The Raw Compiler Project

A. Agarwal, S. Amarasinghe, R. Barua, M. Frank,
W. Lee, V. Sarkar, D. Srikrishna, M. Taylor
MIT Laboratory for Computer Science
Cambridge, MA 02139
http://cag-www.lcs.mit.edu/raw

Abstract

Compilers today are capable of inferring detailed information about program parallelism and analyzing whole-program behavior. However, the traditional interface between the compiler and the processor, as defined by the instruction set architecture (ISA), is unable to communicate much of the compiler knowledge to the processor. The approach taken by modern processors such as superscalars is to incorporate purely run-time algorithms in their hardware to perform analyses and optimizations such as detection of instruction-level parallelism. However, these complex hardware implementations can only exploit a small fraction of the parallelism information available to the compiler.

The Raw architecture developed at MIT aims to maximally utilize the compiler by fully exposing the hardware and by delegating the hardware's control completely to the software system. The Raw microprocessor, a set of simple RISC-like processor tiles interconnected with a high-speed 2D mesh network, does not provide hardware implementations for any of the complex algorithms found in conventional microprocessors. Instead, the compiler and the run-time software system fully orchestrate the Raw hardware resources, and they implement run-time analyses and optimizations tailored to the need of each individual application. This novel approach provides many opportunities and challenges for the Raw compiler and run-time system.

1 Introduction

The advent of RISC revolutionized the microprocessor by making it simpler. This simplicity, achieved mainly by off-loading the task of choreographing complex instructions from the hardware to the compiler, resulted in a cost-effective, high performance processor. However, subsequent generations of microprocessors failed to take heed from this lesson about hardware simplicity. They focused on fully hardware-based approaches for many optimizations, resulting in complex hardware implementations of algorithms such as branch prediction, instruction level parallelism detection, register renaming, and out-of-order execution.

Although these microprocessors clearly benefit from compiler optimizations, they do not take full advantage of the compiler. Performing the analysis at compile time can simplify and eliminate many of the complex algorithms in the hardware. Furthermore, unlike hardware-based approaches which must work under heavy resource and time constraints, compiler-based analysis can be more rigorous, leading to an increase in the effectiveness of the optimizations.

Of course, compiler-based analysis cannot completely eliminate run-time analysis. Run-time analysis is necessary when the information required to perform the optimization is not available at compile-time. But run-time analysis and optimization need not be implemented in hardware. A run-time software system can be utilized in many cases. Unlike hard-coded custom logic, a software system can adapt to the individual requirements of the program, and it can readily accept changes and updates to the algorithms.

There have been numerous attempts to use compiler and run-time software technology to eliminate the complex algorithms from hardware. A prominent example of this approach is the VLIW processor [8]. However, most of these approaches had concentrated on off-loading individual algorithms and optimizations from hardware to software.

We are developing a novel approach that fully exposes the low-level details of the hardware architecture to the compiler. We call machines based on this approach *Raw architectures* because they implement only a minimal set of mechanisms in hardware. These mechanisms are fully exposed to the software, allowing efficient implementations of high-level application programs.

A corollary to the Raw philosophy of keeping the hardware simple is that the software system must bear more responsibilities. The Raw software system includes both a run-time system and a compiler. The run-time system manages dynamic mechanisms historically handled in hardware, including branch prediction, caching, and speculative execution. The Raw compiler faces issues such as resource allocation, inter-tile exploitation of fine-grained parallelism, communication scheduling, and the use of configurable logic.

The remaining sections are organized as follows. Section 2 describes the Raw architecture. Sections 3 and 4 discuss issues, opportunities and challenges in developing the Raw run-time system and the Raw compiler respectively. Section 5 details the current status of the SUIF-based Raw compiler. We close with the project status and conclusions in Section 6.

