A MULTIPORTED NONBLOCKING CACHE FOR A SUPERSCALAR UNIPROCESSOR

BY

JAMES EDWARD SICOLO

B.S., State University of New York at Buffalo, 1989

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

ACKNOWLEDGMENTS

I would like to thank Professor Wen-mei Hwu for his guidance and inspiration. Dr. Hwu's exciting and enlightening discussions are the among the highlights of my experiences at the University of Illinois.

In addition, I would like to thank William Chen for his close monitoring of the completion of this thesis. William proposed the initial direction of this thesis and has helped provide solutions to the problems encountered along the way.

I would also like to thank Randy Cetin for providing the computing power on which this research was done.

Finally, I wish to thank all of my friends and relatives who have made my life in graduate school very enjoyable.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Superscalar processors provide increased performance by allowing the simultaneous issue of multiple instructions. To effectively support this enhanced execution rate, multiple functional units are needed. In the case of memory instructions, aggressive superscalar processors will require cache memory systems capable of handling multiple hits and misses simultaneously. Current first-level cache designs support only a single memory request at a time and are unsuitable for future aggressive superscalar processors.

There are two ways in which current first-level cache designs should be modified to support superscalar execution. First, they should provide multiple ports so that multiple requests can be made and serviced in parallel. It is easy to see how a bottleneck may emerge in some programs if an aggressive superscalar processor is implemented but constrained by a single memory instruction issue rate.

The second modification is to make caches nonblocking, thus allowing the handling of further requests to be overlapped with the servicing of cache misses. The best performance comes when a nonblocking cache is used with a split-transaction bus, and multiple misses are overlapped inside the second-level memory system.

The subject of this thesis is to examine the design of a multiported nonblocking cache. The design of each subcomponent is presented and discussed. Trace driven simulations are performed to evaluate the cost effectiveness of certain design decisions.

This chapter describes a general model of an aggressive superscalar microprocessor and considers the needs of a cache for such a processor. The general design of a multiported nonblocking cache is presented. Chapter 2 discusses the design of each subcomponent in detail and presents some design issues. Chapter 3 presents the simulation environment and describes the features of the microprocessor and cache modeled in this

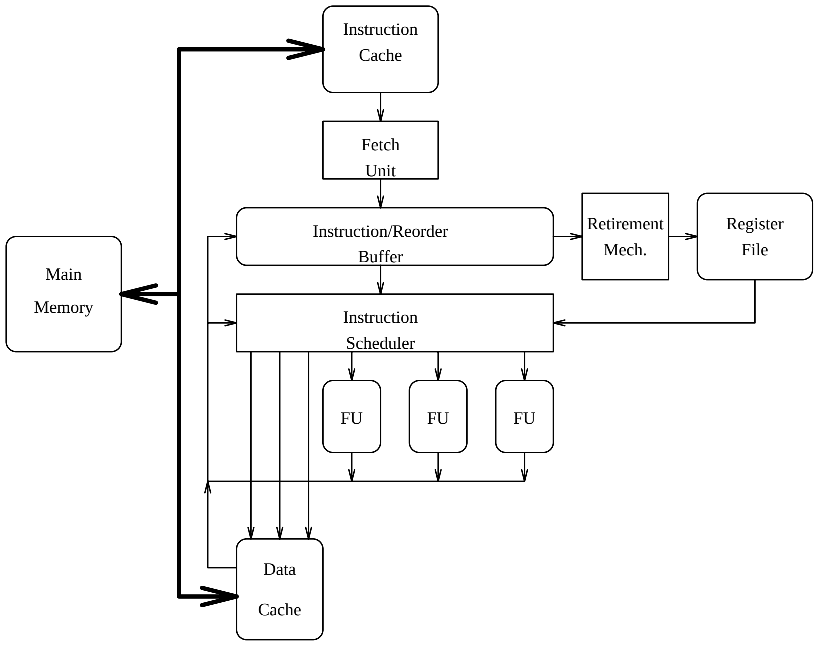


Figure 1.1 Superscalar microprocessor with multiple cache ports

study. Chapter 4 presents the simulation results and discusses how they apply to the design issues presented in Chapter 2. Finally, Chapter 5 presents concluding remarks.

1.1 Superscalar Microprocessors

In this section, the general model of a superscalar microprocessor is discussed. For this model, an aggressive processor design is chosen, employing such performance enhancing features as register renaming, out-of-order execution, and speculative execution. The reason for this is that a processor capable of exercising the memory to a high degree should be used so that the merits of the memory system design can be correctly evaluated.

Shown in Figure 1.1 is a diagram of a superscalar microprocessor based on [6]. Multiple instructions are fetched from the instruction cache into an instruction buffer. Next, the instruction buffer is examined by an instruction scheduler which attempts to issue instructions to the functional units. An instruction is issued to the proper functional unit when all of its operands have been evaluated and a functional unit of the proper type is available. Next, the functional unit performs the computation and returns the result to a reorder buffer where it awaits retirement into the register file. To provide for precise interrupts, an instruction is allowed to retire only if all other instructions ahead of it have been retired or are retiring in the current cycle. Bypass paths exist so that a computation's result may be used by a currently issuing instruction without waiting for it to be written to the reorder buffer. Since there is a one-to-one mapping between instructions and results, the reorder buffer and instruction buffer are incorporated into one unit as in [6] and [7].

Register renaming is performed by storing a result into space reserved in the instruction/reorder buffer; these spaces are associatively searched by incoming instructions for the latest value of the register. This removes any dependencies arising from register storage conflicts. Out-of-order execution is provided by allowing instructions in the rear of the buffer to issue despite unissued instructions ahead of them. Speculative execution occurs by allowing instructions to issue although an unresolved branch is ahead of them. This implies some type of branch prediction to tell what instructions to bring in to the machine.

1.2 Caches for Superscalar Microprocessors

This section introduces the demands that superscalar processors put on first level memory systems. Solutions are introduced which meet these demands and lay the groundwork for future discussions. The two main problems of concern are providing simultaneous cache access for multiple hits and reducing the performance degradation caused by sequentializing cache misses.

1.2.1 Multiported caches

The data cache in Figure 1.1 is connected to the processor by three independent paths, each capable of issuing a memory instruction during a given cycle. Assume that three memory instructions are issued each cycle and that they are all hits in the cache. Obviously, peak performance is obtained if all three of the memory requests are satisfied simultaneously each cycle.

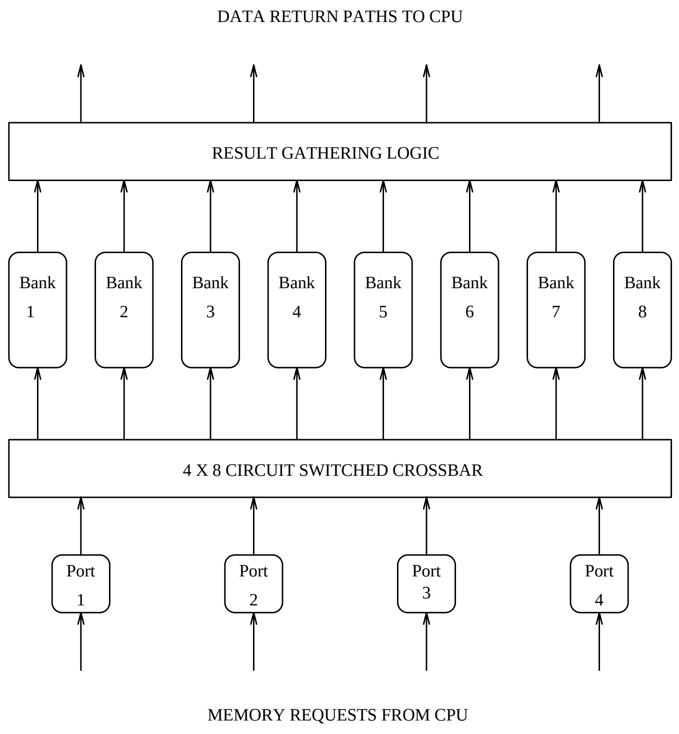


Figure 1.2 Providing multiple cache ports with interleaved cache banks

To allow for this feature of simultaneous access, a way of providing multiple access to the entire block of cache memory is needed. Sohi discusses ways this might be accomplished in [8]. Interleaved cache banks and duplication of cache memory are mentioned as possible solutions. Interleaving is the better solution since duplication is wasteful and allows for only one write at a time. Interleaving requires no duplication and allows multiple writes provided they are to different cache banks. When a bank conflict occurs among multiple accesses, only one of the accesses is allowed to access the bank and the others are stalled. The exact interleaving scheme is an important problem and will be discussed in detail later. Figure 1.2 shows how interleaving can be used to provide multiported cache access. Requests are placed on the cache ports by the cpu. A crossbar is then used to route the request to the proper cache bank. The access is performed at the cache bank and, in the case of a read, the results are gathered and returned to the cpu. The cpu considers a write request to be completed after it is placed at the cache port.

1.2.2 Blocking and nonblocking caches

The above discussion assumed that all cache accesses were cache hits. When a cache miss occurs, a request must be made to main memory to service the miss. This introduces the problem of what to do with cache accesses that occur while a miss is being serviced. In conventional systems, the processor is unable to issue a second memory instruction during the handling of a cache miss, although other computations may be initiated. This type of cache is referred to as a *blocking* cache. In a high-performance system, other alternatives should be considered during a cache miss. For instance, one alternative is to buffer future misses and allow the cache to continue to service hits. A better alternative is to allow multiple cache misses to be overlapped. This type of cache is referred to as a *nonblocking* cache. In [8] Sohi shows that nonblocking caches can be potentially instrumental in meeting the memory bandwidth needs of superscalar processors.

To see how a nonblocking cache can improve performance over a blocking cache, consider the following sequence of memory accesses: { miss to cache block 1, miss to cache block 2, hit to a cache block }. Assume that each access is a read and that the leftmost miss is the first issued. Assume that the memory latency is 5 cycles and that it takes one cycle to request the line and two cycles to receive the line from main memory. Also assume that only a single path is provided from the cpu to the cache and from the cache to the cpu. Thus, only one memory request can be made, and only one data item can be returned at a time. Also assume that cache hits can be handled in a single cycle.

The sequence of events for a blocking cache handling the described reference stream is shown in Figure 1.3. In this case, the cpu must wait for a memory request to be completely serviced before issuing a subsequent memory request. The only overlap occurs when the cpu issues a memory instruction because the cache has informed it that the cache will be free the next cycle. In this case, the total time for the three operations is 18 cycles. Notice that the final hit also completes execution on the 18th cycle.

The case for the nonblocking cache is shown in Figure 1.4. Unlike the blocking cache, the nonblocking cache is allowed to continue processing memory requests after an initial miss. Notice that the total sequence completes in 12 cycles and that the hit is serviced on the fourth cycle.

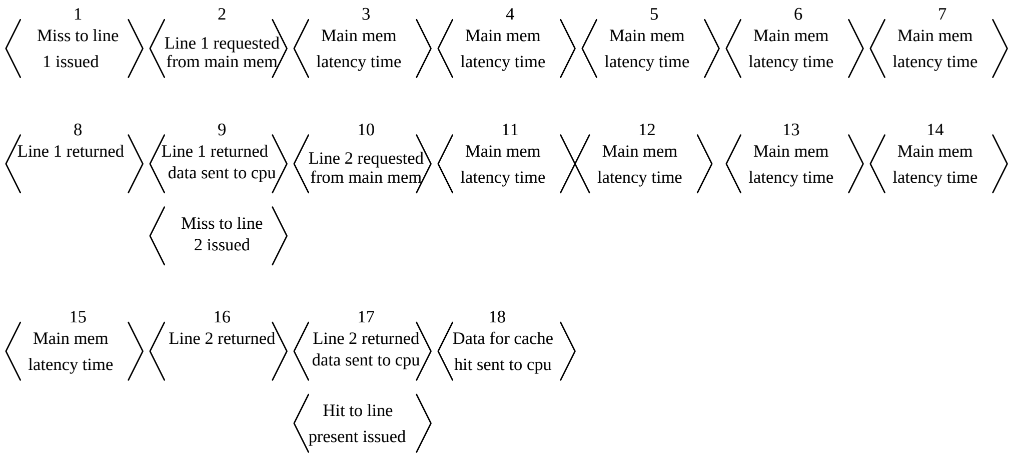


Figure 1.3 Performance of a blocking cache for a {miss1, miss2, hit} sequence

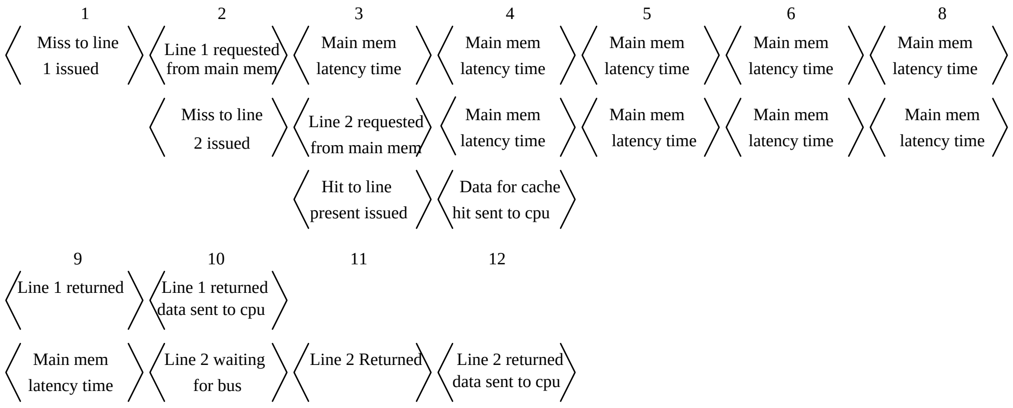


Figure 1.4 Performance of a nonblocking cache for a {miss1, miss2, hit} sequence

As can be seen from Figure 1.4, one of the main reasons for the speedup of the nonblocking cache is the ability to overlap main memory access. This occurs during cycles 4 through 8. It is assumed that the second-level memory system can handle multiple requests, and that the cache line read operation is split into two parts, the request of the line and the receiving of the line.

Another reason for the speedup of the nonblocking cache is the fact that future hits are not blocked during a cache miss. In this case, the hit was completed at cycle 4 rather than cycle 18 as in the previous case. It will be shown later that this type of speedup accounts for only a small amount of performance increase in nonblocking caches.

The hardware requirements for a nonblocking cache are now introduced. Kroft proposed a scheme in which special registers called MSHRs (miss-info, status-holding registers) are allocated for each missed cache line [5]. The MSHRs contain all of the information necessary to return the data to the proper cpu register when the cache line returns from memory, and to write the line to the proper cache buffer address, and to handle any further accesses to the cache line prior to the line returning from memory. The number of MSHRs sets an upper limit on the number of outstanding misses that can be overlapped at a single time. This thesis will later introduce a scheme different from Kroft's, while still using the name MSHR to describe the basic element of the data structure necessary for providing nonblocking cache access. The details of the scheme are not considered here; instead, a diagram is presented to show the general hardware requirements of a nonblocking cache.

Figure 1.5 shows a simple block diagram of a nonblocking cache. The MSHR queue is a queue, in which each entry is a single MSHR. Each valid MSHR represents a cache line that is currently being fetched from main memory. The MSHR queue is searched in parallel with the conventional cache buffer. A hit in the MSHR queue indicates a currently outstanding miss to the same cache line. In this case the MSHR corresponding to the correct line is modified to indicate the effects of the current memory request. This modification is described in detail later. If the line is absent from both the cache buffer and MSHR queue, an MSHR is allocated and a request is made for the line to be sent from memory. This new MSHR is now examined by future requests. There is also circuitry to request and receive cache lines from memory. Requests should be made in

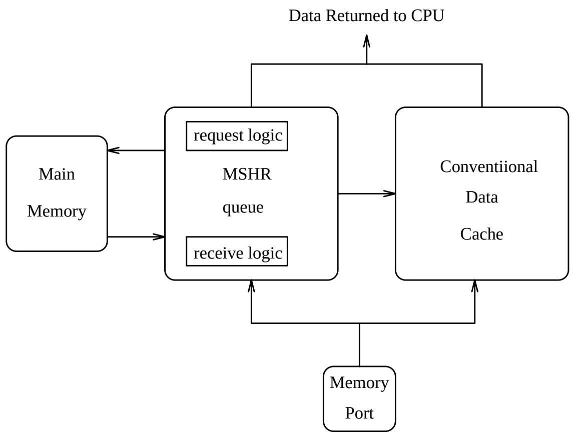


Figure 1.5 Block diagram of a nonblocking cache

the order in which the MSHRs were allocated. When a line is returned from memory, it is transferred to the cache buffer and the corresponding is MSHR cleared.

1.2.3 Multiported nonblocking caches

In this section, the multiported and nonblocking features are combined. Figure 1.6 shows a block diagram of a multiported nonblocking cache (MPNBC). The MPNBC shown has two ports and four interleaved cache banks. Note that the MSHR queue needs to be dual ported so that the two incoming requests can search the MSHR queue simultaneously. The tag store for the cache buffer also requires dual porting.

Also shown in Figure 1.6 is a replacement buffer. This is used by a write-back cache to hold lines before they are written back to main memory. Sometimes the data being requested may be present in the replacement buffer. In this case, it may be possible to forward data directly from the replacement buffer to the cpu. This thesis will examine the use of the replacement buffer as an auxiliary storage space. Of particular interest is the frequency of requests which hit in the replacement buffer and how that changes as the cache's associativity changes. The replacement buffer also requires dual porting.

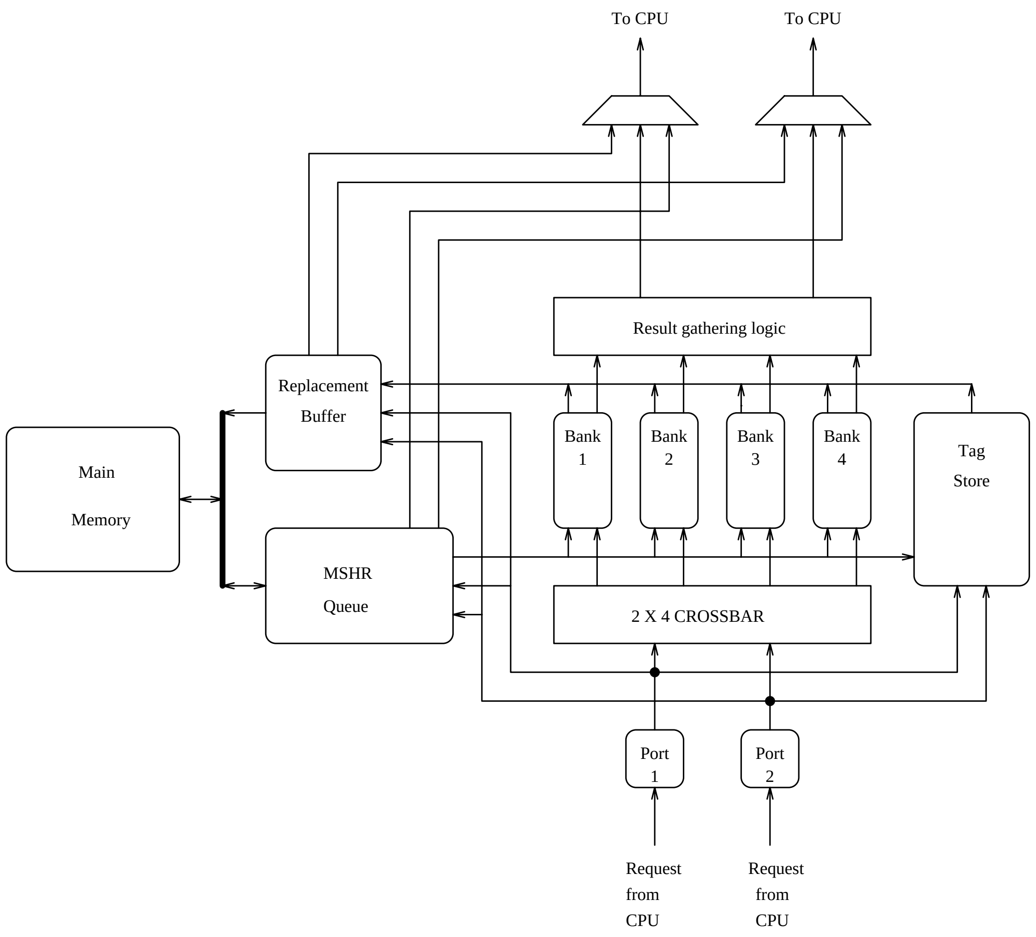


Figure 1.6 Block diagram of a multiported nonblocking cache with replacement buffer

CHAPTER 2

DESIGN ISSUES

In this chapter, the operation and requirements of the subcomponents of the multi-ported nonblocking cache (MPNBC) are examined in detail.

2.1 Cache Ports

The cache ports are a series of input latches to the MPNBC, each containing a memory instruction. When the cpu issues a memory instruction, it is sent to a cache port and latched in. Each cycle, the cache examines the cache ports and attempts to execute valid memory instructions. When a memory instruction is executed, the port is vacated and is ready to receive another request from the cpu.

It is assumed that the cpu sends the memory instructions to the ports in some type of implied order. The memory system does not have to execute the instructions in this implied order but the results must be the same as if they were. This implies that the cache ports themselves must have some type of implied ordering or priority with respect to the execution logic.

