

The Effects of Architecture on the Performance of Latency Hiding Via Rapid Context Switching

Rajat Mukherjee
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
rajatm@watson.ibm.com

John K. Bennett Jay A. Greenwood
ECE, Rice University
Houston, Texas 77251
{jkb,jayg}@ece.rice.edu

Abstract

We study the effects of cache organization, caching policy and network capacity on the performance of latency hiding via fast context switching in large-scale shared memory multiprocessors. We describe a technique that supports hardware or software-initiated switches that works on a commercially available processor with register windows. Significant performance improvements (120%) can be achieved with latency hiding when cache miss latency is high, even with low cache miss rates (1 - 2%) and relatively large thread switch overhead (30 cycles). Write-back, set-associative caches provide best latency hiding performance, especially with constructive cache interference. We also show that increased processor utilization can significantly increase contention on the underlying network.

Key Phrases: Shared Memory Multiprocessors, Latency Hiding, Context Switching, Architecture.

1 Introduction

Memory system performance is critical in large-scale shared memory multiprocessors since limited interconnection bandwidth and delays due to network traffic can significantly increase memory access latencies. Memory hierarchies, consisting of processor registers and multiple levels of cache memory, are used to reduce average memory access latency. However, misses in the processor or lower-level caches can incur latencies of hundreds of cycles, e.g., DASH (>135 cycles) and KSR-1 (570 cycles) [9]. Large memory latency is a major impediment to achieving high performance as it can significantly reduce processor utilization.

Memory latency can be masked via rapid context switching [1]. When a thread misses in the cache, it is possible to switch to an alternate thread of computation while the access is being satisfied. This allows processors to be more effectively utilized. The performance of this technique depends on the expected access latency, as well as the time consumed in performing a thread switch.

We present a study of the effects of architectural variations on the performance of latency hiding via context switching. With cache miss latencies of 150 cycles, common in large-scale multiprocessors, we have obtained speedups of up to 120%, even with low cache miss rates (1 - 2%) and thread switch times of about 30 cycles. We show that direct-mapped caches are unsuitable when operating system code is highly sensitive to cache misses, as in the case of context switching trap code. Set-associative caches with a write-back policy are most conducive to the success of latency hiding via context switching. We show that the success of this technique depends on the ability of the underlying communication network to support the increase in processor utilization.

The remainder of the paper is organized as follows: Section 2 briefly describes the simulation environment used. Section 3 describes an approach of using register windows to switch between multiple threads that belong to the same application. In Section 4, we address possible deadlock implications of such a scheme. Results from simulation are presented in Sections 5 and 5.1 in the context of uniprocessors and typical multiprocessor clusters found in large-scale shared memory systems. We present a survey of related research in Section 6 and present our conclusions in Section 7. The context switching technique described in this paper has been implemented in SALSA, an operating system for Willow, a large-scale hierarchical shared memory multiprocessor [2]. Further details on this work can be found in [10, 11].

This research was supported in part by the NSF under Grant CCR-9010351 and by an IBM Graduate Fellowship.

2 Simulation Environment

Our simulation environment used MPSAS, a detailed instruction-level simulator that was developed at Sun Microsystems, Inc. and modified for Willow. The kernel, run-time system and programs were compiled using the standard Sun compiler (which issued unnecessary save and restore instructions for procedure call (Section 3)). The executable image was loaded into the memory of the simulated system. The kernel, run-time system, and program are all simulated in detail, including register, cache, and memory transfers as well as virtual memory translations, thus accurately accounting for resource contention and delays.

We used a mix of symbolic and integer programs - **LIFE** (Game of Life), **MULT** (Matrix Multiply), **ISO** (Subgraph Isomorphism), **SIEVE** (Finding Primes), **STRING** (Pattern Matching), **MST** (Minimum Spanning Tree) and **QSORT** (Quicksort). All programs used lightweight threads [3] as latency hiding is effective only in this environment. Note that for the purpose of this study, the program characteristics are more important than the algorithms used.

