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

Technical Report UW-CSE-98-08-02
August 18, 1998
Revised September 26, 1998

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 meant to assess the impact of the increasingly dominant memory access times of future processors. 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 .

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

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. The importance of such unparallelized loops is evidenced indirectly by the continuing intense interest in parallelization techniques, and directly by empirical studies of the effectiveness of current techniques [8, 11, 17, 23, 25]. Whether loops are left unparallelized because of inherently sequential semantics or limitations of the parallelization techniques, Amdahl's law tells us that these sequential code segments are a limiting factor to performance. Thus, our goal is to speed up the execution times of these sequential code segments.

Parallelized applications present a special opportunity to speed up the execution of sequential code segments that we exploit in our technique. In particular, by definition, these applications are run on multiple processors. When a sequential code segment is reached, it is executed by a single processor, with the others going idle[1]. Our technique takes advantage of these otherwise idle processors, in the context of shared-memory multiprocessors.

We focus on reducing the execution times of sequential code segments by reducing the number of cache misses that occur. In programs that manipulate large in-memory structures, a characteristic typical of many compiler-parallelized applications, memory access costs can be a substantial fraction of execution times [26]. It is well known that cache miss penalties are becoming increasingly costly as memory access time fails to keep up with increasing processor speed and instruction-level parallelism. Thus, not only will sequential portions of compiler-parallelized applications become a bottleneck, but specifically, the memory accesses in these sequential segments will dominate performance.

We call our technique *cascaded execution*. Specifically, we propose optimizing the memory state of a processor (for example, by pre-loading data values into its caches) before it executes a portion of the sequential loop. The loop itself is executed sequentially, but its execution is cascaded 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. Those processors not currently executing the loop body spend their time optimizing their memory states.

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.

The remainder of the paper is organized as follows. Section **2** describes cascaded execution in more detail. Section **3** describes our experimental setup and presents the performance results. In Section **4**, we describe related work on hiding memory latency and making use of otherwise idle processors during sequential code segments. Section **5** concludes the paper.

[1] While it is possible for the system scheduler to reallocate the idle processors to other applications during these intervals [16, 19], this can entail considerable overhead and is not commonly done.

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.

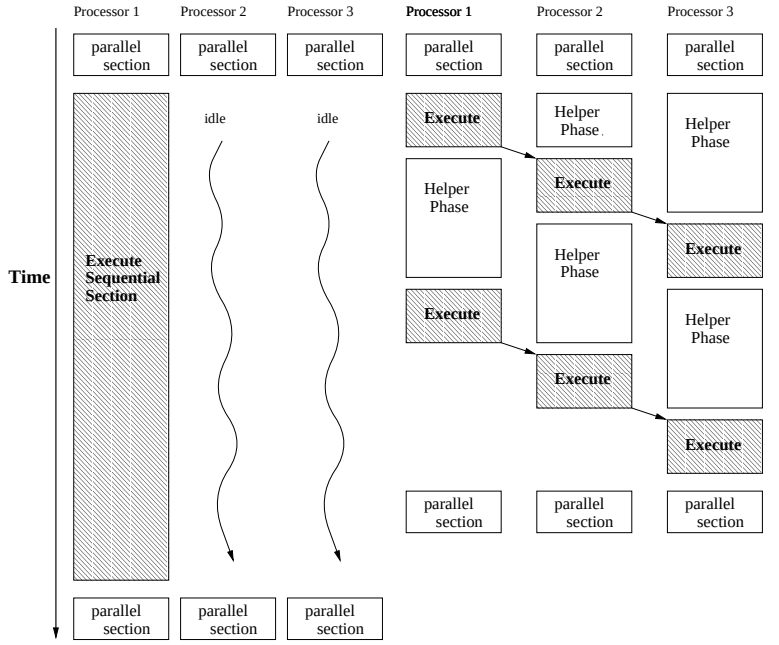

Figure 1: *Cascaded Execution*

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 execution 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.

2.1 What Functions Should Be Performed During the Helper Phases?

We have developed two different cascaded execution helper techniques: *prefetching* and *sequential buffer data restructuring*.

2.1.1 Prefetching

The simplest helper technique is for a processor to prefetch needed data into its caches. To do so, we execute a shadow of the original loop body. The shadow loop index runs over the range of iterations that will be performed during the next execution phase, and the shadow loop body simply prefetches each operand that will be referenced during execution of the corresponding loop body iterations.

