Cascaded Execution: Speeding Up Unparallelized Execution on Shared-Memory Multiprocessors

Ruth E. Anderson, Thu D. Nguyen, and John Zahorjan
Department of Computer Science and Engineering, Box 352350
University of Washington, Seattle, WA 98195-2350
{rea, zahorjan}@cs.washington.edu, tdnguyen@cs.rutgers.edu

Abstract

Both inherently sequential code and limitations of analysis techniques prevent full parallelization of many applications by parallelizing compilers. Amdahl's Law tells us that as parallelization becomes increasingly effective, any unparallelized loop becomes an increasingly dominant performance bottleneck.

We present a technique for speeding up the execution of unparallelized loops by cascading their sequential execution across multiple processors: only a single processor executes the loop body at any one time, and each processor executes only a portion of the loop body before passing control to another. Cascaded execution allows otherwise idle processors to optimize their memory state for the eventual execution of their next portion of the loop, resulting in significantly reduced overall loop body execution times.

We evaluate cascaded execution using loop nests from wave5, a Spec95fp benchmark application, and a synthetic benchmark. Running on a PC with 4 Pentium Pro processors and an SGI Power Onyx with 8 R10000 processors, we observe an overall speedup of 1.35 and 1.7, respectively, for the wave5 loops we examined, and speedups as high as 4.5 for individual loops. Our extrapolated results using the synthetic benchmark show a potential for speedups as large as 16 on future machines.

1. Introduction

The focus of most of the work on parallelizing compilers has been on finding efficient, legal parallel executions of loops expressed using sequential semantics [3, 5]. This paper addresses a complementary issue, how to most efficiently execute loops for which the compiler cannot find a legal or efficient parallel realization. For correctness, these loops must be executed sequentially. We focus on reducing the execution times of these sequential loops by reducing the number of cache misses that occur. We achieve this with a technique called *cascaded execution*, in which processors alternate between phases of loop execution and memory state optimization. Cascaded execution assures that exactly one processor is executing loop iterations at any one time, resulting in sequential loop execution. The time when a processor is not executing iterations is used to optimize its memory state for its next turn at iteration execution.

This work was supported in part by the National Science Foundation (Grant CCR-9704503), the Intel Corporation, Microsoft Corporation, and Apple Computer, Inc.

We evaluate the performance of cascaded execution using loop nests from wave5, a Spec95fp benchmark application, and a synthetic benchmark designed to simulate the increasing future cost of memory accesses. We present results for two different hardware platforms, a PC with 4 Pentium Pro processors and an SGI Power Onyx with 8 R10000 processors, to illustrate that the performance improvements obtained by cascaded execution are independent of a particular hardware configuration.

Our results show overall speedups of 1.35 (on the PC) and 1.7 (on the Power Onyx) for a number of important loops in wave5, with speedups as high as 4.5 for individual loops. Results for the synthetic benchmark show a potential for speedups of up to 16 on future processors.

2. Cascaded Execution

Figure 1(a) shows how an unparallelized loop in a compiler-parallelized application would typically be executed on a system with three processors. Note that processors 2 and 3 are idle while processor 1 executes the sequential section. Processor 1 must eventually load all the data referenced by this loop into its cache. It is likely that this will cause a high miss rate: the usual compulsory, capacity, and conflict misses of any sequential execution are exacerbated by the fact that parallel applications typically process in-memory structures too large to fit in the caches of any single processor, and by the likelihood that the data was distributed among the other processors during a previous parallel section.

Figure 1(b) shows the application of cascaded execution to the same loop. The loop is still executed sequentially, but all processors contribute to the effort. Each processor alternates between two phases: helper and execution. For correctness, only one processor at a time may be in its execution phase, during which it executes a portion of the loop body. When done, it exits the execution phase and passes control to another processor, which then enters its own exe-

