The Interaction of Architecture and Compilation Technology for High-Performance Processor Design

Sarita Adve
Electrical and Computer Engineering
Rice University

Doug Burger
Computer Science
University of Wisconsin
Madison

Rudolf Eigenmann
Electrical and Computer Engineering
Purdue University

Alasdair Rawsthorne
Computer Science
The University of Manchester

Michael D. Smith
Division of Engineering and Applied Sciences
Harvard University

Catherine Gebotys
Electrical Engineering
University of Waterloo

Mahmut Kandemir
Electrical Engineering and Computer Science
Syracuse University

David J. Lilja
Electrical Engineering
University of Minnesota
Minneapolis, MN

Alok Choudhary
Electrical and Computer Engineering
Northwestern University

Jesse Fang
Intel Corporation

Pen-Chung Yew
Computer Science
University of Minnesota
Minneapolis, MN

April 11, 1997

Corresponding author:
David J. Lilja
Department of Electrical Engineering
University of Minnesota
200 Union St. SE
Minneapolis, MN 55455

Phone: 612-625-5007
FAX: 612-625-4583
E-mail: lilja@ee.umn.edu

1 Introduction

Previous decades have seen dramatic growth in computer performance due to a combination of higher density devices, architectural innovations, and developments in compilation technology. While a computer's architecture can be defined as the line that divides what is implemented in hardware from the operations implemented in software, this line has become increasingly blurred as significant complexity has shifted back and forth between the hardware and the compiler. For example, this shift has exposed the inner structure of the processor to the compiler, allowing sophisticated program analysis techniques to be exploited to hide branch and memory access delays. At the same time, current processors implement register renaming and dynamic instruction scheduling algorithms directly in hardware, which previously had been the exclusive domain of the compiler. A similar shift is occurring in optimizing compilers for parallel machines, where, to parallelize a larger class of applications, compiler writers are moving beyond static transformations that are provably correct and exploring techniques that rely on run-time decisions or hardware support.

The increased blurring of the distinction between compile-time and run-time optimizations raises a variety of new research opportunities. This paper attempts to both identify a general framework for such research as well as identify specific research directions. We begin by describing program optimization as a continuous process that operates over the continuum spanning from the initial compilation of the application up to and including the execution of the application binary. We then provide a taxonomy to categorize the different classes of optimizations within this continuum. For each category, we provide examples from the literature, as well as examples that represent possible future research directions. Finally, we discuss ongoing efforts that completely redefine the hardware-software interface to provide both high performance and architectural flexibility, and conjecture on future research directions enabled by these efforts.

2 The Optimization Continuum

Program optimization is most commonly thought of as a task to be performed entirely at compile-time. Though compilers often have access to profile information during analysis and can consider platform-specific information during the code transformation process, their access to run-time information is severely limited. Thus, they must make conservative assumptions so that correctness under all possible run-time conditions can be guaranteed. Furthermore, today's dynamically scheduled microprocessors perform their own separate set of code optimizations, such as run-time instruction scheduling. Although the hardware has perfect knowledge of the run-time environment, it has lost some analysis information that the compiler knew, which limits its optimization potential. These two optimization approaches can be viewed as end-points in a continuum of program translation steps where decisions about performance optimizations are made.

2.1 The range of opportunity

To better understand the possible points in this continuum, it is helpful to think of an optimization as a process involving the following two phases: an *analysis phase*, where information is gathered and evaluated to determine if an optimization is possible and will produce a performance improvement; and a *transformation phase*, where the actual code transformation is performed. Typically, when the compiler performs an optimization, it analyzes the source code, and, if it determines that an optimization is correct

and beneficial in most cases, it will transform the static code. A run-time optimization performed by the hardware, on the other hand, needs to determine only if the optimization is correct and beneficial in that single instance. That is, the hardware analyzes and transforms the dynamic instruction stream, while the compiler analyzes and transforms the static program executable.

In the simplest form, we can view this continuum as separate points in time where the entire analyze-and-transform process can take place, for instance, at link-time, load-time, and run-time. The motivation for these later optimizations is that the availability of information increases as we go from compile-time to run-time, assuming nothing learned in the earlier phases is thrown away. At link-time, the code of separately compiled modules and libraries is available for analysis, enabling whole-program optimization. At load-time, information about the target machine environment becomes available, enabling further machine-specific optimizations. At run-time, the exact numeric values of program variables are available, enabling exact program analysis and value-specific optimizations.