There are two ways in which priority can be given to the cache ports. The first way is to have some type of sequential logic indicate the priority of the ports. For example, in a system with two cache ports, a flip-flop could be used to indicate which port currently had priority. The execution logic would use this information to select which port to execute. For example, in the case of a conflict, the port with priority would be the one chosen for execution. After the first request is sent for execution, the flip-flop would then be updated to indicate that the other port now had priority.

Another method is to apply a fixed priority to the ports themselves. For instance in the case with two ports, port 1 would always have priority over port 2. If there were a conflict between the ports, the request at port 1 would be executed, and the contents of port 2 would be shifted to port 1 so that an incoming request would be placed on port 2 (and have lower priority).

Fixed priority ports require a shifting network to shift the unexecuted instructions to higher priority ports. However, the logic in the execution portion of the memory system is simpler because there is no decision to be made based on which port has priority to execute.

The data structure for a cache port is now considered. Each port must contain all of the information needed to completely specify a memory request. This information is described below:

1. A bit to indicate that the request it holds is valid.
2. A bit to indicate the type of operation, read or write.
3. A field to indicate the address to be written or read.
4. A field to indicate the data to be written in the case of a write.
5. A field to indicate the register or reorder buffer entry that the result should be returned to in the case of a read.

Note that fields 4 and 5 are exclusive of each other; therefore, the same data space can be used to accommodate both cases.

2.2 Cache Buffer

To support multiported access, the conventional cache buffer must have the ability to handle multiple hits at a time. The handling of multiple misses is not considered here, because additional buffer space and data structures are required as touched upon in the previous chapter. Multiple misses will be addressed in detail in the following section. In this section, the modifications to the cache buffer necessary to support multiple hits are examined.

2.2.1 Cache data storage

As mentioned in the previous chapter, the main modification to the cache buffer is that the cache data storage must be interleaved to support multiple access. Interleaving allows multiple words in the cache to be read or written simultaneously, provided they are in separate cache banks. There are two issues that are of concern when considering interleaving schemes.

The first issue is to avoid the performance degradation caused by bank conflicts. Conflicts depend on the interleaving style of the cache memory and the access pattern of the program. An interleaving scheme that minimizes conflicts as much as possible for a wide range of programs should be chosen, since it is expected that the MPNBC is to be used as part of a general-purpose superscalar processor.

The second issue of concern when choosing an interleaving scheme is the effect that the interleaving style has on the design of the cache buffer itself and how that influences performance. A scheme that places an entire cache line in a single cache bank will have a different cache buffer design from a scheme that spreads the line out across many banks.

Consider the situation in which a line is returning from main memory and needs to be written into the cache banks. In the case in which a complete line is contained in a single cache bank, the bank is stalled for some time to write the new line into the cache bank and remove the replaced line if there is one. The number of cycles required to complete this is determined by the width of the data paths into the cache bank and the cache line size. During this time, this single bank will be unable to handle current memory requests which will be stalled at the ports until the bank becomes available. Other cache banks are still free to handle requests.

In the case in which a cache line is spread across multiple banks, the returning line can be written to all cache banks in parallel. This will stall all cache banks simultaneously for a time shorter than the case described above for a single cache bank. Below, four types of interleaving schemes are described. These schemes are also discussed in [3].

The first type of interleaving introduced is low-order interleaving. In this type of interleaving, the lowest B bits of the word address are used to determine the cache bank where B is $\log_2$ of the number of cache banks. This has the effect of spreading the entire cache line across all banks. If the number of words in a line is greater than the number of

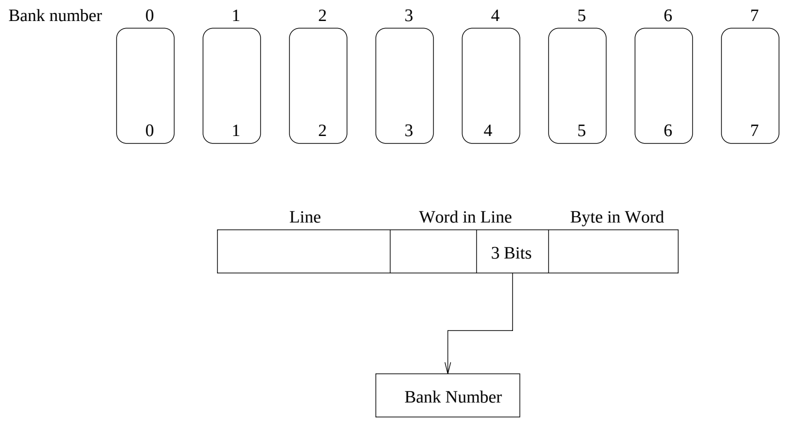


Figure 2.1 Location of block zero and address calculations for low-order interleaving scheme

cache banks, multiple words from a cache line appear in a bank. Low-order interleaving is appealing in the sense that sequential accesses reside in different cache banks. Locality suggests that a superscalar processor may, on occasion, request simultaneous access to adjacent memory locations. With this type of scheme, adjacent locations will never reside in the same cache bank. Figure 2.1 shows the word placement and bank number calculation for block zero using low-order interleaving assuming a cache line size of 8 words and 8 interleaved cache banks. A number i appearing inside a bank numbered j indicates that word i of block 0 with be located in bank j. This convention holds for the following examples as well.

The drawback of low-order interleaving is that relative locations in different cache lines occupy the same cache bank (assuming that the number of cache banks is less than or equal to the number of words in the cache line). To see how this can be hazardous, consider a sequence of instructions which computes the sum of two vectors A and B. Assume that the elements of A and B are arranged in sequential order, and that A[0] and B[0] are positioned at the same relative position in the cache line. In this case, the loads of A[i] and B[i] are independent and could be issued simultaneously. However, inside the memory system, A[i] and B[i] would reside in the same cache bank, and their execution must be sequentialized. The next type of interleaving will reduce this problem.

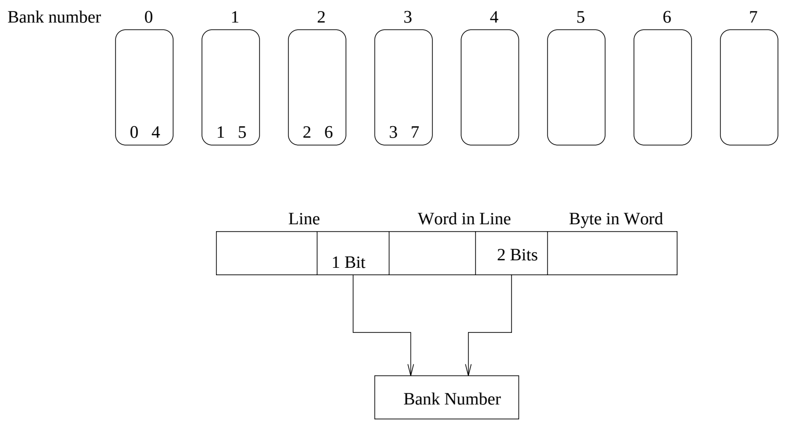


Figure 2.2 Location of block zero and address calculations for block-level interleaving scheme

The next type of interleaving scheme is block-level interleaving. In block-level interleaving, the first B bits of the cache line number are used to determine the bank number. This has the effect of placing an entire cache line in a separate bank. Adjacent cache lines are placed in adjacent memory banks. This type of interleaving seems to be well-suited for the above vector addition case, although there is still a small chance that A[i] and B[i] will map to the same cache bank. Since no two elements of the same cache line can be accessed concurrently, poor performance due to conflicts is expected in programs exhibiting great amounts of concurrent sequential access. Figure 2.2 shows the word placement and bank number calculation using block-level interleaving.

The last type of interleaving is split-N interleaving which combines the best features of the above two styles. In split-N interleaving, a cache line is basically low-order interleaved across N cache banks where N is less than the total number of cache banks. The bank number is determined by concatenating the lower ($\log_2$ N) bits of the word address with the (B - $\log_2$ N) bits of the cache line number. Figures 2.3 and 2.4 show split-2 and split-4, interleaving respectively.

Split-N interleaving is advantageous because it allows adjacent accesses to reside in different cache banks, thus reducing conflicts for sequential accesses. However, accesses in the same cache line may conflict. Split-N interleaving is also advantageous because

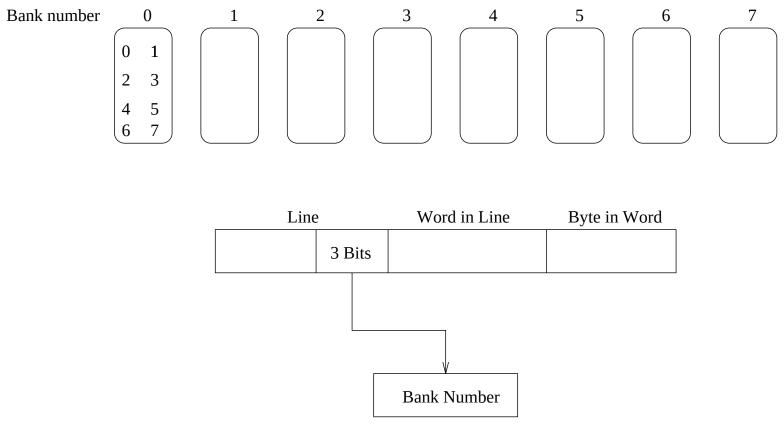


Figure 2.3 Location of block zero and address calculations for split-2 interleaving scheme

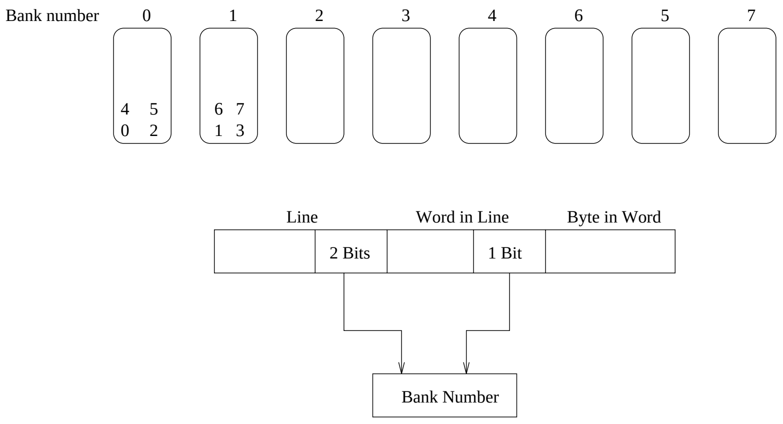


Figure 2.4 Location of block zero and address calculations for split-4 interleaving scheme

there is a chance that relative accesses in different cache lines will not map to the same cache bank.

Now, the repercussions each type of interleaving has on the overall system design are considered. Low-order interleaving, by scattering the words across all cache banks, causes the entire cache to stall at the time a line returns from memory. It will stall, however, only for the time necessary to write the line to the cache banks, which should be relatively short since all of the cache banks can be written in parallel. In this case, the stall logic can be made very simple because either the whole cache is stalled or not stalled. Recall, however, that current bus designs may require many cycles for the entire line to be transferred from memory, and it would be wasteful to stall all the cache banks for the entire transfer time. In this case, the words are buffered as they are returned from the memory system, then simultaneously written into the cache banks. An alternative to stalling the entire cache would be to have a scheme in which there are multiple valid bits per cache line. As the words are forwarded directly from the memory system to the cache bank, the individual valid bits are set to indicate the presence of the word in the cache.

Block-level interleaving will cause only a single bank to lock up at the time the line returns from memory. However, this bank will be locked up for a longer time than is the whole cache in the low-order interleaving case. In this case, it may be possible for the words to be forwarded to the cache directly as they arrive from main memory without buffering.

Split-N level interleaving will cause N cache banks to be locked up for the time necessary to write the cache line back. This time should be greater than for the low-order interleaving case, and less than for the block-level interleaving case. This case may or may not require buffering depending on the widths of the data paths involved. Again, a multiple valid bit scheme may be used instead of buffering the line.

2.2.2 Cache tag storage.

The tag store must be accessed once per cycle by each port to see if the requested items are present in the cache. Thus, the tag store needs to be multiported. Its needs, however, are very different from the needs of the cache data storage memory itself because the tag store memory is required to be read by many ports, but is written only once when a line is written to the cache buffer.

Interleaving the tag store memory as has been done for the cache data memory is not a good idea because each tag store entry corresponds to a single cache line. Simultaneous accesses to the same cache line would cause a conflict in the tag store memory (because they would map to the same tag store entry, hence the same tag store bank), and one of them would be stalled. Thus, an interleaved tag store is incompatible with low-order data interleaving because the benefits gained from low-order cache data interleaving would be not allowed due to the conflicts of the tag store entries. Block-level interleaving, however, goes very well with tag store interleaving because the same interleaving used for the tag store may be used for the cache data itself. Block-level interleaving, however, may impose a performance penalty in some programs due to the extra conflicts it incurs.

Another, better alternative for implementing the tag store memory is simply to add multiple read ports and keep one write port. Multiple read ports allow tag access from any port regardless of its address. Of course, the number of ports to the tag store will be equal to the number of cache ports. This type of tag store design will also provide for simpler stall logic since no stalls will be caused by conflicts in the tag store memory. If multiple read ports present implementation problems for the design of a portion of memory as large as the tag store, replication, though wasteful, will yield the same results. The tradeoff is that more space is required for the replicated tag store.

2.2.3 Dirty bits, valid bits, and LRU storage

The dirty bits indicate whether a cache line has been written and are used by a write-back cache to determine if the cache line should be written back to main memory. Assume that two writes to different cache lines are to be executed simultaneously. In this case, it must be ensured that the dirty bits for both cache lines are set or one of the requests is stalled. Because of the possible need to mark multiple cache lines as dirty at the same time, the solutions presented above for the tag store and cache data are inadequate for handling the dirty bits. A solution is to provide multiple write ports to the dirty bit storage. Again, the number of ports to the dirty bit storage should be equal to the number of input ports to the cache memory system. Also, in this situation, only one output port is needed because only one dirty bit is examined by the replacement circuitry.

The valid bits indicate that the block being accessed contains valid information in the cache. Valid bits are read during each memory access by the tag compare logic for each port to determine if a cache hit has occurred. They are also written when a new cache line is being stored. Thus, the valid bit requirements are basically the same as the tag store requirements and can be included in the tag store. When a line returns from main memory, the valid bits are also examined to compute which line is to be replaced. Note that some caching schemes may require multiple valid bits per line The requirements for this case are the same as those listed above.

The LRU information indicates which block in a set is to be replaced in the event of a conflict with a line arriving from memory. The type of storage required by the LRU information is dependent on the particular LRU scheme used. Consider a simple case for a two-way set associative cache design in which one bit is kept per set to indicate the least recently used cache block. Obviously, this memory will need to be N-write ported to update LRU information for as many as N sets at once. In this case, the LRU logic does not need the current LRU bit value to determine the new LRU value for a cache line being accessed; it is the just the negation of the index of the block that matched within the set. For this reason, the LRU information can be single-read ported.

Note that two of the structures described above require multiple write porting. These are the dirty bits and the LRU information. This is fortunate because these structures require only a relatively small portion of the overall cache area, and the overhead for multiple write porting is large for large amounts of memory.

2.3 Handling Misses

In the last section, the modifications which should be made to a conventional cache buffer in order to support multiported access were discussed. Thus far, little has been mentioned about the features necessary to support nonblocking cache access. In this section, a hardware data structure capable of providing nonblocking cache access is presented. Its operation is thoroughly described, and alternative designs are later presented for a few of the subcomponents. The overall design is shown in Figure 2.5. The design can be broken down into two main parts, the MSHR queue and the read queue. The MSHR queue contains the MSHRs which contain the information about which words have been written during the time the line has been requested from memory. In this thesis, MSHR and MSHR queue entry are used interchangeably. The MSHR queue must also contain the circuitry to request cache lines from memory and to handle their return. It must also contain the associative logic to identify when a memory request has hit in the MSHR queue.

The read queue handles requests to words in cache lines that are currently outstanding misses. However, the read queue handles only read requests to words which have not been written since the initial request that caused the cache miss was made. This means that the read queue handles requests that can be satisfied only by the data returning from memory. The read queue temporarily stores these requests until the line returns from memory, at which time the proper requests are serviced. When a cache line is returned from memory the read queue is searched for read requests that are waiting for data from the returning line. Each matching request is satisfied with the incoming data and the entry cleared.

2.3.1 MSHR queue

As stated above, the MSHR queue contains a set of MSHRs organized as a FIFO queue. Each MSHR contains buffer space to handle writes during the time the line is being accessed from memory. Each time an access occurs, the MSHR queue is associatively searched, looking for a valid MSHR entry whose line number matches the line number of the memory request. This type of match is referred to as a "hit" in the MSHRs. Each time a write hit in the MSHRs occurs, the word is written into proper data space in the matching MSHR, and the V bit corresponding to the written word is set. When a future read request hits in the MSHR queue, the V bit for the corresponding word in the matching MSHR is checked. A V bit for an individual word in an MSHR will be referred

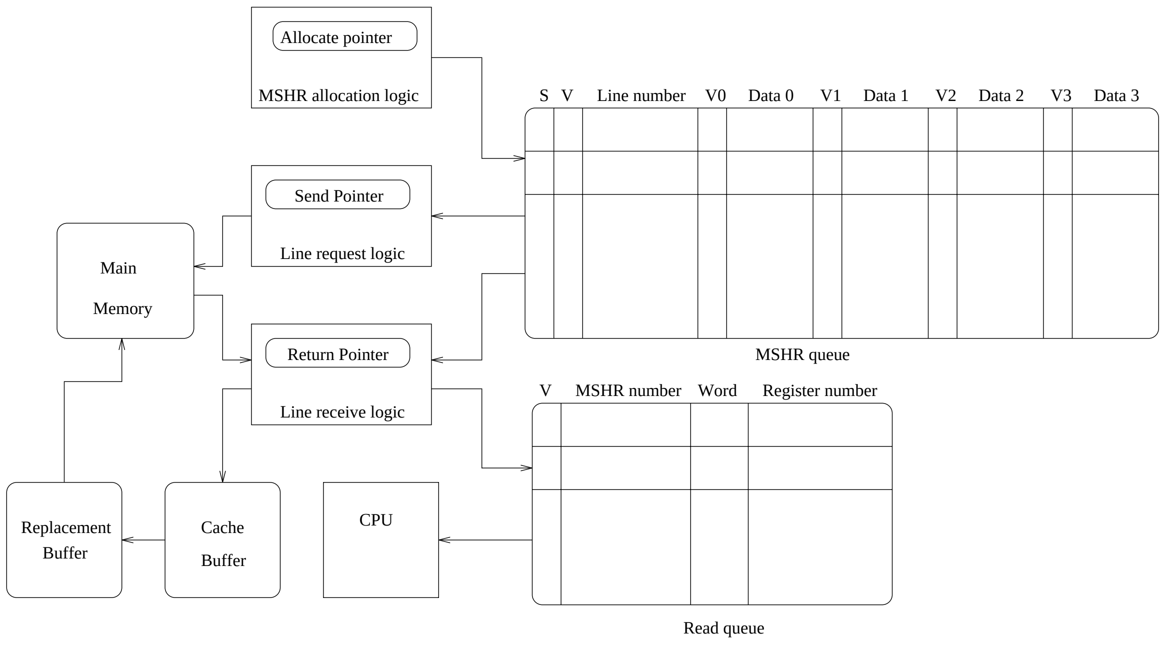


Figure 2.5 Block diagram of the data structures necessary to handle nonblocking cache access for a four-word cache line

to as Vi, as opposed to V which indicates the V bit for the MSHR itself. If the Vi bit is a one, the value in the data space is returned to the cpu. If the Vi bit is a zero, the read request is entered into the read queue and is satisfied when the line returns from memory. The action of the read queue will be discussed in the following subsection. The number of data word entries in an MSHR is equal to the number of words in the cache line. Figure 2.5 shows the MSHR queue for a 4-word cache line. The MSHR queue needs to be N ported to handle the memory requests from N cache ports. The total number of comparators needed is N * M where M is the number of MSHRs.

Note that the in-order return of cache lines is assumed here. In this case, the MSHR queue can be managed as a simple FIFO. Aside from the MSHR queue itself, there exists other logic consisting mainly of pointers which controls the allocation and removal of items from the MSHR queue. This associated logic will be discussed later. The data structure for the MSHR queue is summarized below. The design assumes a cache line size of four words.

1. V bit. This bit indicates that this specific MSHR entry contains valid information.

2. S bit. This bit indicates that this specific MSHR entry needs to request a line from memory.

3. Line number. This is the address of the line that is being requested from main memory.

4. Data 0 - Data 3. These entries are reserved as storage space to hold any writes to the cache line which occur before the line has been returned from memory.

5. V0 - V3. These are the valid bits that indicate that data has been written into the corresponding word.

2.3.2 Read queue

The read queue holds a list of entries that contains information about which words in a returning line are needed by the cpu. When a line returns from main memory, the read queue is associatively searched to find entries corresponding to the MSHR that originally requested the line. The read queue can not be organized as a FIFO the way it stands because there may be reads to one cache line interleaved with reads to another cache line. When a line is returned and all of its entries deleted, there is now a queue with blank spaces between entries. A simple counter cannot be used to point to the next free entry in this case. The data structure of the read queue is described below.

