A Performance Comparison of DRAM Memory System Optimizations for SMT Processors

Zhichun Zhu
Dept. of Electrical & Computer Engineering
University of Illinois at Chicago
Chicago, IL 60607, USA
zhu@ece.uic.edu

Zhao Zhang
Dept. of Electrical & Computer Engineering
Iowa State University
Ames, IA 50011, USA
zzhang@iastate.edu

Abstract

Memory system optimizations have been well studied on single-threaded systems; however, the wide use of simultaneous multithreading (SMT) techniques raises questions over their effectiveness in the new context. In this study, we thoroughly evaluate contemporary multi-channel DDR SDRAM and Rambus DRAM systems in SMT systems, and search for new thread-aware DRAM optimization techniques. Our major findings are: (1) in general, increasing the number of threads tends to increase the memory concurrency and thus the pressure on DRAM systems, but some exceptions do exist; (2) the application performance is sensitive to memory channel organizations, e.g. independent channels may outperform ganged organizations by up to 90%; (3) the DRAM latency reduction through improving row buffer hit rates becomes less effective due to the increased bank contentions; and (4) thread-aware DRAM access scheduling schemes may improve performance by up to 30% on workload mixes of memory-intensive applications. In short, the use of SMT techniques has somewhat changed the context of DRAM optimizations but does not make them obsolete.

1 Introduction

Two technology trends may have strong impacts on future high-performance computer designs: the exploitation of thread-level parallelism (TLP) on the top of instruction-level parallelism (ILP), and the employment of increasingly complex memory hierarchies. In recent years, two single-chip TLP techniques, namely simultaneous multithreading (SMT [30, 29]) and chip-level multiprocessing (CMP [10, 1]), have been well studied and applied to commercial processors. In this study, we focus on the memory system considerations for SMT processors.

Main memory system designs and optimizations have become an increasingly important factor in determining the overall system performance. The main memory speed lags far behind the processor speed, creating a scaring speed gap, which will eventually offset most performance gains from further improvements on the processor speed. For today's multi-GHz and multi-issue processors, a cache miss penalty is equivalent to hundreds of processor cycles, a time long enough for the processor to issue more than one thousand instructions. Thus, the performance of future systems will be increasingly dependent on complex memory hierarchies. This is especially true for systems running memory-intensive applications, which in general have large working sets and irregular memory access patterns.

Although directly reducing the physical memory access latency is limited by the DRAM technology advancement and cost considerations, the advance of modern memory systems has provided many opportunities to reduce the average latency for concurrent memory accesses. For example, the DRAM open page mode can reduce the latency of successive accesses to the same page by eliminating interim row accesses [5]; and the memory access scheduling can reduce the average access latency by reordering concurrent memory accesses [22].

The SMT technique [30, 29] can effectively reduce the memory stall time by executing instructions from other threads when one thread is waiting for its data to be returned from the main memory. This may make the overall system performance less dependent on the memory performance. On the other hand, the SMT technique may increase the contention on main memory systems and demand memory systems with higher performance. Previous studies have shown that memory accesses tend to be clustered together under the single-threaded execution [13, 22, 34], resulting in the access concurrency in DRAM memory systems. When multiple memory-intensive threads are running together, we can expect that the memory access concurrency will be even higher. As a consequence, the memory system should be able to deliver higher bandwidth than that under single-threaded executions.

The emergence and employment of multithreading techniques raise some important questions regarding the memory system designs and optimizations: *whether those memory optimization techniques proposed for single-threaded systems are still effective; and whether a memory system designed for multithreaded systems should take the SMT factor into considerations.* In this paper, we will answer those questions quantitatively.

We extend a high-fidelity simulator based on the Alpha 21264 processor to model an SMT processor, and develop simulators for DDR SDRAM and Direct Rambus DRAM systems. Using both memory-intensive and compute-intensive programs from the SPEC2000 benchmark suite, we analyze the performance of representative program mixes as the memory system configuration changes. We have also searched for thread-aware DRAM optimizations that consider the states of individual threads. Our major findings are:

1. In general, the use of SMT techniques increases the memory access concurrency; and the degree of concurrency increases with the number of running threads. We do observe some exceptions: mixing memory-intensive programs with applications of few memory demands may decrease the concurrency, but not significantly.

2. The application performance is sensitive to memory channel organizations. For example, organizing each physical channel as an independent logic one may outperform combining them as a single logic channel by up to 90% on 8-channel DDR SDRAM systems.

3. The DRAM latency reduction through improving row buffer hit rates becomes less effective because of the increased bank contentions. We find that DRAM row buffer miss rates are consistently higher than the previously reported results on single-threaded systems. In other words, the higher memory access concurrency hinders the utilization of row buffers.

4. Thread-aware DRAM access scheduling schemes may improve the overall system performance by up to 30%, on workload mixes of memory-intensive applications. The considered thread states include the number of outstanding memory requests, the reorder buffer occupancy, and the issue queue occupancy.

The paper is organized as follows. We will first briefly introduce the contemporary DRAM memory optimization techniques in the next section. Then we will discuss three new thread-aware memory optimization techniques in Section 3. After presenting the experimental environment in Section 4, we evaluate the impact of SMT techniques on memory systems and analyze the performance of different memory optimization techniques in Section 5. Finally, Section 6 discusses the related work, and Section 7 summarizes our study.

2 Memory Optimization Techniques

For decades, the DRAM has been used to construct main memories because of its high density, large capacity, and low cost. Although it is called the dynamic random access memory, its access time is not a constant but depends on the current state of DRAM cell arrays. Many optimization techniques exploit this property to reduce the average memory access latency.

A typical high-performance memory system can support multiple memory channels, which can be accessed in parallel. In addition, each memory channel can connect multiple independent groups of chips[1]; and each group consists of multiple independent memory banks. Each bank is organized as a two-dimensional array (row and column). Depending on the status of row buffers (formed by arrays of sense amplifiers), a DRAM access may require different numbers of operations, and thus different access time. A row buffer hit, where the data are already in the row buffer, only requires a column access. A row buffer miss requires a row access and a column access if the bank has already been precharged; otherwise, it requires an additional precharge operation. Thus, an important feature of DRAM accesses is that *concurrent accesses to different rows (also called pages) in the same memory bank have significantly higher latencies than those concurrent accesses to the same page or to different memory banks*.

To fully utilize the contemporary DRAM features, memory system configurations are becoming increasingly complex. Consequently, the performance of a memory system perceived by applications is very sensitive to its configuration. Previous studies have shown that tuning the memory configurations, such as the burst width and channel organizations, can significantly affect the overall performance [4, 34]. The performance depends on a number of factors, including the number of memory channels, the page mode, the DRAM mapping scheme, and the memory access scheduling policy.

