Theme Feature

Changing Interaction of Compiler and Architecture

Program optimizations that have been exclusively done by either the architecture or the compiler are now being done by both. This blurred distinction offers opportunities to optimize perfomance and redefine the compiler-architecture interface.

Sarita V. Adve
Rice University

Doug Burger
University of Wisconsin at Madison

Rudolf Eigenmann
Purdue University

Alasdair Rawsthorne
University of Manchester

Michael D. Smith
Harvard University

Catherine H. Gebotys
University of Waterloo

Mahmut T. Kandemir
Syracuse University

David J. Lilja
University of Minnesota

Alok N. Choudhary
Northwestern University

Jesse Z. Fang
Intel Corp.

Pen-Chung Yew
University of Minnesota

With recent developments in compilation technology and architectural design, the line between traditional hardware and software roles has become increasingly blurred. The compiler can now see the processor's inner structure, which lets architects exploit sophisticated program analysis techniques to hide branch and memory access delays, for example. Processors can now implement register renaming and dynamic instruction scheduling algorithms directly in the hardware—something that was once exclusively the compiler's job.

A similar shift is occurring in optimizing compilers for parallel machines. To parallelize a larger class of applications, compiler writers are moving beyond static transformations that are provably correct and exploring techniques that rely on runtime decisions or hardware support.

This increased blurring of compile-time and runtime optimizations opens many new research opportunities, particularly for program optimization—a task typically performed entirely at compile time. In this article, we describe an optimization continuum with compile time and post-runtime as endpoints and show how different classes of optimizations fall within it. Most current commercial compilers are still at the compile-time endpoint, and only a few research efforts are venturing beyond it. As the gap between architecture and compiler closes, there are also attempts to completely redefine the architecture-compiler interface to increase both performance and architectural flexibility.

OPTIMIZATION CONTINUUM

Program optimization is traditionally viewed as a task to be performed entirely at compile time. Compilers often have access to profile information and can consider platform-specific information when the code is translated. However, because they have no access to runtime information, they must make conservative assumptions so that the program will be correct under all possible runtime conditions. This limits the degree of optimization.

At the other end of the continuum is optimization with knowledge of the runtime environment. Current superscalar microprocessors, for example, perform their own set of code optimizations, such as runtime instruction scheduling. The hardware has perfect knowledge of the runtime environment, but it has lost some analysis information that the compiler knew. This again limits optimization potential.

How optimizations differ

Optimization during compile time consists of two phases. In the *analysis* phase, the compiler gathers information and evaluates if optimization is possible. If so, it then evaluates if the optimization will actually improve performance. In the second phase, *transformation*, if the compiler has determined that an optimization will improve performance, it transforms the code. Thus, a compiler analyzes and transforms the entire program statically.

In a runtime optimization, on the other hand, the hardware must determine only if the optimization is correct and beneficial in a single instance. That is, it analyzes and transforms only the dynamic instruction stream.

Thus, runtime optimization takes a snapshot approach, while compile-time optimization takes a whole-program approach.

At first glance, it may seem reasonable to do optimization solely at runtime. After all, at this point the system should (theoretically) have all the pertinent information. Unfortunately, doing the transformation part of the optimization at runtime causes extra work that may prohibitively increase the program's execution time. An optimization is useful only if the time spent in the analysis and transformation phases does not exceed the reduction in execution time from that optimization. Optimization at compile time is essentially free because it is done before the program is executed. If the hardware could do the transformation, the overhead would be low and the program might even execute faster because of the optimization. However, this "compile time or runtime" view of optimization no longer fits, as we describe later.

