Complexity/Performance Tradeoffs with Non-Blocking Loads

Keith I. Farkas
Dept. of Electrical and Computer Engineering
University of Toronto
10 Kings College Rd
Toronto Ontario Canada
M5S 1A4
farkas@eecg.toronto.edu

Norman P. Jouppi
Digital Equipment Corporation Western Research Lab
250 University Avenue
Palo Alto, CA 94301
jouppi@pa.dec.com

Abstract

Non-blocking loads are a very effective technique for tolerating the cache-miss latency on data cache references. In this paper, we describe several methods for implementing non-blocking loads. A range of resulting hardware complexity/performance tradeoffs are investigated using an object-code translation and instrumentation system. We have investigated the SPEC92 benchmarks and have found that for the integer benchmarks, a simple hit-under-miss implementation achieves almost all of the available performance improvement for relatively little cost. However, for most of the numeric benchmarks, more expensive implementations are worthwhile. The results also point out the importance of using a compiler capable of scheduling load instructions for cache misses rather than cache hits in non-blocking systems.

1. Introduction

A continuing trend in the design of computer systems is the widening gap between microprocessor and memory speeds. This speed discrepancy can significantly impact the performance obtained from the system if the processor stalls whenever a data-cache miss occurs. To prevent such stalls, non-blocking loads and stores can be provided to allow the processor to continue executing instructions while a data-cache miss is resolved. Using non-blocking instructions with sufficient hardware resources, a data-miss induced stall will only occur if the register target of the load is used by an instruction before the register is filled.

There are two common methods for implementing non-blocking stores. The first method entails placing the data to be stored in a write buffer while the cache fetches the line into which the data is to be stored. The second method is to use write policies other than fetch-on-write, such as write-around, which neither fetch data on a write miss nor write the new data into the cache; instead the data is written directly to the next lower level in the memory hierarchy [6]. Both of these methods do not require very complex hardware and are becoming common in microprocessors.

To allow the processor to continue to access the data cache during the processing of a non-blocking load miss, a lockup-free cache [7] is required. Non-blocking loads have only recently appeared in microprocessors [3, 9], and often these implementations have been fairly restrictive. For example, the HP PA7100 [1] allows a maximum of only one miss outstanding in the cache (i.e., "hit under miss"). The only recent appearance of mostly restrictive implementations is in part due to the more significant hardware complexity required to implement non-blocking loads. Yet studies of non-blocking loads have often assumed very unrestricted models. For example, Sohi and Franklin [12] assumed an 8-way banked cache where each bank could support four outstanding fetches and several times more misses. Other studies [2, 5, 11] generally have used unrestricted models while focusing on other aspects of system performance.

In this paper, we investigate the performance obtainable from a number of practical non-blocking load implementations and evaluate the performance obtained in the context of the hardware complexity required. Key to our investigation is careful modeling of the processor and memory system and judicious accounting for the complexities involved with non-blocking loads. Our results suggest that a significant portion of the available performance improvement can be achieved with implementations that are not nearly as complex as the unrestricted implementations assumed in many previous studies.

We begin by presenting in Section 2 a brief description of the hardware implementations considered and in Section 3 an overview of our simulation methodology. We then present in Section 4 the performance of various hardware organizations using a baseline system with a 8KB direct mapped data cache and 32 byte lines. Section 5 considers the effects from variations in the cache size, the cache line size, and the miss penalty. Section 6 explains how our results can be extended to specific processor organizations. We finish by summarizing our results in Section 7.

2. Hardware Options

To support multiple in-flight (i.e., outstanding) misses, lockup-free caches require special hardware resources which we describe below. In discussing non-blocking loads it is helpful to divide the misses into three categories. The first miss to a cache block with a given tag is called the *primary miss* [7]. Subsequent misses to any of the bytes in the block that is being fetched may cause a stall depending

1063-6897/94 $03.00 © 1994 IEEE

on the hardware resources available. If a stall occurs due to such a *structural hazard*, the miss causing the stall is called a *structural-stall miss*. If, however, a structural-hazard-induced stall is not required, then the miss is referred to as a *secondary miss* [7]. Secondary misses require in-flight-miss resources while structural-stall misses do not.

The organization of a lockup-free cache with support for non-blocking loads was first given by Kroft [7]. In Kroft's implementation, registers called MSHRs (Miss Status Holding Registers) are used to hold information on outstanding misses. The MSHRs save enough information on a miss so that when a requested cache block arrives from the next lower level in the memory system, load instructions for the corresponding block can be completed. One MSHR is associated with each fetch request outstanding to the next lower level in the memory hierarchy. A primary miss and several secondary misses can be merged into a single fetch request. Kroft's organization only allows one miss per word in the cache block being fetched. If two misses occur to a word while the block is being fetched, the processor would stall; this second miss is an example of what we call a structural-stall miss. In the remainder of this section we consider four organizations of MSHRs: *implicitly addressed*, *explicitly addressed*, *in-cache MSHRs*, and an *inverted MSHR organization*.

2.1. Implicitly Addressed MSHRs

The organization of a typical basic MSHR which is similar to Kroft's is shown in Figure 2-1. (The typical bit width of each field is listed above each field.) Each MSHR contains a valid bit to signal that it is in use. When a primary miss occurs, the valid bit and block request address of a free MSHR are set. (The processor stalls if there is not a free MSHR.) Assuming a 64 bit virtual address architecture machine with a 48 bit physical address and a 32 byte line size means that 5 bits are required to address within a 32 byte cache block size, and only 43 bits need to be stored as the block request address. Each MSHR has its own comparator so that a collection of MSHRs can be searched associatively when a new miss occurs to find out whether the new miss is a primary, structural-stall or secondary miss. For each word in the block (e.g., four 8 byte words in a 32 byte cache block) there exists a destination address, formatting information, and a word valid bit. These fields are set when a load miss occurs for a particular word so that when the block returns from the next lower level in the memory hierarchy, the actions of the load instruction can be completed. The destination address is typically a full register address including a bit specifying whether it is a fixed point or floating-point register. The format information gives other information provided by the load opcode and perhaps low-order bits of the address which are required for completion of the load instruction. Examples of these are the width of the load (e.g., 1, 2, 4, or 8 bytes), byte address bits for byte loads, and a bit saying whether to sign extend the returning data. Specific instruction set architectures would require additional information. For example, the MIPS R4000 architecture [10] has load-word-left and load-word-right instructions for support of unaligned accesses. Information specifying these load opcodes would need to be saved in the MSHRs as well so that proper shifting and masking of the data can be performed when it is placed in a register.