- *Page modes*: Contemporary DRAM systems, such as SDRAM and Rambus DRAM, support two page modes, *open* and *close*. The open page mode delays the precharge operation and keeps the row buffer data valid after a DRAM access in the hope that the next access to this bank will fall into the same page (the row buffer data are lost after a precharge operation). By contrast, the close page mode performs the precharge operation immediately after a column access, and favors successive accesses that are row buffer misses.

- *Mapping schemes*: Mapping schemes determine how memory addresses are mapped to multiple DRAM channels, chips, and banks. Previous studies have shown that the choice of mapping schemes significantly affects the row buffer hit rate and memory system performance [32, 33, 8].

- *Memory access scheduling*: The memory access

[1] For SDRAMs, multiple chips are grouped together to support the wide and low-speed memory bus; while for Rambus DRAMs, each chip is an independent group and connects to the narrow and high-speed bus.

scheduling reorders concurrent memory operations, i.e. precharge operations, row accesses, and column accesses, according to the current states of memory channels, banks, and outstanding requests. An optimized memory access scheduling scheme can reduce the average memory access time and improve the memory bandwidth utilization [13, 22].

3 Thread-aware Memory Access Scheduling

With the aggressive exploitation of ILP and TLP, higher access concurrency will be exposed to the memory system, making the memory access scheduling more important. Superscalar processors have extensively used out-of-order execution and non-blocking caches. Even though, they will exhaust their instruction scheduling resources (e.g. issue queues or reorder buffers) on cache misses that fall to the DRAM memory. However, the processors may issue dozens of instructions before stalling. Because cache misses tend to be clustered together [19], a good amount of memory concurrency exists. The degree of concurrency may further increase when multiple threads are running together. Because of the internal structure of DRAM systems, there is a good opportunity for the memory access scheduling to reduce the memory stall time.

3.1 Schemes for Single-threaded Processors

Previous studies have shown that the memory access scheduling can effectively reduce the average memory access latency and improve the memory bandwidth utilization for single-threaded processors [18, 17, 13, 21, 22, 16, 34]. For example, a hit-first policy prioritizes memory accesses hitting on the row buffers. Suppose two concurrent memory access streams touch DRAM pages *A* and *B* at the same bank with the sequence of *A-B-A-B-A-B-A-B*. This is a case of severe bank conflicts, which require the precharge operation, row access, and column access for every reference. The memory access scheduling can reorder the access sequence to *A-A-A-A-B-B-B-B*, converting six row buffer misses to row buffer hits. Since a row buffer hit has a much shorter latency than a row buffer miss, this technique effectively improves the memory system performance.

The followings are some commonly used policies in the memory access scheduling.

- *Hit-first*: A row buffer hit has a higher priority than a row buffer miss.
- *Read-first*: A memory read operation has a higher priority than a memory write operation.
- *Age-based*: An older request or a request whose waiting time is longer than a predefined threshold has a higher priority than a newly arrived one.
- *Criticality-based*: A request containing the critical word (which is currently required by the processor to resume its execution) has a higher priority than a noncritical one.

3.2 Outstanding Request-based Scheme

The emergence and employment of SMT techniques introduce a new possible direction in the memory access scheduling: *considering the current states of each thread.* A number of thread states may be used to guide the scheduling, such as the number of pending memory requests, or the number of issue queue entries occupied. In this study, we propose three thread-aware scheduling schemes and analyze their effectiveness.

Our first thread-aware scheduling scheme is based on the current number of outstanding memory requests generated by each thread. In addition to the hit-first, read-first, and age-based policies proposed for single-threaded processors, an outstanding request-based policy is introduced. These policies can be combined with each other in several ways. One scheme is to apply the hit-first and read-first policies on top of the request-based one. This means that a read hit always gets a higher priority than a read miss even if the hit is generated by a thread with more pending requests. Meanwhile, among a group of same-type requests (e.g. a group of read hits), the one generated by the thread with the fewest pending requests will be served before others.

The rationale behind this scheme is two-fold. First, if a thread has fewer cache misses than other threads, statistically more instructions can be committed after each of its cache misses is returned. Note that a load instruction that incurs a cache miss will block its subsequent finished instructions from being committed. For a load miss to the DRAM, which has a very long latency, the processor core will eventually stall due to the full ROB before the data is returned. Second, since cache misses from a thread tend to be clustered together, the scheme helps the thread move faster into the next phase of having no cache misses. The reason to enforce the hit-first policy ahead of the request-based one is that for SMT workloads, the sustained memory bandwidth is more important than the latency of an individual access, since the processor has the ability to overlap computations of one thread with memory accesses of others.

3.3 Resource Occupancy-based Schemes

Another class of thread-aware memory access scheduling schemes that we have studied is based on the number of processor resources occupied by each thread. Similar to the request-based scheme, the hit-first and read-first policies are enforced ahead of the resource occupancy-based ones. Two types of resource information are used here. The ROB-based scheme assigns the highest priority to the requests generated by the thread that holds the most reorder buffer entries. The IQ-based scheme assigns the highest priority to the requests from the thread occupying the most issue queue entries. The rationale behind the IQ-based scheme is that for most workloads, the issue queue entries are the most limited resources. Thus, returning the requests from the thread holding the most issue queue entries can release more entries. As for the ROB-based scheme, the reorder buffer occupancy of each thread indicates the thread's comprehensive occupancy on processor resources. When a thread occupies significantly more reorder buffer entries than other threads, it is very likely that those entries are piled up due to the thread's pending memory requests. If its requests can be served earlier by the memory system, each returned request may release more processor resources than a returned request from other threads; and thus the instruction throughput of the whole system can be improved. Note that the ROB or IQ occupancy information should be piggybacked with the memory request sent to the memory controller when a cache miss is found. Because of the communication latency between the memory controller and the processor core, the memory controller does not always have the most up-to-date information on the ROB or IQ occupancy. However, since the scheduling is heuristics-based, the information does not need to be completely precise.

4 Methodology

4.1 Simulator and Parameters

We use the Sim-Alpha from the SimpleScalar 4.0 release as the base architectural simulator, which has been validated to the Alpha 21264 processor [6]. We have extended the simulator to support the simultaneous multithreading and deepened its pipeline stages. Each active thread has its own PC. The active threads share the bandwidth at each pipeline stage, as well as caches, execution units, issue queues, and physical registers. In the front end, we implement four fetch policies, ICOUNT [29], fetch stall [28], DG [7], and DWarn [3]. Detailed discussions on those fetch policies are in Section 5.1.

In order to study the effects of different memory system configurations, we also extend the memory simulator. Multichannel DDR SDRAM and Direct Rambus DRAM systems are simulated. The memory simulator is implemented using an event-driven framework that allows the modeling of the memory access reordering. The simulator keeps track of the states of each DRAM channel, chip, and bank, as well as all pending requests. Based on the current memory states, memory operations are issued according to the employed scheduling policies. The details of the memory bus are also simulated; and memory accesses are pipelined whenever possible. Table 1 presents the major simulator parameters.

