necessitate revision of the training data and repeating of the learning experiment a few times in order to converge on an effective learned model.

10.3.2.2 Alternative to supervised machine learning: clustering

Clustering, a major type of unsupervised machine learning (Figure 10.5) is of fundamental importance, for two reasons. One is that good manifold learning that precisely maps the structure of the feature space enables *discoveries* of pattern groupings, and relationships among them. For example, we may discover previously not known program or compiler behaviors, or interactions between them. Another reason is that knowledge of the cluster structure can greatly assist in achieving subsequent accurate supervised classification and regression, by enabling fine discrimination of classes with subtle (but consistent) differences. Examples of these in earlier work (from a different application domain), where the feature space comprised hundreds of input features and up to several dozens of classes, include [53, 56, 67, 53, 80, 81].

Another use of clustering that will very likely have a significant role in PACE tasks, is the following. When labeled training data are scarce, we can cluster the feature vectors, take some summary descriptors (such as the averages) of the clusters as the typical behavior of the members of the clusters, and develop expert treatment for these cluster representatives. Then the members of each cluster can be expected to benefit from the same treatment. New feature vectors (at run time, for example) can be assigned to existing clusters by the trained model thereby indicating what treatment should be applied. This is illustrated in Figure 10.6, where the input feature vectors consist of descriptors of the target system, the program to be compiled, and the performance of the program with default compiler settings, and the resulting clusters represent categories of program behaviors. Through the (off-line) post processing indicated by the black rectangles the clusters can be labeled for treatment with appropriate optimization sequences developed for the discovered clusters. Bundled together, the clustering engine and the canned optimization sequences can serve as a run-time oracle-and-optimization unit.

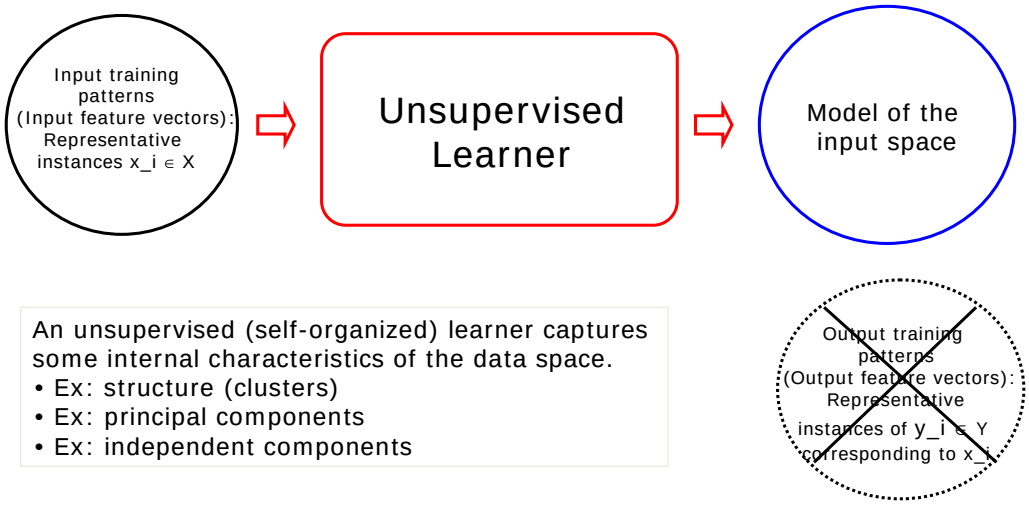


Figure 10.5: Schematics of unsupervised machine learning

10.3.3 Candidate Machine Learning Approaches

10.3.3.1 Neural networks

The methods we will be applying are all non-linear techniques, as–with perhaps a few exceptions–based on prior knowledge we anticipate problem spaces with complex, convoluted relationships among variables, and non-linearly separable classes.

Based on past experience with data sets and problems similar in their nature to those expected in the PACE context, neural computation is high on our candidate list. Neural approaches have demonstrably performed better on learning from such data than a number of other, well known, machine learning algorithms [13, 52, 45]. One of us (EM) has been developing neural modules under NASA funding for clustering (unsupervised learning) and classification (supervised learning) of high-dimensional (hyperpsectral) imagery, which has similarities with PACE data in dimensionality, considerable number of classes, scarce training samples, complex class/cluster structure. We also have experience with using neural computation for function approximation where the domain has hundreds of dimensions [80, 81].

We want to point out here that by "neural network" we do not imply "Back Propagation Neural Network (BPNN)". While our models may include, in simple cases, BPNNs, for complex cases we anticipate using more sophisticated and more robust neural architectures that were developed specifically for complicated high-dimensional data as mentioned above.

In a nutshell, these more robust approaches involve learning the structure of the data manifold first, in an unsupervised manner, and storing that knowledge in the form of a neural map (Self-Organizing Map, SOM) and related descriptors derived from the neural map. SOMs are adaptive vector quantizers (VQs) that place prototype vectors in the data space for optimal matching of the data distribution. SOMs have a unique property among VQs: they also represent the topology of the input data space by organizing (indexing) the quantization prototypes according to the similarity relations of the data. SOMs mimic the biological neural maps observed in various areas of the cerebral cortex. Neural maps form in response to input stimuli (data) and organize the stimuli on the 2-dimensional surface of the cortex while preserving the topological relations (the manifold structure of the input data). This facilitates fast and precise retrieval of patterns. Since SOM learning is unsupervised, the entire available data set can be used for its training, thus the SOM can

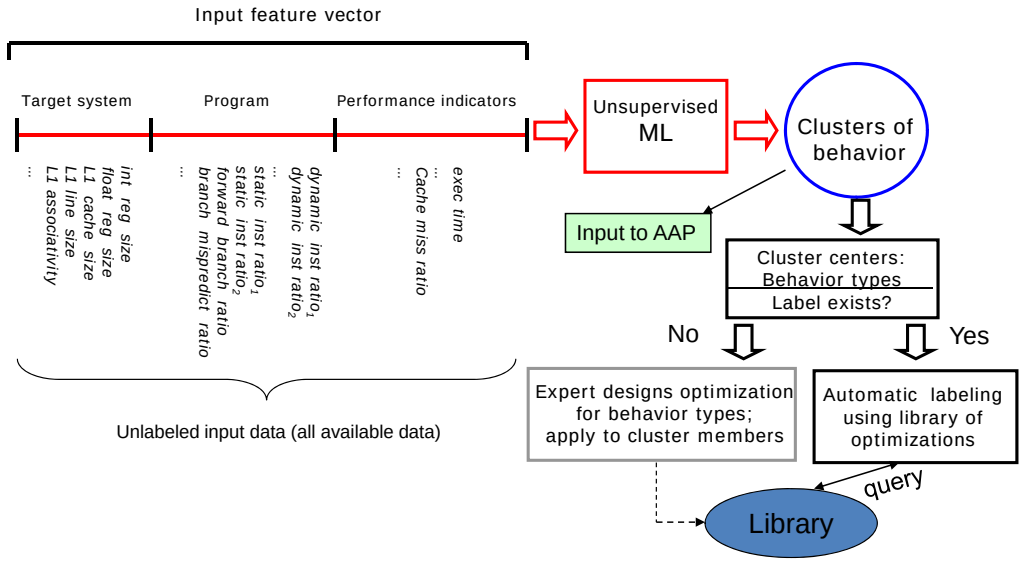


Figure 10.6: Unsupervised machine learning scheme for compiler flag optimization

form its own view of all available details of the manifold structure. A trained SOM can subsequently be snapped into a feed-forward, fully connected supervised network as its hidden layer, where the output layer is trained using the outputs of the SOM as inputs, and using a simple Delta rule (rather than the complicated BPNN rule). During supervised learning by this "SOM-hybrid network, the hidden SOM layer pre-screens the input patterns by indicating where they belong on the manifold. This helps the output layer learn to categorize the patterns into classes based on training labels - fast and precisely. Applications of this network architecture are described in a number of our previous works. An overview with references is given in [54].