Prefetching has the advantage that it is easy to implement. In particular, no dependence testing is required to determine if it is legal; it is always legal to apply prefetching[2]. If a value is updated after it has been prefetched, the hardware cache coherence policy will ensure that the correct value is used in the computation.

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

(a)

```
C Helper phase
      do i = mystop, mystart, -1
           prefetch X(IJ(i))
           prefetch A(i)
           prefetch B(i)
           if (ReadyFlag(myProcID))goto execute
      end do

C Execution phase
wait: if (!ReadyFlag(myProcID)) goto wait
execute:
      ReadyFlag(myProcID) = false
      do i = mystart, mystop
           X(IJ(i)) = X(IJ(i))+A(i)+B(i)
      end do
      ReadyFlag(myProcID+1 % P) = true
```

(b)

Figure 2**:** (a) *An unparallelized loop, and* (b) *cascaded execution with prefetching*

Figure 2(a) shows the code for a simple loop written in pseudo-Fortran, while Figure 2(b) shows the corresponding cascaded execution with prefetching code. Here variable *myProcID* is the index of the processor executing the code, and *P* is the total number of processors being used.

During the helper phase, each processor loads the values that will be required to execute its next set of loop iterations. Prefetching of data is done in *reverse* access order whenever possible, so that the data required first during the execution phase will have been most recently fetched. This minimizes misses during the subsequent execution phase in the case that the prefetched data conflicts with itself significantly. Additionally, the processor abandons the helper phase in favor of its execution phase, even if not all prefetching has been done, as soon as execution is legal. When sufficient processors are available, such "jumping out" of the helper phase is not required. However, its implementation serves as a safeguard for loops that have a high prefetch to execution time ratio.

2.1.2 Sequential Buffer Data Restructuring

A more aggressive use of the helper phase than simply prefetching is to *restructure* the data in a way that optimizes the execution phase memory reference pattern.

Many techniques have been proposed for data restructuring [2, 14]. While any of these could be applied here, we adopt the *sequential buffer technique*. In particular, instead of simply loading the data into the caches during the helper phase, we copy all data that is read-only in the execution loop into a sequential buffer in dynamic reference

[2] We assume a sequentially consistent memory system.

order[3]. During the execution phase, these operands are simply fetched in sequential order out of the buffer. Packing the data in this way improves cache utilization, since each line of the sequential buffer is full with useful data. As importantly, the sequential buffer eliminates conflict misses in the data it contains, since it is read purely sequentially during the execution phase.

Figure 3 shows the application of the sequential buffer data restructuring technique to the loop from Figure 2(a). During the helper phase, the processor references data in the (reverse) order of the execution loop and streams it into the array *temp*. During the execution phase, the processor keeps a pointer into *temp* and simply fetches operands as they are needed, incrementing the pointer as it progresses. Since it is possible for the processor to be signaled to begin execution before it has finished executing all helper loop iterations, we must provide two copies of the execution loop. One copy reads operands from *temp* and the other from the original arrays.

```
        c Helper phase
        cntr = (mystop-mystart)*3+1
        do preidx = mystop, mystart, -1
            if (ReadyFlag(myProcID))goto execute
            temp(cntr) = A(preidx)
            temp(cntr+1) = B(preidx)
            temp(cntr+2) = IJ(preidx)
            prefetch X(IJ(preidx))
            cntr = cntr - 3
        end do
        c Execution phase
wait:   if (!ReadyFlag(myProcID)) goto wait
execute:
        ReadyFlag(myProcID) = false
        do i = mystart, preidx
            X(IJ(i)) = X(IJ(i))+A(i)+B(i)
        end do
        do i = preidx+1, mystop
            regtemp = temp(cntr+2)
            X(regtemp) = X(regtemp)+temp(cntr)+temp(cntr+1)
            cntr = cntr + 3
        end do
        ReadyFlag(myProcID+1 % P) = true
```

Figure 3: *Sequential buffer data restructuring applied to the loop in* Figure 2(a).

While the primary goal of the sequential buffer technique is to reduce misses during the execution phase, a side benefit in many cases is a reduction in the amount of work required during that phase. In particular, it is common that some index function evaluation overhead of the original loop is eliminated. For example, in Figure 3, if the operand *A(i)* were instead *A(IDX(i)-1)*, the index function *IDX(i)-1* would be evaluated in the helper loop and the value *A(IDX(i)-1)* stored in *temp*. Thus, we would have removed one memory reference (to *IDX(i)*) and the subtraction from the execution loop.