4.2 Workloads

The SMT workloads used in our study are mixtures of SPEC2000 applications [26]. There are hundreds of possible ways to mix the 26 SPEC2000 applications together. To meet the space limit but still show representative results, we construct the SMT workloads based on applications' ILP degrees and memory access demands using approaches similar to those used in [28, 3].

Figure 1 presents the CPI breakdown of all the 26 applications running independently on the 2-channel DDR SDRAM system with parameters listed above. For each application, its CPI value is divided into four portions: CPI_{proc}, CPI_{L2}, CPI_{L3}, and CPI_{mem}, which correspond to the execution time spending on the processor core (including L1 caches), L2 cache, L3 cache, and main memory, respectively. Applications are sorted by their CPI_{mem} values in Figure 1.

Table 1: Simulator parameters.

Processor speed	3 GHz
Fetch width	8 instructions
Baseline fetch policy	DWarn.2.8
Pipeline depth	11
Functional units	6 IntALU, 6 IntMult, 2 FPALU, 2 FPMult
Issue width	8 Int, 4 FP
Issue queue size	64 Int, 32 FP
Reorder buffer size	256/thread
Physical register num	384 Int, 384 FP
Load/store queue size	64 LQ, 64 SQ
Branch predictor	Hybrid, 4K global + 1K local (32-entry RAS/thread)
Branch Target Buffer	1K-entry, 4-way
Branch mispredict penalty	9 cycles
L1 caches	64KB instruction/64KB data, 2-way, 64B line, 1-cycle latency
L2 cache	512KB, 2-way, 64B line, 10-cycle latency
L3 cache	4MB, 4-way, 64B line, 20-cycle latency
TLB size	128-entry ITLB/128-entry DTLB
MSHR entries	16/cache
Prefetch MSHR entries	4/cache
Memory channels	2/4/8
Memory BW/channel	200 MHz, DDR, 16B width
Memory banks	4 banks/chip
DRAM access latency	15ns row, 15ns column, 15ns precharge

Using approaches similar to those used in [2, 5], the CPI value of each application is broken down as follows. We first run an application on four systems and collect its corresponding average cycles per instruction:

- $CPI_{overall}$: on the system with the 2-channel DDR SDRAM main memory;
- CPI_{pL3}: on the same system but with an infinitely large L3 cache;
- CPI_{pL2}: on the system with an infinitely large L2 cache; and
- CPI_{proc}: on the system with infinitely large L1 instruction and data caches.

The differences between the CPI values achieved by those systems represent the performance loss caused by introduc-

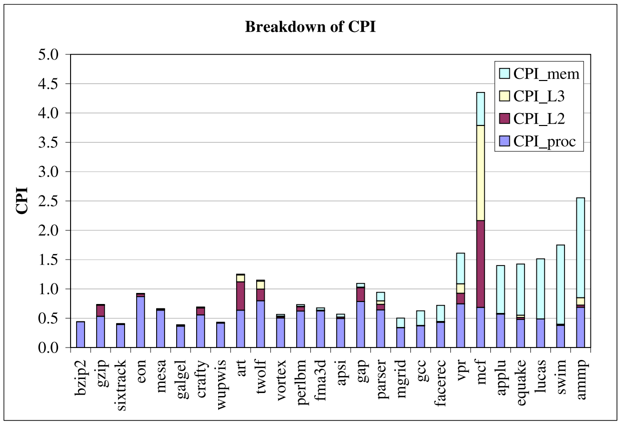

Figure 1: CPI breakdown of SPEC2000 applications. The applications are sorted by the increasing order of their CPI_{mem} values. CPI_{proc} corresponds to the cycles spending on the processor execution and L1 cache accesses; while CPI_{L2}, CPI_{L3}, and CPI_{mem} correspond to the cycles due to accesses to the L2 cache, L3 cache, and main memory, respectively.

ing each level of realistic implementations in the memory hierarchy:

- $CPI_{mem} = CPI_{overall} - CPI_{pL3}$: corresponding to main memory accesses;
- $CPI_{L3} = CPI_{pL2} - CPI_{pL3}$: corresponding to L3 cache accesses;
- $CPI_{L2} = CPI_{pL2} - CPI_{proc}$: corresponds to L2 cache accesses; and
- CPI_{proc}: corresponding to the processor core execution and L1 cache accesses.

Applications with small CPI_{proc} and CPI_{mem} values are categorized as “ILP” applications; while applications with large CPI_{mem} values are categorized as “MEM” applications. In general, applications in the left part of Figure 1 belong to the “ILP” category; and those in the right part belong to the “MEM” category. Table 2 presents the workloads used for 2-, 4-, or 8-thread configurations. The “ILP” workloads consist of only “ILP” applications; and the “MEM” workloads consist of only “MEM” applications[2]. The “MIX” workloads consist of half “ILP” applications and half “MEM” applications.

To control the simulation time while still getting the accurate behavior of applications, each application is fastforwarded and then executed for 100 million instructions as suggested by SimPoint [23]. Caches are warmed up during the fastforwarding. Note that those 100 million instructions are representative clips of SPEC2000 program executions using the Alpha binary code.

Table 2: Workload mixes.

2-thread	ILP	bzip2, gzip
	MIX	gzip, mcf
	MEM	mcf, ammp
4-thread	ILP	bzip2, gzip, sixtrack, eon
	MIX	gzip, mcf, bzip2, ammp
	MEM	mcf, ammp, swim, lucas
8-thread	ILP	gzip, bzip2, sixtrack, eon, mesa, galgel, crafty, wupwise
	MIX	gzip, mcf, bzip2, ammp, sixtrack, swim, eon, lucas
	MEM	mcf, ammp, swim, lucas, equake, applu, vpr, facerec

Multiple metrics have been proposed to measure the performance of SMT processors; for example, the weighted speedup [28], harmonic mean of relative IPCs [15, 7], and throughput [3]. We follow the work in [28] and use the weighted speedup in this paper.

5 Memory System Performance Analysis

In this section, we report and analyze the memory system performance with different numbers of hardware threads. If not mentioned specifically, the memory hierarchy consists of a 512 KB L2 cache, a 4 MB L3 cache, and a 2-channel DDR SDRAM memory system.

5.1 Performance Loss Due to DRAM Accesses

Previous studies have shown that instruction fetch policies are critical to the performance of SMT processors and affect how processor resources are allocated among threads. Figure 2 shows the weighted speedup of four representative fetch policies on a system with the 2-channel DDR SDRAM system. The *ICOUNT* fetch policy [29] assigns the highest priority to the thread that has the fewest instructions in the decode stage, rename stage, and instruction queue; and fetches instructions from up to two threads each cycle, where each thread can provide up to eight instructions (called ICOUNT.2.8 in [29]). It achieves much better performance than the simple policy that fetches instructions from threads in a round-robin way. The three other policies are proposed to avoid under-utilizing processor resources because of long-latency memory accesses. The *Fetch stall* policy [28] stops fetching from threads that have L2 cache misses but keeps at least one thread eligible of fetching instructions. The *DG* policy blocks fetching from threads that are experiencing data cache misses [7]. Instead of stopping fetching from threads with outstanding data cache misses, the *DWarn* policy just lowers the fetch priority of those threads [3]. Threads are divided into two groups. Those without outstanding data cache misses have higher fetch priorities than those with misses. Within each group, threads are prioritized using the ICOUNT policy.

