Software-Controlled Multithreading Using Informing Memory Operations

Todd C. Mowry

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA 15213
tcm@cs.cmu.edu

Sherwyn R. Ramkissoon

Dept. of Electrical and Computer Engineering
University of Toronto
Toronto, Canada M5S 3G4
sherwyn@eecg.toronto.edu

Abstract

To help tolerate the latency of accessing remote data in a shared-memory multiprocessor, we explore a novel approach to switch-on-miss multithreading that is software-controlled *rather than hardware-controlled. Our technique uses* informing memory operations *to trigger the thread switches with sufficiently low overhead that we observe speedups of 10% or more for four out of seven applications, with one application speeding up by 14%. By selectively applying* register partitioning *to reduce thread switching overhead, we can achieve further gains: e.g., an overall speedup of 23% for* FFT. *Although this software-controlled approach does not match the performance of hardware-controlled schemes on multithreaded workloads, it requires substantially less hardware support than previous schemes and is not likely to degrade single-thread performance. As remote memory accesses continue to become more expensive relative to software overheads, we expect software-controlled multithreading to become increasingly attractive in the future.*

1 Introduction

Memory latency is a key performance bottleneck in modern microprocessor-based systems. As we look to the future, the relative importance of memory latency is expected to increase as the gap between processor and memory speeds continues to grow, and as wider-issue processors increase the effective performance penalty of each cycle of latency. While memory latency presents a challenge for all systems, the problem is especially acute in large-scale shared-memory multiprocessors, where accesses to remote memory locations can suffer latencies on the order of hundreds of cycles [7]. Although cache hierarchies are an essential first step toward coping with this problem, they are not a complete solution. To further tolerate latency, one attractive technique is to use a form of *multithreading* [1, 12, 15] whereby a long-latency access from one thread is overlapped with the computation from other parallel threads.

1.1 Previous Work on Multithreading

Several researchers have proposed and evaluated hardware-based multithreading schemes in the past [1, 2, 6, 12, 15]. (Throughout this paper, we use the term "multithreading" to refer to multithreading for the sake of latency tolerance, as opposed to more general forms of multithreading.) These schemes can be broken down into roughly three categories: *fine-grained*, *coarse-grained*, and *simultaneous* multithreading.

The HEP architecture [12] was an early example of *fine-grained multithreading*. The idea behind this approach is to unconditionally switch between threads at a very fine granularity: i.e. once every cycle. The advantage of this approach is its very low thread switching overhead, since the hardware knows ahead of time when thread switches will occur. The disadvantage, however, is that it relies on having a large number of parallel threads to keep the pipeline full. For applications with limited thread-level parallelism (e.g., when there is only a single thread), performance suffers since each thread can utilize only a small fraction of the processing resources. A more recent example of this style of multithreading is the Tera architecture [2].

Rather than switching between threads on every cycle, the idea behind *coarse-grained multithreading* is to allow a given thread to continue running (with the full processor to itself) until it encounters a long-latency operation; only at that point does the processor switch to executing another thread. An example of this coarse-grained approach is the MIT APRIL architecture [1]. In contrast with the fine-grained approach, coarse-grained multithreading offers better single-thread performance and requires a smaller number of parallel threads to hide latency. The disadvantage of this approach, however, is that since cache misses are detected relatively late in the pipeline, the minimum thread switching time is non-trivially large. Hence this scheme is not appropriate for hiding short latencies (e.g., primary-to-secondary cache misses), and it is primarily used to hide the large latencies found in shared-memory multiprocessors.

Finally, a more recent proposal known as *simultaneous multithreading* [15] leverages the register renaming mechanism within dynamically-scheduled superscalar processors to allow instructions from multiple threads to be active simultaneously within the pipeline. The advantage of this approach is that it enjoys good single-thread performance without paying a significant thread switching penalty.

A common feature of all of these multithreading techniques is that the decision of when to switch between threads and the actual switching itself is controlled entirely by hardware. As a result, a non-trivial amount of hardware support is required to manage the multiple threads. For example, to minimize the thread switching latency, coarse-grained multithreaded processors typically replicate key per-thread state such as the register file [1]. Under simultaneous multithreading, the concept of "thread switching" is effectively eliminated at the point where instructions reach the functional units—i.e. when they are buffered in dynamic instruction scheduling queues—since register renaming has already isolated the effects of independent threads. However, simultaneous multithreading does require some non-trivial hardware support to fetch, issue, and retire instructions from multiple threads properly. More importantly, simultaneous multithreading requires a larger register file to accommodate the multiple threads, and this is likely to increase register access latencies and possibly add additional stages to the pipeline [14]. Concern over the potential impact of multithreading hardware support on single-thread performance may be a contributing factor to why we have yet to see hardware-based multithreading in commodity microprocessors.

Rather than relying on specialized hardware support, an alternative approach is to use *software* to implement multithreading. The advantage of this approach is that there is no degradation in single-thread performance (since the processor is not modified); the disadvantage, however, is that the thread switching time is significantly larger than when it is accelerated by special hardware support, and this may limit the types of latency that can be successfully hidden. Previous studies have considered purely software-based multithreading in the context of hiding remote latencies in software distributed shared memory (DSM) machines [10, 13]. Purely software-based multithreading makes sense for software DSMs for two reasons: (i) software is already invoked upon the start of a remote access, and therefore it knows when to initiate a thread switch; and (ii) remote access latencies are so large in software DSMs [3] (typically several orders of magnitude larger than in hardware DSMs [7]) that the overhead of switching threads in software is small by comparison. As a result, both the Mowry *et al.* [10] and Thitikamol and Keleher [13] studies found positive results when using software-based multithreading to hide the large remote latencies in software DSMs.

An open research question is whether software-based multithreading can successfully tolerate more modest forms of latency, such as the remote latencies in *hardware* DSMs (e.g., the SGI Origin [7]). To implement software-based multithreading, we need two software mechanisms: (i) the ability to switch between threads; and (ii) a mechanism for knowing *when* to trigger thread switches. The former mechanism is clearly feasible, since software can save and restore all thread-specific state (e.g, registers, the program counter, any condition codes, etc.). The latter mechanism, however, had been lacking in the past, since there was no way for software to directly observe and react to cache misses in a sufficiently lightweight fashion. (Note that the signal handler mechanism used to trigger thread switches in software DSMs is not applicable to cache misses, since it is too costly and can only react to page-level access violations.) Fortunately, a mechanism which provides this functionality was recently proposed: *informing memory operations* [4].

1.2 Informing Memory Operations

The idea behind informing memory operations [4] is to make cache misses directly observable to software, and to enable software to react quickly to these misses. In essence, an informing memory operation consists of a memory operation that is combined—either implicitly or explicitly—with a conditional branch-and-link operation where the branch is taken only if the reference suffers a cache miss. Since a given reference is far more likely to to be a cache hit than a miss, the hardware speculates that the branch is not taken. As with incorrectly predicted conditional branches, if the reference actually misses, the informing mechanism squashes any subsequent speculatively scheduled instructions before invoking the miss handler. Horowitz *et al.* [4] describe two possible implementations of informing memory operations: one based on a cache-outcome condition code, and another based on a low-overhead trap.

