Proceedings of the 8th International Symposium on High Performance Computer Architecture (HPCA-8), 2002

Fine-grain Priority Scheduling on Multi-channel Memory Systems

Zhichun Zhu Zhao Zhang Xiaodong Zhang
Department of Computer Science
College of William and Mary
{zzhu, zzhang, zhang}@cs.wm.edu

Abstract

Configurations of contemporary DRAM memory systems become increasingly complex. A recent study [5] shows that application performance is highly sensitive to choices of configurations, and suggests that tuning burst sizes and channel configurations be an effective way to optimize the DRAM performance for a given memory-intensive workload. However, this approach is workload dependent. In this study we show that, by utilizing fine-grain priority access scheduling, we are able to find a workload independent configuration that achieves optimal performance on a multi-channel memory system. Our approach can well utilize the available high concurrency and high bandwidth on such memory systems, and effectively reduce the memory stall time of memory-intensive applications. Conducting execution-driven simulation of a 4-way issue, 2 GHz processor, we show that the average performance improvement for fifteen memory-intensive SPEC2000 programs by using an optimized fine-grain priority scheduling is about 13% and 8% for a 2-channel and a 4-channel Direct Rambus DRAM memory systems, respectively, compared with gang scheduling. Compared with burst scheduling, the average performance improvement is 16% and 14% for the 2-channel and 4-channel memory systems, respectively.

1 Introduction

As the performance gap between processor and DRAM memory continues to widen, the memory stall time of a typical memory-intensive application is becoming a dominant portion of the total execution time. On a multi-issue and multi-GHz processor, the latency of a single DRAM access could be equivalent to the time to execute hundreds of CPU instructions. Even for applications with low cache miss rates, the memory stall time due to a small percentage of DRAM accesses can easily exceed the CPU execution time. It is highly desirable to reduce memory stall times of memory-intensive applications.

Configurations of contemporary DRAM memory systems become increasingly complex. Modern memory systems, such as Direct Rambus DRAM systems, can support multiple memory channels, while each channel can connect multiple devices (chips). Each chip consists of multiple banks, where concurrent accesses to different banks can be pipelined. For memory-intensive applications running on contemporary computer systems, the occurrence of multiple outstanding memory requests is frequent. Memory access scheduling can reorder the sequence of concurrent accesses to reduce access latency and improve memory bandwidth utilization [12, 11, 7, 14, 15, 10]. In addition, a memory request for a cache miss can be further split into several sub-requests which can be processed separately. Normally, a cache miss results in a cache line fill request that fetches more data than what is immediately required to resume processor execution. This provides an opportunity to improve performance by splitting the request into multiple sub-requests with smaller sizes and serving the critical ones (containing immediately required data) first. On a multi-channel memory system, such a scheduling method requires a number of considerations, such as how to split a single reference, how to assign sub-requests to channels, and how to schedule concurrent accesses.

A recent study [5] finds that program performance is highly sensitive to the DRAM system configuration, and suggests that tuning burst (sub-block) sizes and channel configurations be an effective way to optimize the DRAM system performance for a given memory-intensive workload. Specifically, they evaluate the performance effect of sub-block size on burst ordering, where each cache block is split into multiple sub-blocks and critical sub-blocks are served before non-critical ones. In their study, all sub-blocks from a cache line are mapped to the same channel and the same page. Thus, in order to exploit concurrency within a single channel, the choice of sub-block size becomes a trade-off between reducing latency of critical data access and lowering system overhead. They find that different applications have optimal performance on different sub-block sizes and the optimal sub-block sizes scale with the channel width.

In this study we show that, by utilizing fine-grain priority access scheduling, we are able to find a workload independent configuration that achieves optimal performance on a multi-channel memory system. In order to fully utilize the available bandwidth and concurrency, our approach splits a memory reference into sub-blocks with minimal granularity, and maps sub-blocks from a reference into different channels. All channels can be used to process a single cache line fill request. In order to increase the parallelism between processor execution and memory accesses, fine-grain priority scheduling is exploited. Sub-blocks that contain the desired data are marked as critical ones with higher priorities and are returned earlier than non-critical sub-blocks. This approach is similar to the method of "critical word first", but it also allows critical sub-blocks of one cache block to bypass non-critical sub-blocks from other cache blocks. By combining with existing DRAM scheduling policies, choosing the minimum sub-block size allows faster access to critical data without increasing the memory system overhead.

Figure 1 gives an example that shows the performance potential of fine-grain priority scheduling. In this example, a 4-channel memory system is processing four cache misses concurrently. Each cache block is split into eight sub-blocks, and the four critical sub-blocks are mapped to different channels. With fine-grain priority scheduling, all the critical sub-blocks finish earlier than non-critical sub-blocks, saving seven time units in fetching all critical data. In this example, the clustering of the four cache misses provides the scheduling opportunity. Our study will show that the cache miss clustering is frequent, i.e., the burstiness of cache misses is high. As a result, the queuing delay can be a major component of access time. Fine-grain priority scheduling can reduce the memory stall time by reducing the queuing delay of critical data.