1. V bit. This bit indicates that the corresponding read queue entry is valid and should be serviced if the line corresponding to its current MSHR returns.

2. MSHR number. This field indicates which MSHR the corresponding read entry was from. This field allows it to identify the returning line that has its information.

3. Word. This field identifies exactly which word in the cache line is to be returned.

4. Register number. This is the register or instruction/reorder buffer entry to which the data is returned.

2.3.3 Additional logic

This section describes the other logic used to implement a nonblocking cache. This logic consists mainly of counters which act as pointers in the FIFO. The structures are described below.

1. MSHR allocation logic. When a request is found that requires the allocation of an MSHR, this logic will allocate an MSHR entry and store the pertinent information. It also will enter the initial request into the read queue in the case of a read, or write the appropriate entry in the allocated MSHR in the case of a write. The allocate pointer is a simple counter which is pointing to the current available MSHR entry. If the MSHRs are all being used, no more misses can be handled and a request that causes a miss is stalled at the ports.

2. Line request logic. This logic continuously monitors the MSHR queue for valid MSHR entries with the S bit set. When it finds one, it arbitrates for the bus and attempts to make a request from main memory for the cache line specified in the line number field of the MSHR entry. Again, since the requests are being made in FIFO order, a simple counter can be used to track the unsent entries.

3. Line return logic. This logic is continuously monitoring the bus for returning cache lines. When it recognizes a cache line returning from memory, it will load the line into the cache buffer while combining it with the data from previous writes to the corresponding MSHR. Again, since the results are guaranteed to return from memory in FIFO order, a simple counter points to the MSHR entry whose line will be returning next. The read queue is associatively searched with this counter, and all valid entries that match this number are returned to the cpu with the correct data from memory.

2.3.4 System operation and hazards

In this section, the complete system operation is described and scenarios are considered which may present hazards. It is explained how the system avoids the hazards to provide correct execution of the program.

Consider a memory access placed on a cache port. Simultaneously, the cache buffer, replacement buffer, and MSHR queue are all searched for the corresponding line. If it is a hit in the cache buffer, the data is returned to the cpu in the case of a read, or written to the cache buffer in the case of a write. There is no need to notify the cpu that the write has been completed. The case of a hit in the replacement buffer will be described later. If it is a miss in the cache buffer, replacement buffer, and MSHR queue, a new MSHR must be allocated. If more than one entry on the current cache ports requires an MSHR allocation, one of them must be stalled or the MSHR allocate logic must be made capable of allocating multiple MSHRs at a time. Stalling the latter request is the better idea, because individual parts of the memory request process can be sequentialized as long as the second-level memory accesses are overlapped. This is true as long as the main memory access time is much greater than the times of the sequential operations.

Once a request is assigned an MSHR, the line number of the requested item is copied into the MSHR pointed to by the allocate pointer and the V bit set. At the same time, a memory request for the cache line is attempted. If the memory request is successful, the corresponding S bit is set to 0; otherwise the S bit is set to 1. The memory request may be unsuccessful due to a busy bus.

If the access that caused the MSHR allocation is a read, an entry in the read queue is allocated. At this time, an available read queue entry needs to be chosen. Note that the read queue is not organized as a FIFO; therefore, more complicated logic is required to select the free entry. At this time, the value of the allocate pointer is copied into the MSHR number entry in the read queue, the corresponding V bit is set, and the word and register number values are copied from the appropriate cache port. Note that it may be impossible to allocate a read queue entry because the read queue may be full. On this occasion, the request is stalled at the cache port and an MSHR is not allocated, even though one may be available. Later a modification will be presented which alleviates the problem of stalling due to a lack of read queue entries even though an MSHR is available.

If the access that caused the MSHR allocation is a write, the corresponding data field is written with the new data and the Vi bit is set.

Consider future requests that hit in the MSHR queue. If they are writes, the data is written and Vi is bit set as described above. If they are reads, they will first examine the Vi bit corresponding to the accessed word. If the Vi is set, the desired word will be forwarded from the MSHR back to the CPU. If the Vi bit is not set, the request will be placed in the read queue and the data will be returned to the CPU at the time the cache line returns from memory.

When the cache line returns from memory, it must be written to the proper address in the cache buffer. The line number field in the MSHR entry specifies the exact index and tag for the cache buffer. If any words have been written into the data fields of the MSHR, they should be written to the cache instead of the stale data arriving from memory. This is achieved by letting the Vi bits select either the data from the MSHR (if Vi=1) or the data from the line returning from memory(Vi=0). This is shown in Figure 2.6. The dirty bit for the cache line is computed simply by taking the OR of all of the V bits. The return pointer identifies which MSHR entry is returning and is used to search the read queue associatively. Valid entries in the read queue with matching tags forward the incoming word (specified by the word field of the read queue entry) to the cpu. Of course, if the cache buffer requires that an entry be replaced, it must be moved into the replacement buffer if it is dirty. Operation of the replacement buffer is discussed in the next section.

To further describe this scheme, one memory hazard and its handling are presented. Consider a memory write followed by another memory write to the same address. Recall from the preceding section that these requests are placed onto the cache ports in an order corresponding to correct program execution Also recall that a cache bank conflict will occur because the requests access the same address. In the case of a conflict, the request with the higher priority will be executed first. There is no way in which the second access can proceed until the first has begun execution. In this way, conflicts actually help to provide correct program execution. The situation is similar for RAW and WAR hazards.

2.3.5 Alternate implementations and limitations

In this section, alternate implementations of two of the above components are considered. First, it is shown how multiple distributed read queues instead of a large complex one can simplify the associated logic and enhance performance. Next, it is shown what

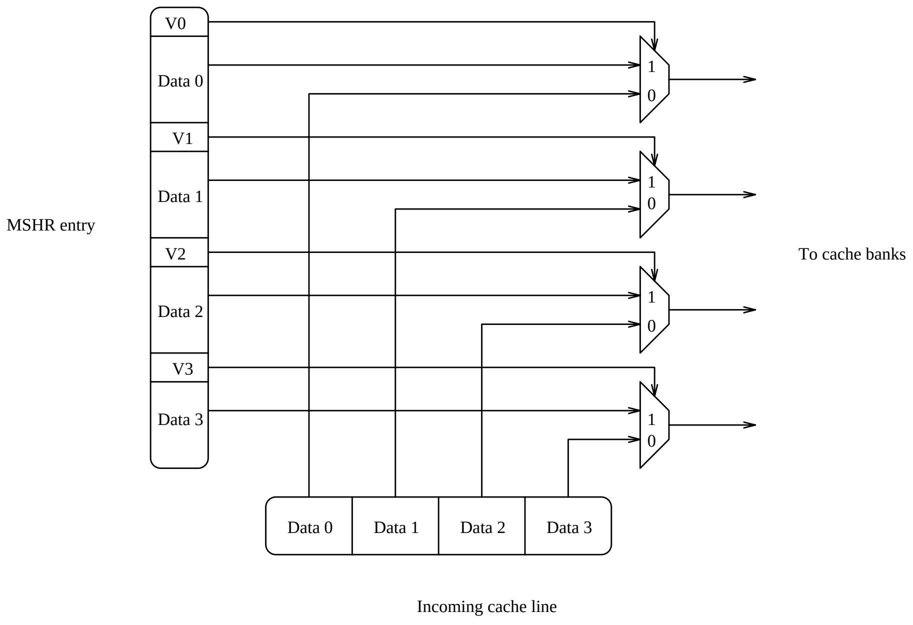


Figure 2.6 Logic which handles the return of lines to cache buffer

modifications to the MSHR system are needed to support cache lines returning out of order. Finally, the limitations of the design are discussed.

The current design has a single read queue which is associatively searched using the MSHR number of the returning line as a key. An alternative is is to subdivide the read queue such that there is one read queue per MSHR queue entry. In this case, when a line returns from memory, only the read queue associated with the MSHR indexed by the return pointer is considered. The associative logic in the read queues can be eliminated altogether since only the V bit of the entries within the proper read queue now has to be examined. The design of the individual read queues can be implemented as a simple counter-based FIFO queue. Each read queue is implemented with a counter that represents a pointer to the next available read queue entry to be written. Each time an entry is added, the counter is incremented. After the line returns from memory and the outstanding read requests are handled, the counter is cleared. Figure 2.7 shows a diagram of the distributed read queues for a system with two MSHRs. Note that the MSHR number field has been eliminated from the read queues' data structures.

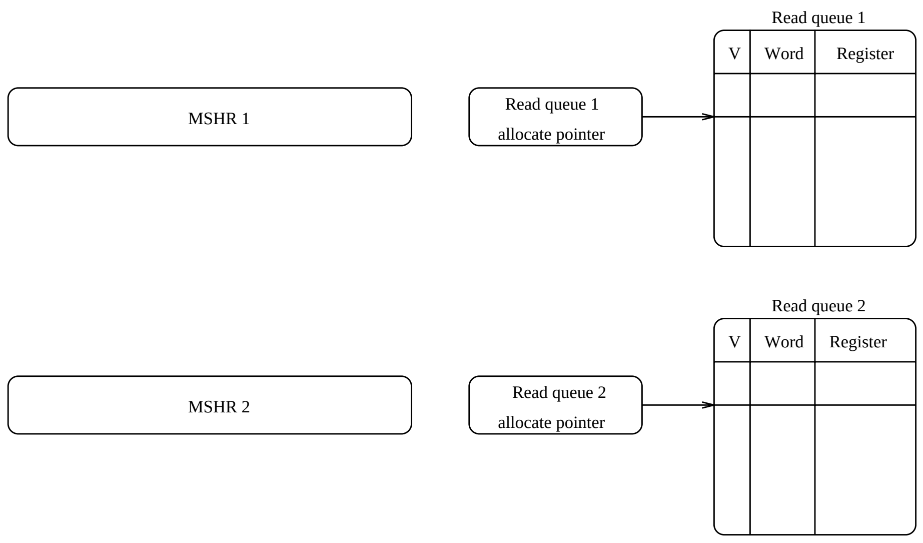


Figure 2.7 Distributed read queue organization for system with two MSHRs

In this alternate design, flexibility is traded for design simplicity. Consider the case of an eight entry read queue compared with four separate read queues of two entries each. Consider then a memory access sequence with four reads to the same cache line which occur before the line returns from main memory. Assume that the locations read are not written by any previous writes. In the distributed case, read requests to the line after the second access will be stalled because the read queue will be unable to accommodate any additional entries. These two requests will tie up two cache ports, thus limiting the number of concurrent requests which can now be handled. The nondistributed case will be able to continue accepting read requests to the same cache line. Note, however, that the distributed read queue entries are cheaper because they do not require the associative logic required by the centralized design. Because of this, the lack of flexibility may be compensated for by increasing the number of overall read queue entries in the distributed case. Note also, that there is always an available read queue whenever an MSHR can be allocated. There will never be a case in which an MSHR is available yet the request is unable to be made because of an unavailable read queue entry.

Thus far it has been assumed that the second-level memory system always returns cache lines in the order in which they were requested. Consider a case in which requests for 3 lines a, b, and c, are made to the memory system in that order. Assume that lines a and b conflict in the second-level memory system and can not be overlapped. In this case, the second-level memory system will have to stall the return of line c because it needs to wait for the access of line b to complete if it is to return the lines in order. Performance may be gained if the cache lines are allowed to return out of order.

To provide for out-of-order return of cache lines, the current design must be modified. Instead of a data return pointer which identifies the MSHR of the next line to return, a tagging scheme is required so that it is possible to link each returning line with the MSHR that requested it. There are two ways in which this can be done. One solution is to add a tag field in the MSHR storage which is returned along with the cache line from main memory. When a line returns, the MSHR queue is associatively searched with the key returning from memory to identify the requesting MSHR. A second way is to use the MSHR number as the tag so that the correct MSHR can be identified when the line returns without an associative search.

This tagging scheme further complicates the other logic. Note that MSHRs may be cleared out of order, leaving empty MSHRs interleaved between valid ones. This is similar to the case of the centralized read queue. When an MSHR is to be allocated, logic is required to look through all of the MSHRs and seek out an empty one. This replaces the allocate pointer described previously. Note that it may be possible to have a scheme that uses an allocate pointer and allocates MSHRs in order, though they may return out of order. The only drawback is that there will be situations in which the next MSHR to be allocated will be busy, while a free MSHR exists elsewhere in the queue. This scheme will be unable to use the avaible MSHR and will pay a performance penalty waiting for the proper MSHR to become available.

To issue line requests to main memory, a separate queue could be used to hold the line requests and issue them in order. This replaces the send pointer. Obviously, the return pointer is also omitted since the attached tag now identifies the returning line. It may be possible to implement the MSHR queue as a shiftable queue that is reorganized each time an entry is removed out of order.

Next, the limitations of the design are discussed. It has been assumed that the smallest addressable unit is a word. Some instruction sets allow operations to be performed on bytes. In this case, the design could be modified such that multiple Vi bits exist per word, indicating that the byte within the word has been written. Incoming requests could examine these bits to see if the word, or portions of the word had been previously written.

Also, it has been assumed thus far that the design has been for a write-back cache. In the case of a write-through cache, a word being written into an MSHR would also be written into the write buffer.

2.4 Replacement Buffer

In a conventional write-back cache, when dirty cache lines are removed from the cache due to a conflict with an incoming line, they are placed into the replacement buffer to be written back to main memory. After they are written, they are removed from the replacement buffer. There is a well known memory hazard that exists with a write-back cache. Consider a cache miss to a line that is present in the replacement buffer. If main memory is accessed before the line is written back, incorrect results will occur. Some solutions to this problem are presented below.

1. One low-performance solution is to flush the replacement buffer every time an MSHR is to be allocated. This is wasteful because time will be spent unnecessarily flushing the replacement buffer instead of first issuing the memory request which may be the critical path of some computation.

2. A better solution is to flush the replacement buffer only if the missed line is present in the replacement buffer. This requires an associative search of the replacement buffer. This method can be improved upon by flushing only the matching line.

3. An even higher performance solution is to forward the data directly from the replacement buffer into the cache. This solution requires data paths from the replacement buffer back to the cache. This idea is similar to a *victim cache* as described in [4].

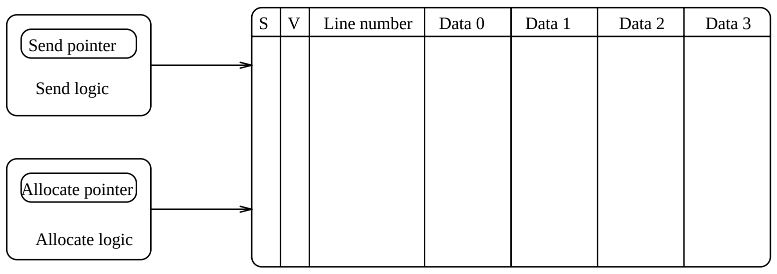


Figure 2.8 Replacement buffer organization for a 4-word cache line

2.4.1 System operation

A solution is presented here that requires no forwarding of results from the replacement buffer to the cache buffer and also allows valid cache lines to be present in the replacement buffer after they have been written. The data is accessed directly from the replacement buffer as long as it is still valid. It is not, however, transferred to the cache buffer after it has been accessed. This decision is based on the assumption that the replacement buffer can be made as fast as the cache. This makes good use of the replacement buffer space because it is now used to hold previously replaced lines which can be accessed by the cache ports. A diagram of the replacement buffer and its data structure is shown in Figure 2.8. A description of the fields in the data structure is provided below. A four-word cache line is assumed.

1. S bit. This bit indicates that the cache line has been written to main memory.
2. V bit. This bit indicates that this particular replacement buffer entry is valid and can supply data in the case of a cache read.
3. Line number. This field holds the line number of the line being sent back.
4. Data 0 - Data 3. These entries hold the data for the cache line.
5. Allocate pointer. This points to the next available replacement buffer entry. Each time a line is placed in the replacement buffer, the allocate pointer is incremented. If the allocate pointer points to an entry which has its S bit set, the buffer is full.

Table 2.1 Possible states of a line in the replacement buffer and the handling of hits

S	V	State	Read hit	Write hit
0	0	Invalid entry	-	-
0	1	Valid and sent	return data to cpu	invalidate and fetch from memory
1	0	can not occur	-	-
1	1	Valid but not sent	return data to cpu	write in new data

This indicates that the allocate pointer is pointing to an entry which has not yet been sent, and the other entries after it have not been sent either.

6. Send pointer. This points to the next entry to be sent. Each time an entry is sent back to main memory, the send pointer is incremented.

The replacement buffer is organized as a FIFO queue. As entries are added to the replacement buffer, the S and V bits are set. Entries are sent to memory in the same order in which they are allocated into the replacement buffer. After an entry is sent to the main memory, its S bit is cleared. Its V bit, however, remains set indicating that this is a valid entry that may provide data to the cpu. The entry remains valid until the V bit is cleared for a case described later, or until the entry is overwritten when the allocate pointer wraps around.

2.4.2 Hits in the replacement buffer

Each memory request placed on the cache ports associatively searches the replacement buffer as well as the MSHR queue and cache buffer. A memory request is a hit in the replacement buffer if its line number matches the line number field of a replacement buffer entry whose V bit is set. Table 2.1 illustrates what situations have occurred and how the hits are handled.

Initially, a line is placed into the replacement buffer in the entry pointed to by the allocate pointer, and the state (SV) is set to 11. If a read hit occurs to this entry, the data can simply be returned from the replacement buffer to the cpu. There is nothing to be gained by keeping the cache line at this time because it is assumed that the replacement buffer can service read requests at the same rate as the cache buffer. Eventually, the line will be overwritten in the replacement buffer, and further accesses will cause an MSHR allocation and the line to be brought in from memory and stored in the cache buffer. If a write hit to this entry occurs (SV=11), the data is written into the replacement buffer location, overwriting the current value. Since the line has not yet been sent, the memory will receive the most up-to-date data. Circuitry must be provided to allow the replacement buffer to be written by the cache ports.

After a replacement buffer entry is written to main memory, the S bit is cleared and the state is now 01. If a read hit occurs in this state, the data can still be returned to the cpu as above. If a write hit occurs at this point, there are two possible solutions. One option (not shown above) is to write the value into the replacement buffer and reset the S bit so that the line can be sent again. This seems like a reasonable solution to avoid the deallocation of a line. However, there are two reasons why this may not be a good solution in a real system. First, multiple writes to the same replacement buffer entry could cause the same line to be resent many times, driving up the bus traffic. Second, because entries in the middle of the replacement buffer could have their S bits reset, a FIFO design for the sending logic would not be permissible. In this case, a separate queue may be used to handle the sending of lines.

The second option is to invalidate the replacement buffer entry and cause an MSHR to be allocated and the line refetched from memory (i.e. set the state to SV = 00). The line will then eventually be reentered into the cache buffer. This method is compatible with the a FIFO design because even though the line is invalidated, incoming lines are still allocated in FIFO order.

2.5 Bus Arbitration and Priorities

For the purpose of this thesis, bus concerns are avoided, and the concentration is on the cache design. However, there are parts of the bus design that are directly relevant to the cache, and these are addressed here.

It has been assumed that three types of transactions can occur on the system bus: the cache requesting a line from memory, the memory sending a line to the cache, and the writing of a line to memory from the replacement buffer. The read transaction is split into two parts. The write transaction is not. Consequently, at any given time, three different things may be contending for the bus. If a particular bus operation is currently underway, all other competing operations must wait for the operation to complete, then arbitrate for the bus. At this time, the arbitration logic must decide which operation is allowed use of the bus.

Since misses are usually on the critical paths of programs, they should be given priority over the writing back of lines. It is therefore clear that read requests and read replies should be given priority over write-back requests. It is not clear exactly whether miss replies should be given priority over read requests. If the read reply time is much longer than the read request time, the read request should be given priority so that the effective memory latency is only slightly increased.

One nontrivial point is what to do when the replacement buffer fills up with unserviced replaceable lines (i.e., the allocate pointer points to a replacement buffer entry with its S bit sent). At this time, if any more replies from main memory are received, an entry may be added to an already full replacement buffer. A solution to this problem is to have the replacement buffer provide a signal indicating to the arbitration logic that it is full. When this line is active, the arbitration logic would unconditionally give priority to the replacement buffer.

Note that in the design of an MPNBC, the replacement buffer and the line request logic will probably be present on the same chip. It is therefore possible that the arbitration between them takes place internally and only one request is sent to the external arbitration logic. Obviously, some signal is necessary to tell the arbitration logic the exact type of request the MPNBC is making. This external arbitration logic must therefore select between the request from the MPNBC and the main memory.

2.6 Example Using a Real Design

The design of a real MPNBC may not be able to implement all of the features as elegantly as described in the previous sections. For instance, a real system design may allow only one MSHR and read queue entry to be allocated per cycle. This section presents an example of a short memory request sequence being executed on an MPNBC partially designed and simulated using Mentor Graphics schematic capture software. Due to time constraints, the design was not fully completed.

Shown in Figure 2.9 is the root design of a system with an MPNBC. Shown are the cpu, memory, MPNBC, and arbitration logic. A complete cpu was not designed. Instead, the cpu was built as a counter which indexed a lookup table (ROM). By programming the lookup table, any sequence of memory instructions could be sent to the MPNBC.