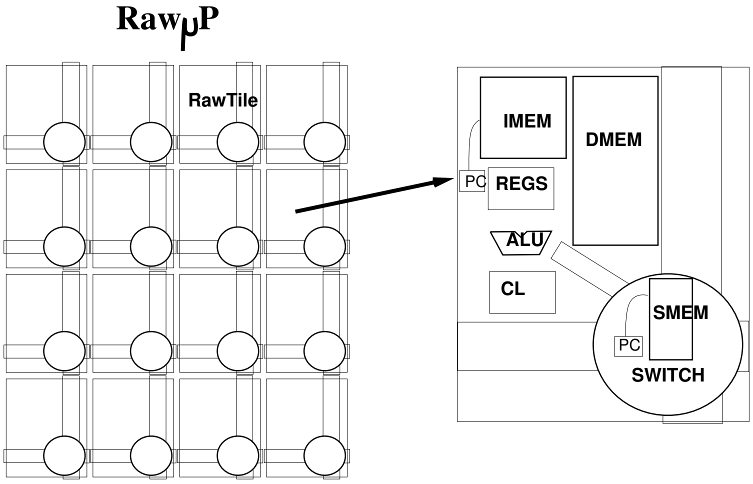

Figure 1: Raw microprocessor composition. A typical Raw system might include a Raw microprocessor coupled with off-chip RDRAM and stream-IO devices.

2 Raw Architecture

The Raw microprocessor chip comprises a set of replicated tiles. Each tile contains a simple RISC-like processor, a portion of memory for instructions and data, configurable logic, and a programmable switch.

2.1 Simple replicated tile

A Raw machine comprises an interconnected set of tiles (Figure 1). Each tile contains a simple RISC-like pipeline and is interconnected with other tiles over a pipelined, point-to-point network. A large number of distributed registers eliminates the small register-name-space problem, allowing the exploitation of ILP to a greater degree. SRAM memory distributed across the tiles eliminates the memory bandwidth bottleneck and provides significantly lower latency to each memory module. The distributed architecture also allows for multiple high bandwidth paths to external RDRAM, as many as packaging technology will permit. The amount of memory is chosen to roughly balance the areas devoted to processing and memory, and to match the memory access time with the processor clock.

Unlike current superscalars, a Raw processor does not bind specialized logic structures such as register renaming logic or dynamic instruction issue logic into hardware. Instead, it focuses on keeping each tile small to maximize the number of tiles that can fit on a chip, thereby increasing the amount of parallelism it can exploit and the clock speed it can achieve. For example, a single one billion transistor die (which might be available in 7-10 years) can carry 128 tiles, each including an R2000 equivalent CPU, floating point unit, configurable logic, 64K-bytes of SRAM, and 128K-bytes of DRAM. Significantly higher switching speeds internal to a chip compared to

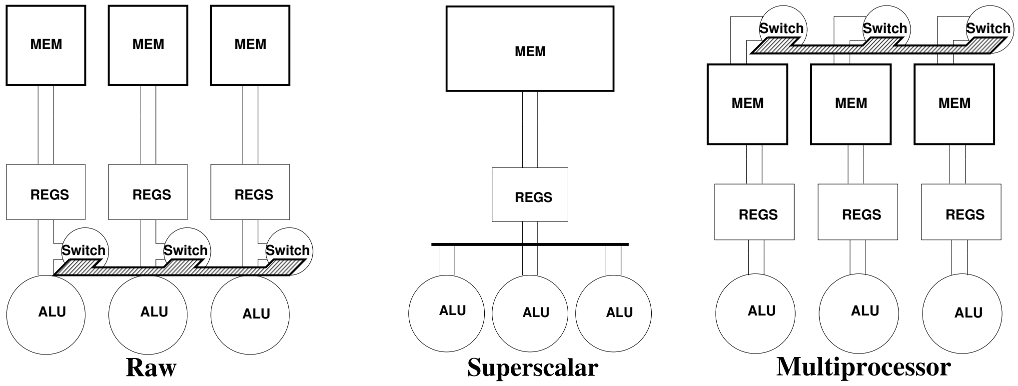

Figure 2: Raw microprocessors versus superscalar processors and multiprocessors. A Raw microprocessor distributes the register file and memory ports and communicates between ALUs on a switched, point-to-point interconnect. In contrast, a superscalar contains a single register file and memory port, and it communicates between ALUs on a global bus. Multiprocessors communicate at a much coarser grain through the memory subsystem.

off-chip communication and DRAM latencies will allow software to replace specialized hardware functionality.

As depicted in Figure 2, a Raw architecture can be viewed as a microprocessor which replaces the bus architecture of superscalar processors with a switched interconnect, and uses software to implement operations such as register renaming, instruction scheduling, and dependency checking. Although reducing the amount of hardware support for these operations is counter to current market trends, this approach makes available more of the chip area for memory and compute logic, results in a faster clock, and reduces the verification complexity of the chip. Taken together, these benefits can make software synthesis of complex operations competitive with hardware for end-to-end application performance.

