Theme Feature

Compilers for Instruction-Level Parallelism

Discovering and exploiting instruction-level parallelism in code will be key to future increases in microprocessor performance. What technical challenges must compiler writers meet to better use ILP?

Michael Schlansker
Hewlett-Packard Laboratories

Thomas M. Conte
North Carolina State University

James Dehnert
Silicon Graphics

Kemal Ebcioglu
IBM T.J. Watson Research Center

Jesse Z. Fang
Intel Corp.

Carol L. Thompson
Hewlett-Packard Laboratories

Instruction-level parallelism allows a sequence of instructions derived from a sequential program to be parallelized for execution on multiple pipelined functional units. If industry acceptance is a measure of importance, ILP has blossomed. It now profoundly influences the design of almost all leading-edge microprocessors and their compilers. Yet the development of ILP is far from complete, as research continues to find better ways to use more hardware parallelism over a broader class of applications.

WHY ILP?

With ever-increasing clock speeds, leading-edge microprocessors are approaching technological limits to processor cycle time. Using ILP improves performance and exploits the additional chip area provided by rapidly increasing chip density.

ILP's key advantage is that it exploits parallelism without requiring the programmer to rewrite existing applications. ILP's success is due to its ability to overlap the execution of individual operations without explicit synchronization. A wealth of opportunities to parallelize programs exist at the fine-grained operation level.

ILP's automatic nature is attractive because it works with current software programs. Despite the rise of novel computer architectures, such as multiprocessors, today's applications are still programmed sequentially, and many will never be rewritten. Sequential performance has enormous *economic* value, which has broadly stimulated commercial interest in ILP.

In a hardware-centric implementation, ILP on a superscalar processor executes a sequential instruction stream. Hardware dynamically detects opportunities for parallel execution and schedules operations to exploit available resources. ILP in a software-centric approach employs a very long instruction word (VLIW) processor and relies on a compiler to statically parallelize and schedule code. Such a partitioning simplifies the hardware. (For a short discussion of hardware architectures, see the "Architectures and ILP" sidebar.)

As chip densities increase, ILP techniques that were previously of use only on supercomputers and minisupercomputers are now broadly applicable to inexpensive and general-purpose computers. Yet exploiting ILP across a diverse set of performance-critical applications will require renewed emphasis on the role of the compiler.

CURRENT ROLE OF ILP COMPILERS

ILP compilers enhance performance by customizing application code to a target processor. Compilers use global knowledge of the application program not readily available to a hardware interpreter as well as a description of the target machine architecture to guide the machine-specific optimizations. Compiler-performed static optimization and scheduling eliminates the complex processing needed to parallelize code, which the hardware would otherwise perform during execution.

ILP compilation is now increasingly important across low- to high-end products as well as general- and special-purpose applications. Compiler techniques originally developed for more specialized products such as minisupercomputers now find more broad use in general-purpose workstations.[1,2] Newly designed embedded processors by Philips, Texas Instruments, and others provide performance using ILP compiler techniques. These uses show that ILP compiler research has already had some commercial success.

ILP compilers originally targeted high performance for loop-oriented scientific applications in which parallelism was abundant and easily recognized. These compilers use trace scheduling or software pipelining to accelerate a broad class of loops with greater efficiency than earlier vector processors.[3,4] For ILP compilers to have a broad impact, however, they must

0018-9162/97/$10.00 © 1997 IEEE

Architectures and ILP

Researchers have discussed the use of ILP to accelerate performance for more than 20 years.[1] Early work addressed the problems of compiling for scientific computers. Machines such as the Floating Point Systems array processor, the Multiflow, and Cydrome all offered substantial amounts of ILP. Each required compiler advances to broadly use parallelism on existing applications.

Superscalar processors

The most visible ILP processors are general-purpose processors, for which superscalar technology is the current design of choice. Several modern superscalar processors, which currently issue about four operations per cycle,[2-5] demonstrate the success of this type of design.