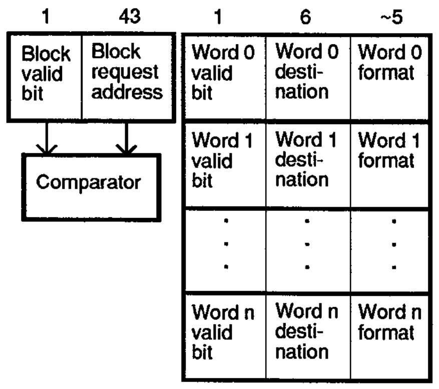

Figure 2-1: Basic implicitly addressed MSHR fields

Note that all lockup-free caches require information to be carried along with returning fetch data in order to match up waiting requests and returning data, unless all data returns in the order it is requested. For example, if there are a small number of MSHRs, the MSHR number might be sent with a fetch request as a tag and then returned with the fetched data. However, since in most systems addresses already need to be sent to the CPU from the memory system for invalidations when maintaining cache consistency, if these addresses are sent with returning fetch data then the MSHRs can be probed with the address of fetch data on its return.

2.2. Explicitly Addressed MSHRs

Even though the basic MSHR of Figure 2-1 is fairly large ((4×12)+44=92 bits in the example above, plus a 44 bit comparator and significant control logic) it has two limitations. First, multiple accesses to the same word while a fetch of their block is outstanding will cause a stall. Even in a machine with a 64 bit virtual address architecture, there may be a fair number of loads and stores of 32 bit data for many years to come. So instead of providing 64 bit word granularity in the word area, the number of word records may need to be doubled by reducing their granularity to 32 bits. This doubling would increase the number of bits in the word section of our example to 8×12=96 bits, making each MSHR 140 bits wide in total. However, this increase still does not allow multiple byte loads to be outstanding to the same 32 bit word in machines with byte loads and stores. A second limitation is that multiple loads to the exact same address will also cause stalls. Therefore, with this type of MSHR organization it is important for the compiler to combine byte operations into word accesses and to use register moves instead of loading from the same address twice.

The word fields of the MSHR in Figure 2-1 are *positionally* addressed (i.e., their position specifies their word address within the block). Another more complicated MSHR is shown in Figure 2-2. This MSHR has a number of generic word fields which explicitly give their word address. An *explicitly addressed* MSHR with 4 sets of word fields could handle four misses to the exact same address without stalling, or four misses to four bytes within one word. Yet, even though bits are required to explicitly store the address within the block, an explicitly addressed MSHR that can hold 4 misses would be only (4×17)+44=112 bits wide. This MSHR is smaller than an implicitly addressed MSHR for 32 byte lines with 4 byte granularity. Explicitly addressed MSHRs work best when there are only a limited number of misses to the same block and these references overlap or are to adjacent bytes or halfwords.

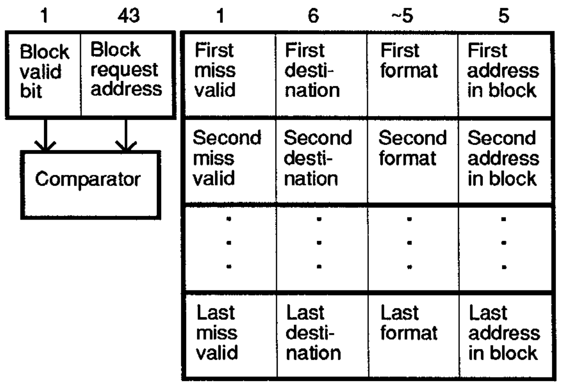

Figure 2-2: Explicitly addressed MSHR fields

2.3. In-Cache MSHR Storage

Implementing a large number of MSHRs each with support for many misses can require large amounts of storage. Franklin and Sohi [4] have observed that cache lines waiting to be filled on outstanding fetches can be used to store MSHR information. This can be done by adding a *transit bit* to each cache line. This bit indicates that the line is in the process of being fetched, that the address in the cache tag specifies the address being fetched, and that the data in the cache line itself gives MSHR information. Using this technique, many secondary misses could be supported whether the MSHR fields were addressed implicitly or explicitly. However, in direct-mapped caches only one in-flight primary miss per cache set can be supported. One thing to keep in mind with this method is that if the read port width of the cache is much smaller than the line size (e.g., if only 8 bytes of a 32 byte line can be read per cycle), it may take several cycles to read the entire cache line when fetch data arrives. Thus it may be advantageous to limit the length of the MSHR information to the width that can be read in a single cycle. Also, even though only one bit is added to each cache line, for very large caches this may require more area than a simpler distinct set of MSHRs.

2.4. Inverted MSHR Organization

A very aggressive lockup-free cache may have many MSHRs. Even if explicitly addressed MSHRs are used, when there are many MSHRs in the system, the total number of MSHR destination pointers provided may exceed the number of destinations in the machine. Even so, there will likely be restrictions on the maximum number of misses outstanding to a single block (e.g., 4 in the explicitly addressed MSHR above), or to the maximum number of blocks being fetched (i.e., the number of MSHRs).

As an alternative organization for an aggressive lockup-free cache, we introduce an *inverted MSHR* (see Figure 2-3). In an inverted MSHR there is one set of fields for each possible destination of fetch data, instead of one set of fields for each outstanding fetch as in a traditional MSHR organization. The possible destinations of fetch data could include all the integer and floating-point general purpose registers in the machine, write buffer entries (for merging with write data when writing into a write-allocate cache), the program counter, and an instruction prefetch buffer (if it exists). Thus a typical inverted MSHR might have between 65 and 75 entries.