This SOM-hybrid neural network is much easier to train than a BPNN network (is not prone to getting stuck in local minima). It has several other advantages over a BPNN network, as well as over other learning algorithms. The knowledge of the underlying SOM about the manifold structure - which is independent of the users knowledge, provided for supervised training as a set of training labels - makes it resistant to learning inconsistent labels. The topology preserving ptorotype based representation of the manifold by the SOM hidden layer allows good quality supervised learning from smaller number of training labels, and enables very precise discrimination of classes whose feature vectors may have slight but meaningful differences for the given application. The price to pay for all these advantages is in the training and interpretation of a high quality SOM. Issues relevant to this have been studied extensively, and tools developed, by the Merényi group. A recent book chapter summarizing related details is [57].

Neural networks have special significance in classification of feature vectors with disparate variables because (supervised) neural networks may be the only way to automatically derive - as part of the learning - appropriate scaling for the mixed types of variables.

Neural computing has also been used, successfully, for assessing the relative importance of the input features, for the purpose of eliminating non-contributing features. While this is quite difficult to do with traditional statistical approaches [13] some supervised neural network methods can naturally produce the weighting based on the training samples. Further, learning of the relevances of the features can be done in a joint optimization for a given classification goal [43, 51].

However, the extremely high number of flag settings (output features, or classes), for example,

exceeds previous experience, presents "firsts" and unknowns, which make compiler optimization and uncharted territory, to be approached with cautious optimism and with a commitment to further research.

10.3.3.2 Genetic algorithms

Genetic Algorithms are the natural choice for some PACE tasks, as earlier work by PACE investigators [30] (and other groups) demonstrated. In particular, the task of finding good order of (good) compiler flag settings involves (potentially) variable length feature vectors, which would be handled poorly by most other machine learners. Genetic Algorithms could also be used to do a fine-grained search in the vicinity of a solution proposed by a different ML algorithm (on a finer grid than the training data for the other ML algorithm was generated), to explore whether significantly better solution may exist. The drawback of Genetic Algorithms is, however, that they do not have a memory (do not build a model), therefore they cannot exploit experience gained from past instances (they have to evaluate each instance through a new search).

10.3.3.3 Other possibilities

Markov Models and Hidden Markov Models have already been applied successfully by us (LC and KP) in earlier research for the prediction of pre-fetching [49]. This software is already part of our arsenal and will be used in the Run-Time subsystem of PACE (box MLxx in Figures 10.1 and 10.3).

10.3.4 Productivity metric for Machine Learning

The performance of Machine Learning for PACE should be assessed on two levels. On the higher level, the *overall effectiveness of* ML will be measured by the quality of advice the ML models will give to the various subsystems (as shown in Figure 10.1) for their decision making. The exact way (or ways) to best characterize this effectiveness will be determined in the course of the development of the PACE system. However, the metric of overall effectiveness should be some combination of (1) the improvement in program performance and (2) decrease in the time needed to achieve the optimization. The ingredients for creating a meaningful metric, in any case, will come from run-time recording of performance improvements in program executions (or in optimization effort) a result of ML advice to the PACE compiler. Below we discuss some details of how these measurements can be done.

10.3.4.1 Quantifying the improvement in program performance

First, we discuss how the improvement in program performance may be quantified. This can be characterized with two different baselines: (a) the performance of the optimized program (running time, code size, memory footprint etc.) with the performance of the unoptimized program as the baseline. (b) the performance of the optimized program with the best possible performance of the program as the baseline. The improvement of the performance of the optimized program over the unoptimized program can be quantified in a relatively straightforward manner by measuring the performance of the unoptimized and optimized versions of the program. Since the performance of the unoptimized program would have been measured in any case to help drive decision making and adaptation in the various subsystems of the PACEcompiler, we do not expect this comparison to incur significant additional resources (time, instrumentation effort etc).

When the performance of the optimized program is compared with the best possible performance of the program as the baseline, it characterizes the *amount of optimization opportunity* discovered by the ML subsystem. For small program regions and for a small set of optimizations, the baseline can be determined by searching the set of all possible optimization decisions. However, for most practical scenarios involving large programs and a large set of possible optimizations,

determining the baseline could prove difficult. In this case, several alternative strategies may be adopted such as

1. Comparing the decisions of the ML engine with those of a human expert. This has several advantages-in particular, not only can the performance of the human expert-optimized and ML-optimized programs be compared, but the *nature of decisions* such as the flags that are set by the human expert and the ML engine, could yield valuable insights for the human expert as well as for the design of the ML models.
2. Generating synthetic program regions with known optimal (and therefore known baseline) performance. For example, a program with a synthetically generated sequence of instructions whose critical path length and instruction latencies are known, may be used to study the effectiveness of an instruction scheduler.
3. Using search strategies which either build on the ML decisions or are independent of the ML decisions to determine if program performance can be improved dramatically. For example, genetic algorithms could be used to search the neighborhood of the decisions made by a different ML approach to determine if better solutions exist

10.3.4.2 Quantifying the decrease in time needed to achieve optimizations

The decrease in time needed to achieve optimization can be quantified under two categories. The first is the reduction in the time needed to perform optimization decisions when the time needed for non-ML (but automated) approach is the baseline. For example, the time taken for the task of determining good tile sizes by (the non-ML approach of) searching, can be compared to the time taken by a trained ML engine to perform the same task. The second is the reduction in time needed when the time needed by a *human expert* to adapt the compiler to a new computer system and/or to perform optimization decisions is taken as the baseline. In both these comparisons, the time needed for the initial one-time training of the ML engine should be considered in some amortized manner.

Before evaluating the overall effectiveness of ML models for PACE, we must, however, measure their performance on a lower level first. The purpose of this is to ensure that the ML engines are well trained for the particular tasks, with the available data. As with any machine learning technique, a supervised model's *prediction capability needs to be assessed* by both a) verification of the learning success on the training data; and b) evaluation of the prediction accuracy on test data (data not used for the training of the model but drawn from the same data distribution), as described in §2.1. Moreover, the reliability (consistency) of the model must be evaluated by building a number of separate models through "jackknifing" (or cross-validation). This means that training and test data sets for each model are obtained through repeated shuffling of the available labeled data and splitting randomly into training and test data sets. The variance of the prediction of the resulting models on the respective test data sets should be small for the result to be credible. Only when trained models are found excellent in their generalization capability (i.e., in their prediction on test data) can one assume that an ML technique is providing advice based on what it derived, by learning, about the relationship between input and output variables. Consequently, only in this case can we attribute any observed improvement in program or compiler performance to Machine Learning.

10.3.5 Infrastructure

One of us (EM) has been developing neural learning modules under funding from the Applied Information Systems Research program of NASA's Science Mission Directorate for clustering and classification of high-dimensional (hyperpsectral) data, which have similarities with PACE data in

dimensionality, large number of classes, scarce training samples, complex class/cluster structure. These learning and data visualization engines have been used to support science investigations in earth and space science, as well as in medicine [55, 67, 36]. The software developed for these applications could be modified and augmented appropriately to interface with PACE data and used for experiments implementing the machine learning tasks outlined in § 10.2.2.