In this study, we quantitatively investigate the miss burstiness for memory-intensive applications from the SPEC2000 benchmark suite on ILP processors with multi-channel Direct Rambus DRAM systems. We also analyze the combination of fine-grain priority scheduling with other DRAM access scheduling techniques, and compare the performance with that of gang scheduling [8] and burst scheduling [5]. Our study provides the following performance results and findings.

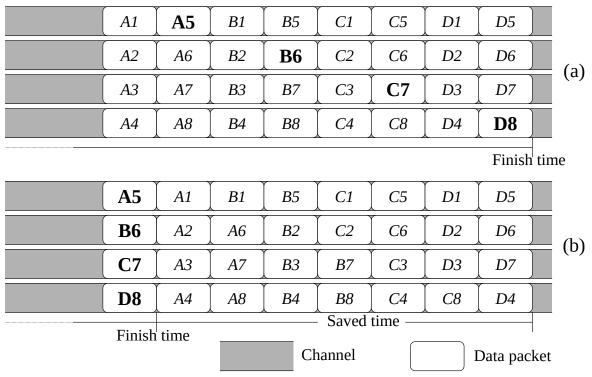


Figure 1. The order of transferring sub-blocks on a DRAM system with four memory channels: (a) without priority scheduling and (b) using fine-grain priority scheduling. The letters A–D represent cache blocks, each of which is split into eight sub-blocks. The boxes with bold letters represent the critical sub-blocks that contain the desired data.

- Fine-grain priority scheduling is effective in reducing memory stall time and increasing IPC (Instructions Per Cycle). Compared with gang scheduling that serves a single cache miss request with multiple channels grouped together, the IPC improvement is 13% on average (up to 34%) for fifteen selected SPEC2000 programs on a 2-channel Direct Rambus DRAM memory system, and 8% on average (up to 22%) on a 4-channel memory system. Compared with burst scheduling that serves multiple sub-requests of a single cache miss with one channel but critical sub-requests first, the average IPC improvement is 16% and 14% on the 2-channel and 4-channel memory systems, respectively. The processor is 2 GHz and 4-way issue.

- Combined with other scheduling policies, fine-grain priority scheduling is able to effectively utilize the memory system resource. For six of the programs, the 2-channel system with fine-grain priority scheduling can achieve performance comparable to that on the 4-channel system with gang scheduling or with burst scheduling.

- We suggest that a DRAM system configuration and its optimization be emphasized on access scheduling and DRAM mapping schemes. Taking this approach, we are able to find an optimal memory configuration that is workload independent.

We briefly introduce the background in the next section. In Section 3, we discuss the issues in fine-grain priority scheduling and its combination with other DRAM scheduling policies. The design complexity of fine-grain priority scheduling is discussed in Section 4.

The experimental methodology is described in Section 5. The results are presented in Section 6. Finally, we conclude our work in Section 7.

2 Memory System Considerations

2.1 Memory Access Scheduling

Contemporary DRAM memory systems can serve multiple accesses concurrently. Memory access scheduling can reduce access latency and improve bandwidth utilization by re-arranging the order and issue time of DRAM operations for a group of concurrent requests [12, 11, 7, 14, 15, 10]. At a given time, a request may require one of the following operations, depending on the state of the bank to be accessed.

- *Precharge*: when the row buffer contains valid but not the desired data. This request is called a row buffer miss.
- *Row access*: when the bank is already *precharged*.
- *Column access*: when the row buffer contains the desired data. This request is called a row buffer hit.

Different operations required by concurrent memory requests may contend for the control bus, the data bus, or the DRAM banks. The contentions can be resolved by prioritizing the requests based on the request type (read or write), the arrival time, or an explicit priority information [15]. For example, *read-bypass-write policy* gives read requests higher priorities than write requests, considering that read requests will block the related load instructions. *In-order scheduling* gives the oldest request the highest priority. This can be combined with the read-bypass-write policy to prioritize old read requests. *Explicit priority scheduling* assigns an explicit priority to each request, giving the processor the opportunity to specify critical requests.

When a bank is activated (the row buffer contains valid data), it is possible that one request to the bank requires a column access while another request asks for a precharge. The column access is usually prioritized over the precharge so as to improve the memory bandwidth utilization [7, 15]. This policy is called *hit-first* in this paper. Operations to different banks may contend for the address bus and the data bus. To increase parallelism at the DRAM side (thus increase the bandwidth utilization), precharges can be prioritized over row accesses, and row accesses can be prioritized over column accesses.

Another scheduling issue is to decide the time to precharge a bank when it has no pending requests. There are two strategies: *close page* and *open page*. The close page strategy issues the precharge immediately after the current column access finishes. The next access to the bank will require a row activation and a column access. In contrast, the open page strategy delays the precharge to hope that the next access is a row buffer hit, thus only the column access is needed. However, if the next access is a row buffer miss, it will require all the three operations. Which strategy performs better depends on the row buffer hit rate.