The low-overhead trap approach works as follows. Two new user-visible registers are added to the architecture: (i) a *Miss Handler Address Register* (MHAR), which contains the address of the miss handler to be invoked upon a cache miss (setting this register to zero disables the trapping mechanism); and (ii) a *Miss Handler Return Register* (MHRR), which contains the return address for resuming execution at the end of the trap (i.e. it contains the address of the instruction following the memory reference that missed). Upon a cache miss, if the MHAR contains a non-zero value, then a branch-and-link occurs to this address, and the MHRR is set appropriately. Unlike traditional trapping mechanisms, this one is extremely lightweight since it occurs entirely at the user level, and the only state that is saved is the MHRR. Hence the run-time overhead is comparable to a traditional branch-and-link instruction, rather than a traditional trap. The earlier study [4] demonstrates how this mechanism can be implemented within modern in-order and out-of-order superscalar pipelines without much additional complexity, since the bulk of the necessary hardware support already exists for handling branches and exceptions. The advantage of the low-overhead trap approach is that it potentially incurs no overhead on cache hits (unlike the cache-outcome condition code approach, which requires an explicit test of the condition code even on cache hits). Hence we will focus on the low-overhead trap approach throughout the remainder of this paper.

There are a number of applications of informing memory operations. For example, since they can be used to collect memory performance information accurately and with little overhead, informing memory operations enable a wide range of new performance monitoring tools which can guide either the programmer or the compiler in identifying and eliminating memory performance problems. In addition, the earlier study [4] also demonstrated that informing memory operations can automatically enhance the performance gains from software-controlled prefetching [8, 9], and that they can accelerate software-based cache coherence with fine-grained access control [11]. Although that study suggested that informing memory operations could be used to implement software-controlled multithreading, there has been no detailed study of this approach until now.

1.3 Objectives of This Study

In this paper, we perform a detailed evaluation of whether software-controlled multithreading based on informing memory operations can successfully improve the performance of parallel applications running on shared-memory multiprocessors with hardware cache coherence. In addition to evaluating our baseline scheme, we also investigate a number of extensions which are designed to further enhance the performance of software-controlled multithreading.

We focus on hardware DSMs rather than uniprocessors for two reasons. First, since applications written for hardware DSMs already contain parallel threads, it is straightforward to extract the additional parallel threads necessary for multithreading. (In contrast, the bulk of applications run on uniprocessors contain only a single thread, and parallelizing requires non-trivial effort.) Second, hardware DSMs tend to suffer more from memory latency than uniprocessors—due to the large latency of remote accesses and the additional cache misses due to communication patterns—and therefore they are an important target for latency tolerance. If software-controlled multithreading on hardware DSMs is successful, then we get the best of both worlds: the benefits of multithreading when it pays off, and maximum single-thread performance when it does not.

Finally, to help quantify the performance tradeoffs between software-controlled and hardware-controlled multithreading, we compare the performance of our scheme with a purely hardware-based implementation of coarse-grained multithreading.

The remainder of the paper is organized as follows. Section 2 discusses the issues involved in implementing software-controlled multithreading. Sections 3 and 4 present our experimental methodology and results. Finally, we conclude in Section 5.

2 Software-Controlled Multithreading

The basic idea behind software-controlled multithreading is as follows. Using the low-overhead trap version of informing memory operations, we store the address of our miss handler in the MHAR once at the beginning of the program. Whenever a reference suffers a primary data cache miss, the informing mechanism squashes any instructions after that memory reference, sets the MHRR to contain the address of the instruction following the offending memory reference, and transfers control to the miss handler. (In essence, the cache miss is treated like mispredicted control speculation, where the incorrectly speculated path was to continue executing past the memory reference, but the correct path was to jump immediately to the miss handler.) In a typical application of informing memory operations, the miss handler would perform a small amount of work and then resume execution immediately after the offending memory reference [4]. In contrast, we use the miss handler to suspend the current thread and resume execution of another user thread. An important insight is that when we switch to a new thread, we begin executing it *immediately*—i.e. while the original cache miss is still in progress—in order to overlap this useful computation with the miss latency.

In the remainder of this section, we discuss the major challenges and tradeoffs involved with implementing software-controlled multithreading. We begin by discussing the hardware support necessary for this scheme, and then present a design of the miss handler software which performs the actual thread switching. Finally, we discuss how our scheme avoids deadlock and handles synchronization events properly.

2.1 Hardware Support

The target architecture for our study is a hardware cache-coherent shared-memory multiprocessor comprised of out-of-order superscalar processors. For the sake of concreteness, we will use the MIPS R10000 [19] as the basis for our discussion, although similar issues apply to other out-of-order superscalar processors.

Our goal is to support software-controlled multithreading with minimal hardware support beyond informing memory operations. (The hardware support necessary for informing memory operations has been described in detail in an earlier publications[4].) There are three issues, however, which may require some additional hardware: the first two involve potential problems that would prevent us from overlapping enough computation with the cache miss, and the third involves our ability to selectively switch threads only upon long-latency misses.

The first obstacle to consider is that when a load suffers a cache miss, it typically cannot retire from the reorder buffer until its cache miss has completed. Since all instructions must retire in-order (even in an out-of-order issue machine), this means that all instructions executed starting in the trap handler (and beyond) after the miss (including thread switching code and the thread that we switch to) must remain in the reorder buffer until the miss completes. (Recall that any instructions following the load in the original user thread have already been squashed and removed from the reorder buffer, and hence they are not a concern.) The problem is that reorder buffers are typically small (e.g., 32 entries in the R10000) relative to the number of instructions that one would need to execute to fully hide a remote cache miss (e.g., several hundred instructions in the SGI Origin). Hence the reorder buffer will fill up quickly upon a thread switch, causing the processor to stall before it can hide the miss latency. For example, the R10000 does not have sufficient buffering to even execute our thread switching code (described later in Section 2.2), let alone the thread that we are attempting to activate. To address this problem, we need a mechanism for specifying that the load should be allowed to retire, despite the fact that its miss is still in progress. In essence, we would like to convert the load into a *prefetch*, since prefetches can retire before their misses complete. Converting the load to a prefetch is acceptable because we do not care about the result of the load—only that it brings the line into the cache—since we will resume execution by re-executing the load that missed (shown later in Section 2.2). While there are many ways to accomplish this, one possibility is to set a flag which indicates to the trapping mechanism that upon a cache miss, the offending load should be allowed to retire (similar to a prefetch). Such an option may be useful in other cases where the miss handler would like to execute a non-trivial amount of code underneath the cache miss, and where the miss handler will resume execution by re-executing the load which invoked the trap, rather than the instruction after it.