3 Fast Context Switching Using Register Windows

We use register windows [5] to cache multiple thread contexts, allowing a context switch to a cached thread to be accomplished in 30 processor cycles. The use of register windows for context caching, as opposed to procedure call, has been previously proposed for Alewife by Agarwal et al. [1]. All switches in Alewife are triggered by hardware traps. Our scheme supports switches initiated by both hardware and software, requires no hardware modifications to the SPARC processor architecture, and is thus usable with commercially available processors.

Thread contexts are cached in the processor, using a window per context. Figure 1 depicts a register file of 8 register windows divided into four banks, each with its own trap window. Up to four threads (1 active) can be *resident* on the processor. A program that sleeps on a synchronization event does so by means of an explicit call to the run-time system. A high latency access switch is triggered by a hardware trap (30 cycles), using *save* and *restore* instructions, traditionally used for procedure call [5]. Traditional use of *save*, *restore* instructions incur overflow and underflow trap overheads (total 109 cycles) and must be avoided.

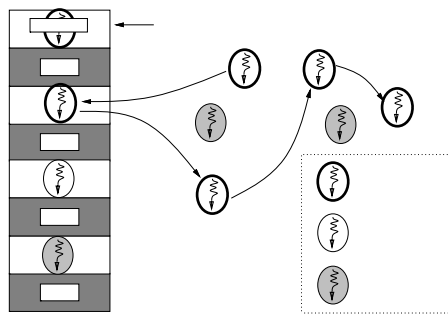


Figure 1: Context Cache Using Register Windows

3.1 Switching on a Synchronization Event

Blocking synchronization constructs, such as relinquishing locks, monitors, condition variables and barriers [3] require context switching. When a thread makes a call to switch, it is *blocked*, and control is transferred to a *scheduler* that runs another *ready* thread. If the target is *resident* (cached context), the scheduler saves its own program counter, moves to the appropriate window, and loads the program counter of the new thread. If the target is not resident, the scheduler finds a *victim* context, saves its registers, and replaces it with the new thread's state. Victim threads are usually *blocked*, as it is inefficient to replace ready threads. Replacing threads that are *waiting* can potentially cause deadlock due to actively held locks.

3.2 Switching on a High Latency Access

Memory access latency can be masked by switching to a *ready/waiting* thread. Latency switches are initiated via a trap to the processor, with the trap handler performing the switch. The trapping thread is placed in *waiting* state, as opposed to one that is *blocked* on synchronization. The trap handler finds a ready resident thread, or switches to a waiting thread, or to the scheduler (never blocked). No thread state needs saving, thus allowing fast switches. An implication of introducing latency switches is that a thread that sleeps on a synchronization event does not have to switch to the scheduler; it can switch to a waiting resident thread. This eliminates context switches to and from the scheduler, improving overall performance.

3.3 Context Caching Performance

It costs **45** cycles for a context switch to a resident thread. Switch to a non-resident thread takes about **130** cycles. Resident thread switch is an order

of magnitude faster than a typical context switch in a system without a context cache [10]. Thus, any scheduling algorithm that maximizes the number of resident threads minimizes context switch time.

The timings for a latency context switch using this scheme are shown in Table 1. If the next thread context is not waiting (non-ideal situation) and there are other ready threads, we switch to the scheduler, incurring the additional overhead. A choice can also be made to check the other contexts for waiting threads; costs for these are also listed.

Next Thread Context Waiting	30 cycles
Next Context Not Waiting Thread in Window+2 Waiting	42 cycles
Next 2 Contexts Not Waiting Thread in Window+4 Waiting	47 cycles
No Waiting Contexts Switch to Scheduler	44 - 48 cycles

Table 1: Timings for a Latency Context Switch

4 Software Issues

Besides hardware requirements such as split-transaction buses (for multiple concurrent bus accesses), lockup-free caches (for concurrent access of the cache by the processor and from the bus) and set-associative caches, there are additional constraints on software organization of the run-time system.

4.1 Latency Traps and Deadlock

Processors ordinarily stall on cache miss. When the access completes, the processor continues and is guaranteed to find the data in the cache. However, if we switch while the access is ongoing, the second thread typically executes until it incurs a cache miss. It is possible for the data returned for the first miss to be victimized (overwritten) by a later access. If this happens, the faulting thread misses immediately upon restart. This phenomenon can repeat, halting forward progress.