10.4 Conclusions

In consultation with the Resource Characterization, Platform Aware Optimization, and Run-Time groups, we have defined an abstract framework for the connection points, input and output variables (the feature space) and the types of learning engines for machine learning tasks that are most likely to benefit the PACE system.

We developed the design of training data and the data collection plan for the first problem we want to target with machine learning, the tile size optimization to be used in the PAO (§ 10.2.2.1). We collected simulator data within the RCacheSim simulator to represent 78 different memory structures of the X86 family. While data collection still continues for this task, we have produced initial neural and other models of the size vs. execution time.

Machine learning for compiler optimization is in its infancy, with much unexplored potential - and potentially with some hard surprises that could require development of new approaches. From what we have researched, combined with our previous experience, we are cautiously optimistic that several effective machine learning components could be developed for the PACE system.

Appendix A

Microbenchmarks Used in Resource Characterization

This appendix contains descriptions of the designs of the resource-characteristic microbenchmarks listed in Table 2.2 on page 16.

Principal Contacts For This Chapter: Keith Cooper, keith@rice.edu

Data Cache Capacity

Description	This microbenchmark measures the capacity of the discernible levels of the data cache hierarchy.
Method	The microbenchmark uses a series of arrays of increasing size and steps through each array in an access pattern designed to maximize cache misses and minimize TLB misses. The number of accesses is held constant for each test. When successive arrays fit within a given level of cache, the time to perform all the accesses should remain constant (within timer error). The microbenchmark starts testing at a small size, say 4096, and doubles that number up to a large size, say 32 MB. The initial and final sizes must be integral powers of two. Between each power of two, we sample a set of three equally-spaced points, to create a series of points that grow in a roughly logarithmic fashion. The access pattern that reveals cache sizes treats the array as a two-dimensional array, with the rows of length *page size*, obtained from the POSIX system call `sysconf()`. The pattern accesses a subset of the elements in each row. It makes all of its accesses within a page before switching pages, to minimize TLB misses. The order of access within a row, and the order in which the rows are accessed, are both shuffled into a random order.
Analysis	The microbenchmark produces an execution time, in microseconds, for each size tested. The analysis treats this result as a series of points $$((size_1, time_1), (size_2, time_2), (size_3, time_3), \cdots (size_n, time_n))$$ that define a piecewise linear function (PLF). It looks for inflection points in that PLF using a number of numerical tests. The transition caused by the L1 cache is particularly sharp; the analysis locates it using a simple test that detects a sharp rise in the PLF. Subsequent levels of cache are more subtle. The analysis uses transitions in the slope of the PLF to identify suspected cache boundaries. It then examines the neighborhood surrounding each suspect at a different scale to confirm or reject the point.
Difficulties	*Variations in timing:* On a real system, the timing results returned by these tests include significant variation. To minimize this effect, the microbenchmark makes multiple runs at each size and keeps the smallest time. It sweeps over the sizes from smallest to largest, then repeats that sweep to decrease the likelihood that an external event disrupts all of the tests at a given size. *Analysis:* Noisy data produces PLFs with inflection points that do not correspond to cache boundaries. We have pursued several techniques to smooth the data, to detect inflection points, and to confirm the suspect points. *Effective vs. Real Boundaries:* Sharing between instruction and data caches and between cores can radically reduce the effective size of a level. Choosing the effective boundary may involve some arbitrary threshold value.
Citation	

Data Cache Line Size

Description	This microbenchmark measures the number of bytes in a line for each level of the data cache.
Method	In a manner similar to the method used to measure the size of the data cache, the data cache line size microbenchmark iterates through an array with a pattern encoded in the array itself. The difference is that this microbenchmark encodes two different, equal-sized patterns in the same array. Each pattern accesses enough memory to fill up the level of cache being tested. The microbenchmark starts by dividing the array into "stripes" of length two. Each pattern accesses one element in alternating stripes, so the first pattern accesses the first element in odd-numbered stripes, while the second pattern accesses the first element in even-numbered stripes. The stripes correspond to an estimate of the line length: if the length of a stripe exactly matches the length of a cache line, the execution time should change. Thus, the microbenchmark starts with a stripe of length two and increases that length until a change in behavior is observed. In a test, we alternate running through the first pattern and running through the second pattern. Running through the first pattern will load all of those values into the cache. Running through the second pattern will cause conflicts with values loaded by the first pattern, except when the stripes fit exactly on the cache line. Then, each pattern will be accessing alternating lines of the cache.
Analysis	Each stripe length is timed and that value is compared against the version with the stripe length set to two. As we increase the length of the stripes, the number of locality hits decreases until the stripe length equals the cache's line length. At that point, there is no spatial locality, but no inter-pattern conflicts, either, and the measured time will drop below the baseline.
Difficulties	To handle the fact that physically mapped caches do not allow control of the placement of arrays in memory, patterns are split across multiple pages. Each pattern gets exactly half of each page allocated to the array, and the patterns completely traverse a given page before traversing another page of memory.
Citation	

Data Cache Associativity

Description	This microbenchmark measures the associativity of a given level in the data cache hierarchy. Because it takes the size of that level as input, it must run after the data cache capacity microbenchmark.
Method	Given a data-cache level of size N words, the words at indices 0, N, $2{\cdot}N$, $\cdots$, $i{\cdot}N$ must all map to the same way. This microbenchmark constructs a series of access patterns using these locations, for two ways (the words at 0 and N), for four ways (0, N, $2{\cdot}N$,$3{\cdot}N$), and so on, out to thirty-two ways. It runs each permutation for the k accesses, where k is chosen to ensure at least 1,000 ticks on the timer. When the number of locations in the permutation exceeds the cache associativity, the time to run the pattern will increase due to the high cost of a cache miss relative to the cost of a cache hit. If all the permutations have the same cost, the cache is assumed to be fully associative.
Analysis	The microbenchmark uses the "sharp rise test" developed for the data cache capacity benchmark. If it detects no "sharp rise" in the permutations from two ways to thirty-two ways, it reports that the cache is fully associative.
Difficulties	*Memory Size:* The microbenchmark assumes that no microprocessor will have associativity greater than thirty two. It will report any number larger than thirty two as a fully associative cache. In principle, testing for larger associativity is simple; in practice, the microbenchmark needs an array with $32{\cdot}N$ words of memory, which may limit its use on larger associativities of larger caches. In practice, a thirty-two way, set associative cache approximates a fully associative cache well enough for most optimizations. *Physically-mapped Caches:* The microbenchmark assumes that adjacent pages in virtual memory map into adjacent cache locations. In a physically-mapped cache, this assumption need not hold. The microbenchmark may have problems with physically-mapped caches; of course, the compiler cannot rely on associativity in optimizing for a physically-mapped cache.
Citation	

Data Cache Latency

Description	This microbenchmark measures the time that it takes to load a value from each level of cache.
Method	To generate data, this microbenchmark uses the method described in the section on finding data cache capacity (page 102). The methods differ in their analysis of the data. The data cache capacity microbenchmark analysis looks for significant upticks in the per-access time. By contrast, the data cache latency microbenchmark analysis looks for periods of little change in the per-access time. The times during periods of little change correspond to the latency for that level of cache. When the access array fits into the cache, all accesses require a uniform amount of time. When the per-access time between two tests is the same, it indicates that each test's array fits into that level of cache and that the latency is precisely that shared measurement.
Analysis	This microbenchmark relies on knowing the size of each level of cache. It then forms a histogram of the times measured for each set of arrays with sizes that fall between the cache sizes of two adjacent levels of cache. The most frequent per-access time measurement is the reported latency for the higher of the two cache levels. For example, if the microbenchmark builds a histogram of the per-access time measurements for arrays with sizes that fall between two cache sizes, L_n and L_{n+1}, the most frequent per-access time measurement is reported as the latency of L_{n+1}.
Difficulties	The difficulties encountered by this microbenchmark are identical to the difficulties described in the section on finding data cache capacity (page 102). In a perfect system, the per-access time for any individual test that falls between two cache sizes could be reported as the latency for the higher of the two levels of cache. However, because of the variability in timing, the most frequently observed per-access time for tests on arrays that fall between two cache sizes is reported for the higher level of cache, effectively discarding per-access time measurements that are probably inaccurate.
Citation	