Figure 2 shows that for the ILP workloads, the four poli-

[2] Application *mcf* is included in the 2-thread MEM workload because it has the highest overall CPI value and a high CPI_{mem} portion.

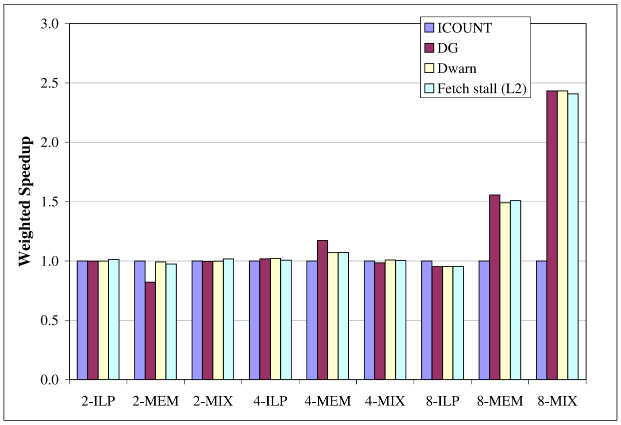

Figure 2: Performance comparisons of four fetch policies on an SMT system with the 2-channel DDR SDRAM system.

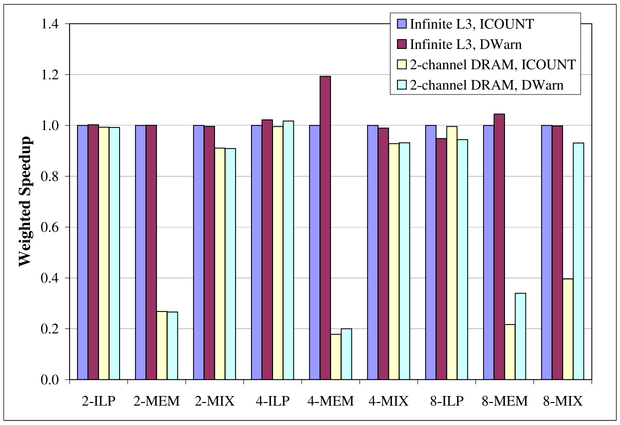

Figure 3: Weighted speedup of two fetch policies (ICOUNT and DWarn) on the SMT system with the 2-channel DDR SDRAM memory (“2-channel DRAM”) compared to that on the system with an infinitely large L3 cache (“Infinite L3”).

cies work comparably to each other. However, for workloads consist of memory-intensive applications and contain a large number of threads, such as 8-MEM and 8-MIX, the three policies (DG, DWarn, and Fetch stall) that alleviate processor resource clogs due to long-latency memory accesses achieve much better performance than the ICOUNT. Our intention here is not to compare the effectiveness of different fetch policies, but to study the impact of memory system optimizations for SMT systems. In terms of handling long-latency memory accesses, the three policies (DG, DWarn, and Fetch Stall) work comparably for the workload and system setup in our study. On average, the DWarn policy works slightly better than others. This result may change when the workload and system setup vary. Due to the space limit, we only use the DWarn policy for the subsequent discussions.

Figure 3 shows the performance loss on the SMT system due to main memory accesses under two fetch policies, ICOUNT and DWarn. The system with an infinitely large L3 cache and using the ICOUNT fetch policy is used as a reference. We can see that for 2-thread and 4-thread configurations, the performance losses due to main memory accesses under these two fetch policies are comparable. However, for 8-thread MEM (8-MEM in short) and 8-thread MIX (8-MIX) workloads, the DWarn policy gains much better performance than the ICOUNT policy. For 8-MEM and 8-MIX workloads, the system with the ICOUNT policy and 2-channel DDR SDRAM memory achieves 21.7% and 39.6% performance of that on the base system, respectively. In comparison, the system with the DWarn policy and 2-channel memory achieves 34.0% and 93.1% performance of that on the base system, respectively. The reason is that when a larger number of threads exist and the workload contains memory-intensive applications, the DWarn policy can alleviate clogs on processor resources caused by memory accesses. This is especially true when memory-intensive applications are mixed with compute-intensive ones. For instances, when running the 8-MIX workload with the DWarn policy, during 92.2% of execution cycles the processor can issue at least one integer instruction. This percentage drops sharply to 43.8% when the ICOUNT policy is applied.

As expected, for ILP workloads, main memory accesses only cause negligible performance losses, especially for those with small numbers of threads. For the 2-thread and 4-thread ILP workloads, on average, every 100 instructions only generate 0.01 and 0.02 main memory accesses, respectively. When the number of threads increases to eight, the cache contentions increase sharply, but the total number of L3 misses is still quite low. The 8-ILP workload generates 0.13 main memory accesses per 100 instructions on average. Even for this workload, the performance only decreases by 3.7% for the SMT system using the DWarn policy, when the L3 cache size reduces from infinitely large to 4 MB and a realistic 2-channel memory system is included. Since the main memory system performance has little impact on ILP workloads, we will focus our subsequent discussions of memory system optimizations on MEM and MIX workloads.

For MIX workloads, long-latency main memory accesses from those memory-intensive applications can overlap with CPU executions of the compute-intensive ones. The performance loss due to DRAM accesses is moderate. For the 2-MIX, 4-MIX, and 8-MIX workloads, the performance losses are 9.8%, 6.6%, and 6.7%, respectively.

For MEM workloads, main memory accesses cause severe performance loss. There are two-sided effects on the performance loss as the number of threads increases. More active threads cause more contentions on caches. On the other hand, as the number of threads increases, the newly added applications to the workload mix are less memory-intensive than those existing ones. On average, 2-thread, 4-thread, and 8-thread MEM workloads generate 3.6, 2.6, and 1.5 main memory accesses for every 100 instructions.

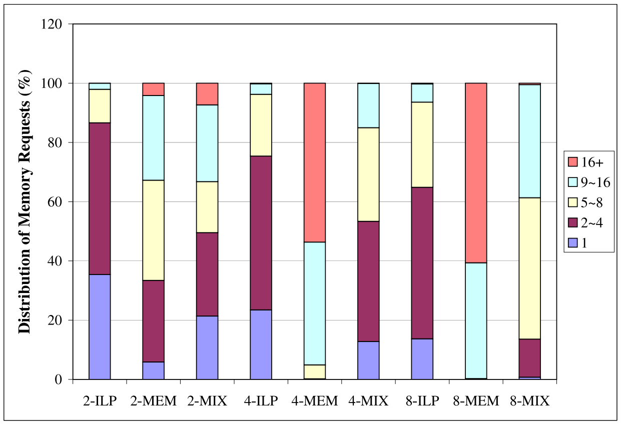

Figure 4: Distributions of the number of outstanding memory requests when the DRAM system is busy.