There are several situations where deadlock can occur due to competition for cache lines, e.g., the trap code maps (or accesses data that maps) to the same line as a thread that missed. Data fetched by the faulting thread will always be invalidated by the trap code invoked on the miss. Another example is a load instruction where the instruction address maps to the same cache line as the data being read. While this

case can be eliminated by the use of set-associative caches, the first two cases cannot. The probability of occurrence can, however, be significantly reduced via increased associativity.

Deadlock can be avoided if we do not fault on the same cache line on consecutive accesses for a thread. In order to account for the case when a thread's instruction and data map to the same line, we are required to avoid faulting if the offending address' cache line number is one of the last two missed for a thread. We use this approach to deadlock avoidance.

4.2 Latency Traps and Spinlocks

In the presence of spinlocks, switching to hide memory latency can lead to another potentially hazardous situation. If a thread that has acquired a lock switches out on a high latency access, and a second thread tries to acquire the same lock, it could spin forever on a local copy of the lock, as the lock owner never gets a chance to run and release the lock.

Timeout schemes can support busy-waiting safely, but preemption or priority scheduling adds significant overhead and complexity in a lightweight threads package. Disabling traps before acquiring a lock and re-enabling traps after release is restrictive and also adds trap overhead for every critical section, making it prohibitively expensive for fine-grained sharing.

The solution we adopt is to invoke a software trap on failed lock acquire operations; this suspends a thread, allowing others to execute. The owner is allowed to make forward progress and eventually relinquish the lock. This solution also avoids wasting processor cycles in spinning. The software trap to switch on failed synchronization is transparent to the program; spinlock primitives invoke the software trap in the run-time library.

5 The Interaction of Latency Hiding with Cache Architecture

While deadlock can be addressed via architectural mechanisms in the cache, other cache design decisions can significantly affect the efficacy of latency hiding using rapid context switching, such as cache set-associativity and cache write policy. In this section, we evaluate these design choices.

We simulated a set of programs on a uniprocessor architecture with a 64 KB cache that generates hardware traps on cache misses. In order to emulate large-scale systems, we varied the bus latencies from 20-150

cycles. Latency traps were taken only on read misses to program data. Other techniques, such as write-buffering, are able to hide write latency [7].

Figure 2 shows the percentage degradation in program execution time with latency hiding enabled (traps vs. no traps) for a miss latency of 20 cycles with direct-mapped, 2-way and 4-way associative caches. The degradation in the case of direct-mapped caches is about 1% for MST and QSORT, but much larger for ISO (9%) and LIFE (300%). With associative caches, the degradation is significantly smaller (0.1% (ISO) and 3% (LIFE) for 4-way associativity).

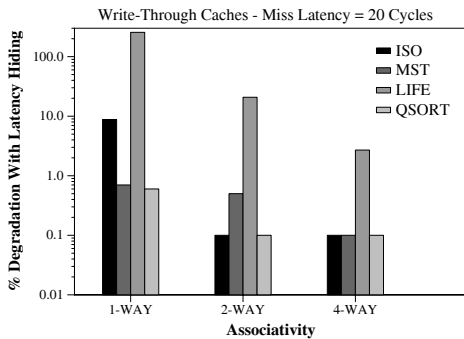


Figure 2: Degradation With Latency Hiding

Bad data or code mappings that exist in the program can be exposed from changes in thread ordering due to context switching, causing severe victimization in a direct-mapped cache. Cache contention between the application and the latency trap code can result in poor cache hit-ratios, increasing effective switch time in direct-mapped caches.

While performance improved with set-associativity, there was no speedup due to latency hiding with write-through caches. Figure 3 depicts the overall improvements due to latency traps with 4-way associative, write-through caches for 5 programs for a range of miss latencies.

The overheads are due to the write-buffers filling due to increased utilization, causing the processor to stall even on write hits. The average queue wait times for read and write for write-through caches are presented in Table 2 for a miss latency of 150 cycles. Due to the load on the bus for every write, there is an increase even in the average queue wait times for read.