TLB Capacity

Description	This microbenchmark measures the capacity of the discernible levels of the translation look-aside buffer (TLB) hierarchy.
Method	The microbenchmark uses a series of arrays, of increasing size, and steps through each array in an access pattern designed to maximize TLB misses and minimize cache misses. The mechanism is the analogue of the data-cache capacity tests (Appendix A, page 102). The difference is that, while the data-cache capacity microbenchmark maximizes the number of data accesses per page in order to minimize the number of misses in the TLB, this benchmark accesses a single data element per page to maximize the number of entries needed in the TLB.
Analysis	Like the data-cache capacity experiments (Appendix A, page 102), the microbenchmark produces an execution time, in microseconds, for each size tested, and the data is examined for significant changes.
Difficulties	*Variations in timing:* On a real system, the timing results returned by these tests include significant variation. To minimize this effect, the microbenchmark conducts multiple runs at each size and keeps the smallest time.
Citation	

Operations in Flight

Description	This microbenchmark measures the number of simultaneous arithmetic operations that can run on the architecture.
Method	The microbenchmark uses a series of interlaced streams as shown in Figure 2.2 (page 18) to measure the number of parallel operations that can execute simultaneously. This is a series of tests, one for each of the four arithmetic types: addition, subtraction, multiplication, and division. For each arithmetic type, we test for four data types: 32-bit integers, 64-bit integers, single-precision floating point, and double-precision floating point. Each interlaced stream is as long as the first stream. So, if the architecture can support, say, two addition operations per cycle, the two-stream executable should run as fast as the single-stream version.
Analysis	Separate tests are run for each <arithmetic operator, data type> pair. The microbenchmark produces an execution time, in microseconds, for each test. For example, the test for 32-bit integer addition is a series of executables with an increasing number of interlaced streams. The runtime for each executable in the test is compared against the single-stream runtime for that test. The runtime doubles when the test exceeds the architecture's resources.
Difficulties	This microbenchmark is subject to the same timing challenges as the other microbenchmarks. This difficulty is moderated by the large time increase observed when the test exceeds the architecture's resources. Because long streams are used to ensure a noticeable increase, as the number of interlaced streams grows, each executable can take a long time to compile and execute.
Citation	

Instruction Latencies

Description	The goal of this test is to determine the execution latency of a set of commonly executed instructions. The test reports latency relative to 32-bit integer addition. If the latency of 32-bit integer addition is one cycle, the reported latency can be interpreted as cycles.
Method	This microbenchmark measures the latency of the main four arithmetic operations (addition, subtraction, multiplication, and division) for the four main data types (32-bit integers, 64-bit integers, 32-bit floating point, and 64-bit floating-point). All times are reported as the ratio against the time to perform 32-bit addition, as this is usually the simplest, fastest instruction on most architectures. The execution time of an executable made up of a stream of 32-bit integer operations as shown in Figure 2.2 (page 18) serves as the base case. the execution times of streams of the same length with different instructions are compared against this base case. On certain platforms, the latency of an instruction depends on the value of the arguments. The execution unit exits the computation early if it encounters input values of a certain form. For example, integer multiplication exits early if one of the operands is zero. To prevent this from happening, the input values to instructions must be controlled. However, the only values that can be controlled are the two initial values in the instruction stream. All subsequent values are determined by the previous results in the stream. On the PowerPC G4, for example, the latency of an integer multiply instruction is reduced by one if the 15 most significant bits of the operands are all zeros or ones. The integer-multiplication test prevents the operands from getting a value that has all bits set or unset by carefully choosing the initial arguments. Because an even starting value results in the stream's value quickly going to zero, the value 3 is employed for both values in tests. After four integer multiplications, the result of multiplying the previous two values has the desired bit pattern; subsequent results in the stream maintain the property that some bits are set and some are not set in the 15 most significant bits. The same issue holds true for floating-point and integer division on many modern processor architectures like the Intel x86, Intel64, AMD64 and Power 6. The execution unit exits early when performing division with certain input values. For double-precision division, using initial values of $9.218868E{+}18$ and $4.607182E{+}18$ results in a repeating bit pattern that produces measurements close to the maximum latency on many architectures.
Analysis	Due to the variable accuracy of each architecture's timing mechanism, each stream is run multiple times and its lowest measured execution time is compared against the lowest time for the same-length stream of 32-bit integer additions.
Difficulties	The values used in each stream were determined experimentally, and there may be values that produce even longer times for each arithmetic operation.
Citation	

Compute-Bound Threads

Description	This test determines the number of compute-bound (vs. memory-bound) threads that can run in parallel before performance degrades.
Method	The test starts with a single compute intensive-thread as a base case and increases the number of threads. It measures the time for each set of threads to finish. The test is repeated until the runtime of N threads is at least two times the runtime of a test with only one thread. A compute-intensive thread contains a loop that repeatedly calls a compute-intensive kernel. The microbenchmark runs tests on streams of additions, multiplications, and divisions for both 32-bit and 64-bit integers, as well as for 32-bit and 64-bit floating-point values. The microbenchmark also runs a test on a loop with a mixture of all of the above instructions. The streams used are described in Section 2.2 (page 18). Using the different kernels, the microbenchmark detects whether certain functional units that implement the above mentioned instructions are shared among hardware computing contexts, which is the case, for example, on the Sun Niagara Processor. The Sun Niagara Processor shares a floating-point unit among all computing contexts, so the performance of the floating-point tests degrades earlier (two threads) than the performance of the integer tests (eight threads). All threads synchronize using a barrier. The timing is started in the main (first) thread when the first barrier is reached, and the timing is stopped after the second barrier is reached. The microbenchmark uses a dissemination barrier [50]. A dissemination barrier does not require native memory synchronization operations (as, for example, test&set or compare&swap), which means that the code is portable, performance is good on a variety of platforms, and implement is easy. The test adaptively determines the number of loop iterations that are required to run integer additions for at least one second for the single-thread case.
Analysis	The microbenchmark runs successively more threads, timing each version. When a time that is more than fifteen-percent higher than the single-thread time is detected, the microbenchmark reports the number of threads used in the immediately preceding test.
Difficulties	This microbenchmark is subject to timing challenges similar to other benchmarks. The microbenchmark reports the results along a continuum because the degradation on different architectures can sometimes be gradual and sometimes be abrupt. This method of reporting the results gives a better picture to the compiler of the architecture's behavior. Some systems do not provide a POSIX thread barrier, in which case this benchmark will not work and will produce no output.
Citation	