The second potential problem is that during a thread switch, any use of the load destination register (e.g., if we attempt to save it to memory as part of saving the thread state) will result in a data dependence that will stall the processor until the load completes. Since we do not care about the result of the load (it will be re-executed later), there is no need to save this register value. One software-based solution would be to save all registers except the load destination; therefore when the register state of the thread we are switching to is restored, the act of overwriting this register will break the original data dependence on the load (due to register renaming), thus avoiding a stall. While this approach will work, the problem is how to quickly determine which register is the load target (since this information is not readily available inside the miss handler) and avoid saving it. One possibility would be to look up this value in a hash table based on the return address in the MHRR; however, this will result in non-trivial software overhead. Another possibility would be for the hardware to make the destination register number directly visible to the miss handler software, perhaps through another special architected register. While this would eliminate the need for a hash table lookup, we would still need to branch to a specialized version of the thread switching code to avoid saving the given register. The most desirable solution would be for the hardware to automatically break the data dependence on the load result when it is marking the load as being able to retire despite its outstanding miss (as discussed earlier). In other words, we would like to fully convert the load to having the same functionality as a prefetch: i.e. it can graduate immediately, and it produces no register result. Breaking this register dependence is realistic for the hardware because the Miss Status Handling Register (MSHR) [5]—the structure which tracks an outstanding miss in a lockup-free cache—already maintains this register number. Since any instructions dependent on the result of this load are already squashed by the informing memory mechanism, we can immediately free this load's destination register. In our experiments, we assume that this latter hardware support is available.

The third area where additional hardware support may be helpful is in identifying (or predicting) whether a given cache miss is likely to suffer a large latency. Since multithreading can only improve performance if the miss latency is larger than the latency of switching between threads—and since our software-based approach requires roughly 55 cycles to switch threads—we cannot hide the latency of primary cache misses which hit in the secondary cache. Hence we only want to switch threads upon secondary cache misses (which are still large relative to our thread switching time). Ideally, we would like an informing mechanism where traps only occur upon secondary misses—however, implementing this may be difficult (or even impossible) given how late the secondary cache tags are checked. Instead, we assume that traps can only occur upon primary cache misses, but that inside the miss handler we can test a flag which indicates whether the primary miss is also a secondary cache miss. This is similar to the condition-code approach that was discussed by Horowitz *et al.* [4].

Note that in all three of these cases, the additional hardware support only affects actions taken upon miss handler invocation, and there is flexibility in how quickly the actions are performed. Hence we would not expect any of these features to slow down the critical path of normal execution. Having described our hardware support, we now discuss how it can be used to implement the miss handler.

2.2 Design of the Miss Handler

We use a single miss handler to implement multithreading, as shown in Figure 1. The MHAR is set to contain this handler address at the start of execution, and is restored after each trap so that we continue using this same handler. As we see in Figure 1, the miss handler begins by subtracting four bytes from the MHRR so that it will eventually restart the thread at the memory reference that missed, rather than at the instruction after it. The reason for doing this is that the original reference

```
HandlerAddr:
       add MHRR, MHRR, -4    // Point MHRR to prev inst
       bne #0, CMF, L2Miss   // Proceed if miss flag set
       j MHRR                // else L2 hit, just return
L2Miss:
       li MHAR, #0           // Disable miss-handler
       li k0, #Membase       // Get ptr to current state
       lw k0, 0(k0)
       sw r1, 0(k0)          // Save integer registers
       sw r2, 4(k0)          // excluding k0,k1,r0
       ...
       sw r31, 112(k0)
       sw fcr31, 116(k0)     // Save fp cond code reg
       sd f0, 120(k0)        // Save fp registers
       sd f2, 128(k0)
       ...
       sd f30, 240(k0)
       sw MHRR, 248(k0)      // Save MHRR
       addu k0, k0, 256      // Get ptr to new context
       and k0, k0, #FFFF0FFF// Assume 16 contexts,
                             // 256 bytes/context,
                             // round robin selection
       sw k0, Membase
       lw r1, 0(k0)          // Restore integer registers
       lw r2, 4(k0)
       ...
       lw r31, 112(k0)
       lw fcr31, 116(k0)     // Restore fp cond code reg
       ld f0, 120(k0)        // Restore fp registers
       ld f2, 128(k0)
       ...
       ld f30, 240(k0)
       ld MHRR, 248(k0)      // Restore MHRR
       li MHAR, #HandlerAddr  // Re-enable miss-handler
       j MHRR                // Jump to new context
```

Figure 1. MIPS pseudo-code representation of the miss handler for software-controlled multithreading.

has been converted into a prefetch by the hardware (as discussed in the previous section), and therefore the reference must be re-executed to complete properly. The handler then tests whether the primary miss was also a secondary cache miss.[1] If so, then the handler switches to a new thread; otherwise, it returns immediately.

To switch between threads, the miss handler saves the state of the current thread, selects a new thread to execute, and restores the state of this new thread. To prevent the memory references inside the miss handler from triggering additional informing memory traps, the trapping mechanism is disabled during the thread switch by writing a zero into the MHAR. Since user code in MIPS-based systems does not use the `k0` register, we use it as a pointer to where the thread state is stored. Assuming that the number of active threads per processor is a power of two, our simple round-robin scheme requires only three instructions to determine the next thread to be executed. Finally, the handler resumes thread execution by jumping to the address in the MHRR.

Hence there are two major dimensions to consider when performing multithreading in software: (i) how to manage the saving and restoring of thread state; and (ii) how to decide when it is desirable to switch threads. We now consider both of these issues in greater detail.

[1] Note that the processor will stall until the secondary cache miss flag is valid. If this is likely to take a non-trivial amount of time, then some of the thread switching code can be scheduled before this test to avoid wasting time (e.g., one can begin saving the current register state to memory, since this work would not need to be explicitly discarded if a thread switch did not actually take place).

2.2.1 Saving and Restoring Thread State

Our multithreading scheme is similar to coarse-grained hardware-based schemes (e.g., APRIL [1]) in that thread switches are triggered by cache misses. An important difference, however, is that while these hardware-based schemes devote special hardware to saving and restoring registers quickly, we must save and restore registers through explicit loads and stores to memory. This overhead accounts for the bulk of our thread switching latency (which is roughly 55 cycles). The good news is that the thread state tends to stay in the primary data cache, which prevents the latencies from being even larger. However, since these non-trivial thread switching times are a potential performance bottleneck, we now considers ways to reduce them further.