An effect of this phenomenon is the increase in the average context switch time for the write-through case, which reduces the fraction of the latency that is hidden. The average switch times are presented in Table 3. The smaller the average context switch, the

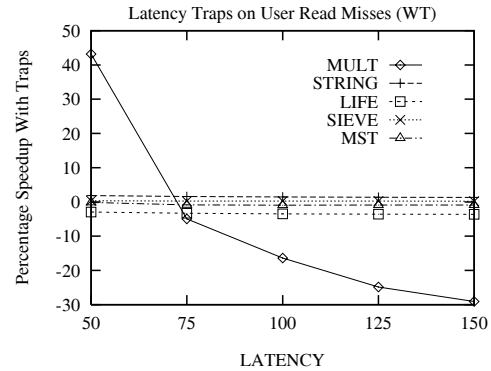


Figure 3: Performance with Write-Through Caches

Prog.	Read-Q Wait Time		Write-Q Wait Time	
	No Traps	Traps	No Traps	Traps
STRING	1112	1099	989	1058
MST	1019	1058	1077	1114
SIEVE	657	678	863	899
LIFE	593	722	914	996
MULT	271	769	221	643

Table 2: WT Caches - Queue Wait Times (150 Cycles)

larger the benefits of latency hiding. The average switch time is reduced by about 20% in STRING and 27% in MST by avoiding stalls in write-back caches.

Program (Miss Latency)	Average Switch Time	
	W-T Cache	W-B Cache
MST (75 cycles)	56.1	40.9
STRING (50 cycles)	61.5	49.4
LIFE (50 cycles)	35.8	34.8
MULT (50 cycles)	36.9	34.6

Table 3: Average Switch Times for WT, WB Caches

Figure 4 shows the improvements due to latency hiding with write-back caches. With write-back, 4-way set-associative caches, the programs ran 3 to 125% faster for a miss latency of 150 cycles.

Table 4 shows the total miss rates, user-level read miss rates (with traps enabled) and improvements for the programs for a 150-cycle miss latency. Figure 5 depicts program instruction and data miss rates with and without context switching. MULT had the highest overall miss rate of all 5 programs, and the majority of misses in MULT were read misses that resulted

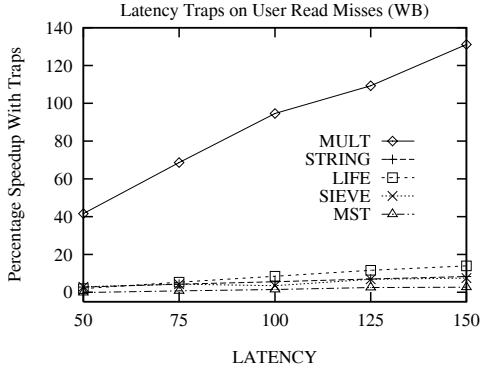


Figure 4: Performance with Write-Back Caches

in latency traps; the other programs had many write misses that were ignored. MULT also exhibits a large decrease in miss rate due to positive interference.

Prog.	Total Miss Rate		Read Miss Rate(Trap)	% Faster
	No Traps	Traps		
STRING	0.15	0.10	0.012	8.32
MST	0.16	0.18	0.099	2.6
SIEVE	0.86	0.77	0.032	7.47
LIFE	0.41	0.40	0.179	13.97
MULT	1.43	0.65	0.602	131.13

Table 4: Miss Rates & Performance (Latency 150 cyc.)

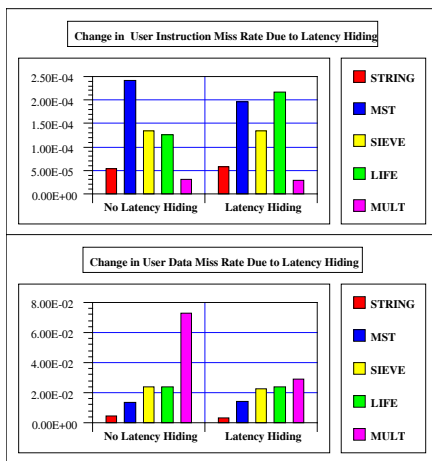


Figure 5: Change in User Cache Miss Rates

There are two effects of latency context switching on program performance: (a) hiding miss latency via

alternate computation and (b) the change in cache performance due to thread reordering. The conventional argument against latency hiding via thread switching is that context switching degrades cache performance. We have observed that in some cases, the reverse is true. Since the threads we are switching between belong to the same address space and are working on the same problem, thread reordering can actually improve cache performance. Latency hiding using context switching can be effective even when there is a negative effect on the cache miss rate, but in the cases where the miss rate decreases, as in STRING and MULT, the gains can be significant. The improved cache performance explains why STRING had better improvements with latency hiding than MST, which had a higher user data miss rate.