The reason that optimizations are not performed solely at run-time, when all information is theoretically available, is that performing the transformations introduces overhead. At run-time, transformations create extra work that does not contribute directly to the result of the program, rather, the work is done to speed the actual computation. Thus, any improvement attributed to an optimization must be reduced by the cost of the analysis and transformation phases. Compile-time optimization is essentially free while run-time optimization using conventional algorithms is prohibitively expensive.

There is a fundamental tension between these two endpoints in that the need for more information drives the optimization process toward run-time, while the overhead costs of the optimization process pull it back toward compile-time. To achieve the best performance after optimization, we must balance the potential benefit of greater information against the overhead of acting on that information. Discarding the traditional viewpoint, we view optimization not as an atomic process that must occur in its entirety at one point in time, but as 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 the later points.

2.2 Optimization decomposition

Viewing the optimization process as a continuum raises the question of how to determine what is the right point in the continuum, or the right decomposition for an optimization, to achieve the proper balance between useful information and optimization overhead. This question has no single answer. This way of viewing optimizations implies that pieces of the analysis and transformation phases for a single optimization can be spread across the continuum, from compile-time to run-time. Cooperation between the compiler and the run-time system will ideally result in low overhead access to information only available late in the continuum.

To better illuminate these issues, consider a fictitious database that holds the results of the analyses performed during compilation, linking, loading, and running. The database contains the important information that transformers need to know, such as static and dynamic information about program input variables, the structure of library routines, architecture and execution environment descriptors, and profile data. Conceptually, the results of the analyses are entered into a database, while the transformations use query functions to obtain the results of the pertinent analyses. Developing such a database, which in reality is distributed across the many optimization stages, is not only an issue of implementing the entry and query functions. Both entry and query operations introduce overhead that may offset the benefits of

optimizing transformations. Another relevant question is how the information in the database is encoded. Keeping the encoding close to the data format of the entry and query functions keeps their overhead low. However, these formats may vary widely between compiler and machine architectures. For example, backwards compatibility with old instruction sets imposes constraints that may greatly limit the utility of the database.

An example of an analysis that could be stored in this conceptual database is *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 the movement of code to a place where it leads to a better instruction schedule, the marking of a loop as parallel, or coarser-grained speculative execution.

In many contexts, however, there is a substantial cost associated with gathering data reference and range information to determine whether two memory accesses conflict. Gathering the access expression, such as the index expression of an array and the expression variable ranges, is a reasonable task for a source-level translator, while using classical analyses to extract this information from the object code may be prohibitively expensive. However, since neither a source-level translator nor a code-generating compiler have access to all of the needed information, the dependence analysis is frequently much more accurate when it can be performed closer to run-time.

One solution to this dilemma is to distribute the analysis portion of the optimization over several analysis phases, while passing the relevant information between the phases. This approach, of course, requires cooperation between the different optimizers in the system. Techniques to both reduce the overhead of performing the transformation and improve the efficiency of querying the conceptual database for the dependence analyses are of great importance. These improvements will allow the actual dependence transformations to be performed closer to (or at) run-time and, in this way, combine the advantages of high-level compiler analysis and accurate run-time information. This approach has the potential to make the efficacy of the transformation far greater.

For example, consider a program section in which a store to some address `y` is followed by a load to some address `x` in the initial static schedule. Assume that performance-enhancing transformations are possible if `x` and `y` are different addresses. If the compiler can determine that `x` and `y` are distinct addresses in all cases, then it is possible to generate a *static transformation* that is always correct. If the compiler cannot determine that `x` and `y` are different addresses, it may generate transformations for each of the two possible cases (identical or different addresses) and decide between the two with a run-time check (run-time *selection* of static transformations). If the compiler determines that one of the two cases is much more likely (e.g., `x` and `y` are different), it can generate the transformation assuming that they are different addresses, and instruct the hardware to execute the transformation, rolling back and repairing the state in those rare instances when `x` and `y` are identical addresses (dynamic recovery from a *speculative* transformation). Finally, the compiler may choose to instruct the hardware to create the transformation for the appropriate case at run-time (a fully *dynamic transformation*), once the addresses of `x` and `y` are resolved and determined to be identical or different. Generating such a transformation at run-time will be fruitful only if the generation cost is less than the benefit of the transformation.