The second-level memory was essentially a normal memory with a simple FIFO added to the output. This simulates the effects of overlapping requests in a conflict-free second-level memory system. Three cycles after the requests come in, the memory system attempts to arbitrate for the bus to send the line back. The arbitration logic shown in the figure gives priority to the line coming back from main memory instead of the requests being made.

Details of the MPNBC are listed below:

- The system has 8 bit words and 20 bit addresses. A word is the only addressable unit in this system. The MPNBC designed here is a write-back cache.
- The MPNBC was implemented with 2 fixed priority cache ports and 4 interleaved cache banks. There are 2 return paths to the cpu. This is similar to the organization shown in Figure 1.6.
- The MPNBC contains 2 MSHRs each having its own read queue of 2 entries.

Figures 2.10 and 2.11 show a simulation of a sequence of memory requests being handled by the MPNBC. The signals and busses shown in the figures are described below.

port1(37:0) and port2(37:0) These are the internal signals directly from the latches themselves. They are displayed because they show exactly what memory instructions are currently being processed. Bits 19 through 0 give the address of the request. Bits 27 through 20 give the word to be written in the case of a write. Bits

Figure 2.9 Schematic diagram of MPNBC root design

Figure 2.10 Simulation of MPNBC for a short memory request sequence (sheet1)

Figure 2.11 Simulation of MPNBC for a short memory request sequence (sheet2)

35 through 28 give the tag to be returned to the cpu in the case of a read. Bit 36 specifies either a read or write (write = 1). Bit 37 indicates that the request is a valid request (valid = 1).

reply1(16:0) and reply2(16:0) These are the busses that return data to the cpu in the case of read requests. Bits 7 through 0 are the data itself. Bits 15 through 8 indicate the tag. Bit 16 indicates that the reply is valid (valid = 1).

data_bus(31:0) and address_bus(19:0) The data bus can accommodate one-half of a cache line. It therefore takes two cycles to complete a transaction involving the sending or receiving of a cache line. The address bus holds the address of the line being requested from memory. The address bus does not have to indicate the address of the line returning from memory because the requests are returning in order.

op(1:0) This indicates the bus transaction that is occurring. The bus is 0 for a line request and 2 to indicate the return of a cache line. A value of 1 indicates the writing back of a line (not shown here).

bus_request and bus_grant The bus_request signal is signal generated by the MPNBC to indicate that it needs the bus for a line request or write-back. The bus_grant line indicates that the MPNBC has been awarded use of the bus.

br_mem and bg_mem The br_mem signal is generated by the memory when it is ready to return a line to the MPNBC. The bg_mem line indicates that the memory has been awarded use of the bus.

A cycle-by-cycle description is given for a sequence of memory operations handled by the MPNBC. Generic, delayless components were used in the design. The example below is not intended to show an exhaustive test of the MPNBC as proof of design correctness; rather it is used to illustrate some of the MPNBC concepts described in this chapter.

Time 0-10 During this time, the system is initialized. The MSHRs and the read queues are cleared.

Time 10-30 Two requests are latched into the memory ports, a write followed by a read to the same address. Only the first access is allowed to proceed. In this cycle, an MSHR is allocated, and the request is made to memory.

Time 30-50 Next, it is seen that the request originally in port 2 has been shifted into port 1 and now has priority over an incoming request. The request in port 1 hits in a previously written word in the MSHRs and returns the value to the cpu. This can be verified by observing the reply1 bus. Concurrently, a miss to a new line is encountered in port 2. This miss causes a second MSHR to be allocated, and because it is a read miss, an entry is made into the read queue of MSHR 2 while a second line is requested from memory.

Time 50-70 Next, two more requests are latched in. They are both read misses to MSHR 1. Because only one read queue entry can be written per cycle, only the first is removed from the ports.

Time 70-90 The previous port 2 has been shifted to port 1 and an entry is made into the read queue for the request. Port 2 also contains a read request which needs to allocate a read queue entry for MSHR 2. Again, it must be stalled because the read queues can be written only once per cycle.

Time 90-110 The previous port 2 request has been shifted to port 1. The cpu is not making any requests at this time, thus an invalid request is latched into port 2. At this time, requests on the ports are not handled because the MPNBC is stalled due to a line returning from memory. In this cycle, the first portion of the returned line is written into the cache banks. At this time, the read queue entries from MSHR 1 supply the values from the returning cache line to the cpu. This can be verified from the reply busses.

Time 110-130 During this time, the second half of the cache line is transferred from the main memory to the cpu while the request(s) at the memory ports remain stalled.

Time 130-150 The first portion of the line corresponding to MSHR 2 is returned. The read queue also returns the correct value to satisfy the initial miss request. The cache remains stalled.

Time 150-170 Here the second portion of the line corresponding to MSHR 2 is returned while the cache remains stalled.

Time 170-190 Now the read request to the line originally requested by MSHR 2 is serviced and is a cache hit. The value is returned to the cpu.

Time 190-210 At this time two new requests are latched in. Both requests hit the cache and both items are returned to the cpu.

CHAPTER 3

SIMULATION METHODOLOGY

A simulation model was developed to analyze some of the tradeoffs involved in the design of a multi-ported non-blocking cache as described in the previous chapter. This chapter describes the simulation methodology and gives a detailed description of the processor and cache designs simulated.

3.1 Simulation Methodology

The simulation process is diagrammed in Figure 3.1. A SPARC executable was traced by a ptrace based tracing program producing a trace file containing dependency and address information for 5 million instructions. This procedure was performed for the eight benchmarks compress, espresso, fpppp, matrix300, sort, tbl, tomcatv, and yacc. Because of its short running time, a trace of approximately 1 million instructions was produced for sort. System executables were used for compress, tbl, sort, and yacc. Espresso, fpppp, matrix300, and tomcatv were compiled with the Sun compiler with all optimizations turned on.

Next, the trace file was read by an architecture simulator that completely modeled a processor, bus, and cache on a cycle-by-cycle basis. The architecture simulator also read in an architectural description which specified exact parameters for the various components. The output from the architecture simulator was a detailed file containing execution and memory access statistics. The architecture simulator was composed of two modules, a processor simulator and a cache simulator. The models of each are described below.

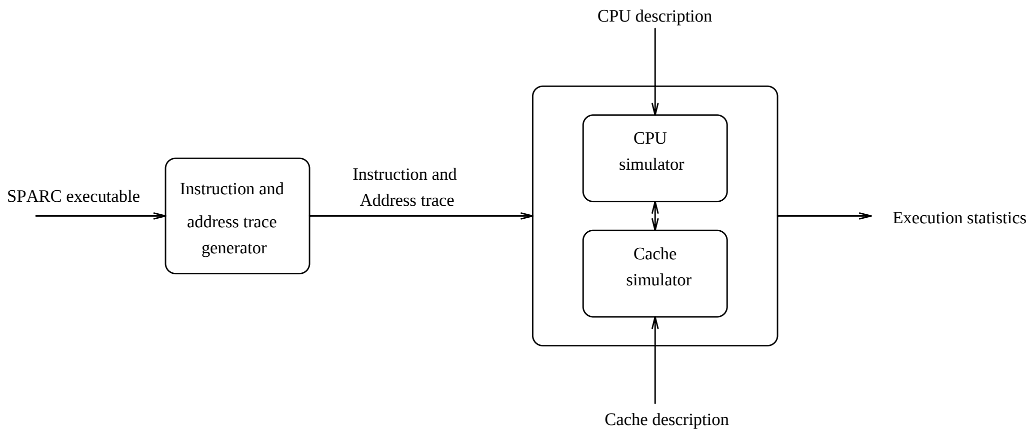


Figure 3.1 Block diagram of the simulation process

Table 3.1 Functional unit classes, multiplicities, and latencies for the simulated cpu

Class	Number	Latency
Integer	4	1
Load	variable	2
Store	variable	1
Branch	2	1
FP Add	1	3
FP Mult.	1	3
FP Div.	1	8
FP Conversion	1	8

3.1.1 Processor model

The processor simulated in this thesis was based on the SPARC instruction set and corresponds to the general model of an aggressive superscalar processor presented in the introduction. The instructions were divided into nine instruction classes. The latencies of each and the number of functional units of each type are shown in Table 3.1. The functional units are fully pipelined meaning that they can accept a new computation every cycle.

The processor is equipped with an instruction buffer whose size can be varied as part of the experiments. Instructions are read from the trace file and placed in the instruction buffer until instruction fetch is blocked. Instruction fetch may be blocked for the following reasons:

1. Full buffer. If there is no more room in the instruction buffer, further instructions cannot be loaded in.

2. Discontinuity in instruction stream. This indicates some type of control transfer in the instruction stream and the processor must wait until the next cycle to begin fetching from the new location.

3. Fetch limit. Only a maximum of 8 instructions can be brought in per cycle. This models a finite data path between the instruction cache and the cpu. A 100% instruction cache hit rate is assumed.

4. Mispredicted branch. An infinitely sized BTB is simulated here. If an incorrectly predicted branch is encountered, instructions after it are not brought into the machine until the branch is resolved.

Next, the instruction buffer is examined by an instruction scheduler. This scheduler will issue instructions all of whose operands have been written back to the reorder buffer, or are being written back this cycle if a functional unit of the desired type is free. This models a reorder buffer with bypass.

There is a special case for memory instructions. Memory instructions must stall if there is an earlier memory instruction that has not computed its address, or an ealier memory instruction that operates on the same address that has not issued and is not issuing this cycle. This constraint does not apply if both memory instructions are reads. No forwarding between instructions that operate on the same address is assumed. Memory instructions are allowed to issue out of order. Address computations are made in zero time at the issue stage. Double loads and double stores are split into two separate load or store instructions by the trace generator.

After instructions are issued, they enter a functional unit of the desired type, and the results appear on the output latches of the functional unit after a time specified by the latency of the functional unit. Memory instructions are given special treatment and are sent to a cache simulator which is responsible for simulating the completion of the memory instruction.

Next, as results return to the instruction/reorder buffer, they are saved and accessed by future instructions. Retirement occurs in order, which frees up the instruction/reorder buffer entries to be used for instructions currently being fetched.

3.1.2 Cache model

In this section, the model of the cache is described. Deviations from the model presented here will be described in the next section as they occur.

Memory requests are initially placed on the cache ports by the cpu according to the order in which they are issued. If the ports are all busy, the cpu is unable to issue futher memory instructions.

The memory system examines the memory requests starting from the memory port with the highest priority. If the instruction is not in conflict with any instructions currently being issued, it examines the MSHR queue, the replacement buffer, and the cache tag store to see if it is a hit in any of them. For all of the experiments performed here except those involving interleaving, the cache banks are low-order interleaved and divided into 32 cache banks. If there is a hit in the cache buffer, the result is returned to the cpu 2 cycles later. The memory instruction is also removed from the port to simulate the effects of pipelining the cache access for hits.

Hits in the MSHR queue are treated as specified in the previous chapter. The simulated design assumes a centralized read queue of infinite size based on the premise that read queue entries are much cheaper than MSHRs. There is also no contraint on the number of MSHR operations that are allowed to be performed in a single cycle. For instance, assuming that the ports contained four complete misses to four different cache lines and that there were four available MSHRS, four MSHRs would be allocated in a single cycle.

Read hits in the replacement buffer (4 entries) return the data directly to the cpu. Write hits to an unsent line cause the data to be overwritten, while write hits to a sent line cause the line to be resent. If the line that is hit is currently being sent, the request is stalled until the transaction completes. The entire cache is also stalled for two cycles when a line is returning from memory.

A cycle-by-cycle modeling of the system bus is performed. Cache line requests take 1 cycle. The return of a cache line from the memory system and the writeback of a line to the memory system are assumed to lock the bus for two cycles. Returning lines are given priority over requesting lines in this simulation. Conflicts in the second-level memory system were not simulated.

CHAPTER 4

SIMULATION RESULTS

In this chapter, the simulation results are presented. As a measure of performance, IPC (instructions per cycle) is used instead of simply reporting hit rates. This is because IPC shows how design changes influence overall program completion time, rather than the performance of a single component.

The first step is to characterize the benchmarks in terms of their parallelism, locality, and instruction class division. This information will help explain later results and relative differences among benchmarks.

Next, simulation results and discussions are presented for a variety of experiments. Issues such as the interleaving scheme, cache size versus memory latency, number of MSHRs versus number of cache ports, number of MSHRs versus cache size, and bus traffic are explored. The effectiveness of the replacement buffer scheme described here under varying associativity is also examined.

Attention is given to the breakdown of the memory references and what sections of the cache they access. This information is particularly relevant to the implementation of a MPNBC. Sections of the MPNBC that receive heavy usage may represent a bottleneck and require an aggressive implementation. For less frequently accessed sections, a less aggressive design may be acceptable to cut down the system cost.

4.1 Characterizing the Benchmarks

In this section, benchmarks are characterized based on their instruction-level parallelism, memory reference locality, and instruction class division.

An architecture with no memory system constraints is used to achieve an upper limit on performance and an idea of the instruction-level parallelism of each benchmark. Each

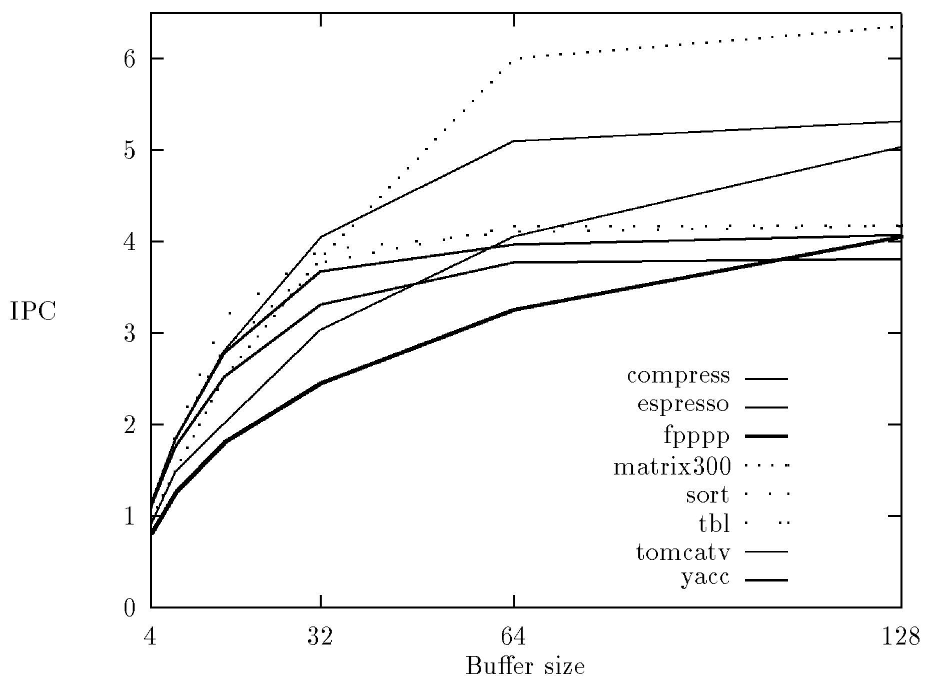


Figure 4.1 Performance with a perfect memory system

benchmark is simulated with a perfect memory system. The perfect memory system has an infinite number of cache ports and every access is a cache hit. Figure 4.1 shows the performance of the previously described processor with varying buffer sizes.

The dynamically scheduled machine is able to exploit significant amounts of instruction-level parallelism. The relative results are approximately the same as those presented in [2] for the benchmarks common to both.

It also can be seen from Figure 4.1 that compress and matrix300 have large amounts of instruction level parallelism. In the case of matrix300, a large instruction buffer is needed to exploit this parallelism due to the in-order retirement restriction imposed here. Many instructions are waiting in the buffer but cannot retire because a long latency arithmetic instruction ahead of them has yet to finish execution. The integer programs sort, espresso, tbl, and yacc all have similar, moderate amounts of instruction-level parallelism. The benchmark fpppp has the worst results of all of the benchmarks. Most of the benchmarks

Table 4.1 Hit rates and memory instruction frequencies for a 32K cache and 10-cycle memory latency. The instructions not accounted for are nonmemory instructions.

		Memory instruction breakdown by class			
benchmark	hit rate	load	store	double load	double store
compress	.881	.18	.10	.00	.00
espresso	.990	.18	.05	.00	.00
fpppp	.978	.36	.10	.05	.02
matrix300	.547	.29	.24	.00	.00
sort	.944	.19	.05	.00	.00
tbl	.988	.24	.14	.00	.00
tomcatv	.848	.28	.12	.07	.01
yacc	.988	.15	.04	.00	.00

typically level off around 3 or 4 IPC for large buffer sizes. Values between 2 or 3 are common for realistic buffer sizes.

Aside from instruction-level parallelism, memory reference locality and memory usage are important issues in the study of a memory system's effect on a program's performance. For instance, on a real system a program with excellent instruction-level parallelism (e.g., compress) may execute more slowly than a program with poorer instruction-level parallelism but superior cache performance (e.g., espresso). Table 4.1 shows the hit rate and instruction class frequencies for each benchmark for a 32K 2-way set-associative nonblocking cache with a main memory latency of 10 cycles. The Figure shown for the hit rate is the ratio of all memory references that hit in the cache buffer to the number of memory instructions issued. Because of nonblocking cache access, values shown here are lower than those obtained with a blocking cache. Recall that some requests may hit in the replacement buffer or MSHRs. This value is still a good indicator of program locality.

It can be seen that in general the floating-point benchmarks have more demanding memory requirements than the integer benchmarks. Matrix300 is the most demanding of the benchmarks; over 50% of its instructions are memory instructions. The floating-point benchmarks contain 28% to 36% loads and 10% to 24% stores. The integer benchmarks contain 15% to 24% loads and 5% to 14% stores. The benchmark tbl has the highest memory usage of the integer benchmarks with 24% loads and 14% stores.

Most of the integer benchmarks have reasonable cache performance. Espresso, tbl, and yacc have hit rates approaching unity for a 32K cache. Compress performs relatively poorly. In the case of the floating point benchmarks, only fpppp performs well while tomcatv and matrix300 perform poorly.

4.2 Memory Interleaving

In this section, the performance of various interleaving schemes is analyzed. Here, a perfect memory system is assumed except for the performance degradation caused by conflicts in the interleaved cache banks. If two or more memory requests are placed on the cache ports and wish to access the same cache bank, the higher priority one is serviced first, and the remaining ones are tried the next cycle. The simulation performed here does not consider the additional factors related to an interleaving scheme (e.g., block write-back time) as discussed in Section 2.2.1. IPC_p is used to represent the system performance without conflicts, while IPC_c is used to represent the system performance with conflicts. Results are shown as the ratio of IPC_c to IPC_p so that the relative amount of performance degradation can be observed. The four interleaving schemes simulated were low-order interleaving, split-2 interleaving, split-4 interleaving, and block-level interleaving. Each benchmark was simulated with each scheme for 4, 8, 32 and 128 cache banks. An instruction buffer size of 32 instructions and a block size of 32 bytes was assumed. The results are shown in Figures 4.2 and 4.3.

The performance degradation due to conflicts varied between the floating-point and integer programs. With 32 cache banks, the floating point programs suffered a performance decrease of 6% to 8% for all types of interleaving except block-level interleaving. The block-level interleaving scheme degraded performance by 10% to 12%.