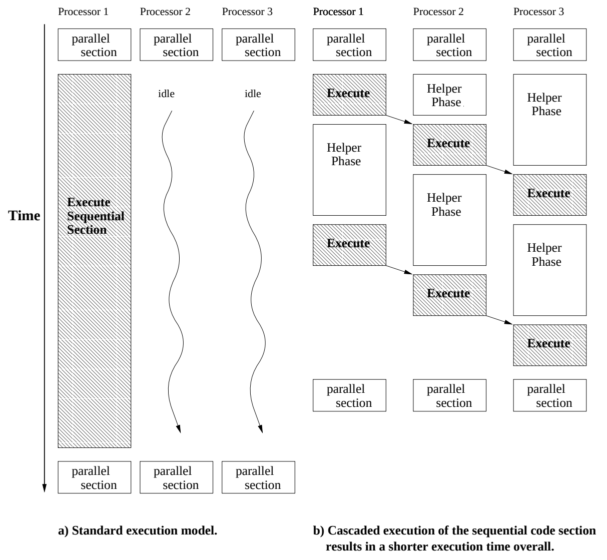
Processor 1
Processor 2
Processor 3
parallel section
Time
Execute Sequential Section
idle
Execute
Helper Phase
a) Standard execution model.
b) Cascaded execution of the sequential code section results in a shorter execution time overall.

Figure 1. Cascaded Execution

cution phase. All other processors are in their helper phases, during which they optimize their memory state by, for example, pre-loading into their caches the data they anticipate will be referenced during their next execution phases. The total time required to execute the loop in this way is the sum of the times the processors spend in their execution phases plus the control transfer overheads. Our goal is that, because of the memory state optimization, the sum of the execution times will be significantly smaller than the execution time of the loop on a single processor, and will more than compensate for the penalty of the required transfers of control.

The central design questions of cascaded execution are "What functions should be performed during the helper phases?" and "How many iterations should be performed during each execution phase?". We now address these questions. A more detailed description of these techniques can be found in [2].

2.1. What functions should be performed during the helper phases?

The simplest helper technique is for a processor to *prefetch* needed data into its caches. During the helper phase, each processor executes a shadow version of the original loop body, loading the values that will be required to execute its next set of loop iterations.

A more aggressive use of the helper phase is to restructure the data in a way that optimizes the execution phase memory reference pattern. In *sequential buffer data restructuring*, instead of simply loading the data into the caches during the helper phase, we copy all read-only data into a sequential buffer in dynamic reference order. During the execution phase, these operands are simply fetched in sequential order out of the buffer. Packing the data in this way has a number of potential benefits. It improves cache utilization, since each line of the sequential buffer is full with useful data. The sequential buffer eliminates conflict misses in the data it contains, since it is read purely sequentially during the execution phase. Reading from the sequential buffer may reduce the number of operations and data accesses required to index array data. Finally, in some cases, computation that involves only read-only data values can be done during the helper phase. This can reduce both the amount of work required during the execution phase and the amount of data that must be stored in the sequential buffer.

2.2. How many iterations should be performed during each execution phase?

The execution phase consists of executing a contiguous chunk of iterations. We choose the chunk size based on an estimate of the number of bytes of data that each iteration of the execution loop will touch. On one hand, we would like the fetched data to fit in the L1 cache, to minimize data access time during the execution phase. On the other hand, because the execution of each chunk ends with a transfer of control, we would like to minimize the number of chunks (to minimize the total transfer of control overhead). To do so, we must use larger chunk sizes.

The effect of chunk size on performance is examined empirically in the next section.

3. Performance evaluation

3.1. Software environment

We evaluate the performance of cascaded execution in two scenarios. First, we measure wave5 from the Spec95fp benchmark suite on two current multiprocessors. In profiling the sequential execution of wave5, we found that one subroutine, PARMVR, dominates the execution time, consuming roughly 50%. PARMVR is called approximately 5000 times and consists of 15 loops. Previous examination of these loops, including our own experience, showed difficulty with parallelization and no effective speedup in this application [9].

The original reference data set provided with wave5 is sized inappropriately for the caches on today's machines: the data set processed by each call to PARMVR is less than 300KB. Larger problem sizes provided with the benchmark grow along the time dimension but not in the space dimension [16]. Since the original data set was too small to be representative of problems likely to be run on today's parallel machines, we enlarged the problem by increasing the amount of data accessed in each loop. In the enlarged problem, the amount of data accessed by each loop ranges from 256KB to 17MB.

Our second set of measurements is intended to estimate the benefits of cascaded execution on future processors, where memory access time will become an increasingly dominant factor in performance. Because we do not have access to tomorrow's multiprocessors, for this evaluation, we use a synthetic loop nest characterized by a larger ratio of memory access to computation than is exhibited by benchmark applications running on current machines.