3 A Taxonomy of Transformations

Given this continuum-based view of the transformation process, it is useful to categorize these different types of transformations into the following taxonomy:

- **Static transformations**, in which a single transformation is performed at compile-time.
- **Dynamic selection of static transformations**, in which the compiler generates multiple versions (with appropriate conditions for correctness and optimality), with the best version chosen at run-time.
- **Dynamic recovery from speculative transformations**, in which potentially unsafe transformations are executed, followed by a check that the transformation was correct (and a rollback or repair if it was not).
- **Dynamic transformations**, in which the bulk of a transformation is performed at run-time, although the transformer may use the results of previous analyses.

The following subsections present additional examples of each type of transformation listed in this taxonomy.

3.1 Static transformations

The majority of existing compiler techniques are static optimizations performed once at compile-time. They include a large body of established scalar and parallel optimizations [BENP93] that are applied by source-to-source restructurers and code-generating compilers. The following examples discuss several current issues with static optimizations.

Load scheduling

The common memory system architecture for current machines consists of one or more levels of on-chip and/or off-chip cache. Given current instruction window sizes, out-of-order processors are not very successful at hiding high latency load misses. For instance, a processor with a 100 cycle load miss penalty that is capable of issuing 4 instructions in each cycle needs an instruction window of at least 400 instructions to fully overlap the latency of the load. One way of reducing the impact of load misses is to schedule multiple load misses within one instruction window so that they can be overlapped [PRA97]. Most compiler scheduling algorithms have assumed either that all loads will hit or that all loads will miss in the cache, although more recent techniques perform latency-sensitive load scheduling [LE95]. Nevertheless, significant challenges, such as compile-time memory disambiguation involving pointer analysis, must be addressed to exploit these techniques well.

Explicit cache control

The impact of load misses can be reduced by improving the locality of accesses using 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 lower levels of caches. While recent

processors have begun to provide some of these hooks, significant work is required for compilers to take advantage of them in a holistic fashion to prevent the various techniques from interfering with each other. Further, since some of the techniques reduce load misses by increasing bandwidth requirements [BGK96] and resource contention, compiler algorithms that use these techniques must perform a resource-sensitive analysis to trade-off the cost with the expected benefit.

Explicit control of the disk-memory interface

While the above discussion focuses mostly on the cache-memory interface, similar observations also apply to the disk-memory interface. Many scientific applications have large data sets requiring out-of-core computation. The performance of these applications is greatly limited by the disk-memory interface which in current systems is managed entirely by the operating system. The performance of the disk-memory interface can be enhanced by compiler optimizations that maximize page reuse and control various parameters, such as the page size, the TLB, and the page replacement policy. The challenges here are similar to those of cache optimizations, although approaches that use explicit file I/O can offer even greater control to the compiler of data layouts on disks, the access pattern, and the striping style of data across disks.

3.2 Dynamic selection of static transformations

With *dynamic selection*, the compiler generates multiple possible transformations for different cases that it cannot resolve at compile-time. The appropriate transformation is then selected when enough information becomes available, typically at run-time. This technique may produce substantially larger programs, but it may also produce a performance advantage if the overheads associated with the run-time decision and the larger code size are less than the benefit obtained from choosing the correct transformation. Since the run-time check may be expensive, hardware support to reduce the overhead of this check often may be useful.

Run-time disambiguation using static checks

Probably the simplest example of code checks on run-time conditions involves the insertion of array bounds checks. Additionally, a compiler can insert similar types of code checks to verify data dependence assumptions made during the optimization process. For example, the run-time disambiguation (RTD) system [Nic89] inserts conditional statements into the compiled code to check aliasing between indirect memory references. These checks are inserted whenever the static alias analysis fails to generate a definitive answer and the rearrangement of these memory operations leads to a better instruction schedule. One branch of the check leads to an optimized code sequence that assumes no aliasing, while the other branch leads to an optimized code sequence that assumes that the references are aliased. This approach improved performance on several benchmark applications by 100-170% [Nic89]. This basic idea could be extended to create simple code checks for other key pieces of critical run-time information that would lead to better optimized code sequences.

Dynamic memory control

The loop-blocking optimization transforms a program to use a primary memory working set that fits in the cache, whereas the original code has a primary working set much larger than the cache. One issue with performing such an optimization at compile time is that it presupposes a particular cache size. This problem can be alleviated by generating code that either uses these unknowns as parameters, or by generating several distinct versions of the code [CM95]. Since a potentially large number of code versions may be produced,

the selection of the cache-specific code could be made at load-time by replacing the cache-size parameter with a constant. Note that guessing wrong in this case only hurts performance, not correctness. Another (hitherto unexplored) possibility for improving cache performance is to put partitioning hooks in a cache, so that when the compiler has enough information to manage it explicitly, the compiler generates instructions to partition the cache, subsequently accessing one partition like registers or memory.