The example in Figure 3 presents another opportunity for speeding up the loop body execution. If arrays A and B are available for copying into the sequential buffer, then the helper loop could instead add them together and store the result into the sequential buffer. This removes the addition, as well as the extra load instruction and cache space for accessing both operands from the execute loop. This form of *loop distribution* [29], while possible in general, fits naturally in the cascaded execution framework.

[3] For this to be legal, it must be certain that the value of the restructured data is up-to-date at the time the helper function is executed. In the worst case, this requires the insertion of a barrier synchronization just before the cascaded loop, although dependence testing may reveal that the barrier is not required.

2.2 How Many Iterations Should Be Performed During Each Execution Phase?

Because we must execute the loop iterations in order, each execution phase must involve a contiguous chunk of iterations. The question is *How many iterations should be in each chunk?*

Note that because we do not want to spend time prefetching when execution is possible, the first chunk of loop iterations must be executed without benefit of a corresponding helper phase. This initial execution phase is unoptimized, and so each iteration in it will take longer than iterations in later execution phases. For this reason, "ramping up" the chunk sizes at the beginning of the cascaded execution of a loop may be necessary to obtain optimum performance [28]. However, thus far we have considered only constant chunk sizes.

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. Expressed this way, the two natural choices for the chunk size correspond to filling the L1 cache and filling the L2 cache.

We are motivated to fetch only enough data to fill the L1 cache in order to minimize the cycles per instruction (CPI) experienced during the execution phase. Larger chunks are guaranteed to miss in the L1 cache and will require accesses to at least the L2 cache. The extent to which these misses affect performance depends on both the architecture-dependent miss penalties and the application-dependent instruction stream.

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. Thus, we are motivated to consider chunks that fill the L2 cache, rather than just the L1.

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

3 Performance Evaluation

3.1 Software Environment

We evaluate the performance of cascaded execution in two scenarios. First, using measurements of a program from the `Spec95fp` benchmark suite running on two current multiprocessors, we assess the benefits of cascaded execution on today's multiprocessors. For this work we selected the `wave5` application from the benchmark suite. 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 [11].

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 300 KB. Larger problem sizes provided with the benchmark grow along the time dimension but not in the space dimension [24]. 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 256 KB to 17 MB.

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 sequentially consistent shared-memory multi-processors: a 4-processor PC server and an 8-processor SGI Power Onyx. The PC server has 4 200 MHz Pentium Pro processors running NT Server 4.0. The SGI Power Onyx has 8 194 MHz 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	Associativity	Line Size	Other info
Pentium Pro	L1	3	8 KB	2	32 bytes	On-chip, write-back
	L2	7	512 KB	4	32 bytes	On package, write-back, unified
	Memory	~58	1.5 GB	-	-	-
R10000	L1	3	32 KB	2	32 bytes	On-chip, write-back
	L2	6	2 MB	2	128 bytes	External, write-back, unified
	Memory	100-200	1 GB	-	-	-

Table 1: *Pentium Pro [12, 13] and R10000 [18] memory characteristics*

On the Pentium Pro, we use the Microsoft PowerStation Fortran compiler and NT threads to support cascaded execution. On the R10000, we used the MIPSpro 7.2 Fortran compiler and the `m_fork` command used by the SGI parallelizing compiler to spawn lightweight processes. Timing and cache miss measurements on the Pentium Pro were obtained using hardware counters that can be accessed in about 60 cycles with the `RDTSC` and `RDPMC` instructions. We measured `DCU_LINES_IN` and `L2_LINES_IN` for L1 data cache and L2 cache misses, respectively [6]. Timings on the R10000 were obtained using a 21 ns. resolution counter that can be accessed in about 100 cycles. Cache misses were obtained from hardware counters via a more expensive system call [32].

3.3 Current Performance

Figure 4 shows the overall speedup[4] 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 5 gives execution times in cycles for the fifteen individual loops in that routine. Figure 6 and Figure 7 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 4, 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 5 shows that results for individual loops vary, from a maximum slowdown of 0.9 to a maximum speedup of 4.5.

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

Our results are somewhat limited by the number of processors available to us. Recall that a processor jumps out of the helper phase as soon as it is signaled to begin execution. 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.