2.2 Programmable, integrated interconnect

As shown in Figure 2, a Raw machine uses a switched interconnect instead of buses. The switch is integrated directly into the processor pipeline to support single-cycle send and receive operations. The processor communicates with the switch using distinct opcodes to distinguish between accesses to the static and dynamic network ports. No signal in a Raw processor travels more than a single tile width within a clock cycle. The tight integration of the switch with the processor and the short wire lengths permit inter-tile communication to occur at nearly the same speed as a register read. An on-chip interconnect also allows channels with hundreds of wires instead of tens – VLSI switches are pad limited and are rarely dominated by internal switch area.

The switch multiplexes two logically distinct networks, one static and one dynamic, over the same set of physical wires. In the static network, routing decisions are made statically at compile time; at run-time the switches blindly follow the routing instructions generated by the compiler. In the dynamic network, routing decisions are made by the network at run-time, similar to how routing is done in NUMAs.

2.3 Configurability

Each tile of a Raw architecture also contains configurable logic. The configurable logic is comprised of a small array of byte-wide ALUs and registers, with sufficient routing resources to connect them into configurations which commonly appear in general purpose computing. There is also a small amount of bit-level and control logic to support special bit-level and conditional operations. A Raw processor is thus coarser than a traditional FPGA-based computer [3], which evolved from structures better suited for random hardware glue logic than datapath oriented computation. Compared to a FPGA computer, the Raw configurable logic has a smaller configuration state, better propagation delays for common operations, lower routing requirements, and smaller area per ALU-like operation. However, it can still achieve the same level of fine-grained parallelism that an FPGA computer can attain when such parallelism exists in an application. Traditional FPGA computers also leverage their bit-level configurability to trade precision for parallelism. Raw explores this idea with some restrictions. The programming and algorithmic analysis issues do not seem worth the effort to provide complete precision flexibility. Instead, the byte-granular configurable logic allows the implementation of multigranular operations much like Sun's VIS, HP's MAX, and Intel's MMX extensions.

3 The Run-time System

Achieving top performance in programs with data dependent branching patterns and pointer based memory operations requires mechanisms that can analyze and react to program behavior at run time. In a modern superscalar processor, these operations are provided by hardware units including caches, branch prediction buffers, register renaming logic, and instruction scheduling units. In a Raw processor these functions are provided by the run-time system.

For example, Raw can do caching in software. The run-time system manages the memory hierarchy by checking each memory access and providing the current mapping of the requested address. As with other systems that use software to manage the memory system [12, 13, 14], the compiler has two responsibilities. It must insert code to perform these checks at each memory reference, and it must optimize away checks that can be statically determined to be redundant.

Inevitably, the baseline cost for software caching is higher than that for hardware caching. However, software caching can handle more sophisticated algorithms than can be conveniently built in hardware, and it provides the opportunity to customize the algorithms. These features can lead to higher hit rates, which in turn can mitigate and possibly completely recover the software overhead. There is also an exciting potential for exploiting inter-tile parallelism and scalability through software caching.

Similar support from both the run-time system and compiler will be required to provide mechanisms like speculative execution. Speculative execution increases the available parallelism of programs with dynamic dependence and branching patterns. The additional costs of performing services such as branch prediction and checking dependences on speculatively issued loads will need to be reduced. Cost reduction can come directly from compiler elimination of unnecessary operations, or it can come indirectly by the use of algorithms which provide superior support. An additional benefit of moving these mechanisms into software is that the hardware can be eliminated, permitting more room on the chip for additional parallelism and better clock cycle times.

Finally, the run-time system has to provide support for performance monitoring and debugging. Because the requested features (*e.g.*, a breakpoint) may change while the program is running, the run-time system needs to support dynamic code scheduling [2, 5, 6, 15]. Because the run-time system inserts code only where it is needed, the user will not need to pay any additional overhead for features they are not currently using.

The overall performance of the Raw system will depend on a number of factors. Because support for dynamic behaviors has been moved into the run-time system, a program will execute more instructions in Raw than in a system with more hardware support. On the other hand, the simpler Raw hardware will execute at a faster clock rate, and it can explore more available parallelism. The end-to-end performance will involve a balance of these factors and will also depend on the compiler's ability to eliminate and optimize the run-time mechanisms.