0018-9162/97/$10.00 © 1997 IEEE

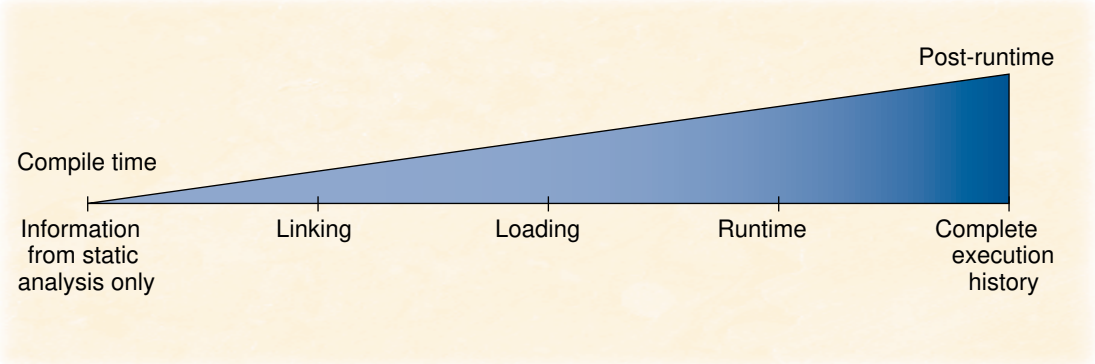

Figure 1. An optimization continuum. As the program progresses from compile time to post-runtime—through linking, loading, and runtime—the system gathers additional information, as the widening bar above the continuum line shows. By post-runtime, it has a complete execution history.

Information flow

In its simplest form, the optimization continuum represents points at which the entire analyze-and-transform process can take place—from compilation to linking to loading to runtime and finally to post-runtime. Figure 1 illustrates these points. As the graduated bar on the continuum shows, the system gains increasing amounts of information as it gets closer to runtime—assuming it loses no information along the way. It begins at *compile time* with information only from static analysis. During *linking*, the code of separately compiled modules and libraries is available for analysis, which lets the system optimize the entire program. During *loading*, it gets additional information about the target machine environment, which means that it can further optimize the program for the specific machine. At *runtime*, the processor knows the exact numeric values of program variables, which means it can analyze and optimize the program to those values. Finally, at *post-runtime*, the system can review the full execution history to adjust for the next execution.

By now the fundamental tension between the two endpoints should be evident: the need for more information drives the optimization process toward runtime; the overhead costs pull it back toward compile time. This tension means researchers and developers must change their view of optimization. It is no longer an atomic process that must occur in its entirety at one point. Rather, it is a process of narrowing choices. At each point, the system should do as much of the transformation as possible given its limited information, offloading some of the optimization overhead from later points. This implies that the parts of the optimization should be broken across the continuum.

Optimization decomposition

If you view optimization as parts scattered along the continuum, the next logical step is to determine what exactly should fall where—that is, how exactly do you decompose optimization to properly balance useful information and optimization overhead? This question has no general answer except that, ideally, cooperation between the compiler and runtime system will allow access to all information in the continuum and at a low overhead.

Cooperation strategies. To see how this cooperation could work, consider a hypothetical database that holds the results of the analyses performed during compilation, linking, loading, and running—that is, the analysis is essentially distributed across many optimization stages. The database contains the important information for transformers, including static and dynamic information about program input variables, the structure of library routines, architecture and execution environment descriptors, and profile data. During transformation, the transformer uses query functions to obtain the results of the pertinent analyses.

Developing such a database is not merely a matter of implementing the entry and query functions. Both entry and query operations introduce overhead that may offset the benefits of the optimizing transformation. Also, how do you encode information in the database? Keeping the encoding close to the data format of the entry and query functions keeps their overhead low, but these formats may vary widely between compiler and machine architectures. For example, what if the machine architecture had to be compatible with old instruction sets? This kind of constraint could greatly limit the database's usefulness.

Suppose you are storing the results of a data dependence analysis. The analysis attempts to determine whether a load and a store access the same address. When the system can determine that the instructions are indeed referencing distinct addresses, many transformations are possible, such as moving code some place that leads to a better instruction schedule, marking a loop as parallel, or conducting a coarser grained speculative execution.

Query efficiency. In many contexts, it is too expensive to gather data reference and range information to determine if two memory accesses conflict. Gathering the access expression, such as the index expression of an array and the ranges for the expression variables, is a reasonable task for a source-level translator, but using classical analyses to extract this information from object code may be prohibitively expensive.