Eliminating unnecessary work

The *common subexpression elimination* optimization is used at compile-time to avoid the recalculation of values that are used several times within a small section of a program. A similar idea can be used at run-time by buffering individual instructions along with their operands and results [SS97]. When the instructions are later fetched to be executed again, the saved result from the previous execution can be used if the current operands match those in the buffer. With compiler and hardware support, it is possible to extend the reuse concept to support *hardware memorization*. With this technique, the compiler identifies a *computational tree*, which is a sequence of instructions where intermediate results within the tree are not used outside of the tree. It then identifies the source operands to the tree so that each time the tree is entered, the hardware can check the input operands with the values used in a previous execution of the tree. If they all match, the previous result can be reused instead of having to recompute all of the instructions in the tree.

The same compiler analysis can benefit DataScalar architectures [BKG97]. These 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 which, since they are running the same program, will also need those operands. When the compiler identifies an isolated tree, each DataScalar node can check to see if it owns all of 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.

Multi-version loops

Parallelizing compilers can apply a multiversion loop transformation technique in several situations. A common example is where the existence of a data dependence is determined by an unknown variable. In this case, the compiler generates both a serial and a parallel version of the code, and chooses between them at run-time, depending on the actual value of the variable. Another example is the "parallelization threshold" introduced by some parallelizing compilers. If a loop does not have a sufficient number of iterations, for instance, a serial loop variant will instead be chosen since the overhead of parallel execution would outweigh any benefit.

3.3 Dynamic recovery for speculative transformations

If the condition that violates a transformation is relatively rare, we can defer the decision-making process and speculate that a transformation is a valid one. After the execution of the code section in question, the run-time system can check to verify that the optimization was correct. If the transformation was incorrect, the system must roll back to a point in the execution before the execution of the offending code section. As long as the cost of the verification, roll-back, and re-execution is less than the benefit of the optimization, the speculation will improve overall performance.

Control speculation

Control speculation, which is a common technique found in today's microprocessors, guesses the outcome of a conditional branch before its evaluation is possible. This speculation allows the microprocessor to continue fetching and issuing instructions down one possible path of a branch before it has resolved whether the path is the correct one. With control speculation, the compiler can move control-dependent instructions above the conditional branches on which the instructions depend [Fis81, H$^+$93]. Without hardware assistance, the compiler must guarantee that these speculative instructions do not destroy program semantics if the outcome of the conditional branch is predicted incorrectly.

The requirements of ensuring safe compile-time speculation severely constrain the ability of the compiler to perform global code motions. As a result, several researchers have proposed new architectures that allow the compiler to perform potentially unsafe code motions 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 the speculation was incorrect. *Boosting* [Smi94] is one early example of such an architectural mechanism. The Multiscalar processor [SBV95], conversely, is an example of a processor that uses compiler support for dynamic speculation, but uses the hardware to keep track both of which instructions are speculative, and of conditions that signal an incorrect speculation. Multiscalar processors execute a serial instruction stream on multiple processing elements by having one stage execute a portion of the program nonspeculatively, while the other processing elements speculatively execute groups of instructions found farther ahead in the instruction stream.

Data-dependence speculation

In addition to speculating on conditional branch outcomes, it is also possible to speculate on dependences between memory operations, on the result of a load operation [LWS96], or on any other currently unknown (or unavailable) piece of process state. The *Memory Conflict Buffer* (MCB) [GCM$^+$94], for example, enables the compiler to move load operations above potentially aliased store operations by maintaining the addresses of speculative loads and checking these addresses against stores originally found before the loads but reorganized to issue later. When a dependence violation occurs, the hardware redirects the execution to a piece of compiler-generated code that repairs the program state. Similarly, the *squash buffer* [MBVS97] holds loads that are likely to cause rollbacks as a result of a dependence. When a suspect load is found in the squash buffer, it is prevented from issuing until all store addresses ahead of it in the reorder buffer have been resolved. With these types of hardware structures, the compiler or hardware can speculatively issue the load earlier while recovering from a misspeculation with the MCB, or misspeculating less often with the squash buffer.