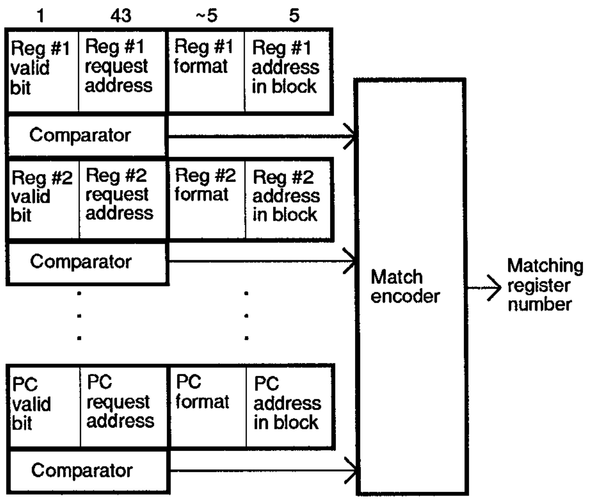

Figure 2-3: Inverted MSHR organization

When a new miss occurs, the inverted MSHR is searched associatively just like a traditional set of MSHRs. If there is already an outstanding fetch for that block, one or more matches will occur. In this case, the miss address is not sent off-chip, but the inverted MSHR entry corresponding to the destination of the fetch data is marked valid and its block request address, formatting information, and address within the block are written. In the event there are no matches, the MSHR entry corresponding to the destination of the fetch data is still written in the same way, but a fetch of the requested block from the next lower level in the memory hierarchy is also begun. When a block of data returns the inverted MSHR must be probed to identify those destinations waiting for data from the block. Then each waiting destination is filled in turn using the information contained in the MSHR. This information specifies the format to be used and indentifies the the portion of the block which is to be loaded into the the destination.

An inverted MSHR can be built with the same basic circuits as a fully-associative translation lookaside buffer (TLB), with the addition of a match entry-number encoder. (The match entry encoder may already be present in a TLB, depending on the replacement strategy used.) An inverted MSHR has the advantage that it has no restrictions on the number of blocks being fetched or the number of misses per block being fetched other than the number of possible destinations of fetch data in the machine.

2.5. Hardware Summary

In this section we have described several mechanisms that can be used to store information about outstanding misses. Many other mechanisms would be possible. We have attempted to list the simplest mechanisms covering a spectrum of non-blocking support. In the following sections we present simulation results that could be expected when using different non-blocking hardware support.

3. Simulation Methodology

The performance achieved with the lockup-free implementations described in the previous section is a function of the number of in-flight misses that are supported. To evaluate the complexity versus performance tradeoff for these implementations, we investigated the performance achieved when restrictions are imposed on the number and the type of in-flight misses. For this investigation, we chose models for the processor and the memory system that isolate the performance available from various non-blocking organizations from other aspects of machine performance. This isolation is achieved by structuring the models so that *all* processor stalls only relate to data accesses. As a result, performance is measure using the average number of memory stall cycles per instruction (CPI). In Section 6, we describe how the results can be extended to systems with more complex processors and memories.

3.1. Processor and Memory Models

The processor model we use has separate data and instruction caches, a multistage pipeline and 3 operand instructions. Since we are only concerned with the behavior of the data cache, all instructions are assumed to hit in the instruction cache. Branch instructions may also introduce stalls if the branch-delay slot(s) cannot be filled by the compiler or if the branch is taken. Branch stalls may occur at the same time as other stalls such as those attributable to accessing a register before its contents are valid. Therefore, to take branch stalls into account requires explicitly modeling them. In addition, the length of a stall is determined by cache hit rates, memory access delays and code scheduling. To avoid having to model a complex memory system and thereby render our results less general, we avoid branch induced stalls by assuming that there are no branch delay slots and that there is a perfect branch-target predictor.

To remove the effects of stalls caused by resource conflicts, we have chosen to model a single-issue processor with single cycle instruction latencies. The register file comprises 32 integer and 32 floating point registers that can be accessed via 2 read and multiple write ports; the need for multiple write ports is explained below.

The memory system model assumes a direct mapped data cache that uses write-around (i.e., no-write-allocate) and write-through policies, and a write buffer situated between the data cache and lower levels in the memory hierarchy. To avoid stalls induced by the write buffer (such as it being full), no memory cycles are required to retire writes from the write buffer. Also, to avoid stalls induced by the main memory, the main memory is assumed to be fully pipelined. Hence, regardless of other memory activity, a constant number of cycles is required to fetch a cache line from the memory into the cache. Data cache references that hit in the cache require a single cycle to be resolved. When a block of data is written into the cache as a result of a primary miss, all registers waiting for the data are updated at the same time; that is, all primary, structural-stall, and secondary misses for a block are simultaneously resolved. This assumption necessitates multiple write ports for the register file.

Given the above assumptions, a stall will *only* occur if there are too many cache misses outstanding (i.e., a *structural-hazard, miss-induced stall*) or if a use is made of a register before a previously issued load completes (i.e., a *true data dependency, miss-induced stall*). As all stalls concern misses, the term miss CPI (MCPI) will be used in lieu of memory stall CPI.

3.2. Simulation Framework

To perform the simulations for this study, we used an object-code translation and instrumentation system. This system emulates the execution of a benchmark as it would run on a target machine by running the benchmark on an existing machine. As a result both the functional behavior and the memory behavior of the application are simulated. The first step in performing a simulation is to compile the benchmark using instruction scheduling rules pertaining to the architecture of the processor to be modeled. We use a modified version of the Multiflow VLIW Compiler [8] for this purpose[1]. Next, the resulting assembly language (i.e., object code) is translated into the assembly language of the

[1]The compiler was modified to produce RISC-like object code for a processor with 32 bit addresses, 32 bit integers and 64 bit floating point numbers. The speculative and predictive code scheduling options were not used. The compiler uses a common backend for both C and Fortran code.

machine on which the simulations are run, namely, Alpha AXP workstations. Instrumentation and modeling code is then inserted into the translated code. Finally, the augmented, translated binary is linked with run-time libraries and support routines. The run-time libraries contain routines (e.g., sin()) that are called from within the benchmark and as such their execution must also be emulated. Hence, these routines have been compiled and instrumented in the same manner as the benchmark.