The integer programs were more immune to the effects of block-level interleaving, but were more susceptible to the effects of bank conflicts in general. The performance degradation varied among the integer programs. Compress suffered a 15% performance degradation while espresso, sort, tbl, and yacc varied from 10% to 14%. Overall block-

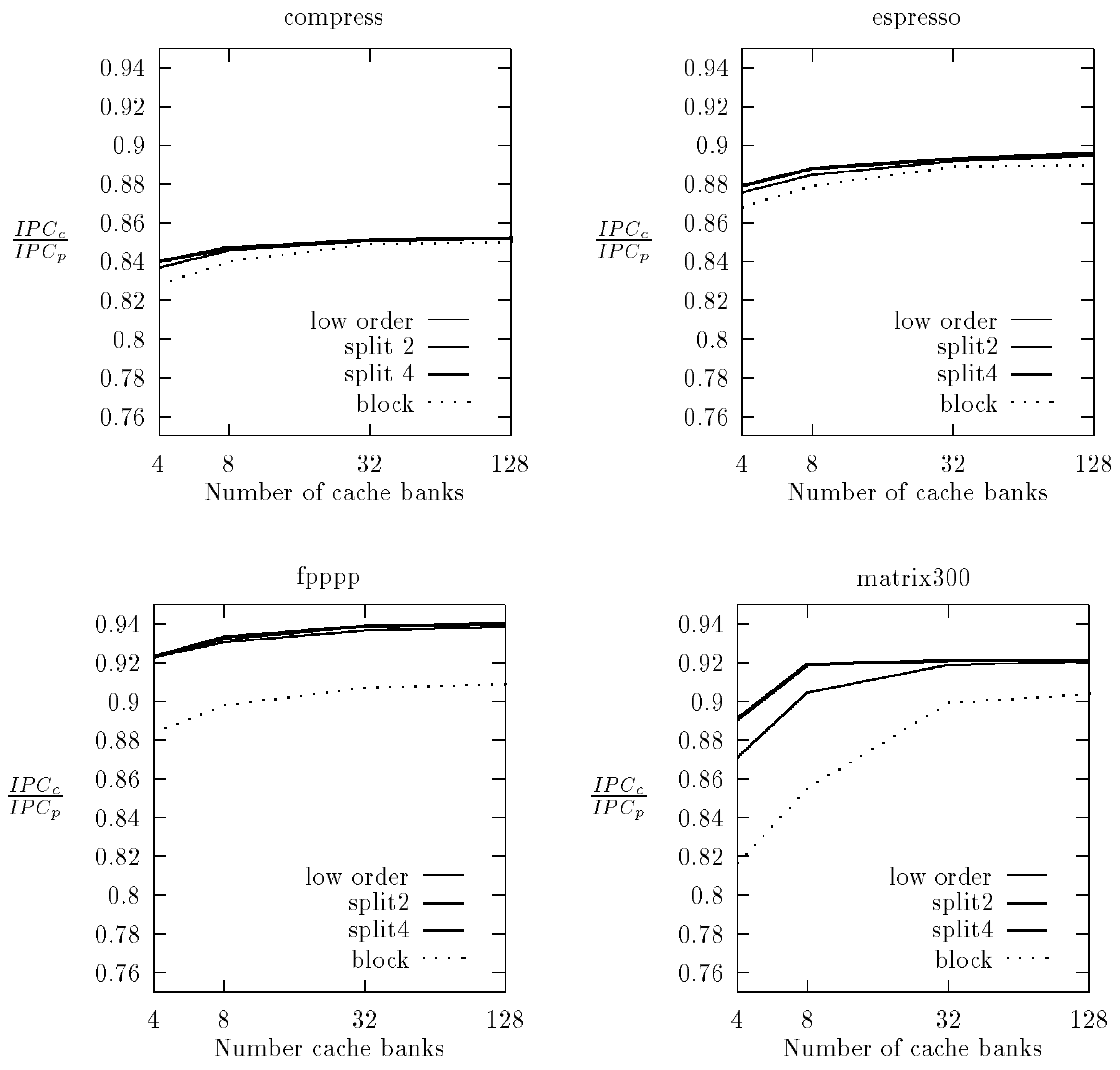


Figure 4.2 Performance degradation with various interleaving schemes for compress, espresso, fpppp, and matrix300

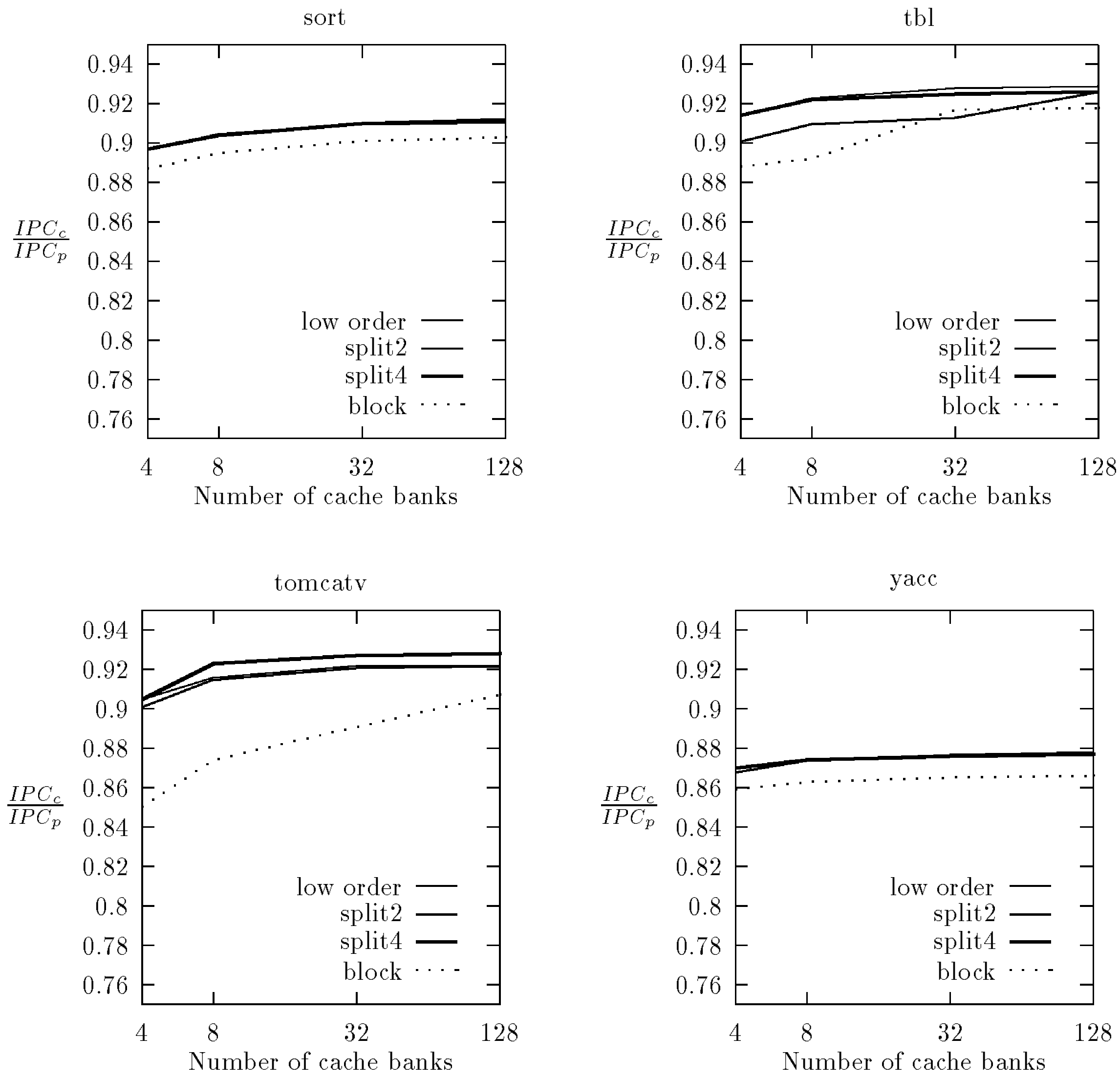


Figure 4.3 Performance degradation with various interleaving schemes for sort, tbl, tomcatv, and yacc

level interleaving was usually about 1% slower than the other types of interleaving for the integer programs.

It is interesting to note that although the floating-point programs access memory more frequently than the integer programs, the performance degradation due to conflicts is less severe as long as block-level interleaving is not used. This may be explained by the fact that many floating-point programs are dominated by sequential accesses, rather than access to random program variables. Sequential accesses are guaranteed not to collide in the cache banks as long as block-level interleaving is avoided. Random variables may or may not collide in the cache. Another possible explanation is that critical paths in floating-point programs are the long execution delays of the arithmetic instructions. There is plenty of "spare time" to do a second memory operation in the case of a conflict. Integer programs have shorter instruction latencies, thus getting the result back quickly from the memory is more crucial, especially if the memory instruction is on a critical path.

Low-order, split-2 and split-4 interleaving all show similar performance. Split-4 does seem to perform the best in the benchmarks shown here, but only by fractions of a percentage in most cases. Notice also that increasing the number of cache banks above 32 does not seem to help performance much in most cases.

4.3 Cache Size Versus Memory Latency

In this section a real cache is introduced. The cache is two-way set-associative with a block size of 32 bytes. In this experiment, the memory latency and cache size are varied. Memory latencies of 5, 10, 20, and 30 cycles and cache sizes of 4K, 8K, 32K, and 128K are used. Four MSHRs are assumed as well as low-order interleaving with 32 cache banks. Results are shown in Figures 4.4 and 4.5. This experiment is similar to that performed in [1] and the figures are similar for the benchmarks in common.

It can be seen that for small cache sizes, the memory latency plays a large role in determining the overall performance, because the miss rate is higher and many of the requests are forced to wait for results from main memory. As the cache size increases, the hit rate increases and the dependence on the memory latency is less. The benchmarks

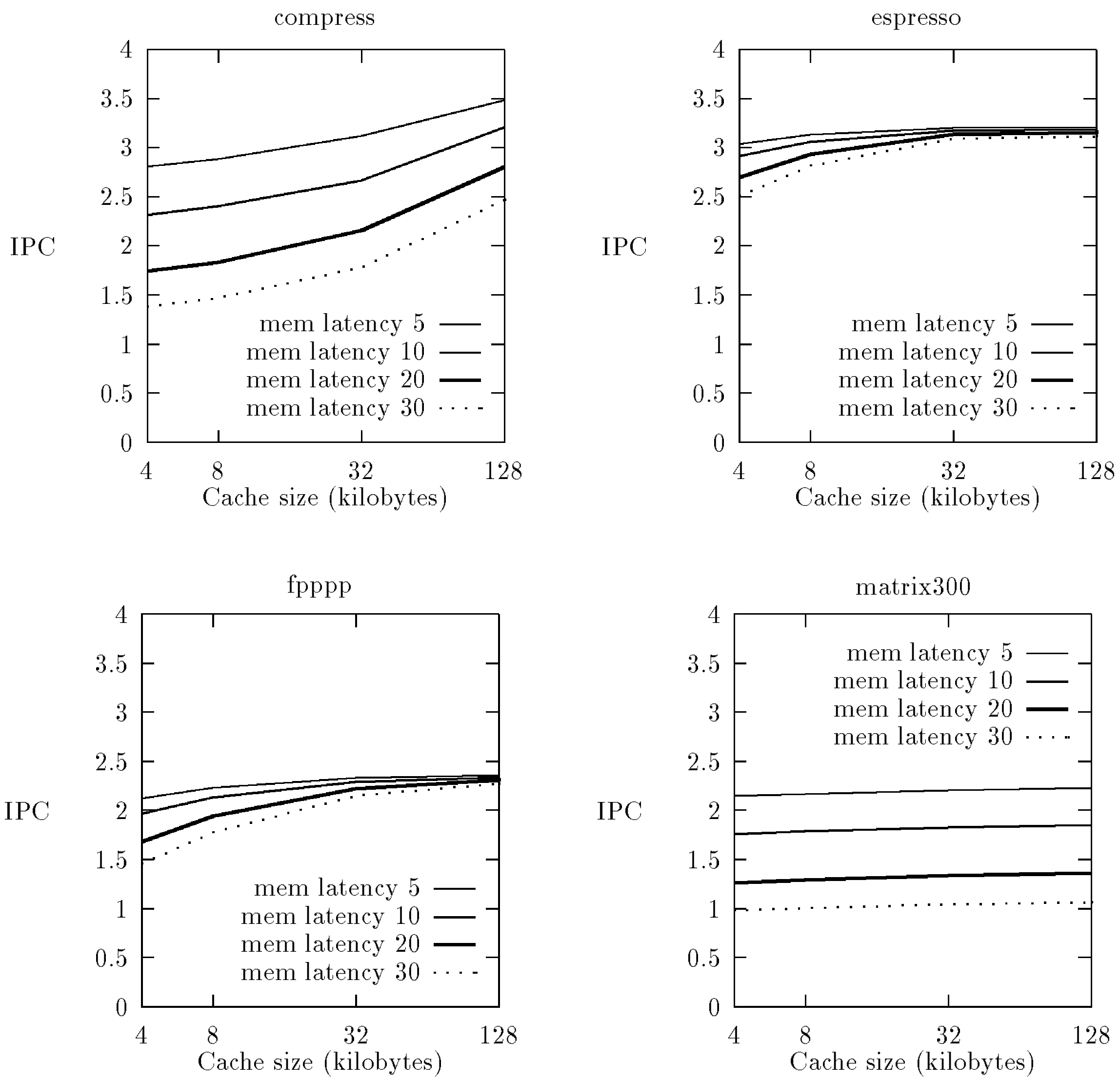


Figure 4.4 Performance with varying cache size and memory latency for compress, espresso, fpppp, and matrix300

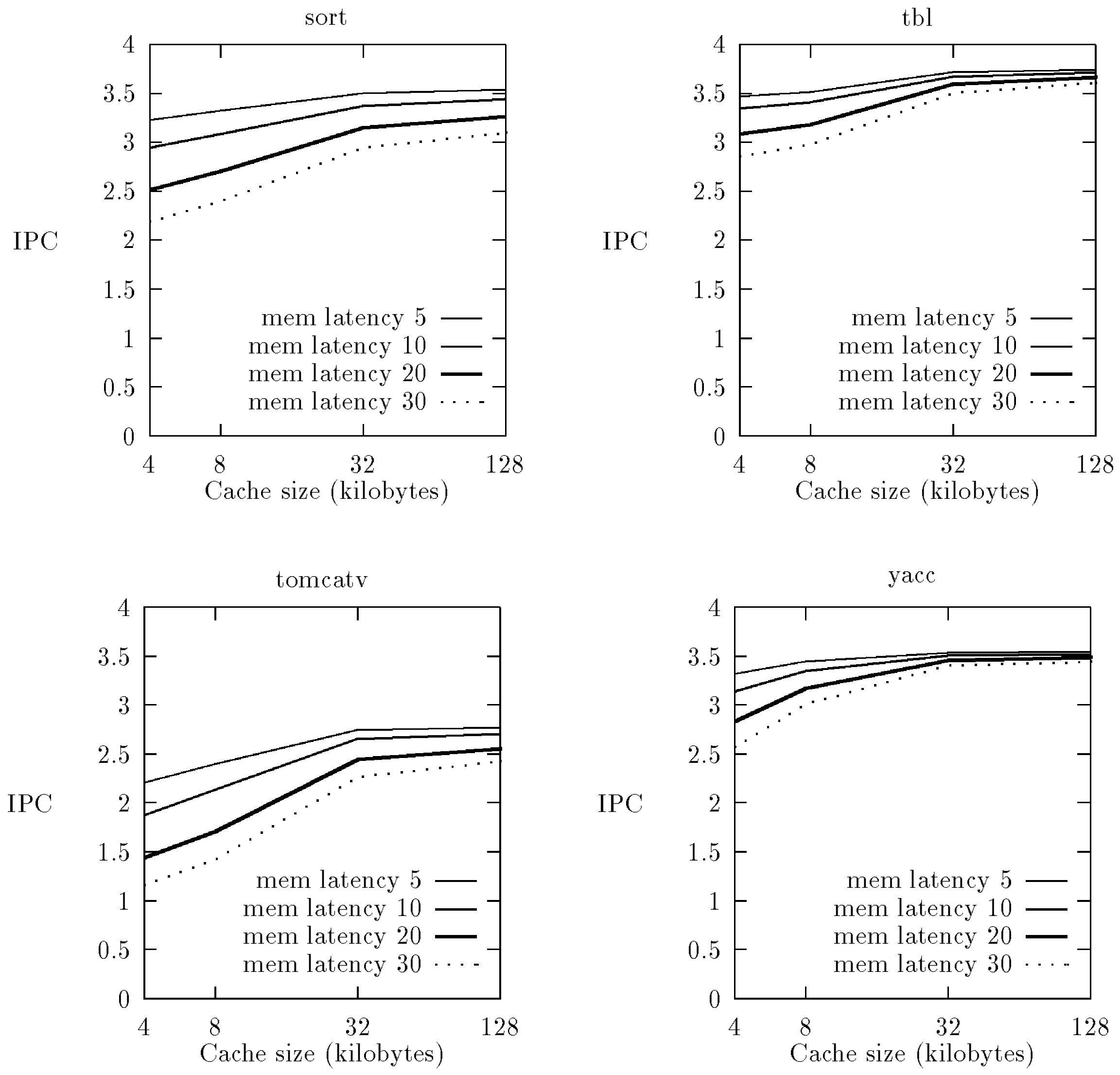


Figure 4.5 Performance with varying cache size and memory latency for sort, tbl, tomcatv, and yacc

with very high hit rates such as fpppp, espresso, and yacc show relatively less dependence on main memory, even for small cache sizes.

On the other side of the spectrum are programs with very poor cache performance that show little performance gain even as the cache size is increased. Both compress and matrix300 are highly dependent on the main memory access time even for large cache sizes. Matrix300 shows no increase in performance as the cache size is increased. This may be because it constantly requires new data from memory and the cached entries are not reused. Compress shows great dependence on memory latency for a large cache size. However, compress' performance increases as the cache size increases, and the relative fluctuation due to memory latency decreases. Compress is therefore able to fit more and more of its working set into the cache as the cache size increases.

4.4 MPNBC Usage

The purpose of this section is to identify the frequency with which parts of the MPNBC are used. As previously stated, this may be helpful in seeing which parts should be aggressively designed and which parts may be less aggressively designed in favor of decreasing system cost.

The first time an access is placed on a memory port, one of five possibilities can occur.

1. It may hit in the cache itself.
2. It may hit in the MSHR queue.
3. It may hit in the replacement buffer.
4. It may cause a new MSHR to be allocated.
5. It may stall due to a conflict or an unavailable MSHR.

Table 4.2 shows the percentages of memory accesses falling into each of these categories for the 8 benchmarks. This breakdown corresponds to the experiments presented in the previous section where the cache size is 32K and the memory latency is 10 cycles. This table represents what happens to an access the first time it is tried. For instance, an

Table 4.2 First-time access results for a cache size of 32K and memory latency of 10 cycles

benchmark	CB hit	MSHR hit	RB hit	allocate MSHR	stall
compress	.818	.0265	.00062	.0639	.091
espresso	.946	.0045	.00212	.0033	.044
fpppp	.908	.0063	.00948	.0050	.071
matrix300	.535	.3322	.00696	.0411	.084
sort	.827	.0361	.00135	.0151	.120
tbl	.882	.0050	.00038	.0036	.109
tomcatv	.763	.0422	.07222	.0235	.099
yacc	.856	.0042	.00235	.0039	.133

access that stalls first due to a conflict and later hits in the cache, is counted as a stall by this scheme.

The first observation is that the percentage of requests that hit in the cache on the first try is somewhat smaller than the overall hit rate given. This is due to conflicts in the cache banks. It will later be seen that conflicts are of major importance in integer programs with high hit rates.

Consider the occasion that permits an MSHR hit. Obviously, an initial miss must have occurred to originally allocate the MSHR. Next, an independent access to the same cache line must occur. Thus, to achieve a high number of MSHR hits, a program requires parallelism among memory accesses and poor locality. Compress and tomcatv have 2.6% and 4.2% of total memory accesses hitting in the MSHRs, respectively. Sort also has a relatively high amount of MSHR hits (3.6%). For other programs with better hit rates such as espresso, fpppp, and tbl, MSHR hits occur less than 1% of the time over the 5 million instructions simulated. It can be seen that matrix300 has a large amount of MSHR hits (33%). Again, this is due to the very poor hit rate and high instruction-level parallelism.

Table 4.3 shows the breakdown of MSHR hits. An MSHR hit can be a write, a read to a word that was previously written, or a read to a word not previously written. It can be seen that reads of previously written information occur vary rarely. This makes sense

Table 4.3 Breakdown of MSHR hits and stalls

	MSHR hit			stall	
benchmark	write	read written	read unwritten	conflicts	MSHR unavailable
compress	.0250	.00003	.0015	.0785	.0124
espresso	.0025	.00005	.0019	.0440	.0000
fpppp	.0027	.00014	.0034	.0709	.0000
matrix300	.1241	.00000	.2081	.0396	.0446
sort	.0232	.00017	.0127	.1202	.0000
tbl	.0019	.00006	.0029	.1080	.0011
tomcatv	.0368	.00004	.0053	.0991	.0001
yacc	.0016	.00013	.0024	.1332	.0000

because a compiler is expected to remove many redundant reads to an address that was just written. This fact presents an interesting tradeoff. It may be possible to have an MSHR queue that does not contain data paths to the cpu nor the associated logic for handling hits to previously written words in an MSHR. When MSHR read hits to written words are encountered, they can be stalled at the cache ports and serviced by the cache when the line returns from memory.

Reads to unwritten MSHRs account for less than 1% of all memory accesses in most of the benchmarks shown. Again, this is because dependencies may not allow future access to a cache line until the initial miss is returned. Matrix300 is an exception to this with roughly 20% of all accesses being read hits in the MSHRs.

The amount of write hits in the MSHRs varies from program to program. Matrix300 is as high as 12% MSHR write hits while compress, sort, and tomcatv contain 2% to 3%. Programs with good hit rates like fpppp, yacc, and espresso have less than 1% MSHR write hits.

Hits in the replacement buffer are rarer than hits in the MSHR queue in most cases. Tomcatv, however, shows an exception to this, and approximately 7% of all of its accesses hit in the replacement buffer. This may indicate that tomcatv suffers from an abundance of cache lines conflicting for the same cache storage space. All other programs have less than 1% of their accesses hit in the replacement buffer. A breakdown of replacement buffer accesses will be described later when replacement buffer performance with decreasing cache associativity is analyzed.

Allocation of an MSHR occurs during access to a cache line not present in the MSHRs, cache buffer, or replacement buffer as long as there is no conflicting instruction of higher priority beginning execution simultaneously. Obviously programs with high hit rates are going to have little MSHR allocation since the working set is contained in the cache. For the programs with high hit rates, MSHRs are allocated for less than 1% of the requests. For compress, matrix300, and tomcatv, MSHRs are allocated for 2% to 7% of the requests. Notice that although compress has a higher hit rate than matrix300, it allocates MSHRs more frequently. This is because matrix300 has many of its requests hitting in the MSHRs, which are not counted as hits.