2.2 Multi-channel Memory Systems

Multi-channel memory systems have been used with high performance processors that require high bandwidth DRAM memories. Each channel can be scheduled independently. Direct Rambus DRAM is such a representative memory system. A Direct Rambus DRAM system generally consists of multiple channels, where each channel supports 1.6 GB/s bandwidth. Each channel has its own row control bus, column control bus, and two-byte wide data bus. The separation of row and column control buses eliminates the contention in the address bus between row operations (precharges and row activations) and column accesses. The bus clock rate is 400 MHz and the data is transfered on both edges of the clock. The row and column addresses/commands and the data are transfered in packets, each taking four bus cycles. The minimal data packet length is 16-byte. Each channel can connect multiple devices (chips). Each device can have 32 banks and 33 half-page row buffers (this may be different according to the configuration). Those banks may be operated independently, which provides high concurrency at the bank level. The Intel Pentium 4 processor supports two channels, and the Compaq Alpha 21364 processor supports up to eight channels.

2.3 DRAM Mapping Scheme

DRAM mapping scheme determines how to map a physical address to a location in the DRAM system. The choice of DRAM mapping scheme directly affects the row buffer hit rate and the memory system performance [20].

A word in a Direct Rambus DRAM system is addressed by the channel index, the device index, the bank index, the row address, and the column address. The first mapping consideration is on how to map the sub-blocks in a cache line onto multiple channels. We use a method interleaving the sub-blocks onto all channels, which is the same as that used in [8]. This interleaving scheme allows the aggregate bandwidth of all channels to be used to transfer a single cache line (assume the number of sub-blocks in a cache line is no less than the number of channels). The mapping scheme in [5] maps all cache lines in a DRAM page-sized block on a single channel. The requests on different channels are scheduled independently. However, this scheme cannot fully utilize the available bandwidth of all channels for a single cache line fill request. In addition, program access locality within the DRAM page-sized block may cause unbalanced usage of memory channels. In contrast, our method groups channels together to serve each cache line fill request, but schedules operations on each channel independently to return critical sub-blocks earlier.

Another mapping consideration is on how to map continuous addresses to multiple banks. Our approach interleaves page-sized blocks onto banks using the XOR-based page-interleaving scheme [20, 8]. It maps a continuous DRAM page-sized block onto a DRAM bank to exploit the locality in the row buffer, and XOR-es two portions of address bits (conventional bank index and cache tag) to permute the mapping of pages to banks. Consequently, accesses causing row buffer conflicts in the conventional page-interleaving scheme are distributed to different banks without changing the locality in the row buffer. The studies in [20, 8] show that the scheme can significantly improve the row buffer hit rate.

3 Fine-grain Priority Scheduling

3.1 Granularity of Scheduling

Current ILP processors have the ability to generate multiple cache misses before stalling. This provides an opportunity for performance improvement by scheduling the concurrent memory requests for those cache misses. In general, only a portion of a cache line contains the currently required data (although other portions may be needed in the near future). The fine-grain priority scheduling tries to exploit this opportunity. It issues multiple DRAM requests for a single cache miss, where each request fetches a *sub-block* of the cache line. Sub-blocks that contain the desired data are *critical sub-blocks*. The requests for critical sub-blocks are given higher priority over those requests for non-critical ones.

Each DRAM system has a limit on the minimal request length. Thus, the sub-block size should be no less than that minimal length. Using smaller sub-block size allows the current request to finish faster and makes newly arrived requests to be issued earlier. However, it is a concern that a small sub-block size may reduce the burst length of DRAM accesses thus increase the system overhead [5]. Nevertheless, we will show that if fine-grain priority scheduling is combined with other scheduling techniques and suitable mapping schemes, the overhead will not exceed that of coarse-grain scheduling on the Direct Rambus DRAM platform. For this reason, we choose the smallest granularity available for Direct Rambus DRAM system as the sub-block size, which is 16-byte, in this study.

3.2 Scheduling Policies

In this paper, we discuss three scheduling policies: fine-grain priority scheduling, gang scheduling, and burst scheduling. Each term actually represents a combination of several basic access scheduling policies, a channel configuration, a DRAM mapping scheme, and a choice of scheduling granularity. We assume a scheduler architecture similar to that presented in [15] (see Section 4) is used to enforce the three policies.

- Pending requests on a DRAM bank are queued in a bank scheduler. Each bank scheduler has an arbiter to determine the next operation on the associated bank.
- Each independent channel has a channel scheduler, which includes a row arbiter and a column arbiter. The row arbiter selects a precharge or a row access (if any) to use the row control bus. The column arbiter selects which column access (if any) to use the column control bus and the data bus.