The instrumentation code is inserted to record the emulated run-time behavior of the benchmark. This code records various statistics including cache miss rates, the number of (simulated) instructions executed, and the number of (simulated) clock cycles. The modeling code is inserted to allow the factoring in of the time required to resolve memory and register accesses. This modeling is accomplished by inserting before every emulated load and store instruction a call to a procedure that models the memory. These calls pass to the procedure the address of the item being loaded or stored and the procedure returns the amount of time required to process the access. For example, for non-blocking loads, this time will be the time required to launch the load whereas for a blocking-load it will be the time required to load the data into the cache if it is missing. A mechanism in the simulator adjusts these addresses so that they do not reflect the presence of the simulation infrastructure. Calls to a scoreboard procedure are also inserted before every emulated instruction that uses the result of a load. This procedure factors in the time required to validate the source registers of the instruction.

3.3. Methodology

To explore the performance of the various implementations, the following software and hardware parameters were varied:

1. **load latency**: The load latency is the time in cycles that the compiler assumes is required to fetch data from the cache on a cache hit and load it into a register. This parameter indicates to the compiler how many instructions it should try to insert between the load instruction and the first use. In contrast, the simulator *always* uses a cache hit load latency of 1. Thus the (scheduled) load latency parameter gives the degree of cache miss tolerance that is expected if the compiler successfully scheduled the code for this latency. *It is important to note that the load latency is a code-scheduling parameter and not a system parameter.*
2. **in-flight misses**: Both the number of primary and secondary misses permitted to be outstanding to each set in the cache, and the number of primary and secondary misses permitted to be outstanding to the cache as a whole.
3. **cache parameters**: The cache size and the line size.
4. **miss penalty**: The miss penalty is the time in cycles required to fill a line in the cache from the next-lower level in the memory hierarchy.

We have simulated 18 of the SPEC92 benchmarks for a wide range of the above parameters. The results we present below represent over 3700 simulations requiring approximately 370 days of run-time. This paper discusses in detail five representative benchmarks of the 18; these five are listed in Table 3-1 along with some run-time characteristics. The three major columns give breakdowns based on instruction, load, and store references. The sub-columns give information about the load latency parameters which resulted in the minimum and maximum number of instructions executed. Individual columns give the number of instructions executed (in millions) and the load latency for which this maximum or minimum number of instructions was executed.

Bench-	Instruction Refs				Data Loads Refs				Data Stores Refs			
	Min		Max		Min		Max		Min		Max	
mark	#	lat	#	lat	#	lat	#	lat	#	lat	#	lat
doduc	1025	1	1035	20	234	1	238	20	701	1	703	20
eqntott	1766	2	1768	10	219	1	220	2	239	3	243	2
su2cor	5114	1	5120	6	1095	3	1099	6	521	1	522	20
tomcatv	1070	6	1091	2	297	6	318	1	100	1	108	3
xlisp	5612	1	5667	10	143	6	144	3	852	6	855	1

Figure 3-1: Benchmark characteristics; references in millions

It is clear from the table that the number of references can change slightly with the load latency. This result is expected as the load latency significantly influences the code scheduling. The compiler tries to meet the specified load latency using a number of techniques including instruction reordering. Because register allocation occurs after instruction scheduling, code schedules prepared with different load latencies are likely to have different register-use profiles. Hence, the number of register spills to memory may vary thereby changing the number of data and instruction references.

Note that the references presented in the table do not include those generated by the operating system as we could not instrument operating system routines.

4. Baseline Performance Investigations

In this section, we explore the performance and the cost effectiveness of non-blocking load implementations for our baseline cache configuration of a 8 Kbyte direct mapped cache with 32 byte lines and a 16 cycle miss penalty. In the ensuing discussion, it is important to remember that the only stalls that can occur are those attributable to true data dependencies or structural hazards.

We begin with *doduc* which best illustrates many of the characteristics common to all benchmarks. The MCPI incurred by *doduc* with several of the simpler non-blocking implementations is shown in Figure 4-1. In this figure, each curve corresponds to a specific cache implementation and the curves show how the MCPI varies with the scheduled load latency. The upper two curves correspond to lockup caches and are given for sake of comparison. These curves have labels that include the term "mc=0".

This term indicates that the implementations allow *zero* outstanding *misses* to a *cache* without stalling the processor, or in other words, are lockup. The term "+wma" included in the label of the upper-most curve indicates that in addition to being lockup, the cache used *write-miss allocate*, and the processor stalls until misses caused by writes have been serviced. The bottom-most curve, labeled "no restrict" shows the MCPI incurred with a lockup-free cache using an inverted MSHR. The other curves correspond to more restricted and lower cost lockup-free implementations.

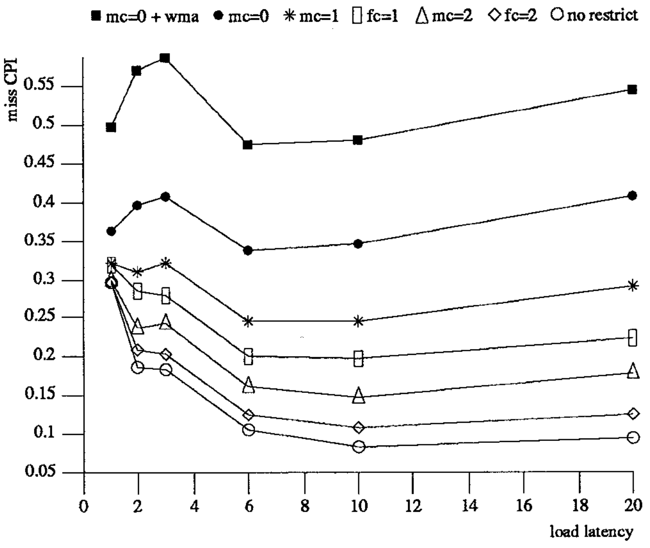

Figure 4-1: Baseline Miss CPI for *doduc*

The curve labeled "mc=1" (*one* outstanding *miss* to the *cache*) corresponds to a hit-under-miss scheme implemented using a single MSHR with one explicitly addressed field. A simple modification to this scheme is to employ an additional MSHR with only one destination address; this scheme allows two in-flight misses, one or both of which can be primary misses. The MCPI for this scheme is given by the curve "mc=2". To support multiple secondary misses but only one fetch operation, the hit-under-miss hardware could be replaced with a single explicitly-addressed MSHR with many destination addresses. The MCPI incurred with this implementation is given by the curve labeled "fc=1" (*one fetch* outstanding to a *cache*), since one primary miss and many secondary misses can be outstanding, but they must all be satisfied by the same cache line refill. For now we assume an infinite number of fields in the MSHR thereby supporting an unlimited number of secondary misses; we will consider the effects from limiting the number later. The final curve corresponds to an implementation with two such MSHRs and is the most complex of the restricted implementations.