Next, the case of stalling is examined. Requests are stalled when there is a bank conflict with a request of higher priority or when they need to allocate an MSHR and there is none available. Note that this stalling may be unneccessary in some cases. For instance, two memory instructions may conflict for the same cache bank, but one is a cache hit and the other is an MSHR hit. It may be possible to design a system which recognizes this case and allows both to proceed. However, this may not be possible if the cache access and MSHR tag search are to be done in parallel. It can be seen that most first-time accesses are stalled from 4% to 13% of the time.

Table 4.3 shows a breakdown of the stalls. In most cases, stalls are caused by conflicts rather than unavailable MSHRs. Stalling in programs with high hit rates is due mainly to conflicts and seldom to unavailable MSHRs. In fact, stalls due to conflicts are the major causes of performance degradation in integer programs with high hit rates. In programs with poor hit rates and high parallelism, conflicts are still a problem, but unavailable MSHRs also come into play. Compress must stall 1% of the requests due to unavailable MSHRs and matrix300 4%.

4.5 MSHR Usage

There is particular concern with the number of MSHRs used because each MSHR greatly influences system cost. Recall that in this model, each MSHR represents an

Table 4.4 Average MSHR usage for 4 MSHRs over 5 million instructions

	MSHR usage				
benchmark	0	1	2	3	4
compress	.561	.300	.114	.017	.007
espresso	.973	.024	.002	.000	.000
fpppp	.941	.044	.012	.002	.000
matrix300	.486	.402	.025	.030	.056
sort	.864	.125	.011	.000	.000
tbl	.959	.028	.005	.002	.006
tomcatv	.684	.206	.108	.001	.001
yacc	.970	.028	.002	.000	.000

additional memory request that can be overlapped in the main memory. Therefore, the phrase "adding an extra MSHR" implies increased complexity in many parts of the memory system, particularly the main memory. Note that an MPNBC may be designed with multiple MSHRs, each contending for a main memory system that supports only a single outstanding request, but this is not considered here.

In this section, MSHR usage is examined both as an average over the 5 million instructions executed, and dynamically, looking at how the instantaneous MSHR usage changes with time. Table 4.4 shows the frequency of usage for varying numbers of MSHRs. For instance, Table 4.4 shows that compress has 2 valid MSHRs 11.4% of the time.

All four MSHRs are rarely used except by matrix300. For programs with high hit rates, 1 MSHR is used less than 5% of the time, and 2 or more MSHRs are used less than 1% of the time. For programs with poor hit rates, MSHR usage is higher. Compress, matrix300, and tomcatv use one MSHR 20% to 40% of the time. Two or more MSHRs are used 10% to 15% percent of the time. Note that these values are highly dependent on cache size and memory latency which are fixed in this experiment at 32K and 10 cycles respectively. The results obtained for MSHR usage may vary greatly if different cache sizes and memory latencies are assumed.

Next, the dynamic usage of MSHRs is considered. Figures 4.6 and 4.7 show the dynamic usage of the MSHRs. Each point shown is the average number of rctive MSHRs per

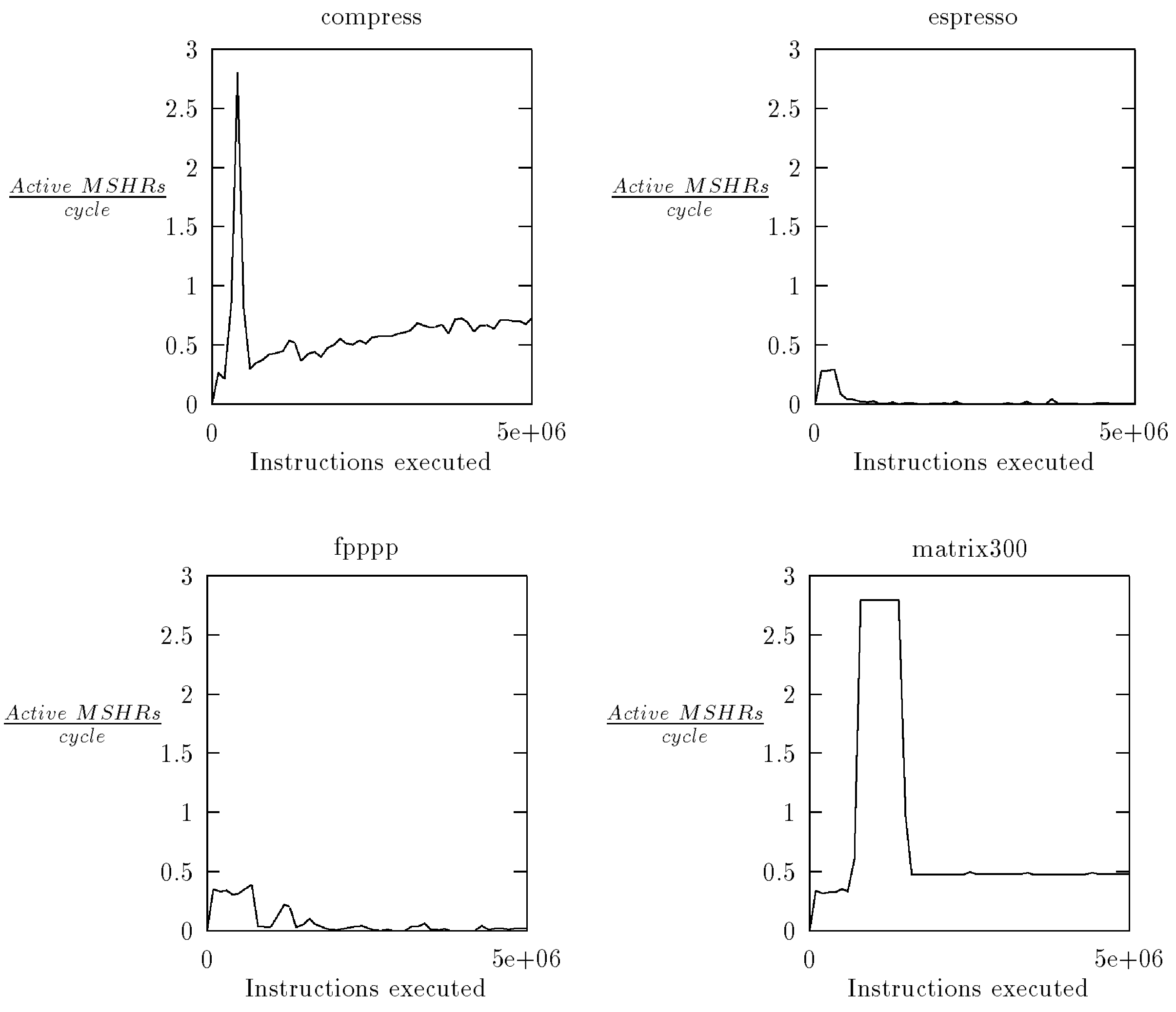


Figure 4.6 Timewise MSHR usage for compress, espresso, fpppp, and matrix300

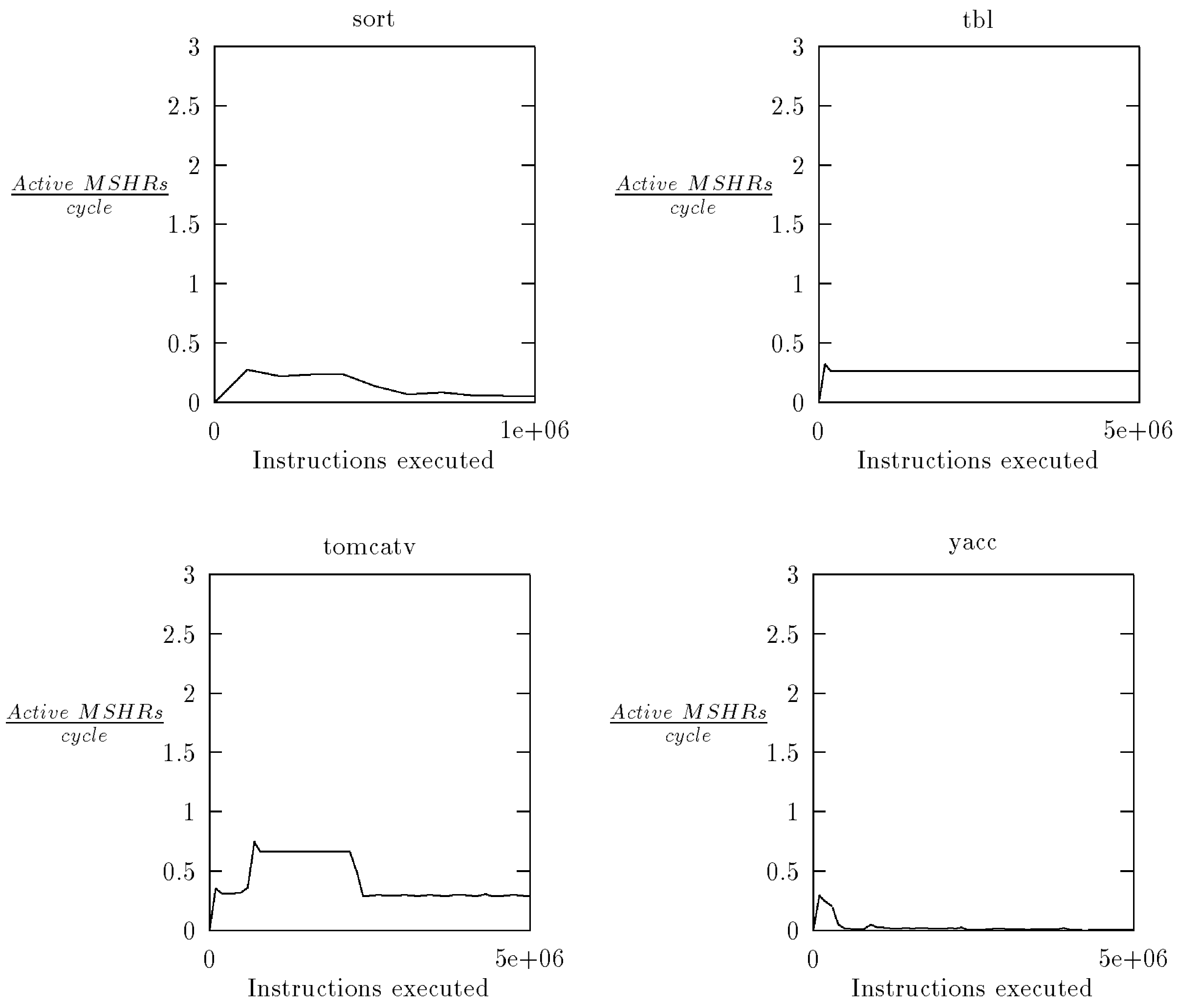


Figure 4.7 Timewise MSHR usage for sort, tbl, tomcatv and yacc

cycle within a 100,000 instruction window. This gives a good indication of instantaneous MSHR usage.

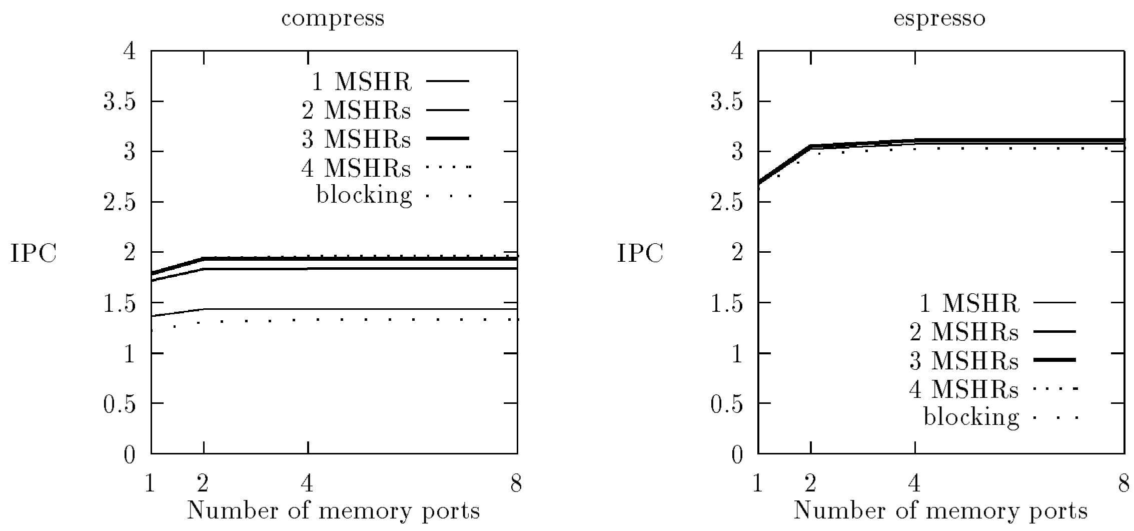
The behavior of benchmarks with high hit rates is easily explained from these figures. In espresso, fpppp, and yacc there is an initial amount of MSHR usage. After the cache fills up with usable data, the number of active MSHRs per cycle drops to near zero for the remainder of the execution time. It can be seen that the programs with poor hit rates (e.g., compress, matrix300, sort, and tomcatv) are constantly using MSHRs to request data throughout the program's execution.

4.6 MSHRs Versus Cache Ports

Both the MSHRs and cache ports are highly influential in determining the cost of the memory system. The purpose of this section is to see how tradeoffs between MSHRs and cache ports affect the system's overall performance for large and small cache sizes. Simulations were performed for each benchmark with the number of ports equal to 1, 2, 4, and 8. The number of MSHRs was varied from 1 to 4. A blocking cache was also simulated. The blocking cache is similar to the case with 1 MSHR except that the blocking cache is unable to handle any other memory requests, even hits, during the time in which the MSHR is valid. For this and the next two sections, the assumed memory latency is 25 processor cycles.

Figures 4.8 and 4.9 show the performance of the benchmarks with a 32K cache. For the benchmarks with moderate and poor hit rates, increasing the number of MSHRs from 1 to 2 demonstrates a notable performance increase. Typically, the worse the hit rate, the more performance is gained by adding an extra MSHR. Notice, however, that the performance gain quickly drops off after a second MSHR is added. Matrix300, compress, and tbl are the only programs which show performance increase when going from 2 to 3 MSHRs, and even so, this gain is slight. The programs with very high hit rates such as espresso, yacc, and fpppp show little performance gain at all by going beyond a single MSHR.

Going from one memory port to two increases performance substantially for most programs. However, matrix300 shows little performance gain when going beyond one



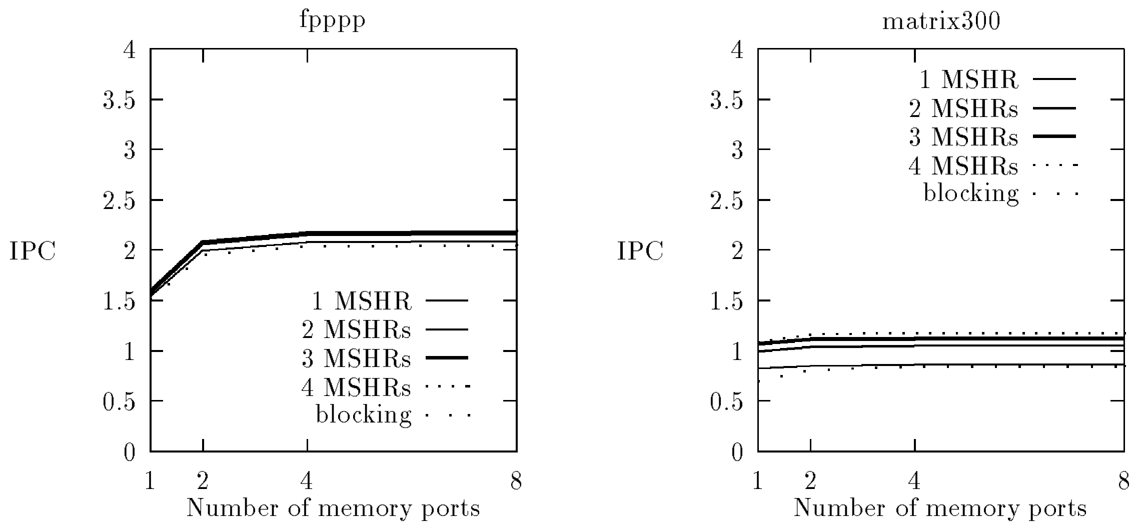


Figure 4.8 Performance with varying MSHRs and cache ports for compress, espresso, fpppp, matrix300 with a 32K cache

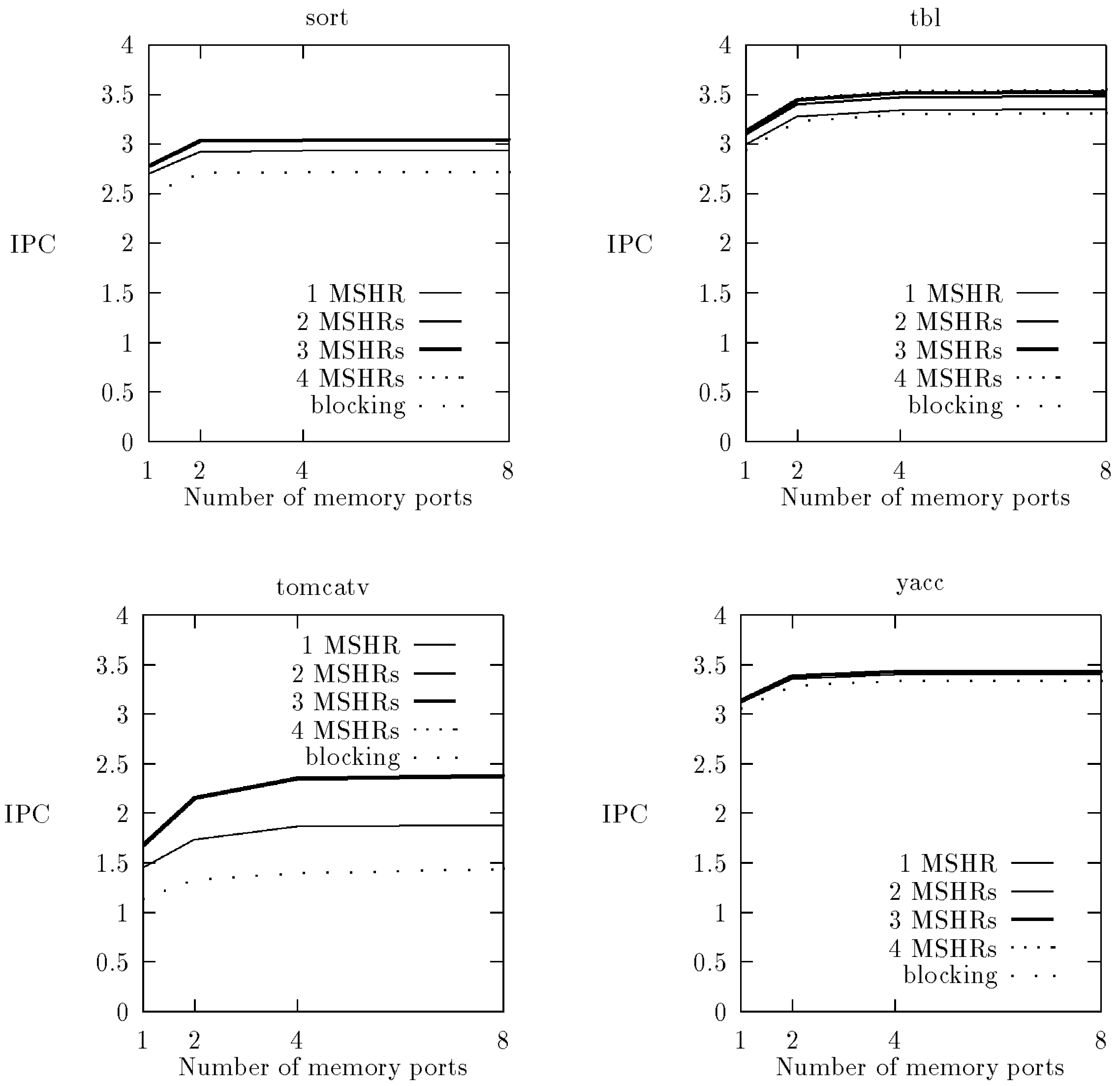


Figure 4.9 Performance with varying MSHRs and cache ports for sort, tbl, tomcatv, yacc with a 32K cache

memory port. Perhaps this is due to the fact that the extra cycles caused by a lack of cache ports are insignificant compared to the time taken to constantly load in new data from memory. Time spent waiting for data to arrive from memory can be used to service requests that were stalled due to unavailable ports. Only tomcatv shows a performance increase when the number of memory ports is increased above two.

The blocking cache performance is similar to the performance of the one MSHR case, only slightly degraded because hits cannot be serviced during the time in which any MSHR is valid. The benchmark tomcatv encounters significant performance degradation with the blocking cache. This may be due to large amounts of cache hits and MSHR hits which are on critical paths. The MPNBC case can service these while a miss is outstanding. A blocking cache can not.

Shown in Figures 4.10 and 4.11 are the results of the previous experiment performed with a 4K cache. For a small cache size, increasing the number MSHRs becomes much more important than increasing the number of cache ports. All benchmarks, even the ones with good hit rates, show a significant performance increase as the number of MSHRs is increased from 1 to 2. Many benchmarks experience further performance increase as the number of MSHRs is increased from 2 to 3. This can be explained by the fact that as the miss rate of the cache increases, more misses are now simultaneously occurring. The more the misses can be overlapped, the better the overall system performance.