All the three scheduling policies are combined with four basic scheduling policies that are enforced in the following order: *read-bypass-write*, *hit-first*, *explicit priority*, and *in-order*. For example, a non-critical read request that requires a column access is issued first even when there is another critical read request that requires a precharge to the same bank. The hit-first policy is enforced before the explicit priority scheduling so that fine-grain priority scheduling would not cause severe row buffer thrashing when multiple requests are mapped onto the same bank. In contrast, enforcing only the explicit priority scheduling may cause more precharges when bank conflicts occur.

There are three levels of explicit priorities for read requests, namely *critical priority*, *load priority*, and *store priority*, from highest to lowest. The critical priority is assigned to critical sub-blocks of read misses, and load priority is assigned to non-critical sub-blocks of read misses. The store priority is assigned to read requests for write misses, as a write-back and write-allocate L2 cache is used in this study.

In fine-grain priority scheduling, each L2 cache miss results in multiple DRAM requests that are mapped to multiple channels evenly; each DRAM request is associated with an explicit priority; and concurrent requests are scheduled based on the policies discussed above. It uses a fixed 16-byte as the sub-block size, which is the smallest granularity with current Direct Rambus technology. Each physical channel is configured as an independent unit and has its own channel scheduler. Instructions stalled for a critical sub-block are resumed when the data of the sub-block is returned.

Gang scheduling uses the cache block size as the burst size. All channel are grouped together as one logical channel, and there is only one channel scheduler. Instructions stalled for a missed block are resumed when the whole block is returned.

In burst scheduling, each L2 cache miss results in multiple DRAM requests that are mapped to the same independent channel; each DRAM request is associated with an explicit priority. In this study, the sub-block size is set to 32-byte. For a 2-channel system, each physical channel is an independent channel. For a 4-channel system, two physical channels are grouped together. There are two channel schedulers in both cases.

When a miss on a cache block happens, the sub-block containing the desired data is marked critical. Due to program locality, it is very likely that when the requests of this miss are being processed, more misses happen on other sub-blocks of the same cache block. In this case, the sub-blocks containing the newly arrived requests become critical ones and gain higher priority. Both fine-grain priority scheduling and burst scheduling will consider this change and update the priority information dynamically.

The read-bypass-write and hit-first policies not only improve the performance by themselves but also help fine-grain priority scheduling avoid the potential increase of system overhead. There is one case that the system overhead may still increase. When the number of banks is very small, the bank conflicts can be severe thus fine-grain priority scheduling may cause more precharges than burst scheduling. Fine-grain priority scheduling always balances the utilization of multiple channels, however, which scheduling performs better will depend on application access patterns. In practice, Direct Rambus memory systems have a sufficient number of banks to avoid severe bank conflicts. SDRAM memory systems usually have large size row buffers which lead to less precharges when the locality in row buffer is good. In addition, large size SDRAM memory systems may also have enough number of banks to avoid severe bank conflicts. The DRAM mapping scheme used in our study produces high row buffer hit rates. Thus, open page mode is used in our experiments.

4 Complexity Analysis

4.1 Complexity inside Processor

Cache and Cache Controller One concern on fine-grain priority scheduling is that it might change L2 cache and/or its controller, because data returns from the memory in a unit of sub-block instead of cache block. Such a change is definitely undesirable. Fortunately, there are existing mechanisms on high-performance processors that can address this issue. For example, the MIPS R10000 has a four-entry incoming buffer that can accept returning data at any rate and at any order [19]. Up to four outstanding read requests to memory are supported, thus each outstanding request is guaranteed to have one allocated incoming buffer entry. The Power-PC 604 has a similar line-fill buffer [16]. The incoming buffer can be used to merge out-of-order returning sub-blocks with only trivial changes. Future high-performance processors that support multiple outstanding memory requests will likely to have such kind of mechanisms.

Address Path to Memory Controller There will be additional lines for transferring priority information. Priority information can be transfered as a bitmap or the position index of a sub-block. Using a bitmap requires more additional lines, but allows priotizing multiple sub-blocks simultaneously, which helps the case when multiple cache misses happen to the same cache line at the same cycle.

Priority Updates The MSHR needs to send priority update to the memory controller when a read miss happens on a non-prioritized sub-block of a cache block that is already missed. A bitmap can be used with each MSHR entry to memorize which sub-blocks have been prioritized[1].

4.2 Complexity in Memory Controller

The basic function of memory controller is to issue DRAM operations (precharge, row activation, or column access) to DRAM banks under the DRAM timing constraints for DRAM access requests. With high-performance processors and high-bandwidth memory systems, the memory controller must have the access scheduling ability to order DRAM operations for multiple outstanding requests. Without this ability, the opportunity to exploit the memory access concurrency allowed by the processor and the memory system will

[1]We assume that the MSHR implementation in [6] is used.

be wasted, and the performance penalty is unacceptable for memory-intensive applications.