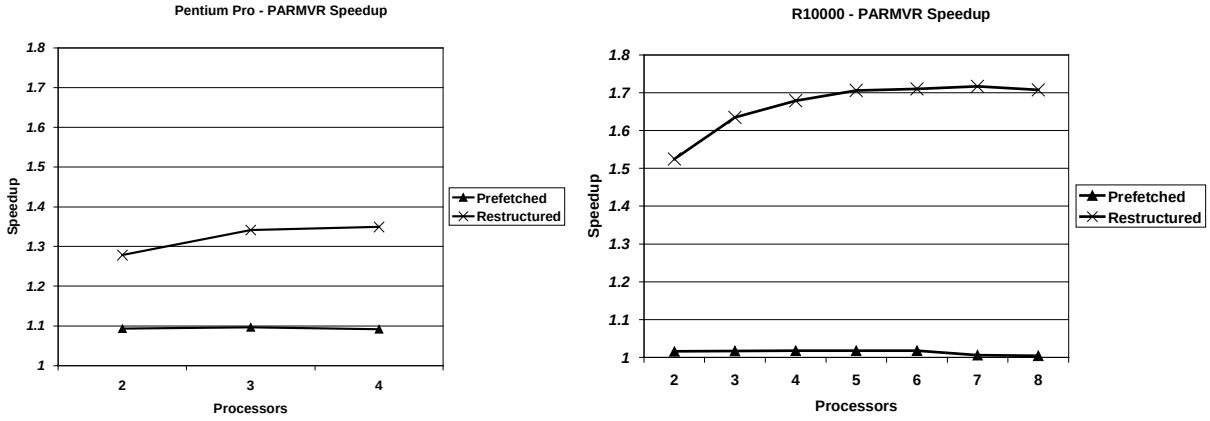

Figure 4: *Overall speedup for PARMVR (64KB chunks)*

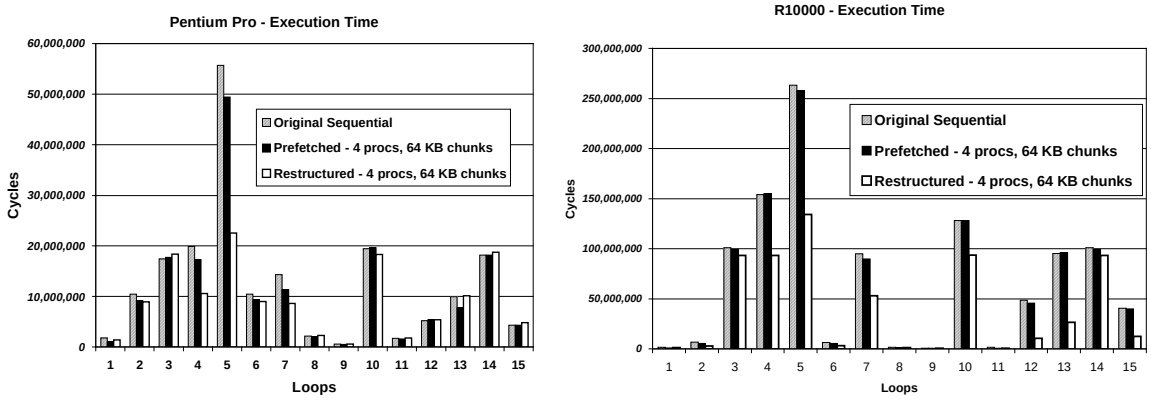

Figure 5: *Execution times of individual loops in PARMVR (four processors, 64KB chunks)*

- *Data restructuring is significantly more effective than prefetching alone.*

Figure 4 and Figure 5 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 less associativity, we see little improvement from prefetching alone. Figure 6 and Figure 7 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 6 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 7 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 is more successful at eliminating L2 cache misses on the Pentium Pro than on the R10000 percentage wise, 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.