The major trick for reducing the thread switching overhead is to avoid saving and restoring registers that do not need to be preserved. As a simple example, some applications do not use floating-point registers at all; by recognizing this fact, we could eliminate roughly half of the thread switching overhead in such applications. In general, the compiler can determine which registers are live at any given point in the program, and it could use this information to select a miss handler that has been customized to only save these live registers. While this approach may sound good in theory, it suffers the following limitations in practice. First, customizing the miss handler on a reference-by-reference basis involves either setting the MHAR before each reference, or else using the MHRR inside the miss handler to hash into a jump table. The Horowitz *et al.* study [4] quantified these types of overheads, which appear to be large enough to offset a non-trivial portion of the expected gains. A related limitation is that creating a large number of customized miss handlers will degrade the instruction cache performance. Finally, while customizing which registers are saved simply involves setting the MHAR as appropriate, customizing which registers are *restored* may involve jumping to a specialized restore handler whose address must be stored with the state of the suspended thread that we are restarting.

A simpler approach to reducing the overhead of saving and restoring registers is to *statically partition* the registers between threads. For example, if we wanted to run two threads per processor, the compiler could compile each thread to use only half of the user registers. (Note that special-purpose registers—e.g., the stack pointer—cannot be partitioned.) The advantage of this approach is that many of the registers would be preserved in the register file itself, thus avoiding the need to save them to memory. The main disadvantage, however, is that each thread may suffer reduced performance due to having fewer available registers. (Another disadvantage is that code replication may degrade the instruction cache performance.) Rather than taking an all-or-nothing approach, there is in fact a continuum of possibilities between saving all registers and partitioning all user registers. For example, it may be beneficial to give each thread one additional register at the expense of slightly increased switching overhead. Waldspurger and Weihl [16] explored a number of these issues in the context of earlier multithreaded architectures, and in this study we focus on the impact of static register partitioning on the performance of software-controlled multithreading, as discussed later in Section 4.

2.2.2 Deciding When to Switch Threads

The second major challenge for software-controlled multithreading is switching threads only when the miss latency is expected to be large relative to the thread switching overhead. For our purposes, this means switching only upon secondary cache misses. Unfortunately, as we mentioned earlier, it is not likely that the result of the secondary cache tag check will be available early enough to trigger a trap. Instead, the strategy which we outlined in Figure 1 is to test whether the primary miss (which triggered the trap) is also a secondary cache miss once we are inside the miss handler. The main disadvantage of this approach is that if the reference does hit in the secondary cache, then we have wasted overhead with no benefit.

To avoid this useless overhead, we would like to predict *a priori* whether a given reference is likely to result in an expensive cache miss. If we believe that it will not, then we can disable the trapping mechanism for that reference. One possibility would be for the compiler to statically analyze the data locality [9, 17]; this technique has mainly been successful for matrix-based codes. Another possibility would be to collect a profile of how frequently each memory reference suffers a long-latency miss, and to feed this information back into the compiler. Finally, another possibility would be to use hardware to predict the conditional probability of a reference suffering a long-latency miss, given that it has suffered a primary cache miss. Such a prediction mechanism could use techniques similar to those used for branch prediction. With this information, the user could specify that they would like informing traps to occur only upon primary cache misses which are *also* predicted to be expensive misses. Implementing this behavior would be feasible since both the primary cache miss signal and the "expensive miss" prediction value would be available early enough to control the trap mechanism.

Of course, the drawback of using a prediction mechanism is that if it incorrectly predicts that a miss will be inexpensive when it turns to be expensive, then it is too late to invoke the thread switching code to hide the miss latency. We will evaluate the potential benefit of such techniques later in Section 4.

2.3 Avoiding Deadlock and Handling Synchronization Properly

By interleaving multiple threads on the same physical processor, multithreading introduces the possibility of deadlock in two ways. First, a repeated pattern could occur where thread A steals resource X from thread B (which is currently suspended, also waiting for resource X), only to suffer a thread switch back to B before A can use X; when thread B restarts, it steals resource X back from thread A, but also switches back to A before B can use X, etc. Such a pattern could be repeated infinitely as the two threads rapidly switch back and forth but neither thread makes progress. This scenario can arise when multiple threads suffer cache misses for unique addresses which map into the same cache entry. To prevent this problem, we swap out a given thread only *once* when it encounters a cache miss. If the miss has not completed by the time the round-robin scheduler reactivates the thread, then the thread stalls at that point until the miss completes (rather than switching to another thread).[2] Hence forward progress is guaranteed.

The second scenario which can result in deadlock is if thread A spin-waits for a resource that is held by thread B, where B is currently suspended on the same processor as A, and A never yields the processor to B in the course of spin-waiting. This scenario can arise with any form of synchronization that involves spin-waiting (e.g., locks and barriers). Our solution is to force a thread switch (i.e. we unconditionally jump to code similar to `L2Miss` in Figure 1) as part of all spin-waiting loops. Not only does this approach avoid deadlock, it also helps us hide synchronization latency.

Table 1. Simulation parameters.

Pipeline Parameters	
Issue Width	4
Functional Units	2 Int, 2 FP, 2 Mem, 1 Branch
Reorder Buffer Size	32
Integer Multiply	12 cycles
Integer Divide	76 cycles
All Other Integer	1 cycle
FP Divide	15 cycles
FP Square Root	20 cycles
All Other FP	2 cycles
Branch Prediction	2-bit Counters

Memory Parameters	
Line Size	32B
Instruction Cache	32KB, 2-way set-assoc
Data Cache	32KB, 2-way set-assoc
Unified Secondary Cache	2MB, 2-way set-assoc
Data Cache Banks	2
Data Cache Fill Time	4 cycles
Miss Handlers (MSHRs)	16 for data, 2 for insts
Main Memory Bandwidth	1 access per 20 cycles
Total Miss Latency to Secondary Cache	14 cycles
Total Miss Latency to Local Memory	78 cycles
Total Miss Latency to Remote Memory	200 cycles (2 hops), 300 cycles (3 hops)

[2] One way to accomplish this is if the informing memory mechanism supports sampling as follows: the user can set a counter value which indicates how many misses must occur before the informing mechanism is invoked (such a sampling mechanism is common in many modern processors which have performance counters). By setting this sampling counter such that the first miss will be ignored, we can avoid this problem.

Table 2. Benchmark characteristics table describes the benchmarks, input data set, and cache miss counts. The total number of misses, misses that hit in local memory, and remote miss counts are given for the 2-processor case.

Name	Description	Input Data Set	Instructions Graduated	Cache Miss Count		
				Total	Local Mem.	Remote
CHOLESKY	Sparse Cholesky factorization	tk14.O input file	44.6M	289K	28.8K	22.5K
FFT	1D fast Fourier transform	65536 complex points	30.1M	256K	123.8K	59.6K
LU-CONT	LU factorization with contiguous partitions	512x512 matrix, 32x32 elem. blocks	184M	755K	45.8K	50.7K
LU-NCONT	LU factorization with non-contiguous partitions	512x512 matrix 32x32 elem. blocks	205M	7508K	66.7K	62.7K
OCEAN-CONT	Large-scale ocean simulation with contiguous partitions	130x130 grid	48.9M	2009K	52.2K	1.5K
OCEAN-NCONT	Large-scale ocean simulation with non-contiguous partitions	130x130 grid	65.6M	2374K	284.4K	15.9K
RADIX	Integer radix sort	262144 keys, radix=1024, max key value=1024	25.9M	197K	23.2K	24.4K

3 Experimental Framework

To evaluate our software-controlled multithreading scheme, we performed detailed cycle-by-cycle simulations of a collection of seven applications from the SPLASH-2 benchmark suite [18] on a shared-memory multiprocessor with out-of-order superscalar processors similar to the MIPS R10000 [19]. Our simulation model varies slightly from the actual MIPS R10000—e.g., we model two memory units, and we assume that all functional units are fully-pipelined. However, we do model the rich details of the processor, including the pipeline, register renaming, the reorder buffer, branch prediction, instruction fetching, branching penalties, the memory hierarchy (including contention), etc. The parameters of our model are shown in Table 1.