Note that all the lockup-free implementations achieve very similar MCPIs for a load latency of 1. This fact, as will be discussed below, is a consequence of the algorithm used to schedule the code. The lockup-free implementations, however, achieve different MCPIs for load latencies bigger than the cache-hit latency. The simplest, hit-under-miss, incurs 2.9 times the MCPI of the unrestricted cache for a scheduled load latency of 10. If the hit-under-two-miss scheme is employed, this factor drops to 1.7, a significant improvement yet one which incurs little additional hardware complexity.

Consider the relative position of the "fc=1" and "mc=2" curves. This ordering indicates that *doduc* benefits more from allowing two primary misses to be in-flight than from allowing unlimited secondary misses to a single block being fetched. This fact is true for many of the other benchmarks. Finally, if the "fc=2" implementation was used instead, the MCPI incurred would be only 1.3 times the MCPI of the unrestricted implementation.

The curves in Figure 4-1 show two peculiarities that are attributable to the load latency. The first concerns the similar performance at a load latency of 1. With a load latency of 1, the compiler often schedules the instruction that uses the target register of a load immediately after the load instruction. Hence, it is rare for there to be more than one outstanding load and thus there is little to differentiate the lockup-free implementations. For *doduc*, we can compute the percent of the run-time that there is more than one miss outstanding from the numbers in Figure 4-2. This figure presents the in-flight miss and in-flight fetch histograms for *doduc* for each load latency and for a 16 cycle miss penalty. For a load latency of 1, there is at least one in-flight miss 27% of the time (the column labeled MIF), while for 92% of this time, there is only one miss. Thus for only $27\% \times (100\% - 92\%) = 2\%$ of the run-time is there more than a single miss outstanding.

load lat-ency	% time with >0 misses in flight (MIF)	in-flight	% of MIF for # in flight						
			1	2	3	4	5	6	7+	max #
1	27	misses	92	8	0	0	0	0	0	12
		fetches	95	5	0	0	0	0	0	5
2	25	misses	69	18	5	4	2	1	1	16
		fetches	80	14	3	1	1	1	0	13
3	27	misses	64	20	6	5	1	1	3	16
		fetches	80	15	3	1	1	0	0	14
6	22	misses	54	25	9	6	2	1	3	16
		fetches	75	18	4	2	1	0	0	13
10	22	misses	51	23	10	9	3	1	3	16
		fetches	70	20	6	2	1	1	0	14
20	26	misses	53	22	10	8	3	1	3	16
		fetches	73	18	5	2	2	0	0	14

Figure 4-2: Histogram of in-flight misses and fetches for *doduc*.

When longer load latencies are used, the compiler tries to insert instructions between the load-use pair and frequently these are load instructions. Hence, there will likely be more in-flight loads and hence more in-flight misses. This increase in outstanding misses and fetches can be seen in Figure 4-2. At a load latency of 20, for 12% of the run time there is more than one in-flight miss, which is 6 times more often than for a load latency of 1. The final column in the figure gives the maximum number of in-flight misses and fetches for the entire run of the benchmark. The maximum number of fetches never exceeds 16 since only one load can be issued in a cycle and the miss penalty is 16. The histograms for *doduc* represent an average case among the 18 benchmarks.

Because the compiler tries to increase the distance between the load and the first instruction to use its result, with longer load latencies there will be a decrease in true-data dependency stalls and a possible increase in the number of structural hazard stalls. This tradeoff is illustrated in Figure 4-3 which shows the portion of the MCPI that is attributable to structural-hazard induced stalls. This percentage is higher for longer load latencies. Note that when a compiler schedules for a load hit on a machine that can issue multiple instructions per cycle and has a cache-hit latency longer than one cycle, the compiler is already scheduling for load latencies greater than one. (More details on scaling our results to multi-issue machines is given in Section 6.)

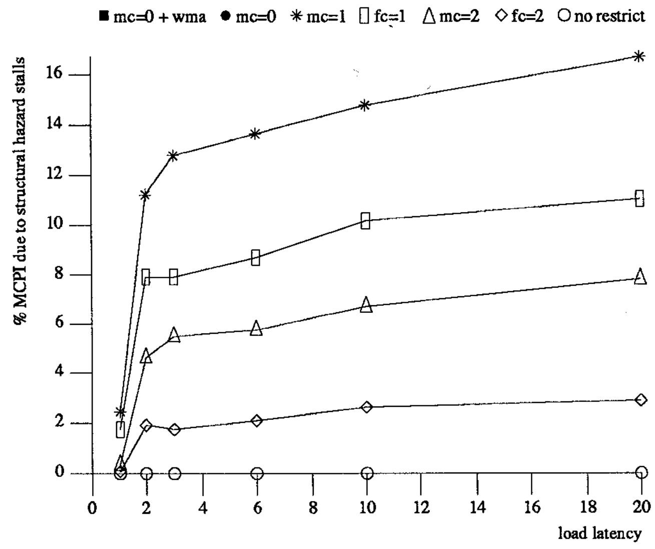

Figure 4-3: Stall cycle breakdown for *doduc*

The second peculiarity concerns the dip in the MCPIs that occurs at a load latency of 6. This dip occurs mainly because the primary and secondary cache miss rate also decreases at this value. This decrease is shown in Figure 4-4 which gives the combined primary and secondary miss rate as well as the secondary miss rate for the various implementations. The rate changes are attributable to the instruction movement and the grouping of load instructions which the compiler performs when trying to schedule for longer load latencies. When several misses are scheduled in close proximity, it is possible that some of these will access data that maps to the same line in the cache. Hence, while the compiler is trying to schedule the code to better tolerate cache misses, the conflict-miss rate may increase. The dip seen at a load latency of 6 corresponds to a code schedule that contains fewer conflict misses. Such discontinuities also exist with many of the other benchmarks.