The main difficulty, however, is that neither a source-level translator nor a code-generating compiler has access to all the necessary information. Thus, dependence analysis should occur closer to runtime, combining the advantages of high-level compiler analysis and accurate runtime information. This might be possible if the analysis part of the optimization is distributed over several analysis phases, while the relevant information is exchanged consec-

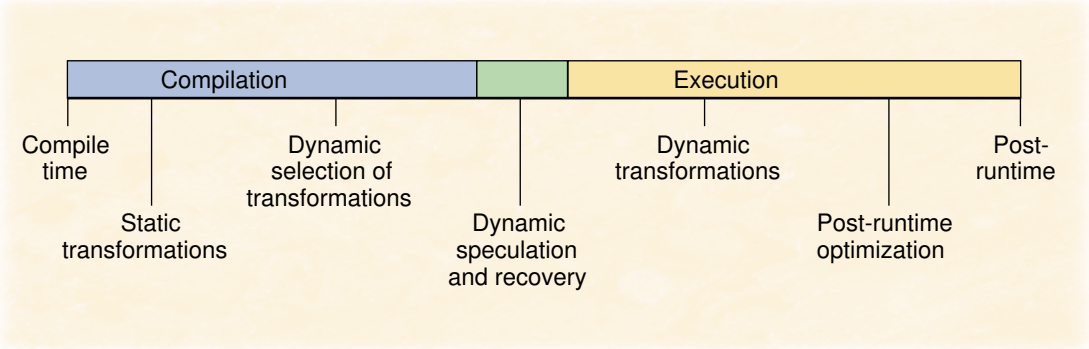

Figure 2. Where various program optimization opportunities fall along the continuum in Figure 1.

utively between phases. This approach, of course, means that different optimizers must cooperate.

Techniques to reduce the overhead of performing the transformation and to improve the efficiency of querying the conceptual database for dependence analyses are important because they will allow the actual transformations to be performed closer to or at runtime.

TAXONOMY OF OPTIMIZATIONS

Figure 2 shows the same continuum as in Figure 1 but with points at which the compiler has opportunities to optimize the program.

Static transformations

In a static transformation, the compiler performs a single transformation at compile time. Most current compiler optimization techniques are static.

Explicit cache control. Most compiler scheduling algorithms assume that either all loads will hit or all loads will miss in the cache, although more recent techniques perform latency-sensitive load scheduling.[1] Nevertheless, to exploit these techniques, researchers must address significant challenges, such as memory disambiguation at compile time, which involves pointer analysis.

One way to reduce the effects of load misses is to improve the locality of accesses. Techniques to do this include *iteration-space transformations, permutation and tiling*, *data-space transformations*, and *hooks* that give the compiler more direct control of the cache. In fact, nearly every cache parameter and policy can be put under compiler control, including the cache-line size, the write policy, the data flush policy, the prefetch policy, whether to bypass the cache, and whether to load data only in the cache's lower levels.

Although recent processors have begun to provide some of these hooks, significant work is required for compilers to keep the techniques from interfering with each other. Further, because some techniques reduce load misses by increasing bandwidth requirements and resource contention, compiler algorithms that use these techniques must do a resource-sensitive analysis to trade off cost with expected benefit. Typically, compilers have done an optimization whenever resources were available; now, however, they must begin to evaluate how using a particular resource will affect another.

Explicit control of disk-memory interface. Many of the issues in explicit cache control also apply to the explicit control of the disk-memory interface. Many scientific applications have large data sets that require out-of-core computation. Their performance is greatly limited by the disk-memory interface, which in current systems is managed entirely by the operating system. Compiler optimizations that maximize page reuse and control various parameters can enhance the interface's performance. These parameters include page size, the translation look-aside buffer, and the page-replacement policy. The challenges here are similar to those of explicit cache control transformations, although approaches that use explicit file I/O can offer the compiler even greater control over data layouts on disks, the access pattern, and the striping style of data across disks.