We model a multiprocessor based roughly on the SGI Origin [7]. We use a full-map directory to implement invalidation-based cache coherence. Remote accesses require either two or three network hops, depending on whether the data can be supplied by the *home* node or whether it must be forwarded from a *dirty-remote* node. We do not model network contention, but we do model memory contention in detail. As shown in Table 1, the two and three hop remote accesses result in nominal latencies of 200 and 300 cycles, respectively, not including additional delays due to memory contention.

We would like to emphasize that we simulate the *actual* thread-switching instructions shown in Figure 1, rather than simply modeling thread-switching as some fixed latency. In addition, we precisely model the timing of the trap mechanism for informing memory operations in the R10000, as described by Horowitz *et al.* [4]. Our thread-switching code consists of a total of 104 instructions—of these, 94 are memory references. Given that our processor has two memory units, the memory references alone would dictate a minimum thread switching time of at least 47 cycles. Since we also model the instruction and data cache misses caused by the miss handler code, data dependences, resource constraints, etc., we observe a thread switching latency that is closer to 55 cycles. (The actual thread switching time varies across applications, and in one case is over 100 cycles, as we will see later in Section 4.)

We performed our experiments using the following SPLASH-2 applications: CHOLESKY, FFT, LU-CONT, LU-NCONT, OCEAN-CONT, OCEAN-NCONT, and RADIX. Table 2 briefly summarizes each application, along with the input data sets and other statistics. Further details on these applications can be found in the study by Woo *et al.* [18]. All applications were compiled using version 2.8.0 of the `gcc` compiler, with `-O3` optimization. We used the *MINT3* MIPS instruction interpreter tool (provided by MIPS) to drive our detailed performance model, thus allowing us to simulate all instructions (including the thread-switching code) in a detailed, execution-driven fashion.

4 Experimental Results

We now present results from our simulation studies. We begin by evaluating the performance of our baseline software-controlled multithreading scheme. To further improve upon this scheme, we evaluate the performance potential of two techniques for reducing overheads: (i) *register partitioning* to reduce the thread switching overheads, and (ii) *miss prediction* to avoid invoking the miss handler upon secondary cache hits. Finally, we compare the performance of software-controlled vs. hardware-controlled multithreading.

4.1 Performance of the Baseline Software-Controlled Multithreading Scheme

The results of our first set of experiments can be found in Figure 2 and Tables 3 and 4. Figure 2 shows the performance impact of multithreading with two threads per processor on two- and four-processor machines.[3] Each bar is labeled with the number of threads on each processor, and the number of processors.

The execution times are normalized to the case without multithreading on two processors, and they are broken down into nine categories explaining what happened during all potential graduation slots. The number of graduation slots is the issue width (4 in this case) multiplied by the number of cycles. We focus on grad-

[3] Since *MINT3* was only able to simulate up to eight parallel threads, we were not able to explore larger machine configurations. By focusing on smaller machine configurations, we tend to underestimate the fraction of secondary cache misses that would be remote in a larger machine—hence our results are conservative since the potential performance gains are likely to be larger in larger-scale machines.

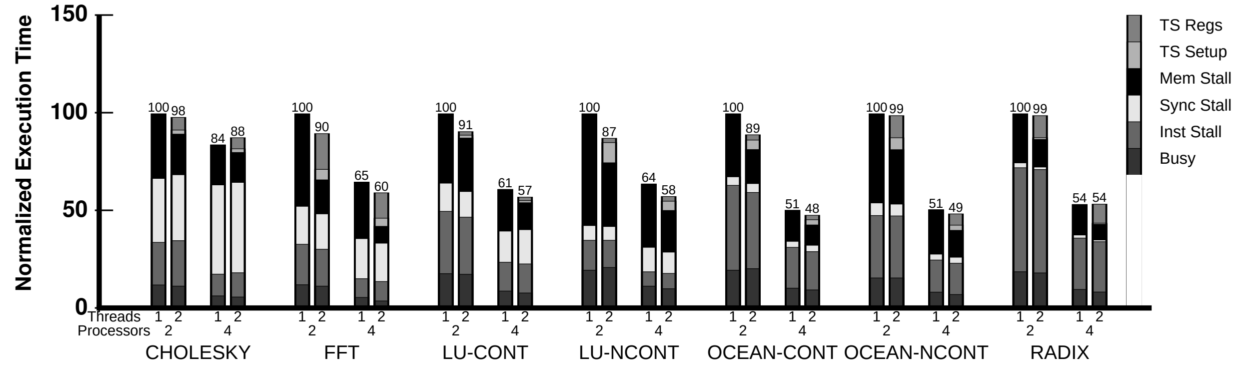

Figure 2. Performance of the baseline software-controlled multithreading scheme.

Table 3. Breakdown of memory stall time for the baseline software-controlled multithreading scheme into misses found in the L2 cache, local memory, or remote memory, and misses combined with other misses. Performance is normalized to the 2-processor case with no multithreading.

Benchmark	# of Procs	Threads per Proc	Stalls Due to L1 Misses Found in Location Below: L2	Mem	Rem.	Comb.
CHOLESKY	2	1	2.2	3.3	5.0	22.7
		2	0.7	1.1	0.0	19.1
	4	1	0.8	2.5	4.8	12.5
		2	0.1	0.5	0.0	14.8
FFT	2	1	1.1	15.5	15.8	15.1
		2	0.0	0.0	0.0	17.5
	4	1	0.5	7.6	15.0	6.1
		2	0.0	0.0	0.0	8.7
LU-CONT	2	1	2.0	1.2	3.2	29.2
		2	0.1	0.2	0.0	27.2
	4	1	0.9	0.8	5.7	14.1
		2	0.1	0.0	0.0	13.8
LU-NCONT	2	1	23.3	1.4	3.4	29.4
		2	0.3	0.2	0.0	32.3
	4	1	8.0	0.1	5.8	18.7
		2	0.2	0.0	0.0	21.3
OCEAN-CONT	2	1	12.7	3.8	0.4	15.5
		2	0.0	0.0	0.0	17.5
	4	1	6.1	0.7	0.9	8.3
		2	0.0	0.0	0.0	10.5
OCEAN-NCONT	2	1	11.2	13.9	1.2	19.5
		2	0.0	0.0	0.0	28.0
	4	1	4.9	7.0	1.0	9.6
		2	0.0	0.0	0.0	13.9
RADIX	2	1	2.1	3.8	8.1	11.2
		2	0.1	0.2	0.0	13.8
	4	1	1.0	1.4	7.6	5.7
		2	0.1	0.1	0.0	7.9