A *memory access scheduler architecture* is proposed in [15], which can enforce a number of scheduling policies. Fine-grain priority scheduling policy can be implemented on that architecture. The scheduler architecture organizes incoming requests by DRAM banks. Each bank has its own arbiter to determine its next operation. A global arbiter determines which bank gets the shared resources, such as the address bus and the data bus. This scheduler architecture can be adapted to work with Direct Rambus memory systems. Each independent channel needs a channel scheduler, and each channel scheduler needs two arbiters, one for row control bus and the other for column control bus. Although Direct Rambus memory systems can have a large number of banks, the bank schedulers can be assigned to busy banks dynamically, thus only a limited number of bank schedulers are needed.

With fine-grain priority scheduling on an n-channel system, n channel schedulers are needed. In comparison, with gang scheduling, there is only one independent channel thus one channel scheduler. On the other hand, fine-grain priority scheduling does not complicate the structure of each individual bank scheduler or channel scheduler. Another change is that the memory controller may split one memory reference request into multiple requests onto those channels, and need to accept priority updates. In this aspect, burst scheduling has almost the same complexity as fine-grain priority scheduling.

5 Experimental Environment

We use SimpleScalar 3.0b [3] to simulate an out-of-order execution processor. An event-driven simulation of a multi-channel Direct Rambus DRAM system is incorporated into the original simulator. Table 1 gives the key parameters of the processor model.

Speed	2GHz, 4-way issue
RUU size	64
LSQ size	32
MSHR size	16
write buffer size	8
L1 cache	4-way 64KB inst./data, 2-cycle hit latency, 64B cache line, write-back
L2 cache	unified 4-way 1MB, 8-cycle hit latency, 128B cache line, write-back

Table 1. Key processor parameters.

We use the parameters of 256 Mbit Direct Rambus DRAM [13] as the parameters of DRAM memory system simulated in our experiments. Table 2 describes the key parameters of this DRAM. We configure the simulated system as 2-channel and 4-channel systems, where each channel has four devices.

Parameters	Values
Precharge delay	8 bus cycles
Row access delay	8 bus cycles
Column access delay	8 bus cycles
Length of packets	16 bytes
Banks per device	32
Page size	2KB
Row buffer	33 half-page size

Table 2. Key parameters of the Direct Rambus DRAM used in the simulation. The bus cycle time is 2.5 ns (400 MHz).

We use the pre-complied SPEC CPU2000 Alpha binaries in [18]. Fifteen programs (five integer programs and ten floating point ones) are selected, which have relatively large memory access demands. For all the applications, we fast-forward 4000M instructions and collect program execution statistics on the next 200M instructions.

6 Experimental Results

6.1 Burstiness in Miss Streams

We first measure the fraction of program execution time with bursty memory accesses. Figure 2 shows the fraction of program execution time with two or more outstanding memory references on a 2-channel system with gang scheduling for the selected SPEC2000 programs. We can see that the fraction of bursty phase is highly application dependent, which ranges from about 6% to 90%.

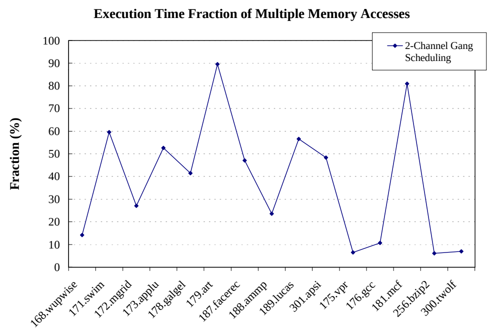


Figure 2. Fractions of bursty phase in execution for SPEC2000 programs.

Figure 3 further presents the distribution of the number of concurrent accesses in the bursty phase. The left figure contains programs with the fraction of bursty

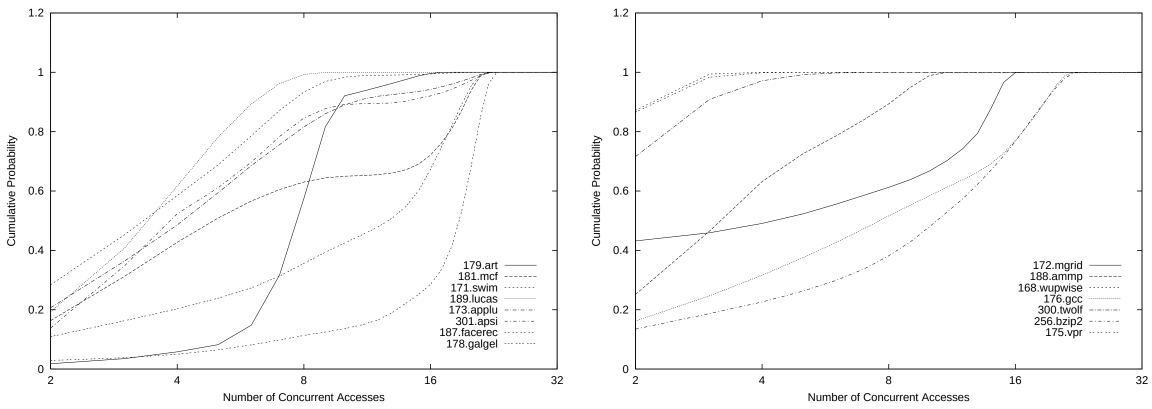


Figure 3. Distribution of the number of concurrent accesses.

phase higher than 40%; and the right one contains programs with the lower bursty phase fraction. Most programs have high burstiness in the bursty phase. In general, programs with a higher fraction of bursty phase tend to have higher probability on large number of concurrent accesses. For all the programs presented in the left figure, more than 40% of bursty references are grouped with at least three other references. Even for some programs with a small bursty phase fraction, the burstiness inside the bursty phase is still high. For example, program *256.bzip2* only spends 6% of its execution time in the bursty phase, however, more than 60% of concurrent accesses are clustered as groups with at least eight references.