Memory-Bound Threads

Description	This test determines the number of memory-bound (vs. compute-bound) threads that can run in parallel before performance degrades.
Method	In a manner similar to the method used to measure the size of the data cache (Appendix A, page 102), the memory-bound threads microbenchmark iterates through arrays of memory. The microbenchmark distributes these arrays to an increasing number of threads until performance degrades noticeably. This microbenchmark measures throughput—defined as the number of memory accesses divided by the total time—instead of simply measuring time. This strategy yields a search along two dimensions: memory size and number of threads. For each of a number of data sizes, we find the number of threads that maximizes performance. The value reported is the number of threads that maximized throughput.
Analysis	The main body of the code executes the two-dimensional search, timing each set of iterations. Each iteration is compared against the base case of a single iteration, allowing for a variance of about fifteen percent before concluding that degradation has occurred. The code also has a threshold for saturation; the search is halted at a memory size when the time measurement, within the margin of error, has been the same for too many iterations.
Difficulties	This microbenchmark works best on a system that allows thread binding. On a system where the threads can migrate during execution, measurements are less accurate because the cost of the execution includes refilling the cache for moving threads. The value in this case is conservative.
Citation	

Simultaneous Live Ranges

Description	This microbenchmark measures the maximum number of simultaneous live ranges supported by the architecture, which is essentially the number of registers allocatable by the compiler on the architecture.
Method	The microbenchmark uses a single stream as shown in Figure 2.3 (page 20) to find the number of registers that the compiler will allocate on the architecture. At each step, the register pressure at each instruction is increased by moving the use of each definition to successively more-distant instructions. When the register pressure at each instruction exceeds the number of available registers, the compiler inserts spill code. The microbenchmark compiles each version of the stream to assembly code and then compares the number of assembly-code instructions generated for different versions. Many architectures and operating systems reserve at least one register, so the number of registers returned by this microbenchmark represents the number of registers available for allocation, rather than the total number of registers on the architecture.
Analysis	The code produced by this microbenchmark is never executed, so the usual limitations of the architecture's timing mechanisms do not apply. Instead, the `wc` program is invoked to get a line count of the assembly code produced for each stream. Each version of the stream is the same length, so the line count remains the same until the allocator inserts spill code. The current version tests streams with up to 256 live ranges, which is sufficient to detect the number of simultaneous live ranges supported by current architectures. Although the stream is as simple as possible, the compiler's register allocator may not produce the most efficient allocation for this simple stream. While the results of this microbenchmark are dependent on the quality of the native allocator, the answer will always be conservative: the reported result will always be less than or equal to the number of available registers.
Difficulties	The native allocator must be able to produce a textual form of assembly code that is amenable to line counting. Every compiler we have encountered has this capability. Comments that the compiler inserts into the assembly code can be problematic because the number of comments as a ratio to the total number of instructions can be very high, causing the effect of spill code to be lost in the noise. To make the results more reliable, an `awk` script is invoked to filter out the comments. This requires the system administrator to record the comment character(s) that the compiler uses in its assembly code, prior to running this microbenchmark. The `awk` script handles two comment structures. The first comment structure is similar to the structure of `C++` comments, which start with one or more opening comment characters and continue to the end of the line. The second comment structure is similar to the structure of `C` comments, which start with one or more comment characters and continue until some closing set of comment characters.
Citation	

Appendix B
Automatic Vectorization in the PACE Compiler

The Platform-Aware Optimizer (PAO) analyzes loops for their vectorizabilility. If the PAO determines that a loop is vectorizable, it marks this loop as such and performs analysis and transformations to enable vectorization. The Rose to LLVM translator transfers this information to LLVM IR. The vectorization pass in the TAO uses the analysis information supplied by the PAO to replace scalar LLVM operations by LLVM vector operations, where a cost model determines if it is beneficial to do so. This document describes the interfaces between the components involved and the TAO pass responsible for vectorization.

B.1 Overview

Vectorization in the PACE compiler is performed as a cooperative effort between the PAO and the TAO. The PAO analyzes whether loops are amenable for vectorization, optionally performs transformations to enable vectorization, and analyzes the code to generate information that is needed by the TAO's vectorization pass. The TAO's vectorization pass requires the following types of information:

- *Alignment information* describes which memory accesses (loads, stores) are aligned with respect to the vector unit alignment requirement.
- *Bundles* describe memory accesses to consecutive memory locations, produced by unrolling the loop.
- *Memory dependence information* describes dependences between loads and stores. The PAO builds a dependence graph and passes the information to the vectorizer as dependence edges between memory instructions. Alternatively, the TAO vectorizer can use LLVM's alias analysis pass to build memory dependence edges.

The PAO annotates SAGE III IR with pointers to the above information data structures. For example, when the PAO performs the alignment analysis pass, it annotates each SAGE III IR array load with a pointer to the node representing the alignment information.

The Rose to LLVM translator takes the annotated SAGE III IR as input and translates the annotated SAGE III IR to LLVM IR, transforming the pointer information into LLVM metadata. The vectorization pass in the TAO uses alignment, bundle, and dependence information and transforms

Principal Contacts For This Chapter: Jisheng Zhao, Jisheng.Zhao@rice.edu, and Vivek Sarkar, vsarkar@rice.edu

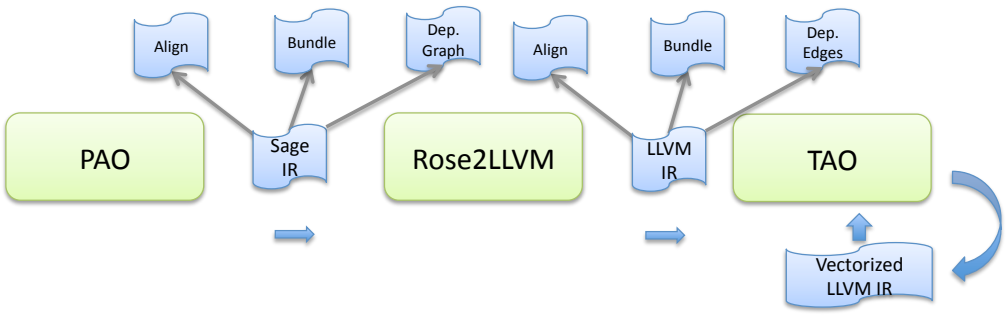


Figure B.1: Vectorization interplay between PAO*, Rose-to-*LLVM *Translator, and* TAO

scalar LLVM instructions to vector LLVM instructions by performing a cost model-based instruction selection algorithm that chooses between scalar and vector instructions.

Figure B.1 illustrates the interplay between the three components.

B.2 Functionality

The PAO recognizes loops that are amenable to vectorization, either because they have statements that can be unrolled and then replaced by vector instructions, or because the loop body contains instructions that can be vectorized. The PAO submodule PolyOpt marks vectorizable loops as such in the SAGE III IR.

The vectorization pass in the TAO operates on straight line code. The PAO tries to generate a longer block of straight line code using the PAO's loop unrolling component. Before unrolling, the PAO uses the PolyOpt submodule to obtain array subscript and dependence information. The loop unrolling component marks accesses to consecutive memory locations, which have been replicated by unrolling, as bundles. For example consider the statement `a[i] =` in a loop body, which, after the loop unroller has unrolled it one time, results in two memory accesses `a[i] =` and `a[i+1] =`. The loop unroller stores the two stores as a bundle.

The loop unrolling component uses the data dependence information to add dependence edge information among the unrolled memory accesses in the loop body. The TAO vectorizer operates on straight line code and, as such, only needs to know which memory access has to happen before which other dependent memory access. The loop unroller translates the dependence graph information, which has dependence edges with a dependence distance, to has-to-happen-before edges in the unrolled loop.

The PolyOpt submodule marks memory accesses with their subscript expressions. The loop unroller uses this information to compute the alignment information of unrolled memory accesses. During unrolling, the PAO annotates the SAGE III IR with links to the analysis data (including the dependence edges and bundles). The process of vectorization in the PAO is illustrated in figure B.2.