uation rather than issue slots to avoid counting incorrectly speculated operations. The bottom section (*Busy*) is the number of slots when instructions actually graduate. The *Mem Stall* and *Sync Stall* sections are any non-graduating slots that can be directly attributed to data cache misses or synchronization, respectively. Table 3 breaks down the *Mem Stall* slots further into four categories: the first three are when a primary cache miss is ultimately found in the secondary cache, local memory, or requires a remote access, respectively; the fourth case (labeled *Comb.*) is when a primary cache miss is combined with another outstanding miss in progress. Returning to Figure 2, the top two sections in the multithreading cases represent slots due to the thread switching code; these are broken down into time spent saving and restoring registers (*TS Regs*) and the remaining miss handler time (*TS Setup*). Finally, the *Inst Stall* section is all other slots where instructions do not graduate. Note that these categories are only a first-order approximation of what is limiting performance, due to the inherent parallelism within an out-of-order superscalar processor and the fact that delaying one dependence tends to exacerbate subsequent dependences.

As we see in Figure 2, software-controlled multithreading results in speedups ranging from 10% to 14% in four of the seven applications (FFT, LU-CONT, LU-NCONT, and OCEAN-CONT), and more modest speedups of 1-2% in the other three cases. We also experimented with using more than two threads per processor but found that in most cases, adding more threads reduced performance rather than improving it. For example, while FFT ran noticeably faster with four threads on each processor, OCEAN-CONT went from a 12% speedup with two threads per processor to a comparable slowdown with four threads. The reason for this, as we will see later in this section, is that two threads are sufficient to fully tolerate the miss latencies.

Let us begin by focusing on the impact of multithreading on memory stall times. We observe that without multithreading, six of the seven applications (all except RADIX) are spending over a third of their time stalled waiting for data when running on two processors; in three of these cases (FFT, LU-NCONT, and OCEAN-NCONT), roughly half of the execution time is lost to memory stalls. By exploiting 2-way multithreading on two processors, we are able to hide 23% to 63% of the memory stall time; in six of the seven cases, multithreading hides over 35% of these stalls. As we see in Table 3, the bulk of the remaining miss latency with multithreading is due to misses that combine with other outstanding misses. For these combined misses, we are able to partially (but not fully) hide the memory latency.

The benefit of reduced memory stall times is at least partially offset by the thread switching overheads. In four of the seven applications (CHOLESKY, LU-CONT, LU-NCONT, and OCEAN-CONT), the switching overhead with two threads each on two processors is less than 30% of the original memory stall time; in the other three cases, however, this overhead is almost one-half of

Table 4. Additional baseline multithreading statistics.

Benchmark	Avg. L2 Cache Miss Latency (cycles)	Average Run Length (cycles)	Average Thread Switch Time (cycles)
CHOLESKY	127	826	71
FFT	114	161	56
LU-CONT	143	1769	57
LU-NCONT	139	1366	53
OCEAN-CONT	84	1125	55
OCEAN-NCONT	83	260	54
RADIX	137	627	108

the original memory stall time. It is not surprising that the thread switching times are non-trivially large, given that all of the thread switching is performed by software. The good news, however, is that the thread switching times are actually small enough that we do see some performance gains. For example, even though FFT experiences a large thread-switching overhead, it still enjoys an 11% speedup with software-controlled multithreading. As we see Figure 2, the bulk of the thread switching overhead is usually due to saving and restoring registers, as opposed to other time spent in the miss handler. (The major exception to this is LU-NCONT, where most of the time is spent entering the miss handler and then deciding not to switch threads due to the reference hitting in the secondary cache.) Later in this section, we will evaluate techniques for reducing this thread-switching overhead.

We observe that multithreading generally had no positive impact on synchronization stalls. The main reason for this is that the bulk of the synchronization stalls in these applications are due to barriers; since barrier stall times are dominated by load imbalance, which is not directly improved by latency tolerance, multithreading has little impact on these synchronization stalls.

To provide further insight into the multithreading behavior, Table 4 shows the following statistics: (i) the average secondary cache miss latency, which is the latency that a thread switch attempts to hide; (ii) the average *run length*, which is how long a thread executes between thread switches; and (iii) the average thread switching latency. (These numbers were collected from the case with two threads per processor on two processors, but the same trends hold in the other multithreading configurations.) First, we observe that the average secondary cache miss latency is significantly larger than the average thread switching latency in all cases. If this were not true, then the overhead of multithreading would offset any potential gains. Aside from the two versions of OCEAN (which are dominated by capacity misses, and where there is sufficient locality such that most secondary cache misses hit in local memory), the average miss latencies in the other applications are over 110 cycles due to a reasonably large fraction of secondary cache misses requiring remote communication. While five of the seven applications have thread switching latencies ranging from 53 to 57 cycles, CHOLESKY and RADIX experience much larger switching latencies: 71 and 108 cycles, respectively. These larger switching latencies are primarily caused by the application displacing both the instructions and data associated with thread switching from the caches between thread switches.

Roughly speaking, we would expect the performance to saturate when the number of additional threads beyond the main thread is equal to $\frac{L}{R+C}$, where L, R, and C are the average miss latency, run length, and thread switching latency, respectively. Given the data in Table 4, we would expect to reach this saturation point with only one additional thread per processor. Indeed, when we simulated the benchmarks with four threads rather than two on each processor, performance only improved in one of the cases, namely FFT. Not surprisingly, FFT is also the case with the smallest average run length.

Finally, we observe that when multiple threads share the same physical cache, they can potentially interfere with each other either *constructively* (by effectively prefetching another thread's working set) or *destructively* (by displacing another thread's working set). While we did not observe any cases where destructive interference was problematic, we did observe a case of positive interference. In LU-NCONT, consecutive threads often access the same cache lines. When these threads are on separate processors, this sharing pattern results in communication and remote accesses. When consecutive threads are assigned to the *same* processor, however (as occurs under multithreading), one thread effectively prefetches the data set of another thread.

In summary, we have seen that our baseline software-controlled multithreading scheme can improve performance. However, a key bottleneck which is limiting further performance gains is the time spent switching between threads in software. To address this problem, we now consider techniques for reducing this overhead.