Dynamic selection of static transformations

With dynamic selection, the compiler generates multiple possible transformations for different cases that it cannot resolve at compile time. It then selects the appropriate transformation when it has enough information, typically at runtime. A simple example of dynamic selection is early microprocessor system code, which had to work both with and without a math coprocessor.

A disadvantage of this technique is that it can produce substantially larger programs because it must have several different code sequences. However, the benefit of choosing the correct transformation may outweigh the overhead associated with the runtime decision and the larger code size. A runtime check may also be expensive, such as in a data dependence test, but hardware support can reduce this overhead.

Runtime disambiguation using static checks. A simple example of runtime disambiguation is to have the compiler insert additional code, such as array-bound checks, so that the processor can verify runtime conditions. It can also insert code checks to verify data dependence assumptions made during optimization.

For example, the RTD runtime disambiguation system[2] inserts conditional statements into the compiled code to check aliasing between indirect memory references. It inserts checks whenever the static alias analysis fails to generate a definitive answer and when rearranging these memory operations would lead to

With dynamic selection, the compiler generates multiple possible transformations for different cases that it cannot resolve at compile time.

a better instruction schedule. One branch of the check leads to an optimized code sequence that assumes no aliasing; the other branch leads to an optimized code sequence that assumes the references are aliased. RTD improved performance on several benchmark applications by 100 to 170 percent. [2] Other researchers could extend this basic idea to create simple code checks for different key pieces of runtime information.

Eliminating unnecessary work. *Common subexpression elimination* is an optimization used at compile time to avoid recalculating values used several times within a small program section. Avinash Sodani and Gurindar Sohi used the same idea for runtime optimization by buffering individual instructions along with their operands and results. [3] When the instructions are later fetched to be reexecuted, the saved result from the previous execution can be used if the current operands match those in the buffer.

With compiler and hardware support, designers can extend this reuse concept to support *hardware "memorization*." Here, the compiler identifies a computational tree containing a sequence of instructions, in which intermediate results are not used outside it. The compiler then identifies the source operands to the tree so that each time it enters the tree, the hardware can check the input operands with the values used in the tree's previous execution. If they all match, the hardware can reuse previous results instead of having to recompute all the instructions in the tree.

The same compiler analysis can benefit DataScalar architectures. [4] These proposed architectures run uniprocessor binaries across multiple processors, each of which is tightly coupled with a fraction of the program's physical memory. Each processor runs the same program, performing redundant computation, and broadcasts needed local operands to all other processors. When the compiler identifies an isolated tree, each DataScalar node can check if it owns all the source operands for the tree. If so, all other nodes branch around the computation. The owner computes the results and broadcasts them to the other nodes.

This technique saves computation at all but one node and thereby reduces total off-chip traffic.

Multiversion loops. For some loops, the compiler can generate two versions, one that could be executed serially and one that could be executed in parallel. At runtime, the compiler chooses the correct version after reading the actual value of the variable that determines the existence of a data dependence.

Another way is to introduce a parallelization *threshold*, the point at which the overhead of parallel execution outweighs any benefit. If a loop does not have a sufficient number of iterations, for example, the compiler will choose the serial loop.

Dynamic recovery of speculative transformations

If the condition that violates a transformation is relatively rare, the compiler can defer the decision-making process and speculate that a transformation is valid. After or while the pertinent section of the code executes, the runtime system can check to verify that the optimization was correct for the conditions that actually occurred. If the conditions were incorrect for the transformation executed, the system must roll back to a point before it executed the offending code section or must repair any changes it made.

As long as the cost of the verification, rollback, and reexecution does not outweigh the benefit of the optimization, the technique will improve overall performance.

Control speculation. In this technique, which is common in current microprocessors, the hardware guesses the outcome of a conditional branch before it can evaluate it. The processor can then continue fetching and issuing instructions down one possible branch path before it has resolved if the path is the correct one. The compiler can move control-dependent instructions above the conditional branches on which the instructions depend. [5] However, it must be able to guarantee, without hardware assistance, that these speculative instructions do not destroy program semantics if it has predicted the outcome of the conditional branch incorrectly.