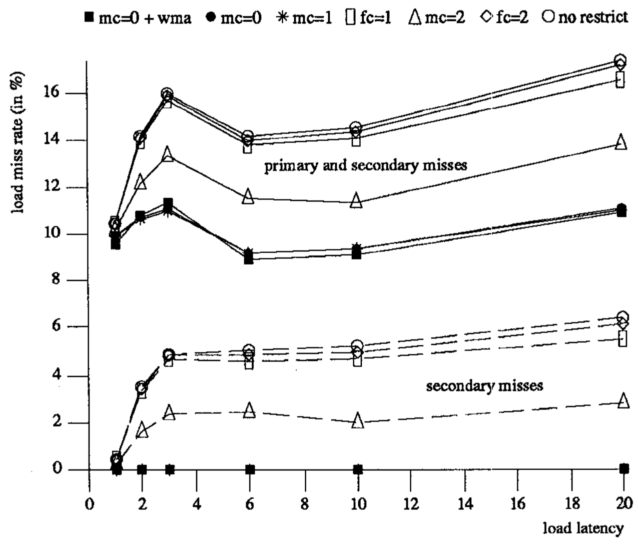

Figure 4-4: Baseline Miss rate for *doduc*

The MCPI graph for *doduc* suggests that *doduc* is able to take advantage of the more sophisticated lockup-free implementations. However, the more sophisticated implementations are not always necessary as the results for *xlisp* illustrate. Figure 4-5 shows the equivalent graph to Figure 4-1 for *xlisp*. The proximity of the curves for the lockup-free cache implementations suggests that the simple hit-under-miss implementation achieves near-optimal performance. In fact, compared to the unrestricted implementation, it incurs only 1.06 times the MCPI for a load latency of 10. The increasing MCPI beyond scheduled load latencies of 2 is primarily due to the conflict-miss problem described above. When the effect of conflicts is removed by using a fully associative cache, the curves become flat (see Figure 4-6). Note that the absolute MCPI is reduced by a factor of two to three by using a fully-associative cache in comparison to the direct-mapped case, due to the high percentage of conflict misses in Figure 4-5. Nonetheless, the same ordering of the MCPI incurred by the non-blocking implementations is maintained.

Another benchmark that does not require a more complex implementation than hit-under-miss is *eqntott*. The MCPI graph for *eqntott* is presented in Figure 4-7. As suggested by the small difference in the MCPI incurred by the different implementations, *eqntott*'s MCPI is dominated by true data dependency stalls; structural hazard induced stalls account for less than 1% of the MCPI. The discontinuity at a load latency of 3 is another manifestation of the effect that produces the dip at a load latency of 2 for *xlisp*.

As shown in Figure 4-8, the *tomcatv* benchmark incurs MCPI values that are an order of magnitude larger than those incurred by *eqntott*. The relative ordering of the curves for the various implementations is the same as those for the benchmarks presented above. Unlike *doduc*,