4 The RAW compiler

The RAW compiler faces many unique opportunities as well as challenges. Compilation issues in RAW include resource allocation, the exploitation of fine-grained parallelism, communication scheduling, the use of configurable logic, and code generation.

4.1 Resource Allocation

When executing a program in the Raw processor, both the run-time system and program execution are mapped to a collection of identical tiles. Rather than fixing the allocation in hardware, the tiles can be allocated in an application-specific manner to perform these tasks. More resources can be dedicated to performance-critical and highly used parts of the run-time system.

The program execution is divided into coarse-grain parallel regions. Each parallel region is executed on multiple tiles, which together behave as a single logical processor. The number of tiles allocated to each region depends upon the availability of fine-grain parallelism within the region. The tiles in a region communicate with each other using mostly static communication, while the regions communicate with each other as well as with the run-time system using dynamic messages. The size and number of regions allocated could be decided by the compiler based upon the nature of the parallelism available in the application. For example, it can use a small number of large regions when high-level fine-grained parallelism is known to the compiler, and a large number of smaller regions when only coarse-grained parallelism is available.

The Raw compiler needs to perform whole program analysis to obtain the run-time resource requirements and the availability of fine-grain and coarse-grain parallelism [9].

4.2 Exploitation of fine-grain parallelism

Raw has the flexibility to explore many forms of parallelism. It can support both the SIMD and MIMD programming models used by conventional multiprocessors. In fact, its low latency network allows it to attain speedup for a bigger set of such applications. More interestingly, Raw introduces a new paradigm for exploiting parallelism. Due to its miniscule cost of communication, Raw can profitably exploit fine-grained parallelism across tiles. In particular, multiple tiles can work together to exploit the ILP of a single original instruction stream.

The Raw approach to exploiting ILP differs from that of superscalar and VLIW. In contrast to a superscalar, Raw exploits ILP without complex and space-consuming hardware logic. Moreover, Raw's software approach does not suffer from the limitations of a small hardware instruction window. Unlike VLIW processors, each of Raw's processing units has its own instruction stream. This approach is more flexible because it does not require instructions to proceed in lock steps. On the other hand, the distributed structure of Raw makes the exploitation of locality an important issue.

Raw offers opportunities for new optimization techniques as well. An example pertains to register spilling. In conventional architectures, registers must spill to memory. Raw provides a cheaper alternative of spilling to registers of a neighboring tile. In effect, Raw maintains a double advantage in available register resources over modern architectures. Not only can a single tile afford more registers than a modern processor on a per-area basis, adjacent tiles can pool and share their registers, an opportunity made profitable by the availability of cheap communication.

4.3 Communication scheduling

The static network makes the Raw compiler novel among traditional compilers. Traditional compilers schedule a single type of event (instructions) along a single dimension (time) [1]. The Raw compiler, on the other hand, faces a more generalized problem: it must schedule both instructions and communication events, and the events must be scheduled both temporally and spatially.

Compilation of code to use the static network provides challenges from the perspective of both correctness and performance. With respect to correctness, the switch code must be deadlock-free. With respect to performance, the compiler must carefully orchestrate the communication to minimize network stalls. Both issues are complicated by the presence of dynamic events such as branches, cash misses, and dynamic messages.

Correct code can be generated for the static network even in the face of compile-time uncertainties. By statically ordering the communication in a way which is deadlock free, the compiler can

[1] If present, the spatial dimension in VLIW and superscalar processors is very small in size.

guarantee that the resultant code will be deadlock free even if the compile-time estimates of event latencies are not cycle-accurate.

The static network can be leveraged in two ways. In the general case, it can be used to provide low-latency communication needed to exploit profitably the fine-grain parallelism across multiple tiles. In this approach, the compiler takes an instruction stream as input, partitions that stream into multiple instruction streams, maps each stream to a tile, instruction-schedules each stream, and schedules the communication between the streams[2]. The partitioning and scheduling framework in the compiler builds on the work reported in [11, 16]. In addition, the compiler can turn certain loops into inter-tile pipelines which leverage the low cost of communication to achieve maximum throughputs.

When both the static and dynamic networks are used in the same application, we need to consider the issues which arise due to their interaction. We have to ensure in the compiler that the resultant programs are correct and deadlock-free. In addition, timing optimizations are likely to be beneficial due to the performance differences between static and dynamic networks. The timing of the static messages is highly predictable, while the delays due to dynamic messages can be large and highly unpredictable.