3.2. Hardware environment

We evaluate cascaded execution on two processor consistent shared-memory multi-processors: a 4-processor PC server and an 8-processor SGI Power Onyx. The PC server has 4 200MHz Pentium Pro processors running NT Server 4.0. The SGI Power Onyx has 8 194MHz IP25 MIPS R10000 processors running IRIX 6.2.

The Pentium Pro and the R10000 are both advanced super scalar processors with out-of-order execution, branch prediction, register renaming, and speculative execution. All caches on both machines are non-blocking, allowing up to four outstanding requests to the L2 cache and to main memory. Table 1 presents the memory hierarchy sizes and access times for the two machines.

Processor	Memory Level	Access Time (Cycles)	Size	Assoc	Line Size
Pentium Pro	L1	3	8KB	2	32 bytes
	L2	7	512KB	4	32 bytes
	Memory	58	1.5GB	-	-
R10000	L1	3	32KB	2	32 bytes
	L2	6	2MB	2	128 bytes
	Memory	100-200	1GB	-	-

Table 1. Pentium Pro [10, 11] and R10000 [13] memory characteristics

3.3. Current performance

Figure 2 shows the overall speedup[1] of the PARMVR subroutine of the wave5 benchmark when run under cascaded execution with 64KB chunks (which was found to perform best on both platforms among the chunk sizes we evaluated). Figure 3 gives execution times in cycles for the fifteen individual loops in that routine. Figures 4 and 5 show the L2 cache and L1 data cache misses, respectively. In these figures, "Prefetched" corresponds to the version of cascaded execution where the helper function merely prefetches operand data, while "Restructured" corresponds to the version where read-only data is streamed into a sequential buffer.

These results lead us to the following conclusions:

Cascaded execution can provide good speedups: we achieve an overall speedup of 1.35 on the Pentium Pro and 1.7 on the R10000. In Figure 2 , we see that for all numbers of processors, a version of cascaded execution achieves noticeable speedup over sequential execution of the original code on a single processor. Figure 3 shows that results for individual loops vary, from a maximum slowdown of 0.9 to a maximum speedup of 4.5.

Our results are somewhat limited by the number of processors available to us. More processors allow more time to complete helper iterations, and thus better performance. In simulations of an unbounded number of processors, some loops were shown to have potential speedups as high as 30. Results on the four and eight-processor machines available to us are more modest. We found that performance is improved by causing a processor to jump out of a helper phase, if necessary, as soon as it is signaled to begin execution. The results presented below are for an implementation that includes this modification.

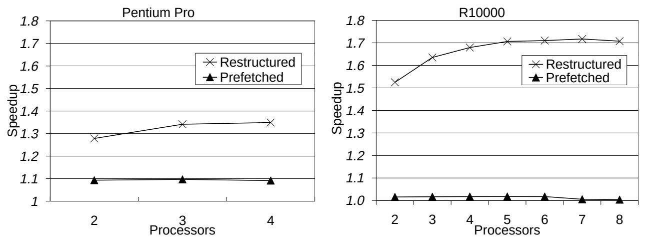

Figure 2. Overall speedup for PARMVR

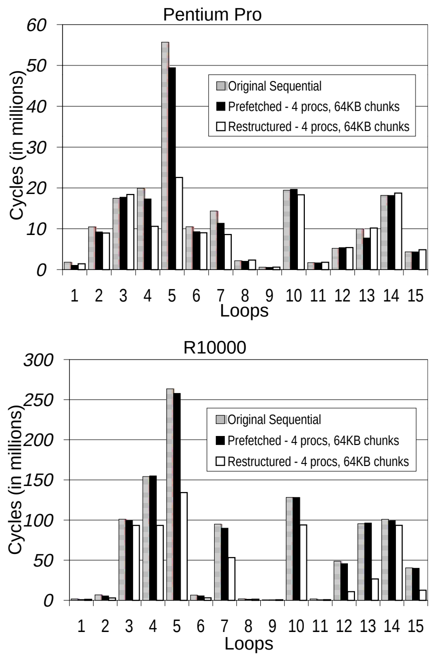

Figure 3. Execution times of PARMVR loops

Data restructuring is significantly more effective than prefetching alone. Figures 2 and 3 show that data restructuring provides a much greater speedup than prefetching alone. We believe that this benefit arises primarily from the elimination of conflict misses that restructuring can provide. In fact, on the R10000, where the L2 cache has lower associativity, we see little improvement from prefetching

[1]We arbitrarily present the timings for the 12th call (out of 5000 calls) to PARMVR - other calls perform similarly.

alone. Figures 4 and 5 show that prefetching does not reduce cache misses on the R10000. We hypothesize that since the MIPSpro compiler inserts prefetch instructions in its optimized code, it may be able to hide much of the latency of memory accesses other than those required for conflict misses. Thus, cascaded execution with prefetching alone provides no additional benefit.