The requirements of ensuring safe speculation at compile time severely constrain the compiler's ability to move code through the program (global code motions). Several researchers have proposed new architectures that let the compiler perform potentially unsafe code moves by indicating which instructions are speculative and on what condition they depend. With this compiler assistance, the hardware can nullify the effect of the instruction if its speculation was incorrect. *Boosting*[6] is an early example of such an architectural mechanism. Conversely, the Multiscalar processor [7] is an example of how a processor can use compiler support for dynamic speculation, but use only the hardware to track speculative instructions and conditions that signal an incorrect speculation.

Multiscalar processors execute a serial instruction stream on multiple processing elements by having one stage execute part of the program nonspeculatively, while the other processing elements speculatively execute groups of instructions found farther along in the instruction stream.

Data-dependence speculation. The compiler can also speculate on dependences between memory operations, on the result of a load operation, or on any other currently unknown (or unavailable) piece of process state.

The MCB (Memory Conflict Buffer), [8] for example,

lets the compiler move load operations above potentially aliased store operations by maintaining the addresses of speculative loads and checking these addresses against stores it found before the loads but which it reorganized to issue later. When a dependence is violated, the hardware redirects execution to a piece of compiler-generated code that repairs the program state.

Similarly, the *squash buffer*[9] holds loads that are likely to cause rollbacks as a result of a dependence. When the processor finds a suspicious load in the squash buffer, it keeps the load from issuing until all store addresses ahead of it in the reorder buffer have been resolved.

With these types of hardware structures, the processor can speculatively issue the load earlier while recovering from an incorrect speculation (MCB), or it can speculate incorrectly less often (squash buffer).

Speculative parallelization. Sometimes the compiler cannot determine if a construct, such as a loop, is parallel at compile time. In speculative parallelization, the construct in question is instrumented with code that lets the compiler verify whether it should have been executed in parallel. [10] Instrumenting a loop, for example, involves setting bits in shadow copies of arrays and using bit operations on the resulting bit vectors. To minimize the runtime overhead of this scheme, the processor compares the variables whose values determine if the loop can be parallelized from run to run of the same loop. If the values do not change, the chosen optimization for the loop's next execution (either serial or parallel) is the same as for the last execution of the loop.

The overhead decreases even more if the compiler can prove statically at compile time that the decision variables do not change between consecutive loop executions. Instructions that support appropriate bit operations could substantially increase this technique's applicability.

Dynamic transformations

Dynamic transformations delay producing the final optimized code until runtime, although they may use the results of previous analyses. For example, most modern microprocessors now use dynamic instruction scheduling, in which instructions may be issued in an order different from that of the static program. Although the compiler generates an instruction schedule, the hardware dynamically tracks dependences among instructions to allow independent instructions to issue ahead of stalled instructions that reside earlier in the static instruction stream. The processor thus continuously transforms the static instruction schedule into a dynamic schedule that can hide unpredictable latencies, such as cache misses. The overhead of this scheme consists mainly of extra hardware, such as register-renaming logic, a reorder buffer, extra reservation stations, and dependence state. However, with the silicon area now available, the hardware overhead is acceptable.

Dynamic code generation is another way to delay the final generation and optimization of a code section until runtime. For example, the runtime system can use data bound at load time and invariant over a specific program run to perform optimizations such as constant propagation and folding, common subexpression elimination, and branch elimination. This technique has also been used in operating systems to eliminate the repetitive checking of environment variables and to improve execution efficiency in languages that use dynamic type information.

Post-runtime optimizations

Here, an object-code editor performs additional transformations on the object code after execution to improve the program's performance next time it executes. Morph [11] is an example of how post-runtime optimization can work. Morph combines compiling, profiling, executable-rewriting, and operating system technology to produce an environment for profile-driven, machine-specific optimizations. Within a Morph system, optimizations that were previously invoked only at compile time are now automatically, continuously, and transparently applied to executables on a user's machine. Thus, the final stages of optimization can now occur after the user has installed and used the application, making it possible to collect representative execution profiles and to exploit all details of the host hardware during optimization. We describe the Morph architecture in detail later.