5.1 Multiprocessor Issues

Since the performance improvements from latency hiding accrue due to increased processor utilization, the uniprocessor results are pertinent to the issue of scalability multiprocessors, where coherence-based invalidations are likely to increase misses. We studied the performance on a single-bus, 4-processor cluster, varying miss penalties from 100 to 200 cycles to emulate high network latency. The improvements were similar to the uniprocessor case. However, the capacity of the network to support the higher processor utilization can be crucial in determining latency hiding performance.

Program (Miss Cycles)	1 processor		4 processors	
	Read Wait	Write Wait	Read Wait	Write Wait
SIEVE (100)	100	102	143	230
STRING (100)	100	102	162	174
LIFE (100)	100	102	199	147
MULT (125)	125	161	147	215

Table 5: Average Bus Wait Times

An implicit assumption in attempting to hide latencies and increasing processor utilization is that the network can handle the increased bandwidth requirements. Table 5 shows the average read and write miss wait times for programs SIEVE, STRING, LIFE and MULT for the uniprocessor and multiprocessor cases. With latency hiding enabled, we observed 17-100% larger read wait times and 33-125% larger write wait times due to increased bus contention, demon-

strating the importance of high-bandwidth networks to the success of this technique. Increased processor utilization with latency hiding will therefore limit the number of processors that can be placed in a cluster of a shared memory multiprocessor.

6 Related Work

Previous proposals for fast context switching [1, 8, 12] mandate hardware modifications to existing processor architectures for implementation, which precludes the use of commercially available processor architectures. We describe a software-based technique, which can be implemented on a commercially available (SPARC) processor architecture without hardware modifications. Our approach supports a combination of software and hardware synchronization primitives, which makes it more generally applicable than previous schemes, such as April [1]. In April, all context switches are initiated in hardware, and the added overhead of maintaining thread state information is not considered. Reported results do not take into account the issues of scheduling, sequential initialization of programs, checks for available ready threads to execute, etc. Traps need to be disabled when in supervisor mode because a synchronous trap that occurs while executing in supervisor mode causes the processor to reset. This overhead is ignored in the context switch times reported for April [1]. The SPARC Version 9 architecture reference allows for stacked traps, which would reduce the switch overhead by 3-4 cycles.

Previous studies have examined the effectiveness of context switch overheads of less than 15 processor cycles, therefore effectively eliminating software approaches from consideration [13]. Our approach is appropriate for large-scale systems where typical miss latencies are hundreds of processor cycles. Moreover, the assumptions used in some studies are not realistic, e.g., the small cache sizes used in [7] result in very large miss rates and therefore tend to reduce the inter-miss distance, which is crucial to the performance of this technique. We do not make any simplifying assumptions about context switch times, cache sizes, etc., but use complete context switch code, and cache sizes based on commercially available caches.

Assumptions of constant cache miss ratios and constant inter-miss distance (cycles between two successive misses) made in previous analytical studies are not accurate. Miss patterns, which determine the benefits due to latency hiding, show significant variability, as demonstrated in [10]. While it may be possible to introduce this variability into statistical models, such

models would still be unable to capture the subtle effects of contention in specific program execution profiles. Cycle-level simulation accurately models data movement in the memory subsystem, as well as contention for resources. We have used detailed cycle-level simulation for our experiments that takes into consideration the effects of cache interference, race conditions, contention, deadlock, etc. Detailed simulation, however, has the disadvantage of being inappropriate for simulating the execution of large programs on large systems.