Cascaded execution is successful at improving application memory behavior. Figure 4 shows that, cascaded execution eliminates 93-94% of the L2 cache misses on the Pentium Pro, and cascaded execution with restructuring eliminates 47% of the L2 cache misses on the R10000. Figure 5 illustrates that, on both platforms, data restructuring eliminates L1 data cache misses in several of the loops. In these cases, we believe that restructuring eliminates conflicts in the L1 cache.

Interestingly, although cascaded execution removes a larger percentage of L2 cache misses on the Pentium Pro than on the R10000, it affords better speedup for the R10000. This is because there are 2.59 times more L2 cache misses in the original sequential execution of wave5 on the R10000 than on the Pentium Pro (perhaps because of the more limited associativity of the Power Onyx's L2 cache). In addition, L2 cache misses are more costly for the R10000.

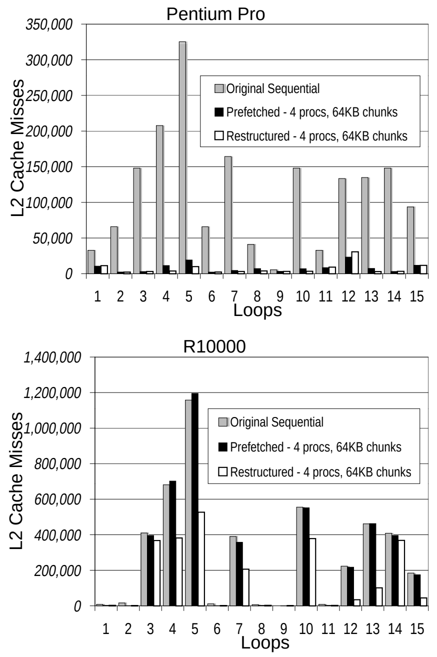

Figure 4. L2 Cache Misses in PARMVR

Chunk sizes larger than the size of the first level cache give the best performance because of the significant cost of transferring control between processors. Figure 6 shows the overall speedup of PARMVR for chunk sizes varying from 4KB to 2048KB. On both platforms, the cost of transferring control is significant: 120 cycles per transfer on the Pentium Pro and 500 cycles on the R10000[2]. The speedups for PARMVR indicate an optimum chunk size in the range of 16KB to 64KB for four processors, which is larger than the L1 cache of either machine.

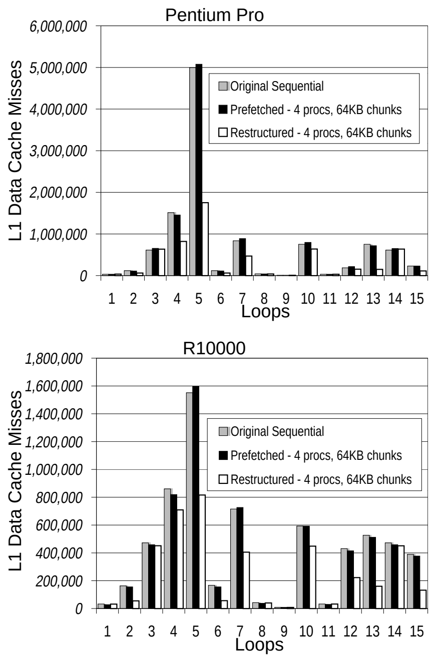

Figure 5. L1 Data Cache Misses in PARMVR

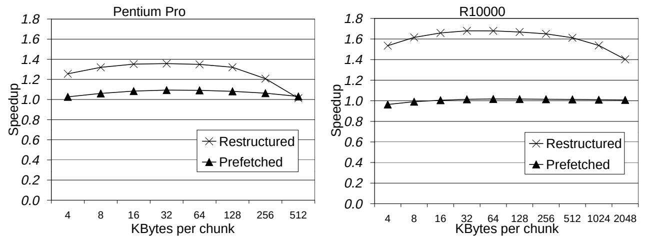

Figure 6. Effect of chunk size (4 processors)

3.4. Future performance

The previous subsection presented results for a benchmark application running on modern multi-processors. In the future, as processors continue to outpace access rates to main memory, we expect that application memory stall times will increase relative to instruction execution times.