During translation of SAGE III IR to LLVM IR, the Rose to LLVM translator converts pointers to the analysis information in the SAGE III IR to information attached as metadata to LLVM's instructions.

Next, the TAO runs its vectorization pass, which examines the loops marked as vectorizable and performs straight line vectorization on them. The vectorization pass uses the analysis information provided by the PAO to guide the instruction selection algorithm that replaces some scalar LLVM instructions by vector instructions. It incorporates instruction cost information provided by the RC. In addition to costs for scalar operations, the cost of vector instructions is also needed. We estimate the cost of a vector instruction by using the cost of its scalar equivalent.

The following two sections describe the input to the TAO's vectorization pass and the output it generates.

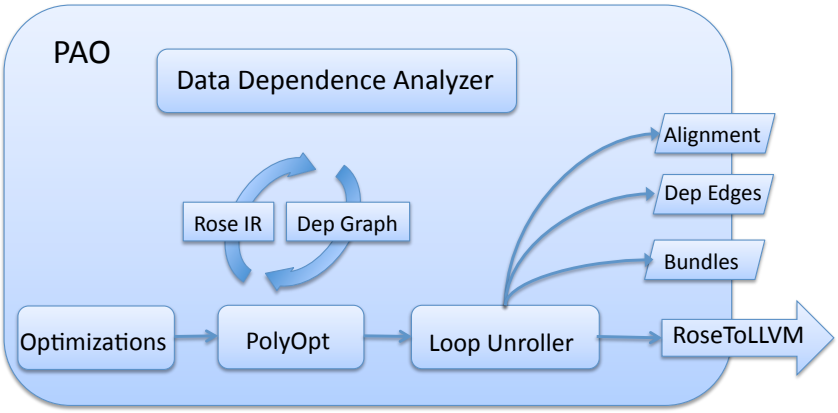


Figure B.2: Vectorization in the PAO

B.2.1 Input

The vectorization pass in the TAO uses the following inputs to perform replacement of scalar operations by vector operations in straight line code.

Scalar LLVM Code The TAO accesses the PAO analysis information as LLVM IR with LLVM metadata that encodes the analysis information.

Vectorizable LLVM IR Loops The PAO marks an innermost loop as vectorizable by annotating the SAGE III IR AST node representing the loop. The Rose to LLVM translator transfers this annotation to the corresponding LLVM IR terminator instructions in all basic blocks within the loop using the metadata tag "`!noivdep !1`".

```
!1 = metadata ! { i1 true }
; ...
for.inc10:
  %.incr15 = add i32 %i.0, 1
  br label %for.cond8, !noivdep !1   ; back edge
```

The vectorization pass in the TAO uses this information to guide which code blocks it should vectorize. It also serves as an indicator that the TAO vectorizer should use the embedded dependence information (`!dep`) instead of using LLVM's alias analysis to approximate dependence information.

Alignment Information Vector instructions involving memory accesses on certain architectures require the access to be aligned at a specific byte boundary. For example, SSE requires memory loads and stores to be 16 byte aligned. Otherwise the programmer must use more expensive unaligned memory move instructions. The PAO tries to generate loops that contain mostly aligned memory accesses, for example, by peeling a loop. The PAO annotates memory accesses (loads, stores), whether they are aligned or not. The TAO accesses this information as LLVM metadata by using the tag `!aligned`.

```
!0 = metadata ! { i1 false }
!1 = metadata ! { i1 true }
; Code: ...  = a[i]; ... = a[i+1];
%elemaddr= getelementptr [2000 x float]* %array, i32 0, i32 %i
```

```
%val = load float* %elemaddr,  !aligned !1, ...
%elem2addr= getelementptr [2000 x float]* %array, i32 0, i32 %iplusone
%val2 = load float* %elemaddr, !aligned !0, ...
```

Memory Dependence Analysis The PAO generates dependence edge information for memory accesses in the loop. It annotates the SAGE III IR memory accesses with has-to-happen-before edges between memory accesses. The Rose to LLVM translator provides access to this information through LLVM metadata by using the tag `!dep`. The has-to-happen-before edges are represented in LLVM IR by a set of incoming edges for every memory access instructions. Every memory access instruction is identified by a unique identifier (a 64 bit value). This is stored in metadata as the first operand of the `!dep` node. All following operands of the `!dep` node represent incoming edges. The example below shows an example where the first store has to happen before the second store.

```
; 0xaffe is the address of the dependency graph node
!3 = metadata ! { /* Dep. source id: */ i64 0xaf10 }
!4 = metadata ! { /* Dep. source id: */ i64 0xaf20, /* Dep. sink id: */ i64 0xaf10 }
; ...
store float* %elemaddr, !dep !3, ...
store float* %elemaddr1, !dep !4, ...
```

Bundles When the PAO unrolls a loop it replicates array accesses. For every array access that it replicates to a contiguous memory location, it builds a tuple that contains all the replicated array accesses. For example, if a `for` loop contains read accesses to `a[i]` and the loop is unrolled four times, the PAO builds a bundle data structure that contains the load of `(a[i], a[i+1], ..., a[i+3])`. Bundles are tuples, so the position is significant. The PAO annotates memory access nodes in SAGE III IR with pointers to their corresponding bundle data structure. The Rose to LLVM translator provides access to the pointers through the LLVM metadata tag `!bun`. The metadata associated with the tag contains not only the pointer but also the index in the bundle tuple. Bundles simplify finding consecutive memory accesses during the vectorization pass in the TAO.

```
!4 = metadata ! { i64 0xf00b, i32 0 } ; 0xf00b is the pointer to the bundle
!5 = metadata ! { i64 0xf00b, i32 1 } ; 1 is the position in the tuple

; Code: ...  = a[i]; ... = a[i+1];
%elemaddr= getelementptr [2000 x float]* %array, i32 0, i32 %i
%val = load float* %elemaddr,  !bun !4, ...
%elem2addr= getelementptr [2000 x float]* %array, i32 0, i32 %iplusone
%val2 = load float* %elemaddr, !bun !5, ...
```

Resource Characterization Information The vectorization algorithm needs the cost of scalar and vector instructions as input to perform instruction selection. It also needs the width of the vector unit to generate vector types of the right size. For example, if the vector length is 128 bits, the vectorization path will try to replace scalar `double` instructions by instructions of the vector <2 `x double`> type.

B.2.2 Output

The vectorization pass replaces scalar instructions by vector instructions if the cost analysis has determined it is beneficial to do so.

```
a0 = load double* %a1ptr, i32 %i,  !bun !4, !aligned !1,
a1 = load double* %a1ptr, i32 %iplus1, !bun !4, !aligned !1, ...
b0 = fadd double %a0, %val1
b1 = fadd double %a0, %val2
```

The TAO vectorization pass would replace the previous code by the following vectorized version.

```
%valvec.0 = insertelement <2 x double> zeroinitializer, double %val1, i32 0
%valvec = insertelement <2 x double> %valvec.0, double %val2, i32 1
a0 = load <2 x double>* %a1ptr, i32 %i, align 16,  !bun !4, !aligned !1,
b0 = fadd <2 x double> %a0, %valvec
```

Note that the pass put the two scalar values, `%val1` and `%val2`, into a vector register and annotated the memory load with the LLVM `align` specification. That alignment specification is necessary so that a native backend will emit an aligned memory move instead of an unaligned one, resulting in better performance.