6.2 Potentials of Fine-grain Priority Scheduling

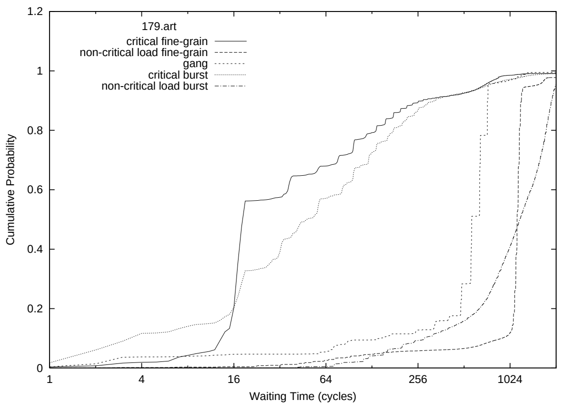
Fine-grain priority scheduling targets at reducing the latency for critical sub-blocks by serving the critical ones before the non-critical ones. However, if all sub-blocks are critical, fine-grain priority scheduling will not make any difference. To evaluate the potential of fine-grain priority scheduling, we measure the percentage of critical sub-blocks in a cache line when the whole cache line fill request completes. Our experiments indicate that on the 2-channel system, for the fifteen programs, this percentage ranges from 15.3% to 57.7%. On average, 33.8% of sub-blocks are critical ones. This indicates that there is a large space left for fine-grain scheduling to reorder requests based on their priorities.

Figure 4 shows the waiting time distribution of critical sub-blocks and non-critical sub-blocks of read misses. We can see that critical sub-blocks have much shorter queuing delay than non-critical ones. Due to space limitation, we use two programs *179.art* and *256.bzip2* as examples here. Program *179.art* has very high bursty phase fraction (about 90%) and high burstiness within the bursty phase. For this application, the waiting time is a significant portion of the total access time. Fine-grain priority scheduling effectively reduces the waiting time for critical sub-blocks. Compared with burst scheduling, the average waiting time for critical sub-blocks reduces from 133 cycles to 104 cycles, and the average waiting time for non-critical load sub-blocks reduces from 1157 cycles to 1070 cycles. With fine-grain priority scheduling, 60% of critical sub-blocks have waiting time less than 36 cycles. In comparison, with burst scheduling, 40% of critical sub-blocks have waiting time longer than 80 cycles. Compared with gang scheduling, the average waiting time for critical sub-blocks reduces from 557 cycles to 104 cycles, but the average waiting time for non-critical load sub-blocks increases from 557 cycles to 1070 cycles. *256.bzip2* has low bursty phase fraction (only 6%) but high burstiness in the bursty phase. Compared with gang scheduling and burst scheduling, the average waiting time for critical sub-blocks is reduced from 42 cycles and 32 cycles, respectively, to 27 cycles.

Figure 5 shows the probability that multiple critical sub-blocks are mapped to the same channel under fine-grain priority scheduling. We can see that for most programs, fine-grain priority scheduling can evenly distribute critical requests to different channels. However, for applications with high burstiness, it is still possible that multiple critical requests are mapped to the same channel. The existence of multiple critical requests in the same channel indicates that fine-grain priority scheduling can reduce the processor waiting time for currently required data.

6.3 Performance Improvement of Fine-grain Priority Scheduling

Figure 6 presents the performance improvement of fine-grain priority scheduling in terms of IPC for 2-channel and 4-channel Direct Rambus DRAM systems. In this figure, the base IPC of each application is the IPC on a system with the perfect DRAM configuration