To simulate this future scenario, we examine the performance of cascaded execution on a simple, synthetic loop that has a larger ratio of memory access time to instruction execution than our benchmark program. This larger ratio

[2]Transferring control requires only that a shared-memory flag be set and that the target processor see its new value. We have optimized this procedure as much as possible, but the large cycle count penalties of accessing main memory lead to these significant control transfer overheads.

is generated by reducing the computational demand of the synthetic loop relative to the benchmark loops, and running on the same multiprocessors as before (thus keeping memory access times constant). Results for this synthetic loop are intended only to give a rough indication of the benefits of cascaded execution on future machines; it is clearly infeasible to attempt to represent all applications, or the details of all future machines, as would be needed to make more precise claims.

The synthetic loop used in our simulation was:

```
do i = 1, n, k
   X(IJ(i)) = X(IJ(i))+A(i)+B(i)
end do
```

In this loop, all operands are integers, and the index array IJ is simply the vector $1..n$.

To examine a range of memory access to instruction execution ratios, we consider two versions of this loop. In a "dense" execution, the loop step size k is set to one, causing the loop to walk sequentially through words of memory. In a "sparse" execution, the step size is set to eight, which corresponds to the number of integers that fit in an L1 cache line on both machines. Thus, in the sparse case, the original loop body has no spatial locality whatsoever, which magnifies the memory costs and thus the memory access to execution ratio.

To avoid limiting observable speedups to the number of processors on the machines available to us, we simulate cascaded execution by running on a single processor, which alternates between helper and execution phases. Helper loops are allowed to run to completion, which models a system with enough processors that each completes each helper phase before being signaled to begin a new execution phase. Overall execution time is calculated by summing the time spent in the execution phases and adding in the cost of control transfers (one transfer per chunk). To obtain speedup we compare this sum to the execution of the original loop running on a single processor.

Figure 7 shows observed speedups for chunk sizes ranging from 1KB to 256KB. From it, we see that in the likely future scenario where memory access time becomes an increasingly dominant factor in program execution time, cascaded execution can provide significant benefits. In the dense case, cascaded execution provides speedups of around 4 for both systems. Speedups are even more impressive for the more memory-intensive sparse case: 16 for the Pentium Pro and close to 14 for the R10000

4. Related work

Numerous hardware [4, 6, 18] and software [7, 8, 14] techniques have been proposed to tolerate memory latency in sequential programs. The approaches most relevant to our work are prefetching and multithreading.

In *software-controlled prefetching* [7, 14], the compiler analyzes the program and inserts prefetch instructions for

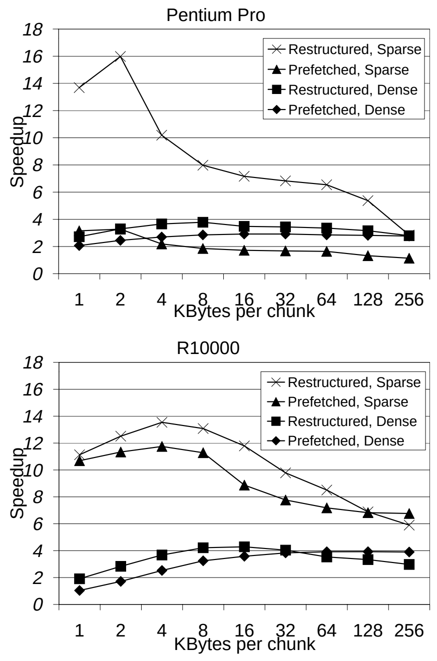

Figure 7. Cascaded execution speedups with increased memory access costs

accesses that will most likely result in cache misses. Accurate analysis is crucial because prefetching can displace useful values in the cache, increase memory traffic, and increase the total number of instructions that must be executed.

Multithreading [1, 17] tolerates latency by switching threads when a cache miss occurs. This technique can handle arbitrarily complex access patterns, but must be implemented in hardware to be effective. Furthermore, sufficient parallelism must be available in the application to fully mask memory latency; this amount of parallelism may not always exist.