Programs which use both networks can benefit from optimizations which attempt to minimize the time wasted when multiple nodes blocked on static messages due to a large delay of a dynamic message to a single node. One such optimization applicable to some cases is to continue to send static messages containing variables stored locally to remote locations, even if the local processor is blocked on the dynamic network. This optimization is possible if the static messages do not depend on the information carried by dynamic messages. For example, variables for which a processor acts as a passive home location, rather than an active generator of new values, may safely be sent to remote processors while the local processor is blocked on the dynamic network.

4.4 Configurable logic

The presence of Raw's configurable logic provides many opportunities for optimization. The simplest manifestation of Raw's configurable support is the ability to perform multigranular operations. This model mirrors SIMD and vector processors. Existing vectorizing compiler work can be leveraged to provide automatic compilation for these operations. Such a compiler would also be useful to current commercial processors with extensions like VIS and MMX.

Additionally, the configurable logic can be used to replace frequently executed patterns found in program codes. In particular, the configurable logic will be helpful in reducing the overhead of the run-time system. For instance, the run-time system might require that additional parallel checks be done to support memory dependence checking and speculative execution. Parallel structures can be configured to accelerate these operations, making the overhead of doing these software operations more similar to the hardware overhead. If these structures are no longer needed, the logic can be

[2]Note that the compiler needs not perform these tasks in the order given in this list, nor are the tasks required to be independent of each other.

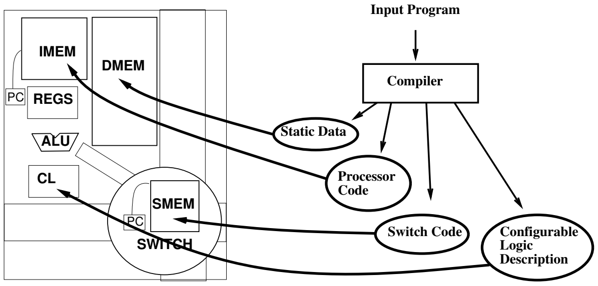

Figure 3: Components of the code emitted by the Raw compiler and their relations to the architecture.

reconfigured to perform other useful, common operations.

The configurable logic can also be used to accelerate signal processing algorithms with medium sized compute-intensive cores. Data would be pipelined through configured logic that mimics the program graph to attain parallelism that a superscalar could never attain due to the limitation in instruction sequencing resources. The reconfigurable logic within each of the tiles would be chained together to provide support for extremely long pipelines. This sort of structure could also be built with the tiles themselves. However, the reconfigurable logic does not require the sequencing or memory resources that a tile does. It thus provides improved computation density and is more suitable for stream-based multimedia applications.

One of the most difficult tasks of the compiler will be to identify the regions of code well suited to configurable implementation. We are exploring the possibility of using tree-pattern matching techniques [1, 10] to assist in identifying these regions. Once these regions are identified, hardware compilation algorithms can be employed to compile the critical program patterns into the configurable hardware [4].

4.5 Code generation

Because the Raw compiler is hardware-omniscient, its code generation phase is more involved than that of a traditional compiler. In additional to the traditional task of data partitioning and tile code generation, the Raw compiler must also generate code for the static switches and emit bit sequences for programming the configurable logic. Each of these components is an opportunity for developing new optimization techniques. See Figure 3 for how the compiler outputs relate to the architecture.

5 Compiler Implementation

We are using the SUIF compiler infrastructure to implement the Raw compiler. SUIF has many features which make it an ideal framework for the Raw compiler project [7]. It provides a single tool under which all compiler development, from whole program analysis for resource allocation to switch and tile code generation, can be performed. We plan to leverage on many aggressive analyses such as data-flow analysis and pointer alias analyses available for the SUIF infrastructure. Development of many new techniques for this unique architecture requires us to perform a lot of prototyping, an objective SUIF is designed to handle well. Finally, the availability of many traditional optimizations passes and strong front-ends to popular languages such as C and FORTRAN will help us evaluate the Raw architecture using a wide range of benchmarks and real programs.

Our short-term goal is to develop a end-to-end compiler path which can take in a subset of sequential C programs and map it to multi-tile code runnable on the Raw simulator. The compiler currently handles mostly static programs with arbitrary control flow, but it forbids memory references which cannot be disambiguated at compile time. We have yet to implement any of the optimizations.