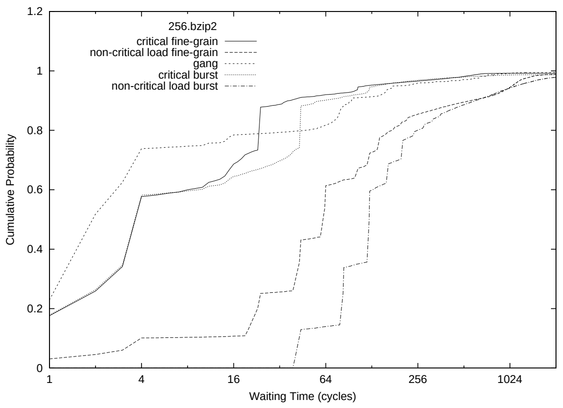
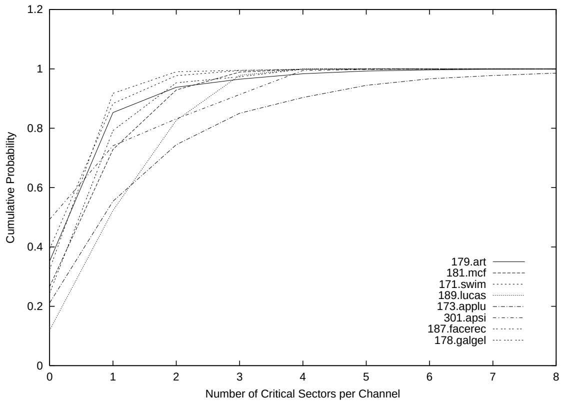


Figure 4. Waiting time distribution of critical and non-critical load sub-blocks.



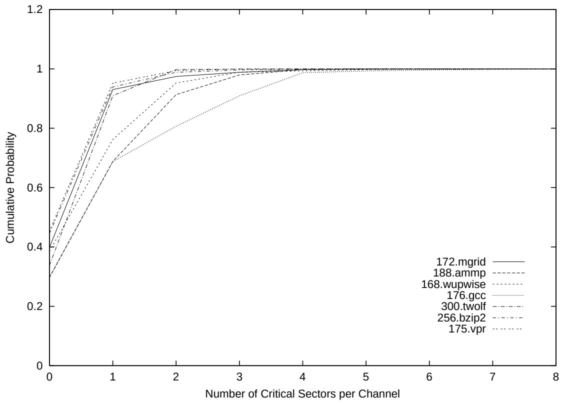


Figure 5. Probabilities of multiple critical sub-blocks mapping to the same channel.

that has the latency of L2 cache hit and an infinite bandwidth. The base IPC reflects the ideal performance after eliminating the memory stall time.

Compared with gang scheduling, fine-grain priority scheduling can increase the IPC by up to 34.2% for the 2-channel DRAM system. For the fifteen selected programs, the average IPC increase is 12.5%. Four programs (*172.mgrid, 173.applu, 181.mcf*, and *300.twolf*) have performance improvement higher than 15%. This implies that fine-grain priority scheduling effectively increases the parallelism between processor execution and DRAM accesses by reducing the latency for critical accesses.

Compared with burst scheduling, fine-grain priority scheduling can increase the IPC by 16.3% on average (up to 38.7%) for the 2-channel DRAM system. This indicates that fine-grain priority scheduling can better utilize the available concurrency of DRAM systems by spreading requests evenly onto multiple channels.

Fine-grain priority scheduling is especially effective for applications with a relative large memory stall portion, modest memory bandwidth demand[2], and high burstiness in miss streams. For applications with small memory bandwidth demand and relatively fewer cache misses, the performance improvement from fine-grain priority scheduling is modest. For example, the memory bandwidth demand of *256.bzip2* is only 0.8 GB/s, the fraction of bursty phase is only 6%, and the number of L2 cache misses per 100 instructions is only 0.11. For this application, the memory stall time is not a significant portion of the total execution time. The fine-grain priority scheduling improves the performance modestly by 4.6% and 3.1% compared with gang scheduling and burst scheduling, respectively.

For applications with extremely high memory bandwidth demands, such as *179.art*, fine-grain priority scheduling improves performance modestly (6.0%) compared with gang scheduling. This is not surprising. The bandwidth demand of the program is so high (64.0 GB/s) compared with the available bandwidth (3.2 GB/s). Returning critical data earlier does not provide a large improvement in this case. Compared with burst scheduling, the performance improve-

[2]We use the memory bandwidth achieved by the application on the perfect DRAM system as the bandwidth demand of the application.

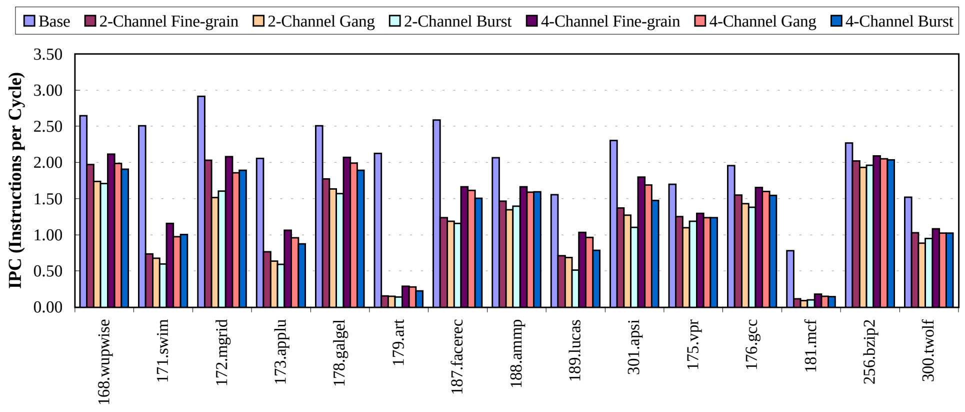


Figure 6. IPC on 2-channel and 4-channel Direct Rambus DRAM systems.

ment is promising (13.2%). This indicates that fine-grain priority scheduling can better utilize the available bandwidth and concurrency by evenly distributing sub-requests to multiple channels.