The superscalar processor's dynamic-scheduling capability is especially important for operations with variable latency, such as a load operation with potential cache miss. Superscalar processors also provide object code compatibility, which allows existing applications to run on new machines without recompiling. Object code compatibility is critical to many users who upgrade hardware running applications that are not easily recompiled.

VLIW processors

A key benefit of very large instruction word architectures is their ability to support large amounts of hardware parallelism with relatively simple control logic. While VLIW processors have been built that issued as many as 28 operations per cycle,[6] superscalars have not yet demonstrated such high levels of parallelism. Yet applications that require only modest parallelism may not justify the cost of superscalar hardware, as in the embedded-computing marketplace. In this arena, processors accelerate highly concurrent applications (such as digital signal processing) at low cost. Thus several processors for embedded computing incorporate VLIW technology. Examples of these are the Philips TriMedia, Texas Instruments TMS320C62xx, and Chromatic MPACT.

Common problems

From the perspective of ILP compilers, the architectural debate is irrelevant. Whether for a superscalar processor or VLIW processor, compilers *can* analyze the program to expose parallelism, transform the program to enhance parallelism, and schedule the program to exploit parallelism. No matter the architecture, performance is greatly influenced by the quality of the compiler's generated code. In compilers for any of these processors, the trend is toward the use of a detailed target processor description to achieve the best performance.

References

1. B.R. Rau and J.A. Fisher, "Instruction-Level Parallel Processing: History, Overview and Perspective," *J. Supercomputing*, Vol. 7, No.1/2, 1993, pp. 9-50.
2. D. Papworth, "Tuning the Pentium Pro Microarchitecture," *IEEE Micro*, Apr. 1996, pp. 8-15.
3. K.C. Yeager, "The Mips R10000 Superscalar Microprocessor," *IEEE Micro*, Apr. 1996, pp. 28-40.
4. M. Tremblay and J.M. O'Connor, "UltraSparc I: A Four-Issue Processor Supporting Multimedia," *IEEE Micro*, Apr. 1996, pp. 42-50.
5. D.A. Dunn and W.C. Hsu, "Instruction Scheduling for the HP PA-8000," *Proc. 29th Ann. IEEE/ACM Int'l Symp. on Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 298-307.
6. P.G. Lowney et al., "The Multiflow Trace Scheduling Compiler," *J. Supercomputing*, Vol. 7, No.1/2, 1993, pp. 51-142.

accelerate the nonlooping scalar codes prevalent in most applications. Although trace and superblock scheduling have been successfully used to exploit parallelism in scalar programs,[3,5] research in this area is far from complete.

Optimization criteria

ILP compilation presents technical challenges not addressed in traditional compilers.

Traditional compilers minimize the number of executed operations in a program by eliminating redundancy and selecting efficient sequences of operations. On a sequential RISC processor, this effectively minimizes program runtime.

For ILP processors, decreasing the number of executed operations isn't necessarily the key to reducing runtime. ILP compilers often use techniques that increase performance but do not decrease operation count. In fact, ILP compilers often employ techniques that increase performance but require executing *additional* operations.

ILP compilers use a *processor model* to minimize the number of cycles needed to execute an operation sequence. The processor model allows static optimization for hardware that has visibly exposed parallelism. This means that hardware parallelism is exposed to the compiler so that it can evaluate the cost and benefits of candidate ILP transformations. The model must be similar enough to the target hardware so that optimization for it produces high-quality code for the target processor.

To produce high-quality code, machine models reveal details of the actual processor implementation, such as the number and types of functional units, functional-unit latencies, and other parameters.

Statistical compilation

An important strategy employed by ILP compilers is the use of statistical information to predict the outcome of conditional branches and improve program optimization and scheduling.

Traditional compilers optimize and schedule code without making assumptions about likely control flow paths. Instead, optimization is performed as if all branches are equally likely and only one basic block is scheduled at a time. However, typical programs provide only a small number of operations and a small amount of parallelism within each basic block. Thus, achieved levels of parallelism when blocks are scheduled one at a time are very disappointing.