Cascaded execution is applicable in only a much more restricted domain than the techniques listed above. However, in that domain it is complimentary to them. Each may be used to reduce the time required to execute a sequential loop on a single processor. Cascaded execution can be combined with any of them to help mask any memory access latency that remains. At the same time, cascaded execution may enhance the performance of these techniques by simplifying and improving the memory reference behavior.

Several *speculative* and *run-time parallelization* methods have been proposed to attempt parallel execution of loops that cannot be analyzed sufficiently accurately at compile time [12, 15]. Like cascaded execution, these techniques make use of processors that would otherwise be idle if the compiler resorted to simple, sequential execution. In cases where enough parallelism is available at run-time to overcome the overheads associated with run-time parallelization, or when memory stalls are not a significant contributor to execution time, run-time parallelization may achieve higher speedups. However, when loops contain little parallelism and when memory stalls contribute significantly to execution time, cascaded execution should provide higher speedups.

5. Conclusions

We have identified a previously unexamined problem confronting parallelizing compilers, how to maximize the performance of portions of the code for which no parallel execution can be found. We have introduced a new technique, cascaded execution, to speed up sequential loop execution. Cascaded execution uses processors that would otherwise be idle during sequential loop execution to optimize memory state in a way that leads to improved cache behavior, and so improved performance.

Experiments run on a Pentium Pro multiprocessor and an SGI Power Onyx show that cascaded execution is able to speed up sequential execution of otherwise unparallelized loops from a Spec95fp benchmark application by up to a factor of 4.5, with no significant slowdown in any case. Experiments using a synthetic loop intended to mimic the increased memory access penalties of future processors indicate that the benefits of cascaded execution are likely to be even larger in the future; we observe speedups as high as 16 in this case.

Acknowledgements

We thank the National Center for Supercomputing Applications for access to SGI Power Challenge machines in the initial stages of this project. We thank Frederic Mokren for porting wave5 to the Pentium Pro. We thank Jan Cuny for access to SGI machines at the Computational Science Institute at the University of Oregon.

References

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the International Symposium on Computer Architecture*, pages 104–114, Seattle, WA, May 1990.

[2] R. E. Anderson, T. D. Nguyen, and J. Zahorjan. Cascaded execution: Speeding up unparallelized execution on shared-memory multiprocessors. Technical Report UW-CSE-98-08-02, University of Washington, Department of Computer Science and Engineering, (*http://www.cs.washington.edu/research/zahorjan/cascade/cascade.abstract.html*), Sept. 1998.

[3] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[4] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, pages 176–186, Albuquerque, New Mexico, Nov. 1991.

[5] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[6] D. Burger, S. Kaxiras, and J. R. Goodman. Datascalar architectures. In *Proceedings of the International Symposium on Computer Architecture*, pages 338–349, Denver, CO, June 1997.

[7] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proceedings of the International Conference on Supercomputing*, pages 354–368, Amsterdam, The Netherlands, June 1990.

[8] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the International Symposium on Computer Architecture*, pages 254–263, Toronto, May 1991.

[9] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1996.

[10] Intel Corporation, P.O. Box 7641, Mt. Prospect, IL 60056-7641. *Intel Architecture Software Developer's Manual*, 1997.

[11] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance characterization of a quad Pentium Pro SMP using OLTP workloads. In *Proceedings of the International Symposium on Computer Architecture*, pages 15–26, Barcelona, Spain, June 1998.

[12] S.-T. Leung and J. Zahorjan. Improving the performance of runtime parallelization. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–91, San Diego, CA, May 1993.

[13] MIPS Technologies Inc., 2011 North Shoreline, Mountain View, CA 94039-7311. *R10000 Microprocessor User's Manual-Version 2.0*, 1997.

[14] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Boston, MA, Oct. 1992.

[15] L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 218–232, La Jolla, CA, June 1995.

[16] J. P. Singh, J. L. Hennessy, and A. Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *Computer*, 26(7):42–50, 1993.

[17] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the International Symposium on Computer Architecture*, pages 191–202, Philadelphia, PA, May 1996.

[18] Y. Yamada, T. L. Johnson, G. E. Haab, J. C. Gyllenhaal, W.-m. W. Hwu, and J. Torrellas. Reducing cache misses in numerical applications using data relocation and prefetching. Technical Report CRHC-95-04, Center for Reliable and High Performance Computing, Apr. 1995.