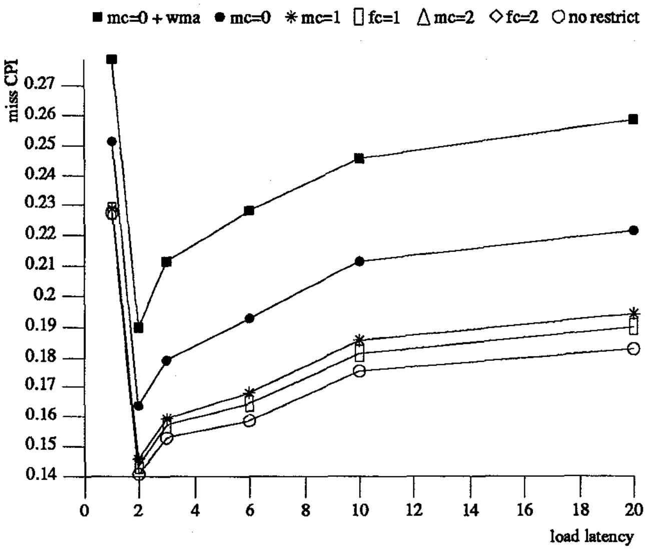

Figure 4-5: Baseline Miss CPI for *xlisp*

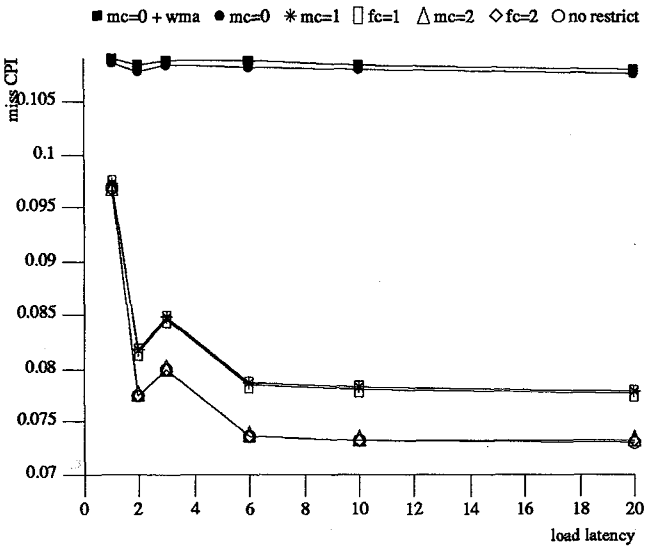

Figure 4-7: Baseline Miss CPI for *eqntott*

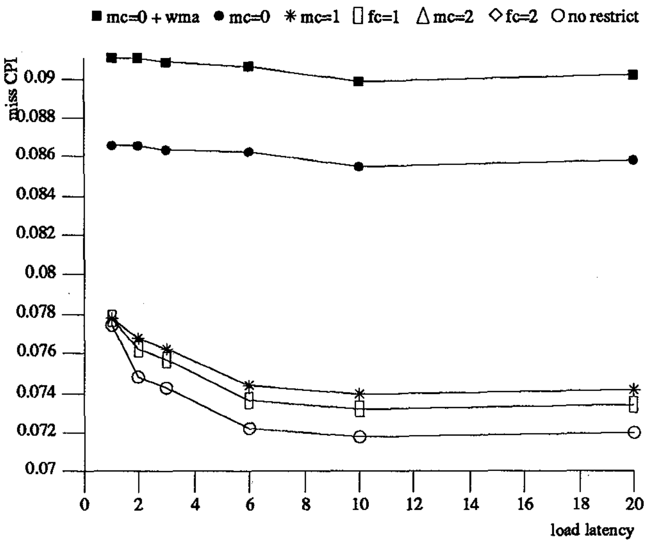

Figure 4-6: Miss CPI for *xlisp* with a fully associative cache

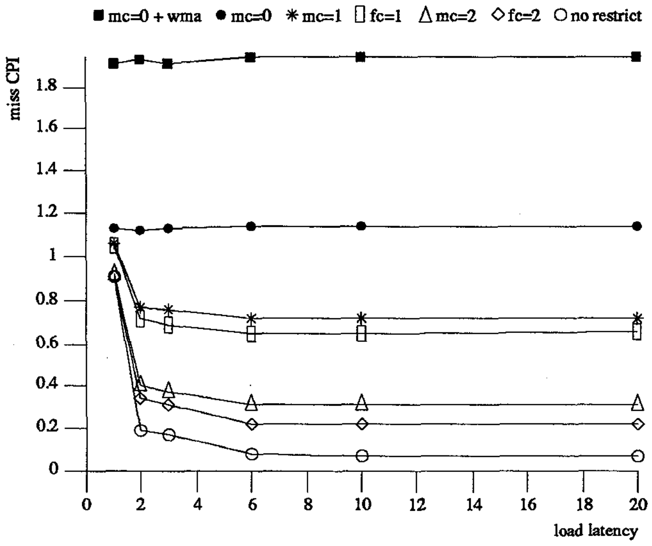

Figure 4-8: Baseline Miss CPI for *tomcatv*

however, *tomcatv* incurs an almost constant MCPI for load latencies 6 and larger. *Tomcatv* contains two nested loops which are unrolled many times by the compiler, and for load latencies of 6 and larger, the resulting code schedules are nearly identical. The performance for *tomcatv* as the load latency is varied corresponds to what intuition suggests would occur, namely, the MCPI monatomically decreases and the rate of decrease is smaller as the load latency becomes larger. This intuitive behavior is usually not exhibited by the other benchmarks, due to changes in the in-flight load profile brought about by changes in the load latency.

The above discussion focused on the baseline performance for five of the 18 SPEC benchmarks we investigated. These five were chosen to illustrate typical MCPI trends. The performance data for the various hardware organizations for all 18 SPEC92 benchmarks is presented in Table 4-9. This table gives the MCPI for each hardware organization and the ratio of this MPCI value to that for the "no restriction" organization. (The MCPI for the no restriction organization is given in the column labeled "∞".) As can be seen from the table, for a large number of the benchmarks, very good performance is obtained with the simpler implementations.

4.1. Implicit vs. Explicit Addressing

In the MCPI figures shown above, the curves labeled "fc=1" and "fc=2" correspond to lockup-free caches supporting an infinite number of in-flight secondary misses. However, implicitly addressed MSHRs limit the number of outstanding misses to one per sub-block where a sub-block is the number of bytes of the cache line for which there is

	mc=0		mc=1		mc=2		fc=1		fc=2		∞
	mcpi	×	mcpi	×	mcpi	×	mcpi	×	mcpi	×	mcpi
alvinn	0.494	1.4	0.398	1.1	0.371	1.0	0.394	1.1	0.367	1.0	0.365
doduc	0.346	4.1	0.245	2.9	0.147	1.7	0.197	2.4	0.109	1.3	0.084
ear	0.094	2.0	0.067	1.5	0.050	1.1	0.067	1.5	0.050	1.1	0.048
fpppp	0.434	7.1	0.234	3.8	0.119	1.9	0.197	3.2	0.091	1.5	0.062
hydro2d	0.708	3.7	0.466	2.5	0.246	1.3	0.457	2.4	0.242	1.3	0.189
mdljdp2	0.314	1.9	0.231	1.4	0.193	1.2	0.227	1.4	0.190	1.1	0.167
mdljsp2	0.154	3.4	0.088	1.9	0.057	1.2	0.070	1.5	0.052	1.1	0.046
nasa7	1.865	3.6	1.452	2.8	0.753	1.5	1.360	2.6	0.670	1.3	0.519
ora	1.000	1.0	1.000	1.0	1.000	1.0	1.000	1.0	1.000	1.0	1.000
su2cor	1.266	14	1.055	11	0.437	4.7	1.002	10	0.394	4.2	0.093
swm256	0.297	4.4	0.110	1.6	0.070	1.0	0.109	1.6	0.069	1.0	0.067
spice2g6	1.092	1.2	0.958	1.1	0.903	1.0	0.945	1.1	0.896	1.0	0.891
tomcatv	1.140	17	0.714	11	0.310	4.7	0.649	9.8	0.219	3.3	0.066
wave5	0.277	2.6	0.194	1.8	0.132	1.2	0.183	1.7	0.126	1.2	0.107
compress	0.453	1.3	0.354	1.0	0.349	1.0	0.351	1.0	0.348	1.0	0.348
eqntott	0.108	1.5	0.078	1.1	0.073	1.0	0.078	1.1	0.073	1.0	0.073
espresso	0.209	1.2	0.176	1.0	0.170	1.0	0.174	1.0	0.170	1.0	0.169
xlisp	0.211	1.2	0.185	1.1	0.176	1.0	0.181	1.0	0.176	1.0	0.176

Figure 4-9: Baseline MCPI for 18 SPEC92 benchmarks.

one destination tag. On the other hand, explicitly-addressed MSHRs allow several misses per sub-block, but have only one sub-block per line. Simulations of *doduc* and *tomcatv* were performed to investigate the performance tradeoffs between these two types of MSHRs; simulations of *eqntott* and *xlisp* were not undertaken as both of these incur near optimal MCPI values with the hit-under-miss implementation. Figure 4-10 presents the results of the simulations for *doduc*. This table gives the MCPI incurred and the ratio of the MCPI to that of the unrestricted cache for several implementations of the baseline cache and a scheduled load latency of 10; the MCPI for the unrestricted cache is given in the row marked by the symbol for infinity. The first column in the table corresponds to a cache using an implicitly-addressed MSHR whereas the first row corresponds to the use of an explicitly-addressed MSHR; non-edge entries correspond to a hybrid of these two.

number of sub-blocks	misses per sub-block					
	1		2		4	
	MCPI	ratio	MCPI	ratio	MCPI	ratio
1	0.150	1.79	0.114	1.36	0.085	1.01
2	0.144	1.72	0.086	1.03		
4	0.091	1.09				
8	0.084	1.00				
∞	0.084	1.00				