ILP scheduling techniques use branch statistics to enhance the performance of frequently taken paths or

traces through the program. Performance improves despite the potential expense of longer execution times for infrequent paths, as often happens with trace scheduling. Branch profile information gathered from sample runs assists in trace formation by providing accurate information about the likely flow of control.

Traditional optimizers also made no assumptions about the likelihood of program branches. An ILP compiler can place higher priority on computations with results that are likely to be used, while de-emphasizing those with results less likely to be used. Code produced in this manner often executes fewer operations than that produced by classical optimization.

Statistical information about the location of operands in cache, the probability of a memory alias, or even the likelihood that an operand has a specific value may all be important to future ILP compilers.

PROMISING AREAS OF RESEARCH

ILP compiler techniques are evolving from a scientific-computing technology into a broadly useful scalar technology. However, many obstacles inhibit the efficient use of hardware parallelism in scalar applications, for which

- branches occur frequently and may not be easily predicted,
- programs are often large and may not spend their execution cycles in small local regions, and
- programs often use pointers to reference complex data structures.

In addition, a compiler must exploit scalar parallelism while addressing trade-offs that require complex heuristics. Beneficial techniques can generate detrimental side effects such as unnecessary computation, increased use of registers, and increased code size. However, the rewards for proper use of ILP techniques are substantial performance gains.

ILP represents a paradigm shift that redefines the traditional field of compilation. The rate at which novel technology flows into ILP compilers remains disappointingly low, and several areas of research deserve more attention.

Increasing hardware parallelism

The increasing density of very large scale integration has enabled processor designs that incorporate more functional units. Current designs also attempt to aggressively use more of these units to provide increased hardware parallelism. Companies typically introduce hardware parallelism in high-end products first, then gradually introduce it into lower priced products as chip costs decrease. Thus, technologies once used only in supercomputers or minisupercomputers have become appropriate for workstations, desktops, and finally even low-priced products for embedded systems.

The trend toward increasing hardware parallelism manifests itself in two forms. First, the number of functional units increases as the chip area per functional unit decreases. Second, as clock speeds increase, there is a trend toward deeply pipelined functional units. Most visible in memory reference units, this latter trend will soon lead to inexpensive computers that have memory-access latencies similar to those of the supercomputers of a few years ago. As the discrepancy between processor clock speed and memory access time increases, memory pipeline latencies are increased to allow one or more memory ports to execute a single reference every cycle.

The number of operations "in flight" (those issued but not yet completed) measures the amount of parallelism the compiler must provide to keep an ILP processor busy. Using operations in flight as a measure accounts for both the number and latency of functional units. Early RISC processors, for example, had at most one or two operations in flight. The current trend is toward processors that have 10 to 100 operations in flight. As we scale up the amount of hardware parallelism, compilers take on increasingly complex responsibilities to ensure efficient use of hardware resources. Techniques that are inappropriate or of little consequence for sequential or modestly parallel processors become critically important with higher levels of hardware parallelism.

The number of operations "in flight" (those issued but not yet completed) measure the amount of parallelism the compiler must provide to keep an ILP processor busy.

Dynamic compilation

Traditional ILP focuses on static compilation, which targets a specific processor architecture. Static compilers often tune code to a single implementation of that architecture or even a specific configuration of system memory. Obtaining optimum performance thus requires compiling applications for each specific system configuration. This complicates sales, support, and networked distribution of software.

Dynamic compilation transparently customizes an executable file during execution. It can use information collected from the application data set, the system software configuration, target processor details, or any other information not known when the software was distributed. Dynamic compilation can optimize an executable file for distinct implementations of a single architecture or translate it to run on an entirely new architecture.

Popular programming languages and their environments are also evolving. One key trend is toward object-oriented programming and the use of virtual method calls, which makes it more difficult for a static compiler to exploit parallelism. The increased use of dynamically linked libraries preclude inlining or other techniques that statically and jointly optimize user and library

Dynamic compilation is in its infancy, and researchers are still addressing basic questions about how to best use the technique and its potential capabilities.

code. Dynamic compilation can use information acquired during execution to overcome these obstacles.