Speculative loop parallelization

Another speculative transformation is *speculative loop parallelization* in which a loop that cannot be determined to be parallel at compile-time is instrumented with code that allows the verification of the correctness of a speculative parallel execution of the loop [BDE$^+$96]. To minimize the run-time overhead of this scheme, the variables whose values determine if the loop can be parallelized are compared from run to run of the same loop. If the values do not change, then the chosen optimization for the next execution of the loop (i.e. either serial or parallel) is the same as for the last loop invocation. The overhead can be reduced further if it can proven statically at compile-time that the decision variables do not change between consecutive executions of the loop. Finally, the instrumentation of the loop involves setting bits in shadow copies of arrays and using bit operations on the resulting bit vectors. Instructions supporting appropriate bit operations could substantially increase the applicability of this speculative parallelization

technique.

3.4 Dynamic transformations

Dynamic transformations delay producing the final optimized code until run-time, 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 than the static program order. 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. With the silicon real estate now available, the hardware cost of dynamically generating a schedule is acceptable.

Dynamic code generation is another example where the final generation and optimization of a piece of code is delayed until run-time. For example, data bound at load time and invariant over a specific run of an application program can be used at run-time to perform optimizations such as constant propagation and folding, common subexpression elimination, branch elimination, and so forth [APC+96]. This technique also has been used in operating systems to eliminate repetitive checking of environment variables and to improve execution efficiency in languages that use dynamic type information [CU89].

4 Redefining the Hardware-Software Interface

The taxonomy of compiler optimizations described above assumes a fairly fixed interface between the hardware and the software. A fixed interface ensures backward compatibility so that existing software applications are guaranteed to run on new implementations of an architecture. Since complex applications may take several years to develop and are expected to provide useful service for a decade or more, backward compatibility is a foremost commercial goal. However, as discussed above, there is a clear performance advantage in allowing software to evolve as hardware evolves. Yet, the separation of the compilation process from the run-time optimization process in today's systems inhibits any automatic and persistent evolution of the code in a shipped application.

This section examines the benefits of systems that allow for the automatic and continuous optimization of application software. It begins with a look toward the future of microarchitectures and the problems of increasing heterogeneity in processor and memory system implementation and then discusses two environments for the automatic and transparent optimization of application software. Both increasing system heterogeneity and the development of the optimization environments described below will increase the need for sophisticated code generation and optimization techniques that can effectively handle complex and application-specific hardware structures.

4.1 Heterogeneity in processor architectures

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 DSP processors where hardware features such as small non-homogeneous register sets, specialized functional units, restricted connectivity, and highly irregular datapaths are common [LDK+95, Me95]. The DSP processor must also meet tight timing constraints with a very small code size, typically on the order of 1K instructions. The use of conventional code generation techniques and compilers specifically designed for commercial DSP processors tends to produce very inefficient code. The need for decreasing time to market, development costs, and maintenance costs of modern DSP chips demands the use of high level language compilation. All of these factors create significant challenges for writing efficient and retargetable code generators for such DSP processors.

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 applications. The latest wide-issue superscalar processors are also clustering register banks with certain functional units and creating non-orthogonal forwarding paths, much like the early VLIW machines of the 1980s. Thus, in the near future, compilation challenges similar to those faced by DSP processors are likely to be critical issues for general-purpose processors as well.

4.2 MORPH

Morph [ZWG+97] is a combination of compiler, profiler, executable rewriting, and operating systems technology that provides a practical environment for profile-driven, machine-specific optimizations. Morph provides an environment for optimizations that were previously invoked only during compilation to occur automatically and transparently on the end-user's machine. With Morph, the final stage of optimization occurs after the end-user has installed and used the application, making it possible to consider all details of the host hardware during optimization. It also allows profile-based optimizations to incorporate the idiosyncratic usage patterns of a specific user and track changes in the way a program is used.

Each Morph environment is specific to a single instruction set and programming interface. The centerpiece of this environment is the Morph executable, which is an executable that includes the supplementary information required by the re-optimization process. Profile collection is typically done using an operating system pseudo-device to sample the program counter of all running applications, which increases application run-time by only 0.2 percent. Currently, profile processing and application optimization occur off-line after the application has terminated, although load-time or run-time optimizations could also be used. Finally, the PostMorph tool rewrites legacy executables to contain the annotations needed by the automatic optimization process and to provide the system with a way to evolve.