B.3 Method

Generating good quality vector code for a straight-line piece of IR fragment is paramount to the performance of the program in processors that support a short vector SIMD unit. As stated in prior work, the process of vector code generation can either be easy or cumbersome. As an example of the former, the compiler can find consecutive memory operations, combine them, and subsequently combine their dependent statements until no more instructions can be combined. As an example of the latter, the compiler can use depth-first search, backtracking, and branch-and-bound techniques to find the best possible way of combining operations to generate vector code. In this document we propose a different approach to automatic vector code generation that is driven by a cost model. The cost model guides the vector code generation steps and prunes many search paths that would lead to suboptimal vector code generation. The cost model is combined with a dynamic programming technique to evaluate the best cost for a sequence of IR instructions.

B.3.1 Dynamic Programming

Each TAO IR instruction has an associated cost[1]. As stated earlier in this chapter, the dependence information is readily available from PAO. Using this dependence information, we build a dependence graph at the TAO IR instruction level. A dependence node is an IR instruction and a dependence edge $a \rightarrow b$ implies that b is dependent on a. Such a dependence graph is made single sink by adding synthetic nodes as needed. We use two cost metrics: (1) *scost*: cost of evaluating an operation in scalar fashion; (2) *vcost* is the cost of evaluating some operations in a vector fashion—the number of such operations can be determined by the *vector length* of the underlying machine[2]. When we consider instruction tuples for vectorization we use alignment and bundle information to guide which memory instructions can be put in vector tuples.

Our proposed algorithm starts a bottom-up pass of the dependence graph to compute the costs of evaluating various operations in the dependence graph in both scalar and vector fashion, choosing the minimum cost along the traversal path. The overall cost of the sink node denotes the overall cost of generating vector code. A second top-down pass over the dependence graph identifies those operations that need to be evaluated in scalar fashion and those operations that need to be evaluated in vector fashion. Finally, the vector code for the dependence graph is automatically generated by making a bottom-up pass. A detailed account of this algorithm has been published [7].

The above algorithm needs to pay special attention to dependence graphs that are DAGs rather than trees. Several approaches have been proposed in the literature to break a DAG into trees and then compute the overall cost of each tree. We compute the cost of a tree the first time it is shared between several nodes and use this cost as input for subsequent nodes.

The complexity of the above algorithm is bounded by three passes over the dependence graph.

[1] The cost of each IR instruction is computed relative to the cost of an integer-add IR operation.

[2] RC provides such information to TAO.

Bibliography

[1] F E Allen, John Cocke, and Ken Kennedy. Reduction of operator strength. In Steven S Muchnick and Neil D Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 3, pages 79–101. Prentice-Hall, 1981.

[2] Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[3] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 231–239, New York, NY, USA, 2004. ACM.

[4] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, NY, NY, USA, 1997. ACM.

[5] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, june 1991.

[6] U. Banerjee. Unimodular transformations of double loops. In *Advances in Languages and Compilers for Parallel Processing*, pages 192–219, Irvine, August 1990.

[7] Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. Efficient selection of vector instructions using dynamic programming. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 201–212, Washington, DC, USA, 2010. IEEE Computer Society.

[8] Muthu Baskaran, Albert Hartono, Sanket Tavarageri, Thomas Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *CGO*, April 2010.

[9] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'04)*, pages 7–16, Juan-les-Pins, september 2004.

[10] C. Bastoul. *Improving Data Locality in Static Control Programs*. PhD thesis, University Paris 6, Pierre et Marie Curie, December 2004.

[11] Cedric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, , and Olivier Temam. Putting polyhedral loop transformations to work. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, pages 23–30, 2003.

[12] Cédric Bastoul and Paul Feautrier. Adjusting a program transformation for legality. *Parallel processing letters*, 15(1):3–17, March 2005.

[13] J. A. Benediktsson, P. H. Swain, and O. K. Ersoy. Neural network approaches versus statistical methods in classification of multisource remote sensing data. *IEEE. Trans. Geosci. and Remote Sens.*, 28(4):540, 1990.

[14] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction*, pages 132–146, 2008.

[15] Uday Bondhugula, Albert Hartono, J. Ramanujan, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Programming Languages Design and Implementation (PLDI '08)*, 2008.

[16] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, 1992.

[17] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *PLDI '92 Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language and Design Implementation*, pages 311–321, 1992.

[18] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3), May 1994.

[19] Michael Burke and Linda Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, 15(3):367–399, July 1993.

[20] Martin Burtscher, Byoung-Do Kim, Jeff Diamond, John McCalpin, Lars Koesterke, and James Browne. Perfexpert: An easy-to-use performance diagnosis tool for hpc applications. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[21] *C Language Standard, ISO/IEC 9899:TC3*, 2007.

[22] Candl, the Chunky Analyzer for Dependence in Loops. Available at `http://cse.ohio-state.edu/ pouchet/software/pocc`.

[23] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 185–197, Washington, DC, USA, 2007. IEEE Computer Society.

[24] John Cavazos and Michael F. P. O'Boyle. Method-specific dynamic compilation using logistic regression. *ACM SIGPLAN Notices, Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 41(10):229–240, 2006.

[25] CLooG, the Chunky Loop Generator. Available at `http://www.cloog.org`.

[26] Cristian Coarfa, John Mellor-Crummey, Nathan Froyd, and Yuri Dotsenko. Scalability analysis of SPMD codes using expectations. In *ICS '07: Proc. of the 21st annual International Conference on Supercomputing*, pages 13–22, NY, NY, USA, 2007. ACM.

[27] Keith D. Cooper and Anshuman Dasgupta. Tailoring graph-coloring register allocation for runtime compilation. In *CGO '06 Proceedings of the International Symposium on Code Generation and Optimization*, pages 39–49, March 2006.

[28] Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steve Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Exploring the structure of the space of compilation sequences using randomized search algorithms. *Journal of Supercomputing*, 36(2):135–151, 2006.

[29] Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. ACME: Adaptive compilation made efficient. In *Proceedings of the 2005 ACM SIGPLAN Conference on Languages Compilers and Tools for Embedded Systems (LCTES 05)*, pages 69–77, June 2005.

[30] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. *Proceedings of the ACM SIGPLAN workshop on Languages, compilers, and tools for embedded systems*, pages 1–9, 1999.

[31] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. Operator strength reduction. *ACM Trans. Program. Lang. Syst.*, 23:603–625, September 2001.

[32] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. To appear., 2011.

[33] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, October 1991.

[34] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[35] Paul J. Drongowski. Instruction-based sampling: A new performance analysis technique for AMD family 10h processors. `http://developer.amd.com/Assets/AMD_IBS_paper_EN.pdf`. Last accessed: Dec. 16, 2009., November 2007.

[36] W. H. Farrand, E. Merényi, , J.F. Bell III, J. R. Johnson, S. Murchie, and O. Barnouin-Jha. Class maps of the mars pathfinder landing site derived from the imp superpan: Trends in rock distribution, coatings and far field layering. *The International Journal of Mars Science and Exploration*, 4:33–55, July 11 2008.

[37] P. Feautrier. Dataflow analysis of scalar and array references. *Intl. J. of Parallel Programming*, 20(1):23–53, February 1991.

[38] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *Intl. J. of Parallel Programming*, 21(6):389–420, dec 1992.

[39] Jeanne Ferrante, Vivek Sarkar, and Wendy Thrash. On Estimating and Enhancing Cache Effectiveness. *Lecture Notes in Computer Science*, (589):328–343, 1991. Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing, Santa Clara, California, USA, August 1991. Edited by U. Banerjee, D. Gelernter, A. Nicolau, D. Padua.

[40] Nathan Froyd, John Mellor-Crummey, and Rob Fowler. Low-overhead call path profiling of unmodified, optimized code. In *Proc. of the 19th annual International Conference on Supercomputing*, pages 81–90, New York, NY, USA, 2005. ACM Press.