Research in dynamic compilation represents a spectrum in which one extreme performs compilation when the program is loaded. The other extreme performs compilation just before execution of a small region of a program. Dynamic compilation is in its infancy, and researchers are still addressing basic questions about how to best use the technique and its potential capabilities.

Program analysis

Program analysis, especially memory reference analysis, is vital to the performance of code generated by ILP compilers. Improved memory reference analysis provides several key benefits.

- Eliminating unnecessary dependences among memory references exposes more parallelism in the program graph and improves program schedules.
- The ability to recognize references to a common memory location enables optimizations such as load and store elimination, improving code quality.
- The ability to differentiate large and small data structures enables code generation techniques that better manage the use of the cache hierarchy.

These and other memory optimizations are especially critical because of the long memory latencies and limited memory bandwidth inherent in modern ILP processors.

Consider the problem of accurately determining dependences among memory reference operations. When memory locations are referenced using pointers, ILP compilers often conservatively assume that pointers might alias or point to the same location when, in fact, they cannot. The compiler inserts dependences between memory references to ensure correctness. On ILP processors, this sequentialization of memory references often degrades the quality of program schedules by a factor of two or more and greatly reduces performance. Analysis techniques that delivered satisfactory results on earlier sequential processors may produce poor results on ILP processors.

We need improved techniques to enhance both the accuracy and the efficiency of memory reference analysis. This problem is especially difficult in languages like C that allow pointers to be passed as parameters to procedures. In this case, inspecting only a single procedure cannot identify relationships among pointers within a procedure. Analysis techniques must analyze more than one procedure or possibly even an entire application. Performing analysis jointly over large amounts of code can be unacceptably slow and consume too much memory. Compiler writers must then limit the scope of the analysis and make conservative assumptions to limit compile time and the complexity of the analysis. Analysis accuracy and application performance often suffer as a result.

Program transformation

Traditional optimizations minimize total operation count, which, for sequential processors, optimizes program performance. Optimizing for ILP processors is not so easy. ILP compilers use a program graph (to represent application parallelism) and a machine model (to represent hardware parallelism) to find fine-grained parallelism. This is an explicit representation of program and hardware parallelism. It allows the compiler to transform and schedule the program to achieve a minimum number of execution cycles on the machine model.

Researchers have developed many nontraditional transformations that support ILP. These include expression reassociation, loop unrolling, tail duplication, register renaming, and procedure inlining. These transformations are examples of techniques that previously received little attention because they provide no substantial benefit in traditional optimization for sequential execution. Further research could improve these transformations as well as identify other transformations to support ILP.

Critical paths thread through both data and control (branch) dependences within a program. The length of a critical path through the program graph limits achievable performance. For these reasons, a scalar application may not initially provide enough parallelism to fully use an ILP processor. However, techniques are being developed that enhance the amount of available scalar parallelism.[6]

Interprocedural transformations are important because procedure-call boundaries present critical bottlenecks to ILP performance. Transformations such as procedure inlining produce complex trade-offs because they both improve the program's schedule but also increase code size. When and how to best implement procedure inlining in general is not well understood, and exciting opportunities exist for partial inlining and interprocedural optimization. These techniques do not inline entire procedures but instead inline the code segments from a procedure that provide the greatest performance improvement.

ILP scheduling

To achieve high performance, ILP compilers must jointly schedule multiple basic blocks. Schedulers typically operate either on entire procedures or on program regions excerpted from a procedure. Regions may also include code taken from multiple procedures due to inlining or other interprocedural code transformations. Common types of region include the

trace, superblock, and innermost loop. Each type favors some control flow paths through the program at the expense of others.

The formation of scheduling regions and scheduling are best performed using control flow statistics. While control flow in looping codes is relatively easy to predict, control flow in scalar programs often is not. Branch profile information provides an important tool for predicting control flow for scalar programs.[7] The use of trace or superblock scheduling is most applicable when branch profile data is available, profiles are stable from one data set to another, and frequent branches are biased—often either repeatedly taken or repeatedly not taken. This allows a branch to be statically predicted as either taken or not taken. When profile statistics are unavailable or branches are balanced (taken at about the same frequency as not taken), inaccurate static branch prediction can lead to premature exit from scheduling regions and poor performance.