4.3 Dynamite

Dynamite [RS97] is an execution environment designed for experimentation with pure run-time optimizations. Dynamite operates on a program consisting of *subject instructions* that are never directly executed by the underlying target processor. Instead, every subject instruction is translated into an *intermediate representation* when it is first executed. As jumps are encountered, the intermediate representation for the previous block of instructions is compiled into a block of target instructions and executed. On subsequent

executions of the same instruction, the translated instructions can be used directly. The environment also dynamically profiles the running program, and as it detects "hot-spots" (groups of frequently-used blocks of instructions), it invokes an optimizer to generate higher-quality code for these instruction groups. 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 further, larger and larger groups of blocks are combined, 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 current behavior of the program. The cost of analysis is minimized by limiting its scope to the group of blocks that is currently being optimized. Analysis that needs to proceed beyond this scope is avoided by planting the appropriate tests at the entry to this optimized group. Since no subject instruction is ever directly executed, there is no requirement for subject and target architectures to be the same. In fact, the Dynamite system is constructed with replaceable front-ends and back-ends to construct a family of cross-platform dynamic binary translators.

5 Summary

The old performance optimization model in which all program transformations were performed exclusively at compile-time has been replaced with something more flexible and amorphous. Both the analysis and the transformation portion of an optimization may occur at compile-time, run-time, in between the two, or in any combination thereof. This paper has presented these choices as a continuum of steps, and discussed how an optimization may be made anywhere along this continuum. A taxonomy was also developed that categorized different strategies for performing transformations, specifically statically, selectively, speculatively, and dynamically. Examples from both the literature and untested potential research directions were presented for each of these possibilities. Finally, the effect of how increasing heterogeneity in processor architectures would drive the need for architecturally independent optimizers was discussed, along with examples of two major projects performing research in this area.

Many of the optimizations presented 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 have been proposed. The rate of innovation in microprocessors will determine whether there is a need for architectural independence. A slow, gradual introduction of new features would give software the time to catch up. Conversely, a relentless rate of new hardware feature introduction would mandate systems that do not require a complete recompilation to take advantage of each new feature. The industry's situation currently resembles the latter, and, if anything, will be growing more so as designers have orders of magnitude more on-chip resources available in the near future.

References

[APC+96] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, Effective Dynamic Compilation. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, May 1996.

[BDE+96] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger,

and Peng Tu. Parallel programming with Polaris. In *IEEE Computer*, pages 78–82, December 1996.

[BENP93] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David Padua. Automatic Program Parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993.

[BGK96] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 79–90, May 1996.

[BKG97] Doug Burger, Stefanos Kaxiras, and James R. Goodman. Datascalar architectures. In *Proceedings of the 24th International Symposium on Computer Architecture*, May 1997.

[CM95] S. Coleman and K.S. McKinley. Tile size selection using cache organization and data layout. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[CU89] Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for SELF. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–160, June 1989.

[Fis81] Joseph A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *Transactions on Computers*, C-30(7):478–490, July 1981.

[GCM+94] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wen mei W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 183–193, 1994.

[H+93] H. Hwu et al. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, 7(1/2):229–248, 1993.

[LDK+95] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Code optimization techniques for embedded DSP microprocessors. In *32nd Design Automation Conference*, June 1995.

[LE95] Jack L. Lo and Susan J. Eggers. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1995.

[LWS96] Mikko H. Lipasti, Christopher B. Wilerson, and John P. Shen. Value locality and load value prediction. In *Proceedings of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, 1996.

[MBVS97] Andreas Moshovos, Scott E. Breach, T.N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th International Symposium on Computer Architecture*, May 1997.

[Me95] P. Marwedel and G. Goossens (eds.). In *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[Nic89] Alexandru Nicolau. Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies. *Transactions on Computers*, C-38(5):663–678, May 1989.

[PRA97] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. The Impact of Instruction Level Parallelism on Multiprocessor Performance and Simulation Methodology. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, 1997.

[RS97] Alasdair Rawsthorne and Jason Souloglou. Dynamite: A framework for dynamic retargetable binary translation, Technical Report UMCS-97-3-2. Technical report, The University of Manchester, March 1997.

[SBV95] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 414–425, June 1995.

[Smi94] Michael D. Smith. *Architectural Support for Compile-time Speculation*, pages 13–49. Kluwer Academic Publishers, 1994. edited by D. Lilja and P. Bird.

[SS97] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th International Symposium on Computer Architecture*, May 1997.

[ZWG+97] X. Zhang, Z. Wang, N. Gloy, B. Chen, and M. Smith. Operating System Support for Automatic Profiling and Optimization. In *Submitted to the 16th ACM Symposium on Operating Systems Principles*, October 1997.