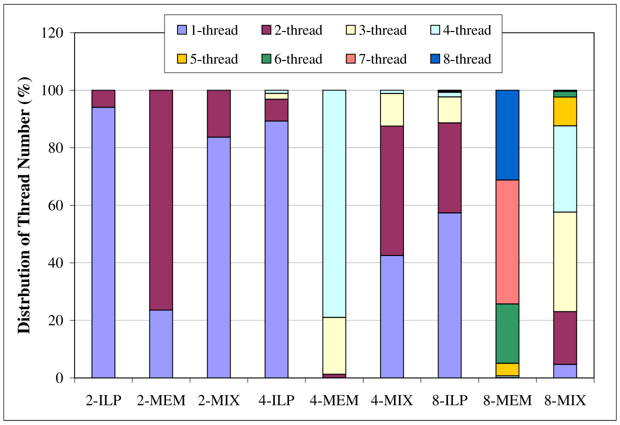

Figure 5: Distribution of the number of threads that generate outstanding memory requests when multiple requests are presented to the DRAM system.

Those main memory accesses cause the weighted speedup to drop by *73.4%*, *83.3%*, and *69.7%* compared to that on the system with the infinitely large L3 cache. This indicates that if multiple memory-intensive threads are running together, the main memory system is the performance bottleneck, even after applying fetch policies that can alleviate resource clogs due to long-latency memory accesses. For SMT systems, there is still plenty of space left for memory system optimizations.

5.2 Memory Access Concurrency

Previous studies have shown that it is common that a single-threaded processor may issue multiple memory accesses to the main memory system in a short time period [18, 17, 13, 21, 22, 16, 34]. The SMT technique affects this phenomenon in two opposite directions. The existence of multiple threads may increase the memory access concurrency due to multiple memory accessing streams. On the other hand, fetch policies that handle long-latency memory accesses, such as the DWarn policy, lower the fetch priorities of threads with outstanding cache misses and may reduce the memory access concurrency as a side effect. Figure 4 shows the distribution of the number of outstanding memory requests in the 2-channel DDR SDRAM system when the memory system is busy. We can see that for SMT workloads, even after applying the DWarn policy, multiple memory requests still group together in most cases. Even for the workload with the lowest memory access concurrency, 2-ILP, 64.6% of memory requests are clustered with at least one more request. For the memory-intensive workloads, such as 4-MEM and 8-MEM, almost all memory requests happen in groups.

The trends in the distribution of the number of outstanding memory requests are quite predictable. For workloads consisting of the same number of threads, the MEM workload has the highest probabilities to make large numbers of concurrent memory requests. For instance, the probability that more than eight requests are presented to the DRAM system when it is serving at least one request is 95.3% for the 4-MEM workload. In comparison, for the 4-ILP and 4-MIX workloads, this probability drops sharply to 3.8% and 15.0%, respectively. Another trend is that when the number of threads increases, it is more likely to see a larger number of concurrent memory requests. As an example, when the number of threads doubles from two to four then to eight, the probability that the MEM workloads make more than sixteen concurrent requests to the DRAM system jumps from 4.2% to 53.7%, then to 60.7%.

The above results indicate that it is common that SMT processors make multiple main memory requests concurrently. Thus, the memory optimization techniques that are proposed for single-threaded processors and utilize the memory access concurrency should still be able to improve the performance of SMT systems. We will analyze their performance impact in details in subsequent sections. Before that, let us see whether those concurrent requests are generated by multiple threads or just a single one.

Figure 5 presents the probability that X threads (e.g. X can take one or two for 2-thread workloads) generate all the requests when multiple memory requests exist. As discussed above, compared to the MIX and MEM workloads, the ILP workloads are less likely to have concurrent memory requests. Even when multiple requests happen closely, it is very likely that they are generated by a single thread; the probability ranges from 94.1% for 2-ILP to 57.4% for 8-ILP. However, for MEM workloads, those concurrent requests are generated by most threads. For example, with probabilities of 76.4% and 79.0%, those multiple requests are generated by all the threads for 2-MEM and 4-MEM workloads, respectively. For the 8-MEM workload, concurrent requests come from at least seven threads with the probability of 74.3%.

In summary, for SMT workloads, even after applying the

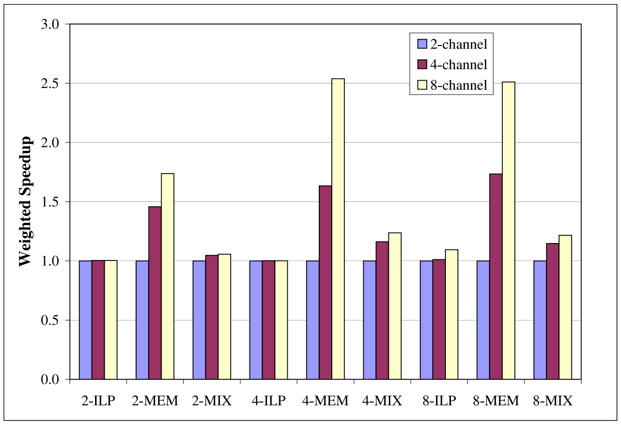

Figure 6: Performance comparisons as the number of memory channels increases.

DWarn policy, the memory access concurrency is still quite high. In addition, the concurrent accesses normally come from multiple threads. This indicates that for those applications, thread-aware memory optimization techniques may gain some performance by reordering the multiple requests coming from different threads. We will discuss their performance potential in Section 5.5 in details.

5.3 Memory Channel Configurations

Increasing the number of memory channels effectively improves the memory bandwidth. Memory channels may be ganged (clustered) in several ways to achieve the best performance [4]. Figure 6 presents the performance as the number of memory channels increases from two to four and eight, when each channel can serve one request independently. The data are normalized to that on the 2-channel system for each workload.

The results confirm that the memory bandwidth is a major performance bottleneck for workloads consisting of multiple memory-intensive applications. In general, increasing the memory bandwidth is more effective in improving the performance for workloads with a larger number of threads. For 2-MEM, 4-MEM, and 8-MEM workloads, quadrupling memory channels from two to eight can achieve weighted speedup of 73.7%, 153.8%, and 151.1%, respectively. As shown in Figure 4, 2-MEM, 4-MEM, and 8-MEM workloads issues more than eight memory requests in short time period with probabilities of 32.8%, 95.1%, and 99.8%, respectively. Thus, for those workloads, doubling and quadrupling memory channels enable more requests to be served concurrently and thus improve the overall performance. Increasing the memory bandwidth is less effective in improving the performance for MIX workloads, since they are less bandwidth-bounded. Quadrupling memory channels can gain the overall performance improvement by 5.6% (for 2-MIX) to 23.7% (for 4-MIX). For 2-ILP and 4-ILP workloads, since the 2-channel system has already achieved the performance within 1% losses compared with the infinitely large L3 cache, increasing the memory bandwidth has negligible impact on them. For the 8-ILP workload, the 8-channel system can narrow the performance loss to 1.0% from the 3.7% loss on the 2-channel system.