As the number of channels increases to four, the congestion at the main memory system is reduced because of the increasing bandwidth and concurrency. As expected, the speedup by using fine-grain priority scheduling drops for most applications. For the fifteen programs, fine-grain priority scheduling increases the IPC by 7.8% on average (up to 22.3%) compared with gang scheduling. However, the speedup for some programs with high memory bandwidth demands increases. For example, the performance improvement for *171.swim* increases from 9.1% to 18.8% as the number of channels doubles. It indicates that, when the bandwidth pressure is alleviated for bandwidth-bounded applications, the performance potential of fine-grain priority scheduling increases. Compared with burst scheduling, the average performance improvement of the fifteen programs is 13.9%.

It is interesting to observe that for six of the fifteen programs, the performance on the 2-channel DRAM system after applying fine-grain priority scheduling is comparable or even better than that on the 4-channel DRAM system with gang scheduling. For *168.wupwise, 176.gcc*, and *256.bzip2*, the IPC on the 2-channel DRAM system with fine-grain priority scheduling is within 3% lower than that on the 4-channel DRAM system with gang scheduling. For *172.mgrid, 175.vpr*, and *300.twolf*, the IPC on the 2-channel DRAM system with fine-grain priority scheduling is higher than that on the 4-channel DRAM system with gang scheduling by up to 10%. Compared with burst scheduling on the 4-channel DRAM system, fine-grain priority scheduling gains comparable or better performance on the 2-channel DRAM system for these six programs.

Compared with the 2-channel system, the 4-channel system not only doubles the available bandwidth, but also doubles the number of memory chips. Fine-grain priority scheduling can better utilize the existing resources and achieve performance comparable to that on a system with much higher cost. Of course, for those applications whose performance is limited by the available bandwidth, paying more cost to increase the bandwidth is the most effective way to improve performance.

For all applications on both the 2-channel and the 4-channel systems, fine-grain priority scheduling always achieves the best performance. In comparison, for gang scheduling and burst scheduling, which one performs better is application and configuration dependent.

7 Conclusion

Although careful tuning of DRAM parameters can effectively improve memory performance, its workload dependent feature may limit its usage in practice. In order to address this limit, we present a workload independent approach by focusing on optimizing fine-grain priority scheduling, and show its effectiveness using SPEC2000 benchmark programs. In addition to supporting workload independent configurations, fine-grain priority scheduling can increase the parallelism between processor execution and DRAM memory accesses, and improve the resource utilization of the memory system.

Hardware or software prefetching [17] is an effective approach to increase the parallelism between processor execution and DRAM operations. Recently, an increasing number of studies have been done on precomputation-based prefetching techniques [1, 2, 4, 9, 21]. By using speculative execution threads to detect future cache misses, those techniques not only increase memory access concurrency but also provide priority information on the prefetched data. However, the prefetch lookahead time is usually limited so that returning critical data early is desirable. We believe fine-grain priority scheduling can make an effective match for those techniques.

Acknowledgment:
We thank the anonymous referees for their constructive comments and insightful suggestions. This work is supported in part by the National Science Foundation under grants CCR-9812187, EIA-9977030, and CCR-0098055. This work is also a part of an independent research project sponsored by the National Science Foundation for program directors and visiting scientists.

References

[1] M. M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 52–61, June 2001.

[2] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Dynamically allocating processor resources between nearby and distant ILP. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 26–37, June 2001.

[3] D. C. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.

[4] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 14–25, June 2001.

[5] 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, June 2001.

[6] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 133–143, June 1997.

[7] 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.

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

[9] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 40–51, June 2001.

[10] 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.

[11] 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.

[12] 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.

[13] Rambus Inc. *256/288-Mbit Direct RDRAM (32 Split Bank Architecture)*, 2000. http://www.rambus.com.

[14] 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.

[15] 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, June 2000.

[16] S. P. Song, M. Denman, and J. Chang. The PowerPC-604 RISC microprocessor. *IEEE Micro*, 14(5):8–17, Oct. 1994.

[17] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, June 2000.

[18] C. Weaver. http://www.simplescalar.org. *SPEC2000 binaries*.

[19] K. C. Yeager. The MIPS R10000 superscalar microprocessor: Emphasizing concurrency and latency-hiding techniques to efficiently run large, real-world applications. *IEEE Micro*, 16(2):28–40, Apr. 1996.

[20] 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 IEEE/ACM International Symposium on Microarchitecture (MICRO-33)*, pages 32–41, Dec. 2000.

[21] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 2–13, June 2001.