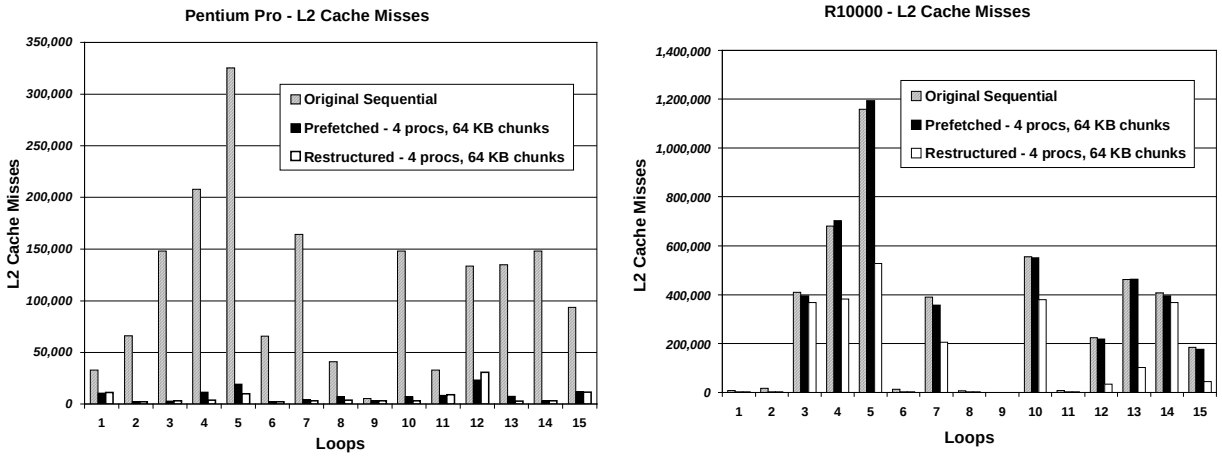

Figure 6: *L2 Cache Misses in PARMVR (four processors)*

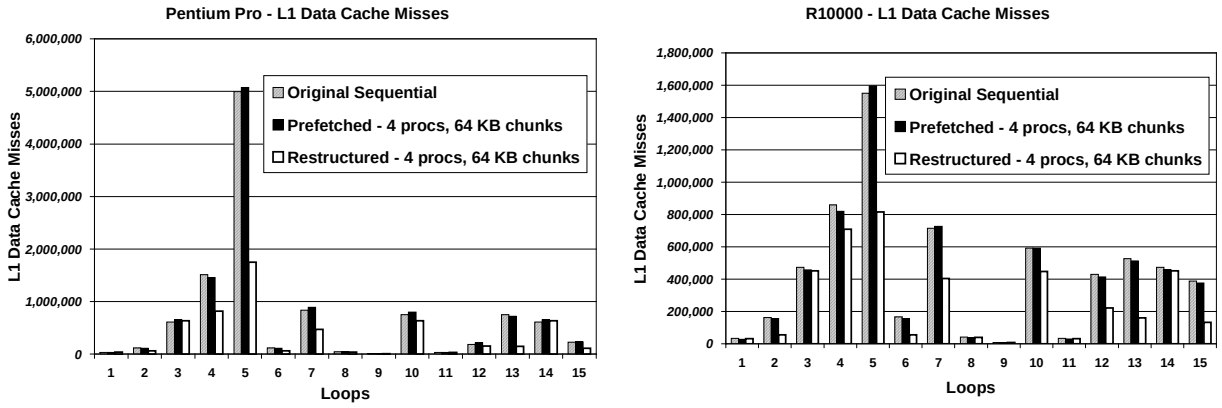

Figure 7: *L1 Data Cache Misses in PARMVR (four processors)*

- *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 8 shows the overall speedup of PARMVR for chunk sizes varying from 4 KB to 2048 KB. On both platforms, the cost of transferring control is significant: ~120 cycles per transfer on the Pentium Pro and ~500

cycles on the R10000[5]. The speedups for PARMVR indicate an optimum chunk size in the range of 16 KB to 64 KB for four processors, which is larger than the L1 cache of either machine.