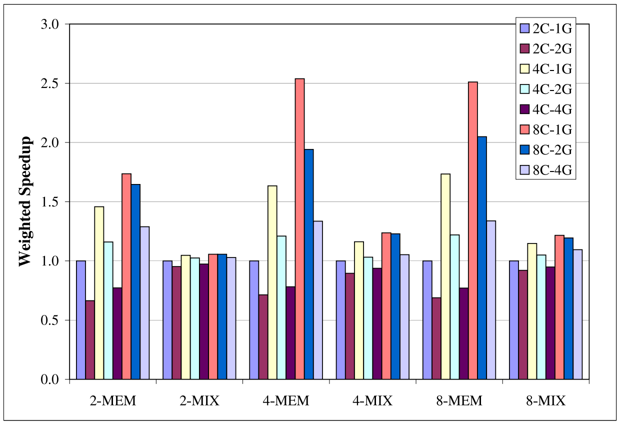

Figure 7: Performance comparisons of clustering physical memory channels to logical ones. “2C-1G” stands for two physical channels where each channel is also a logical one; while “4C-2G” means four physical channels where every two of them are clustered as one logical channel.

Multiple physical channels can be ganged together as a logical channel to serve one memory access. A wider logical channel can shorten the bus transfer time for a single memory request; however, it also reduces the number of requests that can be served concurrently and enlarges the queueing delay and the overall latency if multiple requests happen together. When multiple physical memory channels exist, they can be clustered in several ways to form logical channels. For example, an 8-channel memory system can be configured to have eight independent logical channels (8C-1G), or four independent channels (8C-2G), or two independent channels (8C-4G)[3]. Previous studies have shown that the memory system performance is sensitive to how channels are organized for single-threaded processors [4]. Figure 7 compares the performance of different channel organizations for MEM and MIX workloads[4].

We can see that for MEM and MIX workloads, their performance is quite sensitive to how multiple channels are organized. For instance, when the 2-channel system gangs both channels as a single logical channel, the performance of 2-MEM decreases by 33.6%, compared with configuring each channel independently. When the number of threads increases, the performance gap between different channel organizations is even wider. For the 4-MEM workload, the 8C-4G organization only achieves 52.8% of the performance of the 8C-1G organization. Thus, we need to be careful in

[3] Because the L3 cache line is 64B wide and each physical channel is 16B wide, it does not make sense to gang eight channels together.

[4] Since the performance of ILP workloads is not sensitive to memory channel configurations as shown in Figure 6, ILP workloads are not included here.

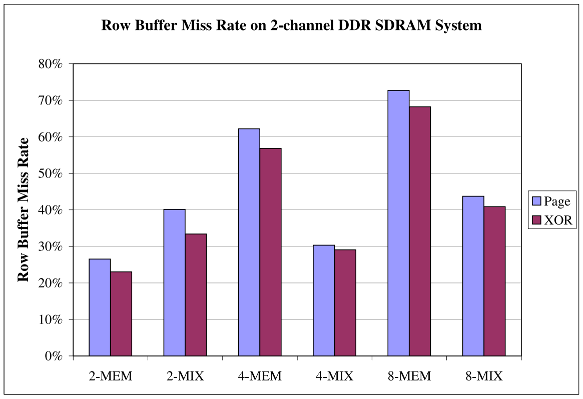

Figure 8: Comparisons of row buffer miss rates under two different mapping schemes (page and XOR) on the 2-channel DDR SDRAM system.

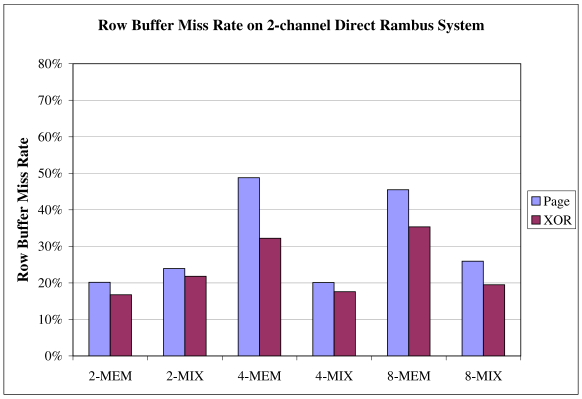

Figure 9: Comparisons of row buffer miss rates under two different mapping schemes (page and XOR) on the 2-channel Direct Rambus DRAM system.

configuring multi-channel memory systems. For the SMT workloads presented here, it is always beneficial to let each physical channel be an independent logical channel. For the workloads of high memory access concurrency, serving multiple requests concurrently is more important than reducing the bus transfer time of a single request.

5.4 Mapping Schemes

As discussed in Section 2, DRAM row buffer hits have significantly shorter latencies than row buffer misses, thus improving the row buffer hit rate is effective in reducing the memory stall time. Previous studies have shown that DRAM mapping schemes have significant impacts on row buffer miss rates for single-threaded processors [32, 33, 8]. SMT processors may cause extra contentions in accessing memory banks, thus may increase row buffer miss rates. Figure 8 compares row buffer miss rates under two DRAM mapping schemes on the 2-channel DDR SDRAM system.

The page mapping scheme assigns DRAM pages to memory banks in a round-robin way; while the XOR-based mapping scheme permutes DRAM pages to memory banks by using the exclusive-OR of the bank index and a portion of address bits from the cache set index [33, 8]. From the figure, we can see that in general, as the number of threads increases, the row buffer miss rate also increases. This is because the number of active accessing streams (from different threads) increases. The XOR-based mapping scheme reduces the row buffer miss rate moderately compared with the page mapping scheme. For example, the row buffer miss rate is reduced from 40.1% to 33.4% for 2-MIX workload. Note that the row buffer miss rates do not always increase with the degree of multithreading. For example, the miss rates for 4-MIX are slightly lower than those of 2-MIX, while 8-MIX has the highest miss rates. One of the programs used in those MIX workloads is *mcf*, which is more memory intensive than any other program. In 4-MIX, the additional programs dilutes the memory access intensity, resulting in the slightly lower miss rates. In 8-MIX, however, the factor of memory contentions significantly increases the row buffer miss rates. The row buffer miss rates are still quite high for workloads with large numbers of memory-intensive applications even after applying the XOR-based mapping scheme, because the DDR SDRAM system used in our study only has a small number of independent banks (eight for the 2-channel system). As we discussed earlier, MEM and MIX workloads generate more than eight concurrent requests during most of their execution times.