The compiler path to handle these simple programs is as follows. We stress that this is the base approach for programs, and we expect that some classes of special programs, such as regular programs, and several others, will be handled as special cases for optional optimization. We do expect however, that this approach will provide a correct fall-back implementation for most programs.

First, we use SUIF's cfg library to divide a program into its basic blocks. Basic blocks are merged into superblocks which contain sufficient parallelism to be profitably parallelized. Then a series of compiler passes are performed on individual superblocks. First, a renaming pass renames variables as well as memory so that most storage locations accessed in a superblock are single assignment [3]. This renaming serves two purposes. It exposes all the inherent parallelism within a superblock, and it simplifies the later task of switch code generation.

After the renaming pass, the partitioner partitions the superblock into multiple instruction streams which can be executed in parallel. The number of streams generated depends on the amount of parallelism in the superblock and is limited to a maximum of the number of processors in the system. Also, communication instructions are inserted into the resultant streams to preserve the semantics of the original superblock.

After partitioning, the placer maps each stream to a physical processor. Then the machsuif backend for the Mips architecture translates each instruction stream from SUIF to Mips assembly to be run on the Raw tiles. Next, these code sequences are instruction scheduled while taking into account inter-tile communication latencies. Finally, switch codes corresponding to these tile codes are generated.

The above compiler passes turn each superblock into parallel instruction streams for Raw. A final pass uses the information from the control flow graph to stitch together the set of instruction streams for each processor. The final result is a MIMD parallel program using static communication which is semantically equivalent to the original program.

[3] Renaming is not performed when it incurs too much overhead e.g., the copy overhead that can result from renaming array variables.

6 Conclusion and Current Research

We have embarked on developing a complete compiler and run-time system for the Raw architecture. This paper describes the opportunities and challenges in developing these software.

Currently our initial compiler can generate unoptimized code for a small set of programs. We are extending the compiler in both its ability to handle a wide range of programs and its capacity to perform Raw-specific aggressive optimization. Our goal is to achieve performance that is at least comparable to that provided by scaling an existing architecture, and also to achieve performance which is orders of magnitude better for applications in which the compiler can discover and exploit fine-grained parallelism, data access patterns and other features.

7 Acknowledgments

This research was funded by DARPA contract #DABT63-96-C-0036 and by an NSF PYI Award.

References

[1] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM TOPLAS*, 11(4), October 1989.

[2] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *Proc. PLDI '96*, May 1996.

[3] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal. The raw benchmark suite: Computation structures for general purpose computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, Apr. 1997.

[4] J. Babb, R. Tessier, and A. Agarwal. Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators. In *Proceedings IEEE Workshop on FPGA-based Custom Computing Machines*, pages 142–151, Napa, CA, April 1993. IEEE. Also as MIT/LCS TM-491, January 1993.

[5] B. Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In *Proc. 1994 ACM SIGMETRICS Conference*, pages 128–137, May 1994.

[6] D. R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *Proc. PLDI '96*, pages 160–170, May 1996.

[7] R. W. et al. SUIF: A Parallelizing and Optimizing Research Compiler. *SIGPLAN Notices*, 29(12):31–37, December 1994.

[8] J. A. Fisher. Very Long Instruction Word Architectures and the ELI-512. In *Proc. 10th Annual Symposium on Computer Architecture*, pages 140–150, June 1983.

[9] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S. Liao, and M. S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing*, San Diego, CA, Dec. 1995.

[10] T. A. Proebsting. Simple and efficient BURS table generation. *Proceedings of the ACM SIG-PLAN '92 Conference on Programming Language Design and Implementation, San Fransico, California.*, June 1992.

[11] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.

[12] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Cambridge, Massachusetts, October 1–5, 1996.

[13] I. Schoinas, B. Falsafi, A. Lebeck, S. Reinhardt, J. Larus, and D. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proc. ASPLOS VI*, pages 297–306, Oct. 1994.

[14] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. In *Proc. Fourteenth ACM Symposium on Operating System Principles*, pages 203–216, Dec. 1993.

[15] R. Wahbe, S. Lucco, and S. L. Graham. Practical Data Breakpoints: Design and Implementation. In *Proc. PLDI '93*, pages 1–12, June 23–25, 1993.

[16] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.