Branch profiles are gathered using sample input data to execute instrumented program runs. We can eliminate this unpopular step by using compile time analysis to predict branch profiles. Compile time prediction alone can be improved but may never match the accuracy of sample runs.

An alternate solution might be to develop larger and more general types of regions. It is possible to generalize the linear control flow required by traces and superblocks to support scheduling of nonlinear program regions. Global schedulers, for instance, move code over larger components of a program such as entire procedures.[8] Since such schedulers are not limited to any specific type of region, they may provide more performance when confronted with difficult-to-predict control flow. However, global schedulers must carefully balance execution performance and compilation speed. Algorithmic efficiency is critical when schedulers process large amounts of code with arbitrary program structure. Global schedulers are also at risk of speculatively executing too many operations from paths that are, in fact, never executed.

ILP schedulers address complex trade-offs using heuristics based on approximations needed to achieve acceptable compilation speed. Because these approximations simplify complex problems, they sometimes yield inefficient results. For instance, schedulers may speculate too aggressively, introducing excessive and redundant computation. Scheduling and register allocation interact in a complex manner that can introduce too much register spill code. Code size can also be exaggerated when code is prepared for scheduling using loop unrolling, tail duplication, or procedure inlining. The scheduler may also directly add too much code in the form of compensation code, which glues together adjacent regions. We need techniques to better balance these complex trade-offs.

Although superscalar architectures use hardware to schedule operations dynamically, their performance often depends on the compiler-specified, static operation schedule. Such schedules are sometimes difficult to identify and may not effectively account for the complex interaction between compile time and runtime scheduling. The interaction between the two is not well understood for complex superscalar architectures.

Software pipelining is an effective scheduling technique for accelerating loops, but it can have costly limitations in important situations. For instance, having too many conditionals within a loop may either preclude the use of software pipelining or yield inefficient code. In addition, software pipelines traditionally complete iterations at a rate independent of the path taken through the loop, which unduly penalizes short paths. Advanced software pipelining techniques may alleviate these deficiencies.

To achieve high performance, ILP compilers must jointly schedule multiple basic blocks.

Architectural support

The evaluation of real-application performance is far more difficult than evaluating segments or kernels, for which handcoding machine instructions might suffice. That is why compilers are the only way to evaluate an architecture's performance on real applications.

Novel architectures. To assess new architectures, compilers must incorporate proposed architectural features. Speculation, for example, has long been used to enhance ILP performance by allowing compile time movement of code across basic-block boundaries. Using speculation or other forms of code motion in the presence of exception processing and debugging presents difficult compiler challenges. Exception processing and debugging often reveal the effects of code transformations that should have transparently accelerated code. Processing an exception or returning control to the debugger exposes results inconsistent with the sequential view of the program.

To alleviate this problem, new ILP architectures provide hardware support for speculative execution.[9,10] Such hardware support mechanisms tag data to differentiate legitimate and erroneous data. The hardware allows the speculative movement of code at compile time while presenting the illusion of sequential program execution. Compilers may need to generate complex recovery code to preserve this illusion.

Predicated execution has been introduced to ILP architectures to enhance performance of difficult-to-predict branches.[4,9,11] Predication can also be used to accelerate code containing sequences of dependent branches.[6] Although a few RISC architectures now partially support predication, predicated-execution research is far from complete.

Exploiting a high degree of parallelism also requires highly concurrent register access, yet it is difficult to provide fast, parallel, and global access to a *single* reg-

A variety of ILP-friendly enhancements to existing languages and application development methodologies could improve the performance of future systems.

ister file. So some processor architectures may incorporate distributed register files that restrict access to provide more parallel register access with simpler hardware. This requires additional compiler capabilities to distribute operands across multiple register files. Although the Multiflow processor and compiler use distributed register files for a limited class of applications,[3] further research is needed to facilitate the broad use of distributed register files.