Cache ports are not as important for programs which are constrained by a lack of MSHRs. The extra cycles spent waiting for results to return from memory can be used to issue the memory instructions which were stalled due to unavailable ports. Most of the 1 MSHR plots are relatively flat with respect to the number of cache ports for most benchmarks. As more MSHRs are added, the effect of additional cache ports becomes slightly more pronounced.

The blocking cache case showed a slightly worse performance than the 1 MSHR case for all benchmarks. Unlike the case for the 32K cache, the blocking case for tomcatv is only slightly worse than the case for 1 MSHR. This may be due to the small cache size. With a small cache, the accesses made during a miss will be misses and will not improve performance anyway.

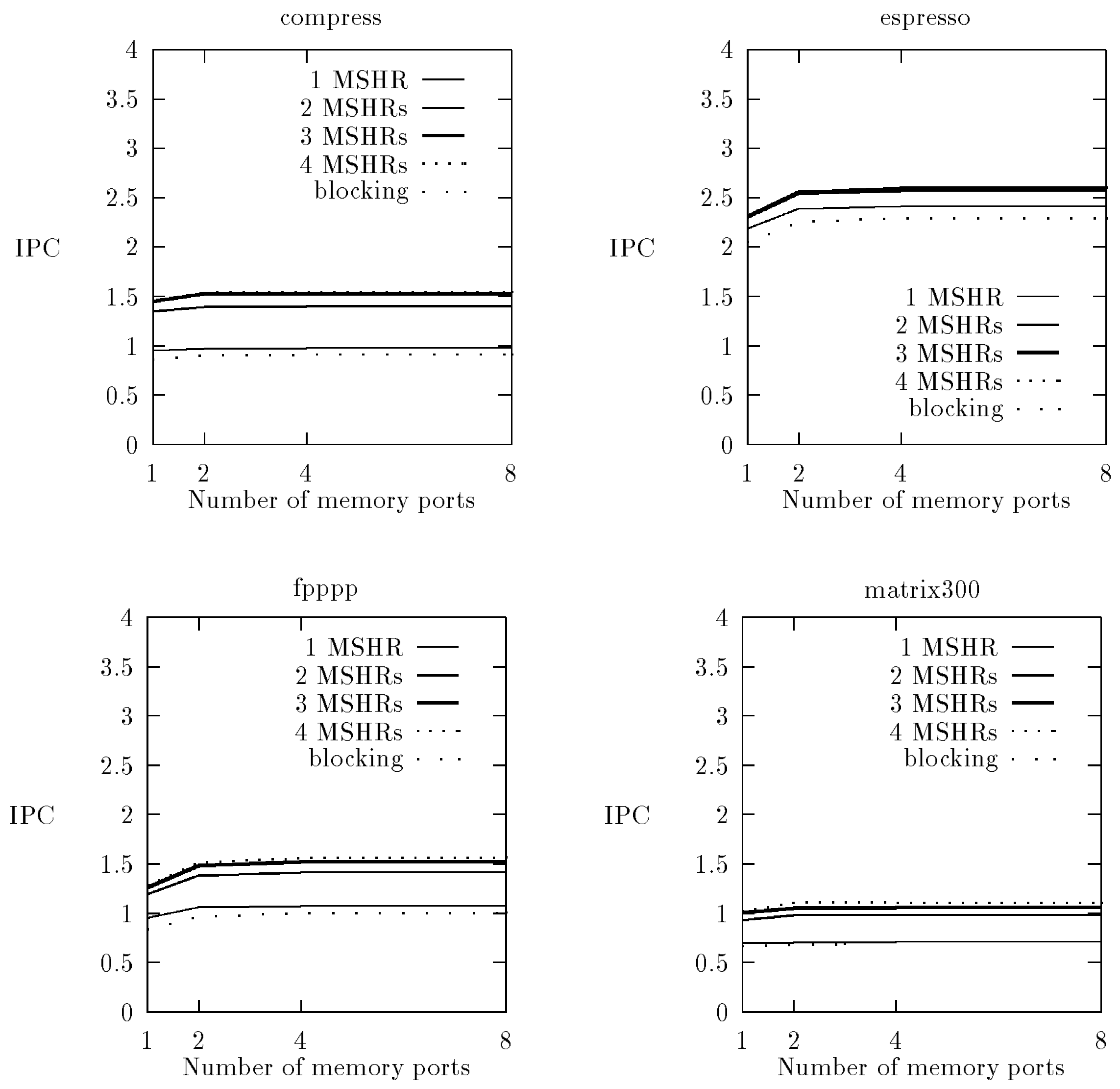


Figure 4.10 Performance with varying MSHRs and cache ports for compress, espresso, fpppp, matrix300 with a 4K cache

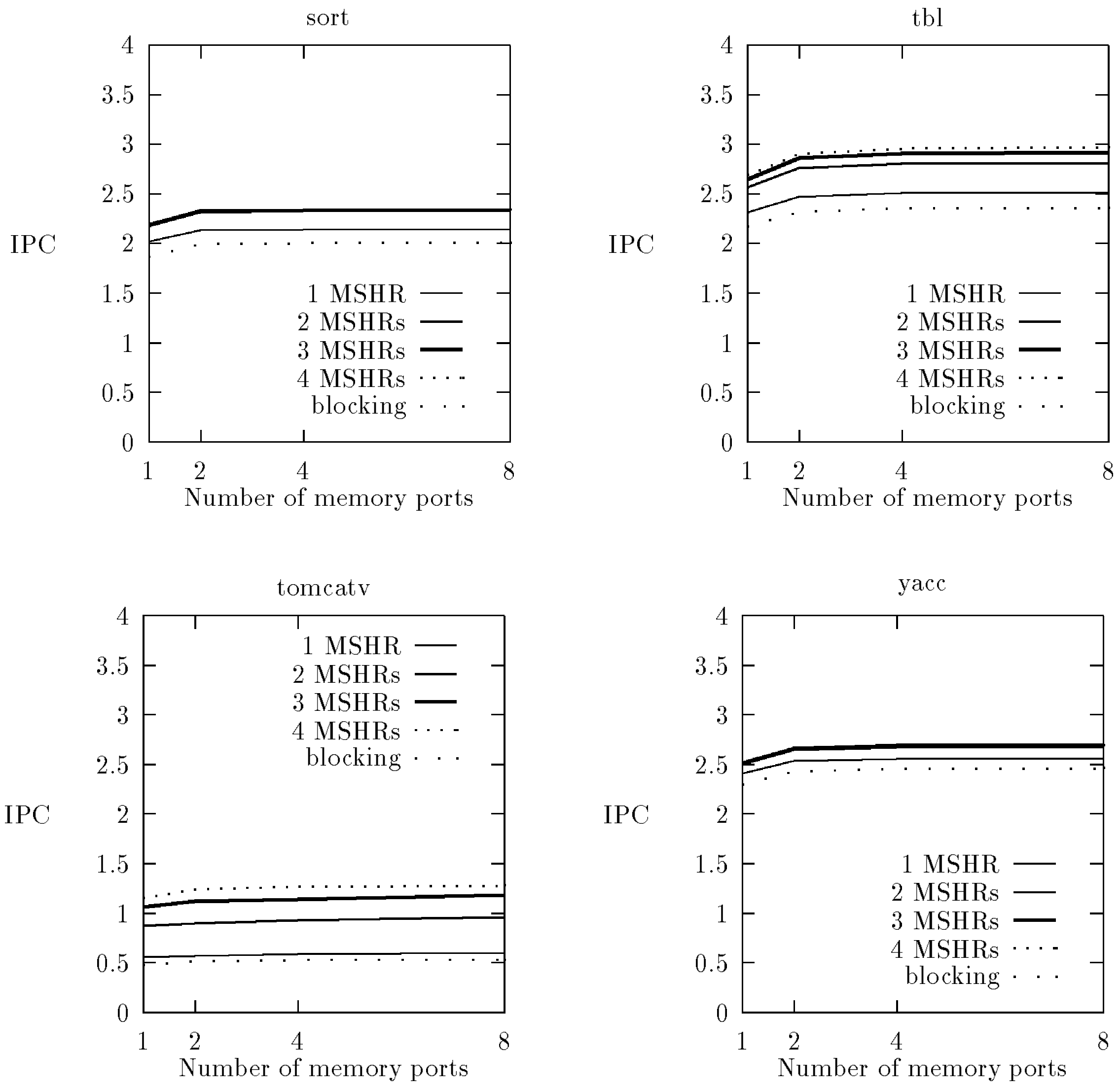


Figure 4.11 Performance with varying MSHRs and cache ports for sort, tbl, tomcatv, yacc with a 4K cache

4.7 MSHRs Versus Cache Size

From the above section, it can be seen that as the cache size is made large, the incremental value of the MSHRs becomes smaller because the hit rate has increased. In this case, the cache ports are an important factor in determining system performance for many of the benchmarks. For smaller caches, it is seen that the ability to overlap misses becomes the dominating factor and the number of MSHRs plays a more important role. However, some programs such as matrix300 make poor use of the cache, and it is expected that cache size will not affect performance.

Figures 4.12 and 4.13 expand on this by providing results with varying cache size and MSHRs. The number of cache ports is fixed at four. The cache size is varied from 4K to 128K, and the number of MSHRs is varied from 1 to 4.

4.8 MSHRs Versus Memory Latency

This section investigates the design of the second-level memory system. The interest is in whether it is more beneficial to allow for more overlapping in the second-level memory system (i.e., more MSHRs), or to provide for faster access. The number of MSHRs and the memory latency are varied from 1 to 4 and from 5 cycles to 30 cycles, respectively. It has been seen that, for some programs, using a cache size of 32K shows little about the second-level memory system since the cache hit rate is high. Therefore, this experiment uses a reduced cache size for the benchmarks with high hit rates. This experiment uses cache sizes of 1K and 4K for the benchmarks espresso, fpppp, tbl, and yacc. Cache sizes of 4K and 32K are used for compress matrix300, sort, and espresso.

Figures 4.14 and 4.15 show the simulation results for small cache sizes and Figures 4.16 and 4.17 show results for large cache sizes. It can be seen that there is definite tradeoff between extra MSHRs and decreasing memory latency. For instance, tomcatv and fpppp benefit more going from one to two MSHRs than from reducing the memory latency by 15 to 20 cycles. Additional MSHRs provide benefits as well. Other programs such as sort and yacc depend more on memory latency, though increasing the number of MSHRs does provide a notable increase in performance.

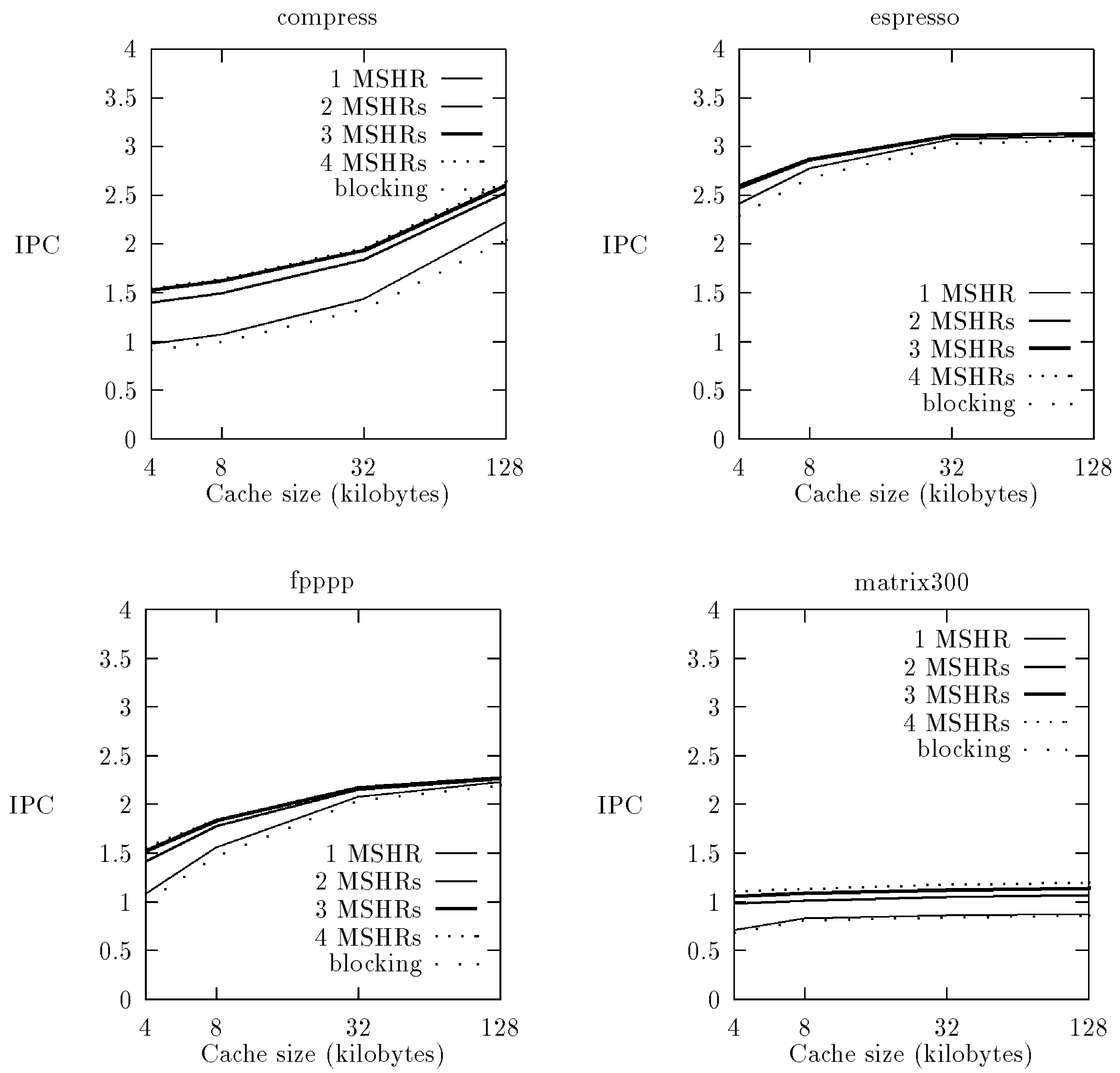


Figure 4.12 Performance with varying MSHRs and cache size for compress, espresso, fpppp, matrix300

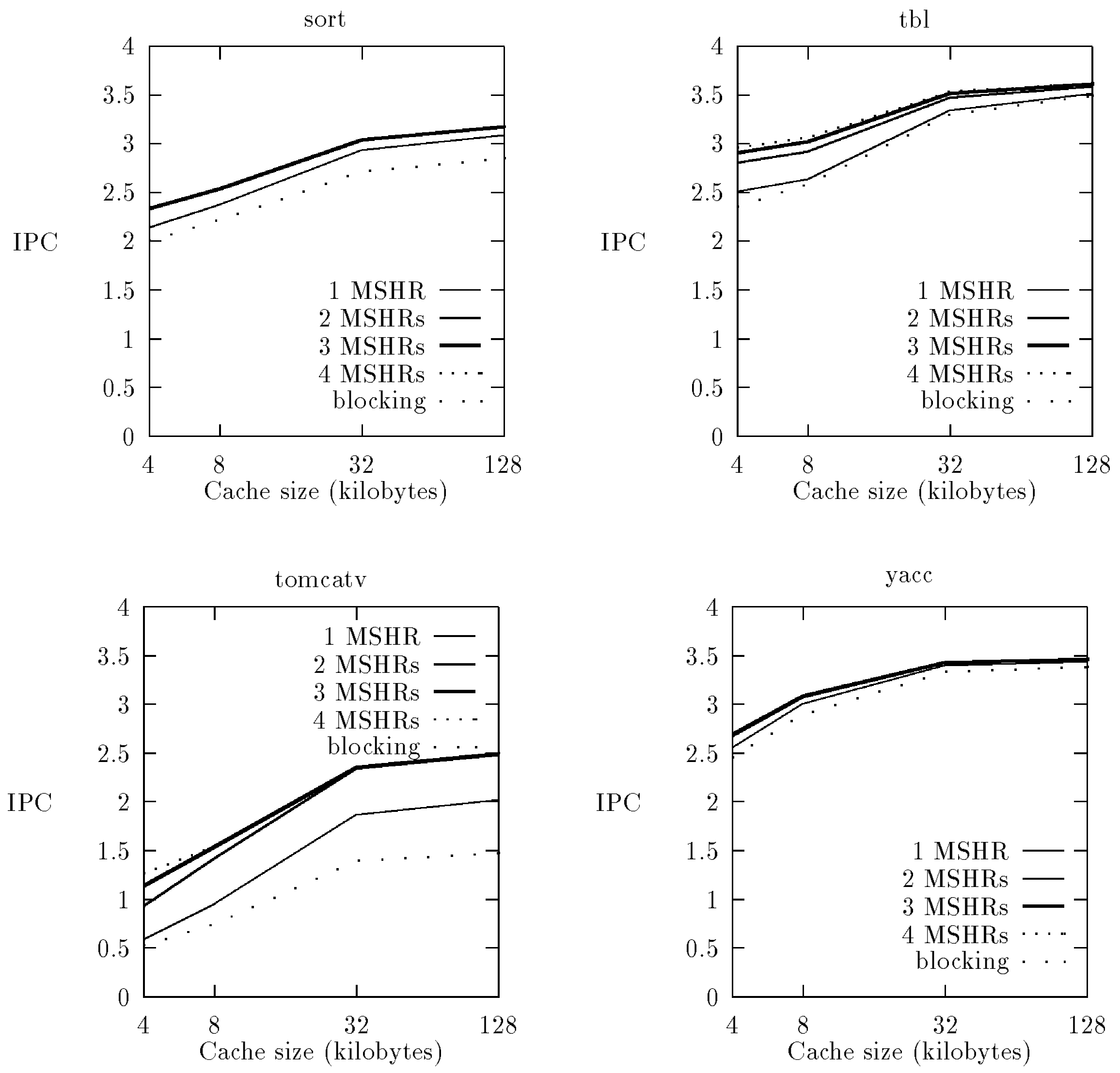


Figure 4.13 Performance with varying MSHRs and cache size for sort, tbl, tomcatv, yacc

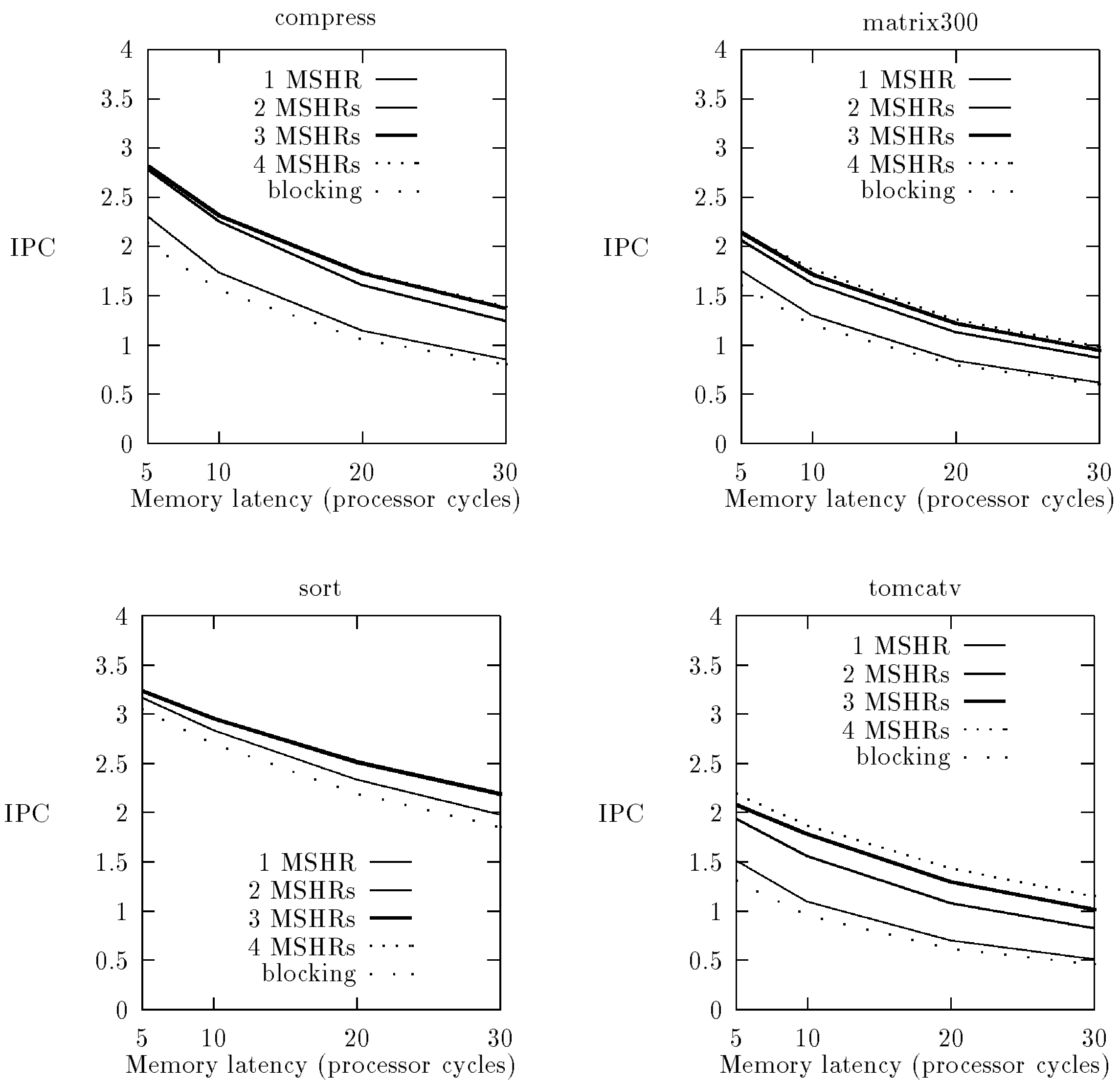


Figure 4.14 Performance with varying MSHRs and memory latency for compress, matrix300, sort, tomcatv with a 4K cache