4.2 Register Partitioning

As we discussed earlier in Section 2.2.1, one approach to reducing the thread switching overhead is to partition the register set between threads, thereby reducing the number of registers that must be saved and restored. To perform these experiments, we recompiled each application using the `-ffixed` flag in `gcc` to control how many user registers could be allocated to a given thread. The following special-purpose MIPS registers could not be partitioned, and must still be saved and restored upon a thread switch: `at`, `v0-v1`, `a0-a3`, `gp`, `sp`, `fp`, `ra` and `fcr31`. By partitioning the remaining 21 general-purpose and 32 floating-point registers between threads, we were able to reduce the thread switching code to only 34 instructions, 24 of which were memory references. This reduced the average thread switching latency to as little as 18 cycles, as shown in Table 5. As we see in Table 5, register partitioning reduces the thread switching latency by at least a factor of 2.5 in all cases.

Figure 3 shows the impact of register partitioning on performance. For each multithreading case, we show two bars: **B** is the base case (shown earlier in Figure 2),

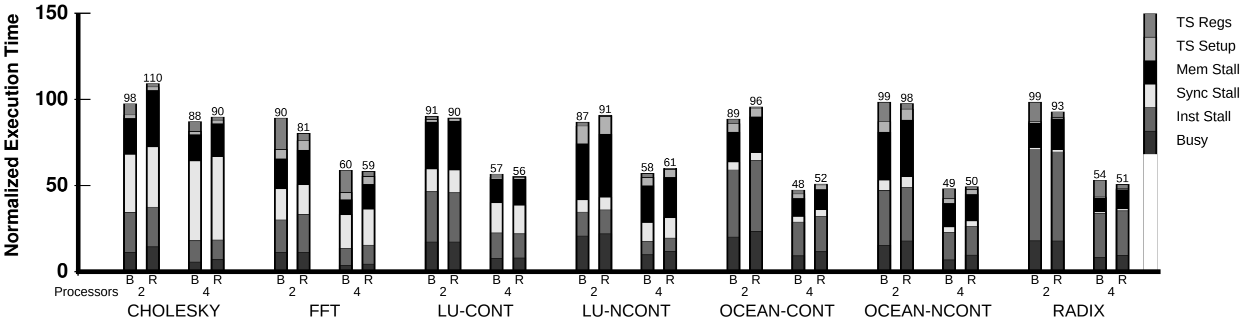

Figure 3. Impact of register partitioning on performance (B = baseline multithreading, R = multithreading with register partitioning). Execution times are normalized to the case without multithreading on two processors.

Table 5. Impact of register partitioning on thread switching latencies.

Benchmark	Avg. Thread Switching Latency (cycles)	
	Baseline Case	Register Partitioning
CHOLESKY	71	26
FFT	56	18
LU-CONT	57	20
LU-NCONT	53	21
OCEAN-CONT	55	20
OCEAN-NCONT	54	19
RADIX	108	37

and **R** is the case with register partitioning. As we see in Figure 3, the results are mixed.

Two applications (FFT and RADIX) enjoy significant performance gains from register partitioning with two threads per processor, and one application (LU-CONT) experiences only a 1% speedup. As we saw earlier in Figure 2, FFT, OCEAN-NCONT, and RADIX each spend over 10% of their time saving and restoring registers to perform thread switches in the baseline case. Hence it is not surprising that we see large performance gains due to register partitioning in FFT and RADIX. In contrast, OCEAN-NCONT has higher register pressure than either FFT or RADIX, and consequently it loses too much performance due to register spilling to make up for the faster thread switching time.

Overall, we see that register partitioning can potentially improve performance by reducing the number of registers that must be saved and restored upon a thread switch. For example, in the case of RADIX, software-controlled multithreading offers almost no speedup on two processors in the baseline case, but it enjoys a 7% speedup with register partitioning. However, register partitioning is a technique that must be used with caution, since it can hurt performance if it causes too much register spilling. Since the decision of whether to perform partitioning is controlled by software, the programmer has the flexibility to choose the option that works best for a given application. A more desirable solution would be for the compiler to make this decision *automatically*, which may be feasible since the compiler is precisely aware of register spilling, and could adjust the degree of partitioning accordingly.

4.3 Miss Prediction

The final optimization that we consider is using prediction techniques to avoid invoking the miss handler upon primary cache misses which hit in the secondary cache (as discussed earlier in Section 2.2.2). The basic idea is to predict the conditional probability of a secondary cache miss given a primary cache miss for a specific reference, and to use this information at the time when a primary miss is detected to decide whether or not to actually invoke the miss handler. In theory, this could allow us to reduce some of the *TS Setup* time shown earlier in Figure 2. However, based on the results of our experiments, this optimization does not appear to be useful in practice. Even with a *perfect* prediction mechanism, the potential performance gain is generally quite small: just a few percent. (Note that only a fraction of the *TS Setup* time can be eliminated, since this overhead is unavoidable for references that do suffer secondary cache misses.) When we experimented with dynamic hardware predictors (e.g., two-bit saturating counters and other mechanisms commonly used for branch prediction), we were unable to achieve any speedup over the baseline case. Stride predictors are not helpful, since both the primary and secondary caches share the same line size. While it is easy to predict that a large fraction of references will hit in the secondary cache (especially those that enjoy spatial locality), most of these references *also* hit in the primary cache, in which case the miss handler would not be invoked anyway.

The fundamental problem is that accurately predicting the conditional probability of a secondary cache miss given a primary cache miss is difficult, and the penalty of a false negative (i.e. failing to predict a secondary cache miss) is extremely large, since we will fail to hide any of the miss latency in that case. (In contrast, the penalty of a false positive is much smaller, since we will quickly discover the mistake after entering the miss handler.) Hence all of the realistic predictors that we considered actually *hurt* performance by generating too many false negatives. The lesson that we have learned from these experiments is that it is far more important to reduce the overhead associated with actually switching threads (the largest component of which is saving and restoring registers) than trying to avoid invoking the miss handler in

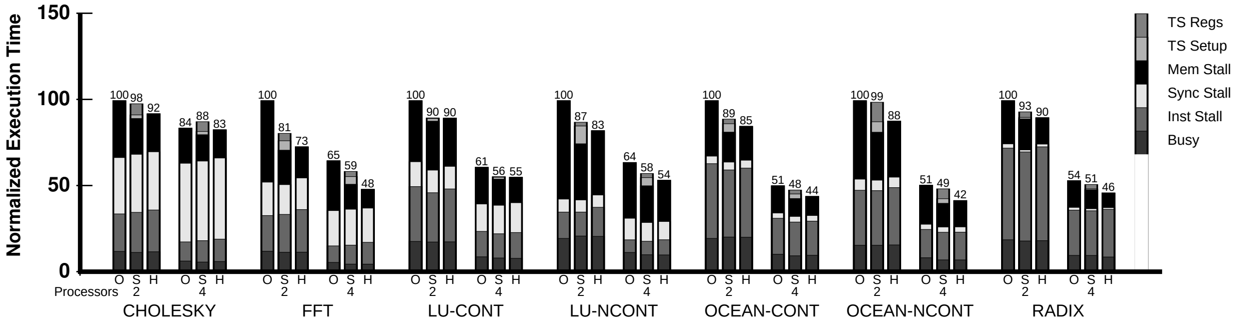

Figure 4. Performance of software-controlled (S) and hardware-controlled (H) multithreading with two threads per processor, as well as the original case (O) with one thread per processor. Execution time is normalized to the original case on two processors.

cases where a thread switch is unnecessary.