Compilers will also help evaluate advanced superscalar architectures that seek to implement higher levels of parallelism while retaining the benefits of dynamic scheduling in hardware.

Advanced memory architectures. Memory references present especially important challenges for ILP because of the severe penalties associated with memory latencies and cache misses. These effects are very costly in applications that manipulate large data sets. Innovation in both compilers and memory architectures could alleviate these effects.

Memory load operations may have either short latencies to access small data sets (those that fit into the cache) or put up with longer latencies when they access large data sets. For static scheduling, a compiler needs to differentiate these loads. Prefetch is one technique to assist an ILP processor in overlapping long-latency memory references that miss in the cache. This technique decomposes a long-latency load operation into a long-latency cache-line prefetch followed by a short-latency load operation.

Another approach improves the program's ability to control the data flow through the cache hierarchy by targeting memory references to specific cache levels.[9] Compilers must now analyze data references to anticipate the flow of operands through the cache hierarchy and generate code that more efficiently uses the available cache.

Data speculation uses additional hardware to improve the amount of ILP in the presence of potentially aliasing pointers.[9,12] When an alias is possible but unlikely, data speculation allows a load and subsequent uses of its result to move upward across a previous store, which improves the operation schedule. The load operation may now yield an incorrect result because of its adjusted position in the schedule. Hardware detects when an alias occurs, and a correct result is calculated after completion of any stores that might alias.

Existing architectures. Architectures that weren't designed for ILP can still incorporate some ILP techniques. For example, a compiler can schedule an operation speculatively on an existing architecture if it can preclude the introduction of an exception. We can adapt several ILP techniques to provide utility on existing processor architectures.

The introduction of multimedia operations into general-purpose architectures creates important compiler challenges. Many of today's general-purpose processors incorporate multimedia extensions. Multimedia operations support SIMD-like parallelism by packing multiple narrow operands into a single, wide data word. This wide word performs up to eight narrow operations at once. If general-purpose programs are to use these extensions, we must develop new compiler technology that exploits multimedia operations when general-purpose programs operate on narrow data.

Techniques to reduce compile time

Compiler complexity escalates with new hardware features and more complex compiler strategies for improving performance. This often results in long compile times that are unattractive in the marketplace. To alleviate this problem, an ILP compiler must provide a software architecture that partitions applications into regions of manageable size. It also must incorporate a variety of analyses and optimization modules that operate on these regions. Careful application partitioning and better algorithms for analysis and optimization of regions can speed compilation.[13]

Language evolution

Even when applications have sufficient parallelism, the compiler is often unable to exploit it because of obstacles imposed by the programming language. Popular languages like C and C++, for instance, have yet to evolve and assist in parallelization; they now often inhibit ILP use. As ILP becomes increasingly important, language support may improve, given the effect of previous hardware advances on programming languages. For example, the introduction of vector and multiprocessor architectures has profoundly affected scientific programming; Fortran evolved to better support both vectorization and parallel processing.

A variety of ILP-friendly enhancements to existing languages and application development methodologies could improve the performance of future systems. The use of branch profile data is one such enhancement. Its acceptance in measuring performance with SPEC benchmarks indicates a growing industry acceptance of ILP. Another enhancement, compiler directives, can substantially enhance ILP performance for languages like C. In the future, applications may be tuned for ILP execution using directives much like those previously used in tuning C for supercomputers.

Exception-processing protocols define roles for system and user-provided exception handlers. Such protocols can erect huge barriers to ILP performance, almost requiring the sequential execution of all operations. Hopefully, the desire for additional performance will stimulate system developers to adopt exception-processing protocols that support ILP.

Almost all ILP research has studied performance for Fortran and C. Such research has yet to consider newer

languages such as C++ and Java. Future work needs to define appropriate languages and environments for ILP as well as quantify relationships between languages and delivered performance. To be of practical value, language enhancements must minimize any departure from popular programming practice.