REDEFINING THE ARCHITECTURE-COMPILER INTERFACE

The taxonomy of optimizations just described assumes a fairly fixed interface between hardware and software. A fixed interface ensures backward compatibility so that existing software applications are guaranteed to run on new implementations of an architecture. This backward compatibility represents an important commercial goal because developers can take up to several years to develop complex applications and then expect them to be useful for a decade or more. However, what processor developers must realize is that a continued separation of compiler and architecture will inhibit future performance improvements.

Systems that allow for the automatic and continuous optimization of application software offer a clear path to greater performance. These systems are a natural consequence of increasing heterogeneity in processor and memory system implementations, which is particularly evident in digital signal processors, as the sidebar "Heterogeneity in Processor Architectures: A

If the condition that violates a transformation is relatively rare, the compiler can defer the decision-making process and speculate that a transformation is valid.

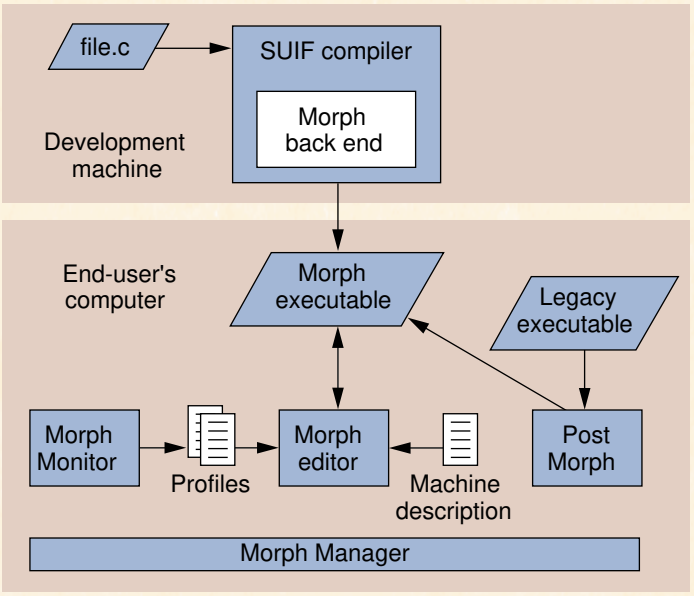

Figure 3. The Morph prototype. Morph provides for post-execution optimization through a combination of compiling, profiling, executable rewriting, and operating system technology.

New Obstacle for Compilation" describes.

Here we describe two optimization environments that are attempting to redefine the architecture-compiler interface by automatically and transparently optimizing the application software.

Morph

Figure 3 shows the general structure of a prototype version of Morph, [11] which was implemented for Digital Equipment's Alpha-based workstations running Digital's Unix. The Morph executable acts as the link between the traditional optimization phase at compile time and the continuous reoptimization phase handled by the Morph components on the end-user's system. This executable includes supplementary information that is normally discarded after compilation but is required for effective reoptimization—for example, a list of target addresses for computed **goto** operations. The Morph editor implements the optimizations Morph provides.

The prototype system implements three code-layout optimizations: branch alignment, procedure splitting, and procedure placement. These optimizations are driven by statistical profile information that the Morph Monitor collects.

The Morph Monitor is an operating system pseudodevice that samples the program counter of all running applications. The prototype has demonstrated that sampling in this manner increases an application's runtime by less than 0.3 percent, which is less than the normal execution-time variability in Digital Unix.

The Morph Manager is a user-level daemon that provides off-line profile management and analysis. It determines when reoptimization should occur. Finally, the PostMorph tool analyzes and rewrites legacy exe-

Heterogeneity in Processor Architectures: A New Obstacle for Compilation

The increasing complexity of microarchitectures is leading to design decisions that compromise homogeneity, making compilation difficult. This issue is already a key research challenge in digital signal processors. DSPs feature small, nonhomogeneous register sets, specialized functional units, restricted connectivity, and highly irregular data paths. They must also meet tight timing constraints with very little code, typically about a thousand instructions. Conventional techniques for commercial compiler-generated code tend to produce very inefficient code for these processors. In fact, hand-coding for DSPs can triple perfomance and result in more compact code. [1]

Thus, most code for DSPs is now written manually. However, the increasing complexity of applications, the demand for shorter time to market, and the lower development and maintenance costs of modern DSPs are pushing designers toward the use of high-level language compilation.