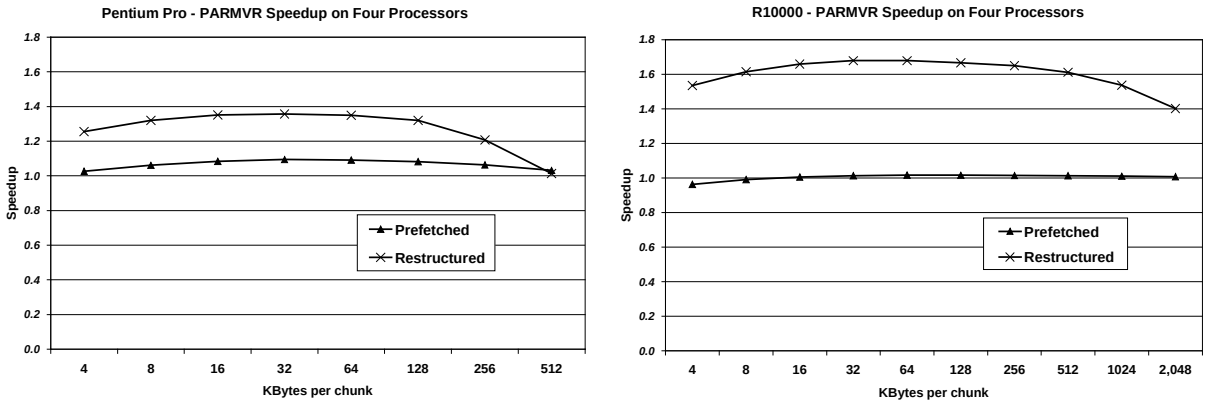

Figure 8: *Effect of chunk size (four processors)*

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 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.

Figure 9 shows the loop used. In this loop, all operands are integers, and the index array `IJ` is simply the vector `1..n`.

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

Figure 9: *Synthetic loop with a high memory access time to computation time ratio*

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.

[5] 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.

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. To calculate overall execution time, we sum the time spent in the execution phases and add in the cost of control transfer (one transfer per chunk) for the number of chunks required for a given chunk size. To obtain speedup we compare this sum to the execution of the original loop running on a single processor.

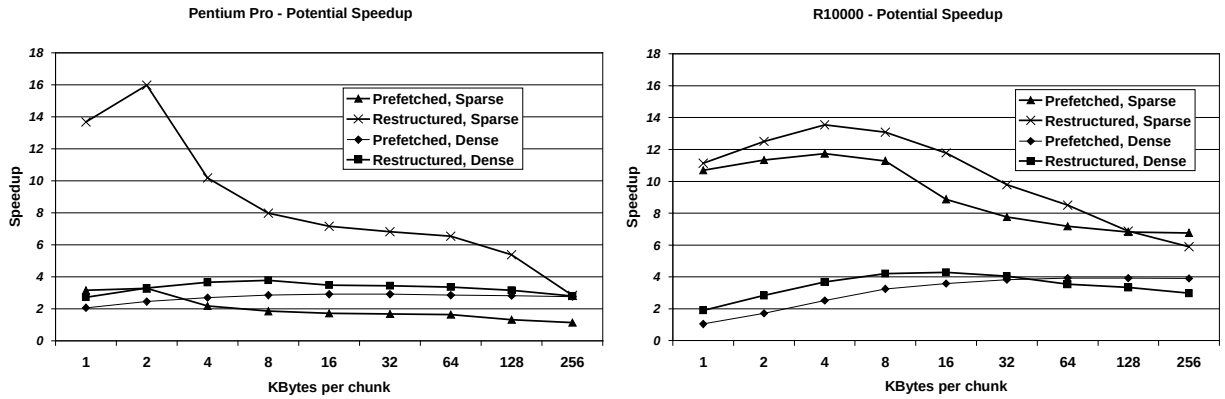

Figure 10: *Cascaded execution speedups with increased memory access costs*

Figure 10 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 [7, 30] and software [9, 10, 20] techniques have been proposed to tolerate memory latency in sequential programs. The approaches most relevant to our work are prefetching and multithreading.

Prefetching may be done in hardware or in software, and may involve prefetching values into cache or into a separate prefetch buffer. Cascaded execution is most similar to software-controlled prefetching [9, 20]. In this approach, the compiler analyzes the program and inserts prefetch instructions for 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.

Hardware prefetching [4] is able to make use of dynamic information not available at compile time, and avoids the instruction overhead that software techniques incur. However, hardware prefetching is usually limited to detecting only constant stride access patterns and can require a nontrivial amount of hardware support both to detect memory accesses that can be prefetched and to retrieve and hold the prefetched values.

Multithreading [1, 27] 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. Further, 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 [15, 21, 22, 31]. Like cascaded execution, these techniques make use of processors that would otherwise be idle if the compiler resorted to simple, sequential execution. The two approaches involve different tradeoffs, however. Speculative and run-time parallelization both require what can often be considerable critical-path processing to either construct a parallel schedule that is guaranteed to be valid or to check that the speculated schedule used was in fact valid. Cascaded execution critical-path overhead is limited to control transfers. Speculative and run-time parallelization require memory overheads that are proportional to the data set sizes touched by the loop, whereas even the restructuring version of cascaded execution has memory overheads bounded by the L2 cache size. Additionally, cascaded execution is applicable even to loops that are inherently sequential, a situation in which the alternatives perform poorly. On the other hand, speedup under speculative and run-time parallelization is limited by the number of processors available, while that of cascaded execution is limited by the fraction of application execution time resulting from memory stalls. For this reason, the maximum benefit obtainable from speculative and run-time parallelization is likely to be larger than that for cascaded execution.