The DDR SDRAM system normally has a small number of memory banks. In comparison, the Direct Rambus DRAM system has a large number of internal memory banks (32 banks per chip) and consists of a narrow but high-speed bus. Next, we will show the impact of mapping schemes on a 2-channel Direct Rambus DRAM system. Figure 9 presents the row buffer miss rates on such a system under the page and XOR-based mapping schemes. The XOR-based scheme effectively reduces the row buffer miss rate. For example, the row buffer miss rate of the 4-MEM workload is reduced from 48.8% to 32.2%. Compared with the results in this figure with those in Figure 8, we can see that as the number of independent banks increases, the XOR-based mapping scheme has more opportunities to permute concurrent accesses and achieves lower row buffer miss rates.

Compared with the results on single-threaded systems reported in previous studies, the XOR-based mapping scheme is less effective for SMT systems in reducing row-buffer miss rates. For SMT workloads, the mapping scheme should take row buffer conflicts from multiple threads into considerations. Further researches on DRAM hardware mapping schemes or OS manipulations of memory allocations (for example, using the page coloring) may help reduce the conflicts from multiple threads.

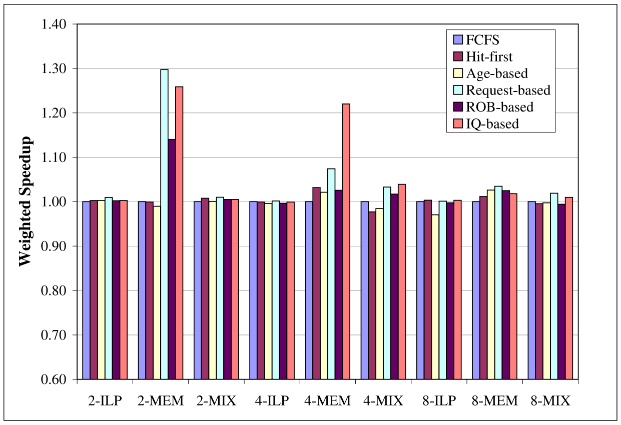

Figure 10: Performance comparisons of thread-aware scheduling schemes (Request-based, ROB-based, and IQ-based) with the schemes that treat accesses from different threads the same (FCFS, Hit-first, and Age-based).

5.5 Effectiveness of Thread-aware Memory Optimizations

In previous discussions, the memory system performs optimizations on accesses from different threads as if they were generated by a single thread. Our experiments have shown that contemporary memory optimization techniques work well under this simple assumption. Next, we will discuss whether additional performance can be gained by considering the running states of each thread.

The reference point (FCFS) in Figure 10 refers to the 2-channel DDR SDRAM system that serves memory requests according to their arrival orders but allows read operations to bypass writes. The "Hit-first" corresponds to the system that exploits the optimization techniques for single-threaded systems, such as the hit-first, exclusive-OR mapping, and open page mode. The "Aged-based" scheme prompts the oldest request when more than eight outstanding requests are presented to the memory. As discussed in Section 3, the request-based scheme assigns the highest priority to the accesses generated by the thread which has the fewest pending memory requests. The ROB-based scheme serves first the requests from the thread which occupies the most reorder buffer entries. The IQ-based scheme assigns the highest priority to the requests generated by the thread that occupies the most integer issue queue entries[5].

Figure 10 indicates that those scheduling policies are the most effective for MEM workloads. The hit-first scheme improves the performance by up to 3.2% (for 4-MEM), compared with a simple FCFS policy. The age-based scheme gets slightly worse performance than the hit-first scheme on most workloads. The three thread-aware scheduling policies can further gain some performance. The request-based scheme can improve the performance by 29.8% for 2-MEM, but only 7.4% for 4-MEM, and 3.5% for 8-MEM. The reason is that for 4-MEM and 8-MEM workloads, the difference of the number of concurrent outstanding requests among different threads is not as large as for 2-MEM workload. The ROB-based scheme can improve the performance by 14.0% for 2-MEM, 2.6% for 4-MEM, and 2.5% for 8-MEM, respectively. The IQ-based scheme achieves speedup of 25.9% for 2-MEM, 22.0% for 4-MEM, and 1.8% for 8-MEM, respectively. The thread-aware scheduling schemes are the most effective for workloads that consist of applications with diverse memory demands. Overall, the above results indicate that when multiple memory-intensive applications are running together, comprehensively considering the processor status and the memory system status is meaningful to improving the memory performance of SMT systems.

Although our observations are based on the DWarn instruction fetch policy, they may also be applied generally to fetch polices that stop fetching for threads with outstanding cache misses, e.g. the fetch stall policy [28]. A good amount of memory concurrency still exists for that policy, because memory requests are generated from all threads and at least one thread is kept active when all threads have outstanding cache misses. Nevertheless, the access scheduling may favor the DWarn policy because it makes more memory requests available for scheduling.

6 Related Work

The related studies can be divided into two categories: those improving the DRAM throughput or reducing the memory access latency for single-threaded processors, and those on the design, evaluation, and optimization of SMT processors. To our best knowledge, there is no existing study on DRAM memory optimizations for SMT processors. In addition, studies on memory systems for SMPs (Symmetric MultiProcessors) have a different focus.

The performance of the DRAM system is characterized by its latency and effective bandwidth. The memory bandwidth has been improved steadily and rapidly. The frequency of memory bus has increased dramatically in the last decade, and the use of the DDR technique further doubles the memory bandwidth. Buses as wide as 256 bits or even wider have been used in high-end workstations. The Direct Rambus DRAM uses narrow buses and increases the data transfer rate to 800 MHz and beyond.

Most approaches to reducing the DRAM access latency utilize the spatial locality in memory access streams and the huge bandwidth inside DRAM chips. Hidaka et al. [11] studied and prototyped a cached DRAM that integrates conventional SRAM caches into DRAM chips. The Enhanced DRAM includes a large SRAM block per bank to cache a whole DRAM page per access, and the Virtual Channel DRAM uses multiple such blocks of smaller sizes. The

[5] For the workloads and systems used in our study, the integer issue queue has higher occupant ratio than the floating-point issue queue.

XOR-based DRAM address mapping schemes improve the hit rate to the existing DRAM row buffers and reduce the latency without the cost of adding SRAM caches [33, 8].

The memory bandwidth may not be fully utilized due to the limited concurrency, bus overhead, and bank conflicts. Nevertheless, the degree of concurrency in memory accesses from superscalar processors is increasing with the aggressive exploitation of ILP and the use of non-blocking caches. The access scheduling [18, 17, 13, 21, 22, 16, 34] has been shown to be effective for streaming applications as well as conventional memory-intensive applications. The system overhead from read/write turnarounds can be reduced by using write buffers [24]. Another overhead from the timing asymmetry of read and write operations may be removed by changing the write timing [4].