Unfortunately it is extremely difficult to design an appropriate compiler, given the combination of heterogeneous architectures, timing limitations, and small code size. Moreover, many embedded systems applications and general computing environments require low power use, which constrains the design of both hardware and software even more.

Although performance improvements generally also improve power dissipation, studies show that for a given performance there may be additional power savings from the way instructions are selected and scheduled. Some tools now support a low-power analysis of architectures, but low-power software compilation has only recently emerged as a research area. The current trend is toward cleaner DSP architectures (typically very long instruction width), although compilation is expected to remain a challenge because of constraints particular to DSP applications.

Increased heterogeneity is beginning to appear frequently in general-purpose processors as well. For example, almost all microprocessor vendors now include some type of specialized support for multimedia. With very small increases in cost, these processors have incorporated significant increases in performance for such applications. The latest wide-issue superscalar processors are also clustering register banks with certain functional units and creating nonorthogonal forwarding paths, much like early very-long-instruction-width machines. Thus, in the near future general-purpose processors are likely to face compilation challenges similar to those DSP designers are now attempting to overcome.

Reference

1. C. Gebotys and R. Gebotys, "Complexities in DSP Software Compilation: Performance, Code Size, Power, Retargetability," in *Proc. Hawaii Int'l Conf. Systems Science*, IEEE CS Press, Los Alamitos, Calif., to be published in 1998, pp. 150-156.

cutables to recover discarded information required by the automatic optimization process and to provide the system with a way to evolve.

Dynamite

Dynamite [12] is an execution environment designed for experimentation with pure runtime optimizations. Dynamite operates on a program consisting of *subject instructions*. These instructions are never directly executed by the underlying target processor; instead, each instruction is translated into an intermediate representation when it is first executed. As the processor encounters jumps, the environment translates the intermediate representation for the previous block of instructions into a block of target instructions and executes it. When the same instruction is reexecuted, the processor can use the translated instructions directly. The environment also dynamically profiles the running program, and as it detects hot spots (groups of frequently used instruction blocks), it invokes an optimizer to generate higher quality code for these spots. The optimizer uses the intermediate representation generated by the initial translator, and benefits from the analysis carried out during the initial translation.

As execution proceeds, Dynamite combines larger and larger groups of blocks, often giving more and more scope for optimization. When conventional optimization reaches its limits in innermost blocks, Dynamite investigates value-specific optimizations that create special-purpose sections of code tailored to the program's current behavior. It minimizes the cost of analysis by limiting its scope to the group of blocks currently being optimized. It avoids analysis that must proceed beyond this scope by planting the appropriate tests at the entry to the optimized group. Early results show speedups of two to four on key inner program loops. A major goal of this research is to make these speedups available over whole program runtimes.

Because no subject instruction is ever directly executed, the subject and target architectures need not be the same. In fact, the Dynamite system is constructed with replaceable front and back ends to construct a family of cross-platform dynamic binary translators.

Many of the optimizations we have described require new hardware support and are thus not compatible with older machines. The large installed base of legacy binaries would also fail to take advantage of many of the new hardware mechanisms that the research community has proposed. The rate of microprocessor innovation will determine if architectural independence is worth the price. A slow, gradual introduction of new features would give software time to catch up. Conversely, a relentless unfolding of new hardware features would mean systems must not require a complete recompilation to exploit each new feature. The industry is closer to the second scenario and is likely to grow even more so, as on-chip resources increase by orders of magnitude. We thus expect to see even more radical interaction changes—an even fuzzier line—between architecture and compiler. ❖