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.

In a theoretical sense, cascaded execution complements work on parallelization techniques in that it affords a useful mechanism to improve performance of inherently sequential code, code to which no future parallelization techniques can apply. In a practical sense, it provides a mechanism that can be used as a fallback whenever the compile-time or run-time techniques actually implemented in a particular compiler fail to find useful parallelizations.

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*, May 1990.

2. J.M. Anderson, S.P. Amarasinghe, and M.S. Lam. Data and Computation Transformations for Multiprocessors. In *Proceedings of the Symposium on the Principles and Practice of Parallel Programming*, July 1995.

3. D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Sureveys*, **26**(4), 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*, November 1991.

5. U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic Program Parallelization. *Proceedings of the IEEE*, **81**(2), 1993.

6. D. Bhandarkar and J. Ding. Performance Characterization of the Pentium Pro Processor. In *Proceedings of the Third International Symposium on High Performance Computer Architecture*, Feb. 1997.

7. D. Burger, S. Kaxiras, and J.R. Goodman. DataScalar Architectures. In *Proceedings of the International Symposium on Computer Architecture*, June 1997.

8. R. Eigenmann, J. Hoeflinger, and D. Padua. On the Automatic Parallelization of the Perfect Benchmarks. *IEEE Transactions on Parallel and Distributed Systems*, **9**(1), 1998.

9. 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*, June 1990.

10. 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*, May 1991.

11. M.W. Hall, J. 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), 1996.

12. Intel Corporation. *Intel Architecture Software Developer's Manual*. Order Number 243190. Vol. 1. Intel Corporation, 1997.

13. 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*, June 1998.

14. S.-T. Leung. Array Restructuring for Cache Locality. Technical Report UW-CSE-96-08-01, Department of Computer Science, University of Washington, Aug. 1996.

15. 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*, May 1993.

16. C. McCann, R. Vaswani, and J. Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, **11**(2), 1993.

17. K.S. McKinley. Evaluating Automatic Parallelization for Efficient Execution on Shared-Memory Multiprocessors. In *Proceedings of the International Conference on Supercomputing*, July 1994.

18. MIPS Technologies Inc. *R10000 Microprocessor User's Manual-Version 2.0*. MIPS Technologies Inc., 1997.

19. J.E. Moreira and V.K. Naik. Dynamic Resource Management on Distributed Systems Using Reconfigurable Applications. *IBM Journal of Research and Development*, **41**(3), 1997.

20. 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*, Sept. 1992.

21. L. Rauchwerger, N.M. Amato, and D.A. Padua. Run-Time Methods for Parallelizing Partially Parallel Loops. In *Proceedings of the International Conference on Supercomputing*, July 1995.

22. 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*, June 1995.

23. J.P. Singh and J.L. Hennessy. An Empirical Investigation of the Effectiveness and Limitations of Automatic Parallelization. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, April 1991.

24. J.P. Singh, J.L. Hennessy, and A. Gupta. Scaling Parallel Programs for Multiprocessors: Methodology and Examples. *Computer*, **26**(7), 1993.

25. B. So, S. Moon, and M.W. Hall. Measuring the Effectiveness of Automatic Parallelization in SUIF. In *Proceedings of the International Conference on Supercomputing*, July 1998.

26. E. Torrie, M. Martonosi, C.-W. Tseng, and M.W. Hall. Characterizing the Memory Behavior of Compiler-Parallelized Applications. *IEEE Transactions on Parallel and Distributed Systems*, **7**(12), 1996.

27. 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*, May 1996.

28. R.Y. Wang, A. Krishnamurthy, R.P. Martin, T.E. Anderson, and D.E. Culler. Modeling Communication Pipeline Latency. In *Proceedings of the SIGMETRICS '98/PERFORMANCE '98 Conference*, June 1998.

29. M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

30. 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.

31. Z. Ye, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *Proceedings of the International Symposium on High Performance Computer Architecture*, Feb 1998.

32. M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proceedings of Supercomputing '96*, November 1996.