Relaxing the memory consistency model hides latency by allowing the overlap of memory accesses with other computation and memory accesses [6]. A relaxed consistency model allows more buffering and pipelining among memory requests. If synchronization points can be identified, it is often possible to defer the completion of write operations until these points. The performance of relaxed consistency models can be further enhanced by dynamic scheduling, which can hide a portion of read latencies. Context switching may be used in conjunction with these techniques, and others such as adaptive caching and prefetching [7], to further improve performance.

7 Conclusions

We have described a technique intended to address the problem of large access latencies in large-scale shared memory multiprocessors such as Willow or Dash. It is possible to effect a rapid context switch to an alternate thread of computation upon the incidence of a high latency event such as a cache miss. Our technique, unlike the April approach, uses the SPARC processor architecture with no hardware modifications. We have obtained significant improvements with latency hiding on cache misses with programs, even with cache miss rates of 1-2 percent, with miss latency in the range of 150 cycles (120% with MULT with a read miss rate of less than 2%).

We have described the limitations of latency hiding using fast context switching, and have investigated the effects of cache associativity and caching policy. We conclude that set associative caches with a write-back caching policy are best, as they result in minimum cache interference between multiple threads, and between threads and trap code. While our results were obtained using unified caches, other related studies [4] have shown that associativity can significantly reduce interference for system code even in the case of separate data and instruction caches.

Context switching on cache misses has two main effects. The first is to improve processor utilization; the second effect is on program cache behavior. By changing thread ordering, program cache behavior can improve when threads prefetch instructions and data effectively for other threads in the same application. Context switching was traditionally considered to have a deleterious effect on the cache performance, but we show that positive cache interference due to context switching can significantly improve performance. In a related study, all programs chosen were reported to demonstrate negative cache interference [13].

When extending the principle of latency hiding to multiprocessor environments, network bandwidth becomes an important concern; if the network is incapable of handling the higher bandwidth required to sustain high-utilization processors, the benefits of latency hiding can easily be offset by network contention. In our experiments with 4 40 MHz. SPARC processors per cluster, the average bus access times were between 17% to 125% worse than the uniprocessor access averages. In a cluster multiprocessor with bus-based clusters, a small clustering factor could ensure that the bus does not saturate due to the increased processor utilization.

References

- [1] Anant Agarwal, Ben Hong Lim, David Kranz, and John Kubiawicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 104 – 114, May 1990.
- [2] John K. Bennett, Sandhya Dwarkadas, Jay G. Greenwood, and Evan W. Speight. Willow: A Scalable Shared Memory Multiprocessor. In *International Conference on Supercomputing*, Minneapolis, MN, November 1992.
- [3] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. Technical Report 87-09-01, University of Washington, Department of Computer Science, September 1987.
- [4] Bradley J. Chen and Brian N. Bershad. The Impact of Operating System Structure on Memory System Performance. *Operating Systems Review*, 27(5):120 – 133, December 1993.
- [5] Robert B. Garner. SPARC Scalable Processor Architecture. *Sun Technology*, pages 42 – 63, Summer 1988.
- [6] Kourosh Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15 – 26, May 1990.
- [7] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd C. Mowry, and Wolf-Dietrich Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 43 – 63, May 1991.
- [8] Yasuo Hidaka, Hanpei Koike, and Hidehiko Tanaka. Multiple Threads in Cyclic Register Windows. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 131 – 142, San Diego, California, May 1993.
- [9] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, pages 63 – 79, March 1992.
- [10] Rajat Mukherjee. *The Interaction of Architecture and Operating System in the Design of a Scalable Shared Memory Multiprocessor*. PhD thesis, Rice University, Houston, Texas, November 1993.
- [11] Rajat Mukherjee and John K. Bennett. Operating System Design Principle for Scalable Shared Memory Multiprocessors. In *Proceedings of the Eighth ISCA Conference on Parallel and Distributed Computing Systems*, Orlando, Florida, September 1995.
- [12] Carl A. Waldspurger and William E. Weihl. Register Relocation: Flexible Contexts for Multithreading. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 120 – 130, San Diego, California, May 1993.
- [13] Wolf-Dietrich Weber and Anoop Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *Proceedings of the 16th International Symposium on Computer Architecture*, New York, June 1989.