References

1. J. Lo and S. Eggers, "Improving Balanced Scheduling with Compiler Optimizations That Increase Instruction-Level Parallelism," in *Proc. Conf. Programming Language Design and Implementation*, ACM Press, New York, 1995, pp. 151-162.
2. A. Nicolau, "Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies," *IEEE Trans. Computers*, May 1989, pp. 663-678.
3. A. Sodani and G. Sohi, "Dynamic Instruction Reuse," in *Proc. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 194-205.
4. D. Burger, S. Kaxiras, and J. Goodman, "Datascalar Architectures," in *Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 338-349.
5. H. Hwu et al., "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *J. Supercomputing*, Jan.-Feb. 1993, pp. 229-248.
6. M. Smith, "Architectural Support for Compile-Time Speculation," in *Architectural Support for Compile-Time Speculation*, D. Lilja and P. Bird, eds., Kluwer Academic, New York, 1994, pp. 13-49.
7. G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar Processors," in *Proc. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 414-425.
8. D. Gallagher et al., "Dynamic Memory Disambiguation Using the Memory Conflict Buffer," in *Proc. Symp. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1994, pp. 183-193.
9. A. Moshovos et al., "Dynamic Speculation and Synchronization of Data Dependences," in *Proc. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 181-193.
10. W. Blume et al., "Parallel Programming with Polaris," *Computer*, Dec. 1996, pp. 78-82.
11. X. Zhang et al., "System Support for Automatic Profiling and Optimization," in *Proc. Symp. Operating Systems Principles*, ACM Press, New York, 1997, pp. 15-26.
12. A. Rawsthorne and J. Souloglou, *Dynamite: A Framework for Dynamic Retargetable Binary Translation*, Tech.

Report UMCS 97-3-2, University of Manchester, Manchester, UK, 1997.

Sarita V. Adve *is an assistant professor of electrical and computer engineering at Rice University. She received a BTech in electrical engineering from the Indian Institute of Technology, and an MS and a PhD in computer science from the University of Wisconsin, Madison.*

Doug Burger *is a PhD candidate at the University of Wisconsin at Madison. He received an MS from the University of Wisconsin, Madison and a BS from Yale University, both in computer science.*

Rudolf Eigenmann *is a faculty member of the School of Electrical and Computer Engineering at Purdue University and chairman of the High-Performance Group of the Standard Performance Evaluation Corp. He received a PhD in electrical engineering/computer science from ETH Zurich.*

Reader Service Number 7

Alasdair Rawsthorne *is a lecturer in computer science at the University of Manchester. He received a BSc in electronic engineering from the University of Southampton.*

Michael D. Smith *is an associate professor of electrical engineering and computer science in the Division of Engineering and Applied Sciences at Harvard University. He received a BS in electrical engineering and computer science from Princeton University, an MS in electrical engineering from Worcester Polytechnic Institute, and a PhD in electrical engineering from Stanford University.*

Catherine H. Gebotys *is an associate professor of electrical and computer engineering at the University of Waterloo. She received a BASc in engineering science and an MASc in electrical engineering, both from the University of Toronto, and a PhD in electrical engineering from the University of Waterloo.*

Mahmut T. Kandemir *is a PhD candidate in electrical engineering and computer science at Syracuse University. He received a BS and an MS, both in computer engineering, from the Technical University of Istanbul.*

David J. Lilja's biography appears on p. 50.

Alok N. Choudhary *is an associate professor of electrical and computer engineering at Northwestern University. He received an MS from the University of Massachusetts, Amherst, and a PhD from the University of Illinois, Urbana-Champaign—both in electrical and computer engineering.*

Jesse Z. Fang *is manager of a compiler and tools research group at Intel's Microcomputer Research Labs. Fang received a PhD in computer science from the University of Nebraska, Lincoln.*

Pen-Chung Yew *is a professor of computer science at the University of Minnesota and former associate director of the Center for Supercomputing Research and Development at the University of Illinois, Urbana-Champaign.*

Contact Lilja at the ECE Dept., University of Minnesota, 200 Union St., SE, Minneapolis, MN 55455; lilja@ece.umn.edu.