The simultaneous multithreading [12, 30, 9] increases the processor throughput by issuing instructions from more than one thread at a single cycle. It overcomes the ILP limits inherent in applications. Tullsen et al. [30] examined the performance potential of SMT technique and demonstrated that it may outperform the wide-issue superscalar design and the fine-grain multithreading, and perform comparably with the chip-level multiprocessing (CMP). A following study [29] examines the SMT performance bottleneck using a more realistic processor model and evaluates the performance of different instruction fetch policies. The instruction fetch policy is critical to the SMT performance and affects how processor resources are allocated among threads. A number of revised policies [28, 3, 7] have been proposed to handle long-latency memory accesses.

It has been confirmed that the SMT technique is effective for OS-intensive commercial workloads as well [14]. In particular, the cache interference between threads may be reduced by using a virtual-physical address mapping called *bin hopping*, which maps virtual pages to physical pages sequentially. A similar mapping is used in our simulation. The OS performance for SMT processors has been studied in [20]. The symbiotic job scheduling [25] in OS increases the system throughput by co-scheduling groups of threads that cooperate better than others. Thus, memory-intensive programs may be scheduled with compute-intensive ones. The performance of Intel Hyperthreading P4 processor has been analyzed in [27] using mixes of SPEC programs.

The SMT and SMP are similar from the viewpoint of DRAM system designs. Surprisingly, there have been few studies on the SMP performance for multiprogramming workloads – the focus is on the memory consistency, cache coherence, and synchronization for parallel workloads. Main memory systems are evaluated in that context. For SMT processors, the cache sharing in parallel programs is usually beneficial, and well-designed synchronizations can be very efficient [31]. Thus, the memory access behavior of parallel programs will be closer to multiprogramming workloads on SMT processors than on SMP systems. To our best knowledge, there is no thorough study of main memory system performance for multiprogramming workloads on SMPs built with contemporary superscalar processors. Thus, little is known or can be applied in answering our questions.

7 Conclusion

In this study, we have thoroughly evaluated contemporary multi-channel DRAM systems for SMT systems and searched for new DRAM optimization techniques. We find that the employment of SMT techniques has somewhat changed the context of DRAM optimizations but does not make them obsolete. The DRAM channel organization becomes a more important factor; the memory access scheduling is more effective when considering the states of each thread; but exploiting the DRAM row buffer locality becomes less effective. In general, for mixes of programs with light or moderate memory demands, the SMT technique can effectively hide the organizational variants of DRAM systems. However, for workloads with intensive memory demands, DRAM optimizations make a larger difference than on single-threaded systems. Since the use of SMT techniques increases the pressure on cache memories, the latter situation becomes more common. Further performance improvement may come from the cooperation between fetch policies and the memory scheduling or between the hardware and the software.

Acknowledgment:
We would like to thank the anonymous referees for their constructive criticism and insightful suggestions which helped us improve the paper.

References

[1] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, 2000.

[2] D. Burger, J. R. Goodman, and A. Kägi. Memory bandwidth limitations of future microprocessors. In 23^{rd} *Annual International Symposium on Computer Architecture*, pages 78–89, 1996.

[3] F. J. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. Dcache warn: an I-Fetch policy to increase SMT efficiency. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 2004.

[4] V. Cuppu and B. Jacob. Concurrency, latency, or system overhead: Which has the largest impact on uniprocessor DRAM-system performance? In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 62–71, 2001.

[5] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary DRAM architectures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*, pages 222–233, 1999.

[6] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings*

of the 28th Annual International Symposium on Computer Architecture, pages 266–277, 2001.

[7] A. El-Moursy and D. H. Albonesi. Front-end policies for improved issue efficiency in SMT processors. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture*, pages 31–41, 2003.

[8] W. fen Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecure*, pages 301–312, 2001.

[9] M. Gulati and N. Bagherzadeh. Performance study of a multithreaded superscalar microprocessor. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*, pages 291–302, 1996.

[10] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, Sept. 1997.

[11] H. Hidaka, Y. Matsuda, M. Asakura, and K. Fujishima. The cache DRAM architecture: a DRAM with an on-chip cache memory. *IEEE Micro*, 10(2):14–25, Apr. 1990.

[12] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 136–145, Gold Coast, Australia, May 1992.

[13] S. I. Hong, S. A. McKee, M. H. Salinas, R. H. Klenke, J. H. Aylor, and W. A. Wulf. Access order and effective bandwidth for streams on a Direct Rambus memory. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 80–89, Jan. 1999.

[14] J. Lo, L. Barroso, S. Eggers, K. Gharachorloo, H. Levy, and S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, pages 39–51, June 27–July 1 1998.

[15] K. Luo, J. Gummaraju, and M. Franklin. Balancing thoughput and fairness in SMT processors. In *International Symposium on Performance Analysis of Systems and Software*, pages 164–171, 2001.

[16] B. K. Mathew, S. A. McKee, J. B. Carter, and A. Davis. Design of a parallel vector access unit for SDRAM memory systems. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 39–48, Jan. 2000.

[17] S. A. McKee and W. A. Wulf. Access ordering and memory-conscious cache utilization. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 253–262, Jan. 1995.

[18] S. A. Moyer. *Access Ordering and Effective Memory Bandwidth*. PhD thesis, University of Virginia, Department of Computer Science, Apr. 1993. Also as TR CS-93-18.

[19] V. S. Pai and S. Adve. Code transformations to improve memory parallelism. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 147–155, 1999.

[20] J. Redstone, H. Levy, and S. Eggers. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *Proceedings of the 9th International Conference on Architectural Support for Programming Language and Operating Systems*, pages 245–256, Nov. 2000.

[21] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 3–13, Nov. 1998.

[22] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 128–138, 2000.

[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, 2002.

[24] K. Skadron and D. W. Clark. Design issues and tradeoffs for write buffers. In *Proceedings of the Third International Symposium on High Performance Computer Architecture (HPCA '97)*, pages 144–155, Feb. 1997.

[25] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic job-scheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the 2002 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS-02)*, pages 66–76, June 2002.

[26] Standard Performance Evaluation Corporation. *SPEC CPU2000*. http://www.spec.org.

[27] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 26–35, 2003.

[28] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 318–327, Austin, Texas, Dec. 2001.

[29] 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 Annual International Symposium on Computer Architecture*, pages 191–202, 1996.

[30] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[31] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 54–58, Jan. 1999.

[32] W. Wong and J.-L. Baer. DRAM on-chip caching. Technical Report UW CSE 97-03-04, University of Washington, Feb. 1997.

[33] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pages 32–41, 2000.

[34] Z. Zhu, Z. Zhang, and X. Zhang. Fine-grain priority scheduling on multi-channel memory systems. In *Proceedings the Eighth International Symposium on High-Performance Computer Architecture*, pages 107–116, 2002.