[41] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations. *IJPP*, 34(3):261–317, June 2006.

[42] M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 106–111, 1998.

[43] B. Hammer and Th. Villmann. Generalized relevance learning vector quantization. *Neural Networks*, 15:1059–1068, 2002.

[44] Albert Hartono, Muthu Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *International Conference on SuperComputing (ICS'09)*, 2009.

[45] E. S. Howell, E. Merényi, and L. A. Lebofsky. Classification of asteroid spectra using a neural network. *Jour. Geophys. Res.*, 99(E5):10,847–10,865, 1994.

[46] F. Irigoin and R. Triolet. Supernode partitioning. In *ACM SIGPLAN Principles of Programming Languages*, pages 319–329, 1988.

[47] Ajay Joshi, Aashish Phansalkar, Lieven Eeckhout, and Lizy John. Measuring benchmark similarity using inherent program characteristics. *IEEE Transactions on Computers*, 55(6):769–782, 2006.

[48] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers'95 Symposium on the frontiers of massively parallel computation*, McLean, 1995.

[49] Jinwoo Kim, Rodric M. Rabbah, Krishna V. Palem, and Weng-Fai Wong. Adaptive compiler directed prefetching for epic processors. In *PDPTA*, pages 495–501, 2004.

[50] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9:21–65, February 1991.

[51] M.J. Mendenhall and E. Merényi. Relevance-based feature extraction for hyperspectral images. *IEEE Trans. on Neural Networks*, 19(4):658–672, April 2008.

[52] E. Merényi. Precision mining of high-dimensional patterns with self-organizing maps: Interpretation of hyperspectral images. In *Quo Vadis Computational Intelligence: New Trends and Approaches in Computational Intelligence (Studies in Fuzziness and Soft Computing, Vol 54, P. Sincak and J. Vascak Eds.)*. Physica Verlag, 2000.

[53] E. Merényi, B. Csató, and K. Taşdemir. Knowledge discovery in urban environments from fused multi-dimensional imagery. In P. Gamba and M. Crawford, editors, *Proc. IEEE GRSS/ISPRS Joint Workshop on Remote Sensing and Data Fusion over Urban Areas (URBAN 2007).*, pages 1–13, Paris, France, 11–13 April 2007. Invited.

[54] E. Merényi, W. H. Farrand, R. H. Brown, Th. Villmann, and C. Fyfe. Information extraction and knowledge discovery from high-dimensional and high-volume complex data sets through precision manifold learning. In *Proc. NASA Science Technology Conference (NSTC2007)*, volume ISBN 0-9785223-2-X, page 11, College Park, MD, June 19 – 21 2007.

[55] E. Merényi, W. H. Farrand, L.E. Stevens, T.S. Melis, and K. Chhibber. Mapping Colorado River ecosystem resources in Glen Canyon: Analysis of hyperspectral low-altitude AVIRIS imagery. In *Proc. ERIM, 14th Int'l Conference and Workshops on Applied Geologic Remote Sensing, 4–6 November, 2000, Las Vegas, Nevada*, 2000.

[56] E. Merényi, K. Tasdemir, and W. Farrand. Intelligent information extraction to aid science decision making in autonomous space exploration. In W. Fink, editor, *Proceedings of DSS08 SPIE Defense and Security Symposium, Space Exploration Technologies*, volume 6960, page 69600M, Orlando, FL, Mach 17–18 2008. SPIE. Invited.

[57] E. Merényi, K. Tasdemir, and L. Zhang. Learning highly structured manifolds: harnessing the power of SOMs. In M. Biehl, B. Hammer, M. Verleysen, and T. Villmann, editors, *Similarity based clustering*, Lecture Notes in Computer Science, LNAI 5400, pages 138–168. Springer-Verlag, 2009.

[58] David Mosberger-Tang. libunwind. `http://www.nongnu.org/libunwind`.

[59] David Patterson. "The Parallel Revolution Has Started: Are You Part of the Solution or Part of the Problem?". Talk at Rice University, February 2010.

[60] Pluto, a Practical Automatic Polyhedral Parallelizer and Locality Optimizer. Available at `http://pluto.sourceforge.net`.

[61] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'08)*, pages 90–100. ACM Press, 2008.

[62] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Conference on Supercomputing (SC'10)*, New Orleans, LA, November 2010. IEEE Computer Society Press.

[63] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop transformations: Convexity, pruning and optimization. In *38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'11)*, pages 549–562, Austin, TX, January 2011. ACM Press.

[64] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Intl. Journal of Parallel Programming*, 28(5):469–498, october 2000.

[65] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–230, 1992.

[66] Rice University. HPCToolkit performance tools. `http://hpctoolkit.org`.

[67] L. Rudd and E. Merényi. Assessing debris-flow potential by using AVIRIS imagery to map surface materials and stratigraphy in cataract canyon, Utah. In R.O. Green, editor, *Proc. 14th AVIRIS Earth Science and Applications Workshop*, Pasadena, CA, May 24–27 2005.

[68] Vivek Sarkar. Automatic Selection of High Order Transformations in the IBM XL Fortran Compilers. *IBM Journal of Research and Development*, 41(3), May 1997.

[69] Vivek Sarkar and Radhika Thekkath. A General Framework for Iteration-Reordering Loop Transformations. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 175–187, June 1992.

[70] Vivek Sarkar and Radhika Thekkath. A general framework for iteration-reordering loop transformations. In *Proc. of the ACM SIGPLAN conference on Programming language design and implementation (PLDI'92)*, pages 175–187. ACM, 1992.

[71] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: improving compiler heuristics with machine learning. *ACM SIGPLAN Notices, Proceedings of the 2003 Conference on Programming Languages, Design and Implementation*, 38(5):77–90, 2003.

[72] Nathan R. Tallent and John Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 229–240, New York, NY, USA, 2009. ACM.

[73] Nathan R. Tallent, John Mellor-Crummey, and Michael W. Fagan. Binary analysis for measurement and attribution of program performance. In *Proc. of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 441–452, New York, NY, USA, 2009. ACM.

[74] Nathan R. Tallent, John Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[75] Nathan R. Tallent, John M. Mellor-Crummey, Laksono Adhianto, Michael W. Fagan, and Mark Krentel. Diagnosing performance bottlenecks in emerging petascale applications. In *Proc. of the 2009 ACM/IEEE Conference on Supercomputing*, 2009.

[76] T. Villmann, E. Merényi, and B. Hammer. Neural maps in remote sensing image analysis. *Neural Networks*, 16:389–403, 2003.

[77] Brian N. West. Adding operator strength reduction to LLVM. Technical Report TR11-03, Rice University, September 2011.

[78] M. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley Publishing Company, 1995.

[79] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.

[80] L. Zhang, E. Merényi, W. M. Grundy, and E. Y. Young. An SOM-hybrid supervised model for the prediction of underlying physical parameters from near-infrared planetary spectra. In R. Miikkulainen, editor, *Advances in Self-Organizing Maps, Proc. 7th Intl Workshop on Self-Organizing Maps (WSOM 2009*, volume 5629 of *Lecture Notes in Computer Science, LNCS*, pages 362–371, St. Augustine, FL, June 8–10 2009. Springer-Verlag.

[81] L. Zhang, E. Merényi, W. M. Grundy, and E. Y. Young. Inference of surface parameters from near-infrared spectra of crystaline H2O ice with neural learning. *Publications of the Astronomical Society of the Pacific*, February 2010. Submitted.