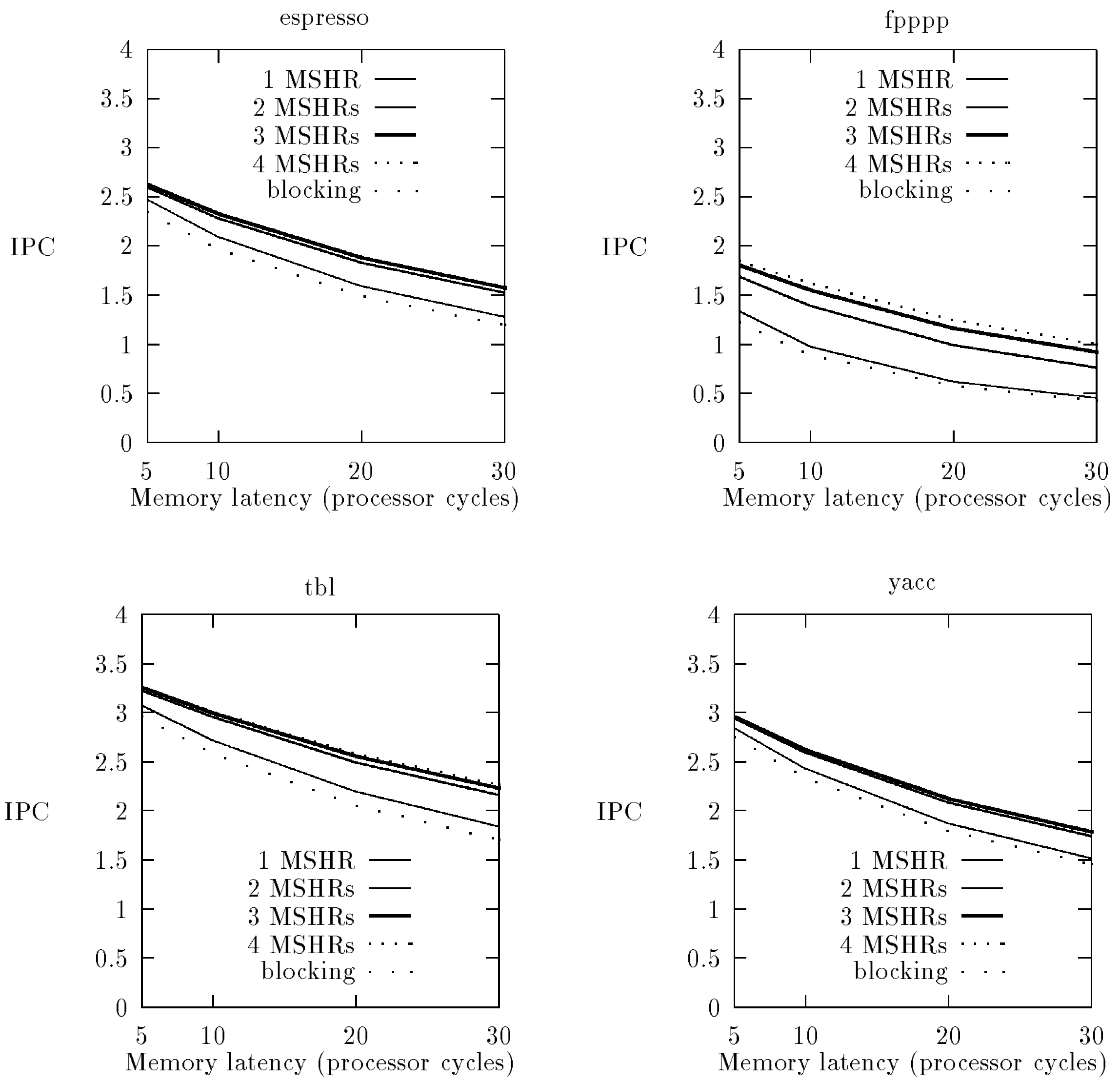


Figure 4.15 Performance with varying MSHRs and memory latency for espresso, fpppp, tbl, yacc with a 1K cache

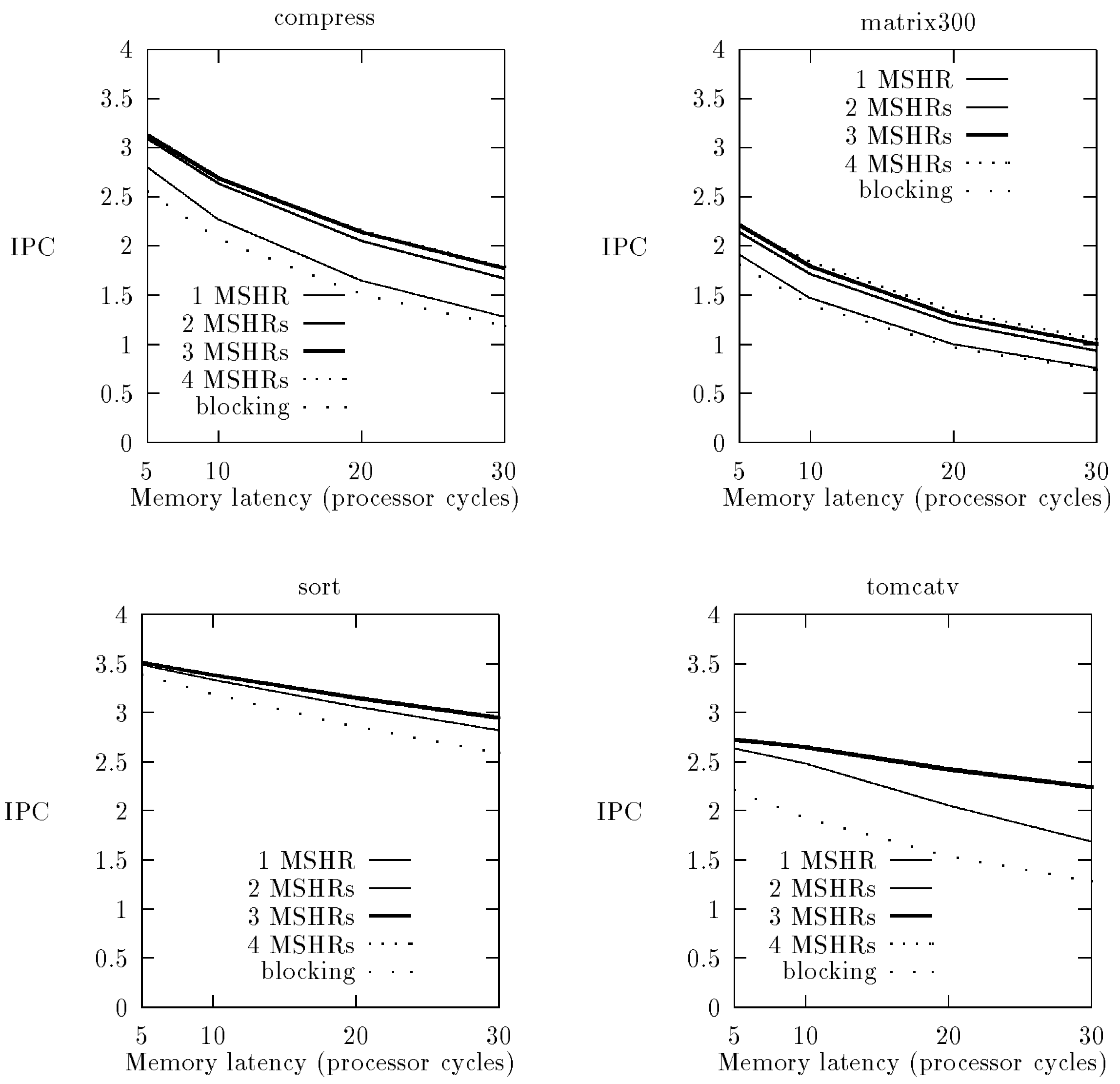


Figure 4.16 Performance with varying MSHRs and memory latency for compress, matrix300, sort, tomcatv with a 32K cache

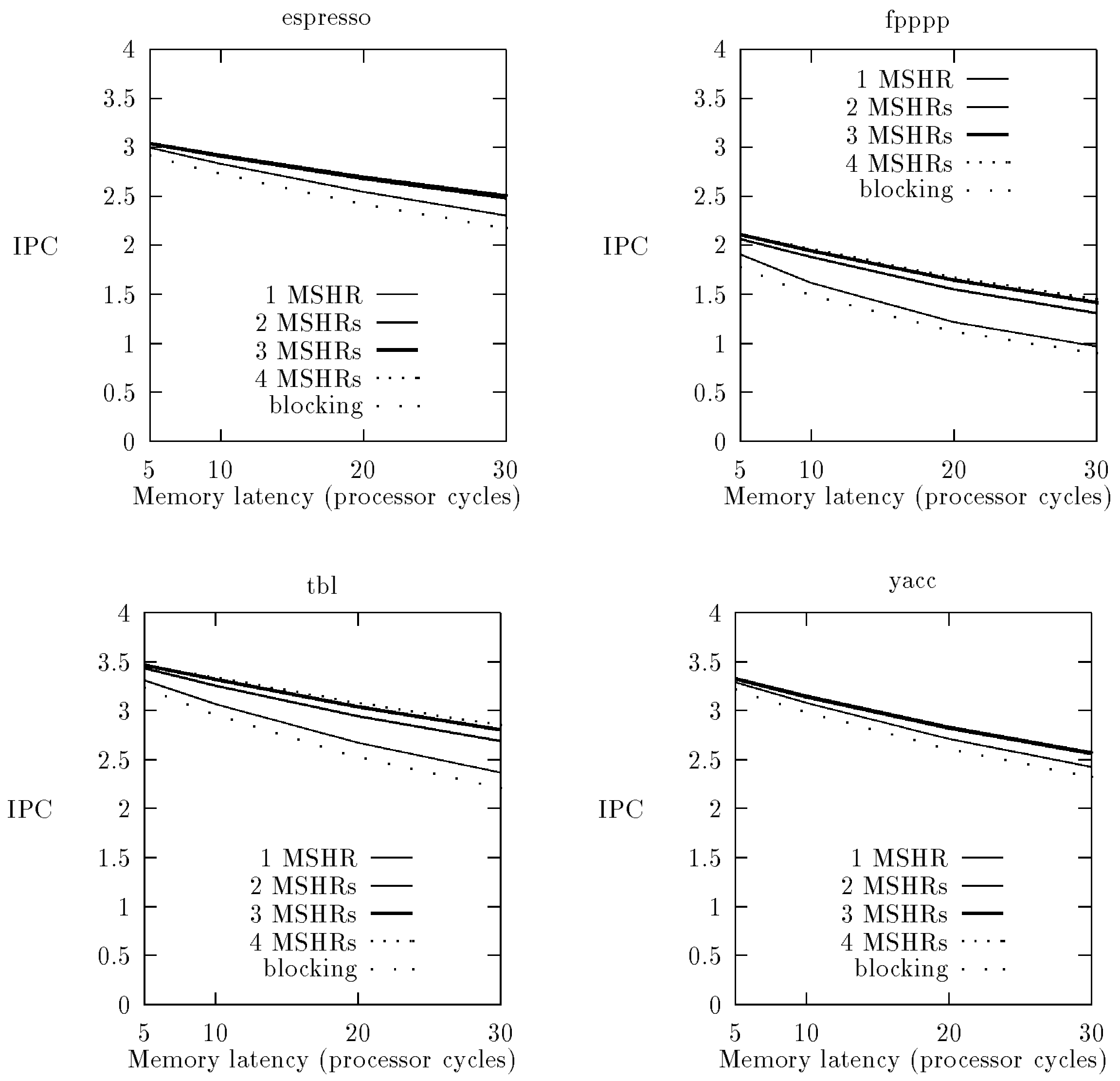


Figure 4.17 Performance with varying MSHRs and memory latency for espresso, fpppp, tbl, yacc with a 4K cache

For a small cache, the benefits provided by extra MSHRs seem relatively stable with respect to memory latency. In other words, increasing the number of MSHRs provides performance increase even when the memory latency is small. For large cache sizes, the importance of additional MSHRs is relatively small for small memory latencies and increases as the memory latency is increased. Overlapping will always be a win when either the cache size is small or the memory latency is large.

4.9 Replacement Buffer Performance

In this section, the breakdown of requests that hit in the replacement buffer is examined. Also considered is the effect of lowering the associativity of the cache. It is expected that lowering the associativity will increase the cache line conflicts, thus increasing the relative number of accesses which hit in the replacement buffer.

Hits in the replacement buffer can be broken down into three categories.

1. Read hits. A read request has hit a valid cache line and the results are returned to the cpu.

2. Write hit unsent. The write request has hit a line which has yet to be sent to main memory. In this case, the data is written to the line.

3. Write hit sent. In this case the hit has been sent to the memory. In this experiment the value is written to the cache line and the S bit is set so that the line will be resent.

Table 4.5 shows the breakdown of the replacement buffer accesses for a direct mapped and a two-way set-associative cache design. This is for the case of a 32K cache with a memory latency of 10 cycles.

There is a wide variation among benchmarks for the distribution of hits in the replacement buffer. Reads are more common than writes. Writes to lines which have been sent are more common than writes to lines that have not been sent except for the benchmarks that tend to constantly request lines from memory. This is easily explained by the fact that in an unbusy system, the cache lines can be sent as soon as they are placed in the replacement buffer whereas in a busy system they will have to wait until the bus is free or the replacement buffer is full. This concurs with the programs having a high hit rate having many more hits to sent lines than unsent ones.

Notice that as the set associativity is decreased, the number of references that hit in the replacement buffer increases dramatically. With a two-way set-associative cache, less than 1% of all requests were hitting in the replacement buffer (except for tomcatv). With a direct-mapped cache, the values are between 2% to 10%. Notice that the IPC remains relatively stable when the cache organization is changed to direct mapped (except tomcatv).

This shows that the scheme may be able to compensate for the performance loss the direct mapped cache can impose due to conflicts occurring from the loss of associativity. However, it cannot compensate for tomcatv. Tomcatv had conflicts occurring in the set-associative case and was simply overwhelmed when the organization was changed to direct mapped. This caused a significant performance drop relative to the other programs.

4.10 Increasing Bus Operation Times

In this section, the effects of increasing the bus operation times are examined for a 32K cache with a memory latency of 10 cycles. Recall that current times are 1 cycle for a line request, 2 cycles to receive the line, and 2 cycles to write a line back to the memory. This section is concerned with the increase in bus traffic and the decrease in performance associated with increasing the bus operation times. Simulations are performed for the above bus times multiplied by two, and multiplied by four. The base results are given as a basis for comparison. The results are shown in Table 4.6.

As can be expected, increasing the bus operation times has little effect on the performance for programs with high hit rates such as espresso, yacc, and fpppp. Note that the bus traffic is increased nearly linearly though. This is because the overall execution time remains the same, while the amount of time used for bus transactions is increased by some factor.

Note that programs with poor hit rates are greatly affected by the bus transaction times. Compress, matrix300, and tomcatv all experience severe performance degradation

Table 4.5 Breakdown of replacement buffer access with varying associativity

benchmark	associativity	IPC	RB breakdown		
			read hits	write hits unsent	write hits sent
compress	2	2.696	.0004	.0000	.0001
	1	2.600	.0719	.0217	.0498
espresso	2	3.182	.0019	.0001	.0001
	1	3.135	.0178	.0020	.0045
fpppp	2	2.297	.0067	.0016	.0011
	1	2.246	.0232	.0046	.0057
matrix300	2	1.834	.0035	.0024	.0010
	1	1.814	.0113	.0079	.0033
sort	2	3.379	.0012	.0000	.0001
	1	3.247	.0409	.0074	.0090
tbl	2	3.677	.0002	.0001	.0001
	1	3.610	.0242	.0376	.0258
tomcatv	2	2.659	.0382	.0161	.0178
	1	2.384	.0321	.0149	.0184
yacc	2	3.510	.0020	.0001	.0002
	1	3.465	.0084	.0012	.0046

Table 4.6 Effects of increasing bus operation times

benchmark	IPC			bus traffic		
	1xBus	2xBus	4xBus	1xBus	2xBus	4xBus
compress	2.696	2.401	1.937	.186	.332	.535
espresso	3.182	3.163	3.125	.008	.017	.032
fpppp	2.297	2.266	2.201	.025	.049	.094
matrix300	1.834	1.508	1.102	.227	.374	.546
sort	3.379	3.281	3.094	.042	.080	.152
tbl	3.677	3.624	3.507	.020	.038	.074
tomcatv	2.659	2.446	2.194	.202	.368	.600
yacc	3.510	3.489	3.442	.010	.019	.037

as the bus operation times are increased. The bus traffic, however, does not increase linearly as in the other cases, because the execution time is also increased as well as the bus operation times.

CHAPTER 5

CONCLUSIONS

This thesis has presented the issues involved in designing a multiported nonblocking cache. A scheme was presented which is capable of handling requests to outstanding cache misses and supervising the return of the cache lines to the cache buffer. Another scheme was presented to allow hits in the replacement buffer to be directly returned to the cache. Some of these concepts were illustrated with an example directly from an actual design. Trace driven simulations using 5 million SPARC instructions were performed to evaluate the cost effectiveness of design decisions based on those schemes.

The following are the highlights of the simulation results.

- It has been shown that performance degradation due to cache bank conflicts is an important issue in the design of a MPNBC. Cache memory should be interleaved using an interleaving scheme that allows simultaneous sequential access to prevent serious degradation in floating-point programs. Further research needs to be carried out to evaluate the effects of stalling various cache banks upon the return of a line as called for by the different interleaving schemes.

- There is a definite tradeoff between cache size and main memory latency. For small caches, the program completion time is highly dependent on the main memory latency. As cache size increases, programs with good locality are immune from the effects of a memory latency. However, programs that make poor use of the cache (eg., matrix 300) exist; they are affected little by increases in cache size and primarily by memory latency.

- It is shown that most of the requests hit in the cache buffer, though some programs have a large portion of requests hitting in the MSHR queue. Reads to unwritten words are the most common type of MSHR hits, followed by writes. Read hits to previously written words are rare as wisdom dictates and may allow for a simpler design without data paths from the MSHR to the cpu. Hits to the replacement buffer are rare in a two-way set-associative cache. Many integer programs have requests that stall due to cache bank conflicts, while programs with poor hit rates and poor cache usage will stall from unavailable MSHRs.

- It has been shown that programs with high hit rates are generally more dependent on additional memory ports for achieving good performance rather than MSHRs. For smaller hit rates, MSHRs influence performance more than memory ports. In most cases, increasing the number of ports above two does not provide much performance increase. Increasing the number of MSHRs above two does provide performance increase for small cache sizes.

- It has been shown that using the replacement buffer to service some cache accesses is an effective way to reduce the performance degradation caused by lowering the associativity. A thorough examination of this topic has not been undertaken making it a good candidate for future research.

- The tradeoff between overlapping main memory access and decreasing access latency has been shown. Some programs benefit greatly from overlap (e.g., tomcatv and fpppp), while others benefit more from decreasing access latency (e.g., yacc and tbl). It has also been shown that for small cache sizes, overlapping the memory accesses provides a notable performance increase across all memory access latencies. For larger cache sizes, overlapping the memory accesses helps for large memory latecies but not as much for smaller ones. Overlapping provides the best performance increase when either the cache size is small or the memory latency is large.

REFERENCES

[1] M. Butler and Y. Patt, "The effect of real data cache behavior on the performance of a microarchitecture that supports dynamic scheduling," in *Proceedings of the 24th Annual Workshop on Microprogramming and Microarchictectures*, pp. 34–41, 1991.

[2] M. Butler, T. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single instruction stream parallelism is greater than two," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 276–286, 1991.

[3] J. W. Fu, "A study of some design choices for improving the performance of a shared cache system," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1990.

[4] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers," in *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 364–373, 1990.

[5] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 81–87, 1981.

[6] V. Popescu, M. Schultz, J. Spracklen, G. Gibson, B. Lightner, and D. Isaman, "The metaflow architecture," in *IEEE Micro*, June 1991.

[7] G. S. Sohi, "Instruction issue logic for high performance, interruptible, multiple functional unit, pipelined computers," in *IEEE Transactions on Computers*, vol. 39, no. 3, pp. 349–359, March 1990.

[8] G.S. Sohi and M. Franklin, "High-bandwidth data memory systems for superscalar processors," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 53–62, 1991.