We expect ILP to provide benefits to an increasing number of products by using inexpensive hardware parallelism to improve performance. To do so, future processors will rely increasingly on important ILP compiler work that addresses complex issues not yet fully understood.

To advance, ILP compilers will require an enormous research effort, much like the one that drove vector compilers to today's relatively mature status. Thus far, however, ILP has yet to receive a similar investment, even though it presents what may be even more complex technical challenges.

ILP research also requires a substantial software infrastructure—a prototype compiler, a processor simulator, and other tools. Although costly to develop, a number of ILP compilers exist both in academia and in industry, and we are just entering an era when processors supporting ILP are generally available. ❖

References

1. R.L. Lee, A.Y. Kwok, and F.A. Briggs, "The Floating Point Performance of a Superscalar SPARC Processor," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1991, pp. 28-37.
2. J. Ruttenberg et al., "Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler," *Proc. Programming Language Design and Implementation*, ACM Press, New York, 1996, pp. 1-11.
3. P.G. Lowney et al., "The Multiflow Trace Scheduling Compiler," *J. Supercomputing*, Vol. 7, No. 1/2, 1993, pp. 51-142.
4. J.C. Dehnert and R.A. Towle, "Compiling for the Cydra 5," *J. Supercomputing*, Vol. 7, No. 1/2, 1993, pp. 181-227.
5. W.W. Hwu et al., "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *J. Supercomputing*, Vol. 7, No. 1/2, 1993, pp. 229-248.
6. M. Schlansker and V. Kathail, "Critical Path Reduction for Scalar Programs," *Proc. 28th Ann. Symp. Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 57-69.
7. J.A. Fisher and S.M. Freudenberger, "Predicting Conditional Branches From Previous Runs of a Program," *Proc. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1992, pp. 85-95.
8. S.-M. Moon and K. Ebcioglu, "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors," *Proc. 25th Int'l Symp. Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 55-71.
9. V. Kathail, M.S. Schlansker, and B.R. Rau, *HPL Play-Doh Architecture Specification: Version 1.0*, Tech. Report HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, Calif., 1993.
10. G.M. Silberman and K. Ebcioglu, "An Architectural Framework for Supporting Heterogeneous Instruction-Set Architectures," *Computer*, June 1993, pp. 39-56.
11. S.A. Mahlke et al., "Effective Compiler Support for Predicated Execution Using the Hyperblock," *Proc. 25th Ann. Int'l Symp. Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 45-54.
12. D.M. Gallagher et al., "Dynamic Memory Disambiguation Using the Memory Conflict Buffer," *Proc. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1994, pp. 183-193.
13. R.E. Hank, W.W. Hwu, and B.R. Rau, "Region-Based Compilation: Introduction, Motivation, and Initial Experience," *Int'l J. Parallel Programming*, Vol. 25, No. 2, 1997, pp. 113-146.

Michael Schlansker is a department scientist at Hewlett-Packard Laboratories. His research interests include computer architecture, compilers, and embedded systems design. Schlansker received a PhD from the University of Michigan, Ann Arbor. He is a member of the IEEE Computer Society and ACM.

Thomas M. Conte's biography appears on p. 37.

Jim Dehnert is a principal engineer in compiler development at Silicon Graphics Inc. His research interests include code generation and scheduling, software pipelining, register allocation, and optimization. Dehnert has a PhD in applied mathematics from the University of California at Berkeley.

Kemal Ebcioglu is manager of the High Performance VLSI Architectures group at IBM T.J. Watson Research Center. Ebcioglu received a PhD in computer science from the State University of New York at Buffalo.

Jesse Z. Fang's biography appears on p. 69.

Carol L. Thompson is an optimizer architect at Hewlett-Packard's Computer Language Operation. Her interests include computer architecture as well as optimization techniques for instruction level parallelism. She received her masters degree in computer science from the University of California at Berkeley.

Contact Michael Schlansker at Hewlett-Packard Laboratories, Bldg. 3L-5, 1501 Page Mill Rd., Palo Alto, CA 94304; schlansk@hplmss.hpl.hp.com.