4.4 Software vs. Hardware Control

To quantify the overheads of our software-based multithreading scheme more precisely, we now compare its performance with a purely hardware-based implementation of coarse-grained multithreading. The hardware-controlled multithreading architecture that we model is identical to our software-controlled scheme (described earlier in Section 3) except for two major differences. First, thread switches are triggered directly by loads suffering *secondary* cache misses. (In contrast, since informing memory operations are triggered by *primary* cache misses, our software-based scheme must first check inside the miss handler whether the primary miss is also a secondary miss before it proceeds with the thread switch.) Second, after determining that a thread switch should take place, the hardware-based approach begins fetching and executing the new thread *immediately* after it flushes the pipeline beyond the load that suffered the secondary cache miss. (In contrast, the software-based scheme must execute miss handler code to save and restore thread state before it can begin executing the new thread.) Hence we expect our software-controlled scheme to perform strictly worse than the hardware-controlled scheme.

Figure 4 shows the results of our experiments. For each application, we show three performance bars for both two and four processors: the original, non-multithreaded case with one thread per processor (**O**); the best software-controlled multithreading case (which includes register partitioning if it was beneficial) with two threads per processor (**S**); and hardware-controlled multithreading with two threads per processor.

As we see in Figure 4, hardware-controlled multithreading outperforms our software-controlled scheme in all cases, as expected. However, in five out of the seven applications (i.e. all but CHOLESKY and OCEAN-NCONT), software-controlled multithreading achieves a significant fraction (often 50% or more) of the speedup attained by hardware-controlled multithreading. On average, we see an 8% speedup for these cases from software-controlled multithreading compared with a 17% average speedup from hardware-controlled multithreading.

Note that although one might expect that we could predict the execution time of the hardware-based scheme by subtracting the two thread switch components (i.e. *TS Regs* and *TS Setup*) from the execution time of the corresponding software-based scheme, this is not quite accurate for the following two reasons. First, the components of the execution time bars are based on graduation slots (as discussed earlier in Section 4.1), which is only a rough approximation of where the time is being spent given all of the parallelism in an out-of-order superscalar processor. Second, to some extent, part of the thread switching code can often be overlapped with fetching and executing the start of the new thread in an out-of-order superscalar processor (i.e. not all registers need to be restored before the first instructions are executed—only the ones on which they depend). Hence, as we see in Figure 4, the execution time of the hardware-based scheme is often somewhat larger than the difference between the corresponding software-based scheme and its thread-switching components.

Although a performance comparison with simultaneous multithreading (SMT) [15] is beyond the scope of this paper, we would expect SMT to offer even higher performance on multithreaded workloads than the coarse-grained hardware-controlled scheme modeled here, albeit at the cost of even greater hardware complexity. In general, there is a spectrum of potential design points for multithreading, ranging from our very low-cost software-controlled approach to more aggressive designs such as SMT. We expect that processor designers might choose any one of these approaches, based on the cost/performance tradeoffs appropriate to their design. The key contribution of our design is that it is a worthwhile approach to hiding latency with a dramatically lower hardware cost than previous multithreading proposals.

5 Conclusions

In contrast with previous studies on using multithreading to tolerate memory latencies in tightly-coupled machines, we have considered a completely new approach: one that is *software-controlled*, rather than hardware-controlled. The advantage of our approach is that due to its much simpler hardware support, it does not run the risk of degrading single-thread performance in applications which cannot benefit from multithreading (e.g., those that do not contain parallel threads). For example, our scheme does not require any modifications to the register file, unlike hardware-controlled schemes which typically require a much larger register file (thereby increasing register access latencies). The primary hardware support required by our scheme is *informing memory operations*, which have already been shown to be useful for a wide variety of purposes other than multithreading, and which are not expected to degrade single-thread performance.

Our experimental results demonstrate that software-controlled multithreading can result in non-trivial performance gains. In our baseline scheme, four of seven applications speed up by 10% or more, with one application speeding up by 14% (LU-NCONT). By judiciously applying *register partitioning* to reduce the thread switching overhead in cases where it does not result in excessive register spilling, we can enjoy even larger speedups: e.g., an overall speedup of 23% in the case of FFT. Since both remote latencies and the amount of remote communication are expected to increase with larger numbers of processors, we expect even greater performance gains on larger scale multiprocessors.

As we look to the future, software-controlled multithreading should become even more attractive as instruction overhead becomes less and less expensive relative to memory latency. Software-controlled multithreading is a gentle path to exploiting some of the performance benefits of multithreading without paying the higher costs of more ambitious hardware-based schemes. The attractiveness of software-controlled multithreading provides another compelling reason for future microprocessors to support informing memory operations.

6. Acknowledgments

We thank Jack Veenstra for providing the *MINT3* simulation infrastructure. Todd C. Mowry is partially supported by an Alfred P. Sloan Research Fellowship.

References

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th ISCA*, pages 104–114, May 1990.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proceedings of the International Conference on Supercomputing*, pages 1–6, June 1990.

[3] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *Proceedings of the 7th ASPLOS*, October 1996.

[4] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing Memory Operations: Memory Performance Feedback Mechanisms and Their Applications. *ACM Transactions on Computer Systems*, 16(2):170–205, May 1998.

[5] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th ISCA*, pages 81–85, 1981.

[6] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. In *Proceedings of the 6th ASPLOS*, pages 308–318, October 1994.

[7] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th ISCA*, pages 241–251, June 1997.

[8] C. K. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the 7th ASPLOS*, pages 222–233, October 1996.

[9] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, March 1994. Technical Report CSL-TR-94-626.

[10] T. C. Mowry, C. Q. C. Chan, and A. K. W. Lo. Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory. In *Proceedings of the 4th HPCA*, pages 300–311, February 1998.

[11] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-Grain Access Control for Distributed Shared Memory. In *Proceedings of the 6th ASPLOS*, pages 297–306, October 1994.

[12] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. *SPIE*, 298:241–248, 1981.

[13] K. Thitikamol and P. Keleher. Multi-threading and Remote Latency in Software DSMs. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, 1997.

[14] 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 23rd ISCA*, pages 191–202, May 1996.

[15] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd ISCA*, pages 392–403, June 1995.

[16] C. A. Waldspurger and W. E. Weihl. Register relocation: Flexible contexts for multithreading. In *Proceedings of the 20th ISCA*, May 1993.

[17] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN PLDI '91*, pages 30–44, June 1991.

[18] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd ISCA*, pages 24–38, June 1995.

[19] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, April 1996.