Figure 4-10: Explicit, implicit, and hybrid MSHRs for *doduc*

As can be seen in Figure 4-10, *doduc* incurs a MCPI within 1% of the unrestricted cache for caches employing either an explicitly-addressed MSHR with 4 misses per line, or an implicitly-addressed MSHR with 8 sub-blocks per line. This 8 sub-block per line granularity corresponds to an entry for every 4 bytes of the cache line. Using the hardware configurations given in Figures 2-1 and 2-2, the hardware costs for a MSHR using 8 implicit addresses is 44+(8×12)=140 bits plus a comparator and control logic. For the 4-entry explicitly addressed MSHR, it is 44+(4×17)=112 bits plus a comparator and control logic. The hybrid approach given in the table (sub-blocks=2, misses=2) offers slightly worse performance but at a cost of 44+(4×16)=106 bits plus a comparator and control logic. (The hybrid organization needs one less address bit in its "address in block" field because it is supplied by the implicit subblock location.)

4.2. In-cache MSHR Storage

Consider the effect of allowing more than one in-flight primary miss per cache set. A direct-mapped in-cache MSHR storage implementation is limited to one primary miss per cache set because the set itself is used to store the MSHR information. However, implementations based on conventional discrete MSHRs can support more than one fetch for a particular cache set once there is more than one MSHR. While many benchmarks, such as those discussed above, achieve a nearly optimal MCPI with only 1 fetch per set, other benchmarks do not. Typical of those that do not is *su2cor*. Figure 4-11 presents the baseline cache configuration simulations for *su2cor*. In this figure, the curves labeled "fs=" correspond to lockup-free implementations that support the specified number of in-flight fetches to a cache set. Thus, for our baseline system with an 8KB data cache and 32 byte line size, "fs=1" would allow up to 256 fetches outstanding, but only one per set in the cache. For a load latency of 10, allowing one fetch per set incurs 2.3 times the MCPI of the unrestricted non-blocking implementation whereas two fetches per set incurs 1.3 times the MCPI. It is clearly advantageous to support multiple fetches per cache set for *su2cor*. By implementing the in-cache MSHR storage method in a set-associative cache, more than one fetch per set could be in progress simultaneously. However, by implementing a set-associative cache, most of these concurrent conflict misses might be eliminated in the first place.

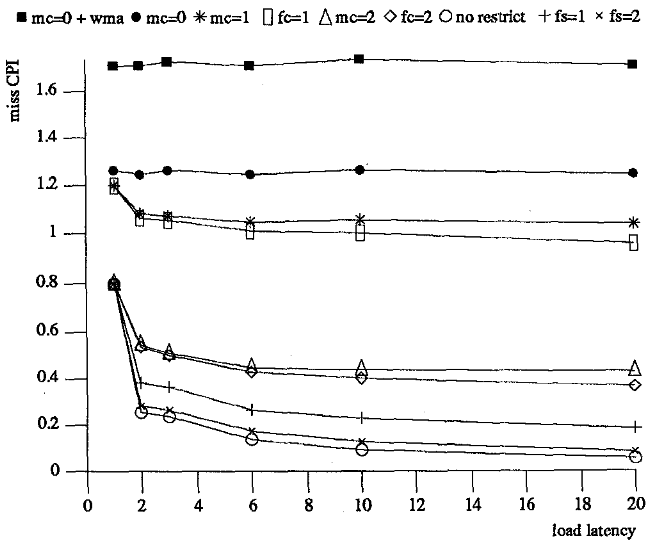

Figure 4-11: Baseline Miss CPI for *su2cor*

5. Variations on the Baseline Configuration

In this section we consider the effects of variations in cache size, cache line size, and miss penalty over the baseline cache configuration performance.

5.1. Variations in Cache Size

The previous results have been for a first-level data cache size of 8KB. In this section we consider the question of how a larger cache size affects the relative benefits of supporting more outstanding primary and secondary misses. Cache miss rates decrease with increases in cache size, resulting in a reduction in the miss CPI incurred by all machine configurations. This reduction might significantly reduce the clustering of misses. If this is true, there may not be significant enough clustering of misses to result in any performance improvement due to more aggressive non-blocking organizations.

Figure 5-1 shows the simulation results for *doduc* with a 64KB cache with 32 byte lines and 16 cycle miss penalty. Although the miss CPI has been reduced by about a factor of five in comparison to the results for 8KB caches in Figure 4-1, the graphs look remarkably similar. This observation indicates that there is still about the same percentage of misses that can be overlapped, even if the total number of misses has been much reduced. Thus, although the absolute performance improvement due to each non-blocking load organization is about a factor of five smaller, more aggressive organizations still provide additional benefits over simpler organizations.

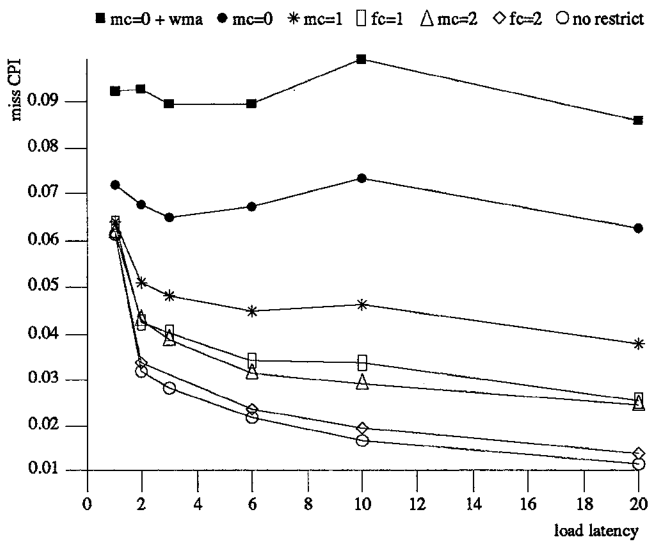

Figure 5-1: Miss CPI for *doduc* with a 64KB data cache

We have also looked at the performance of the other benchmarks with 64KB caches. In general the overall shape of the graphs for the other benchmarks are also similar to those for 8KB caches, although the absolute miss CPIs may be much lower. We have not looked at cache sizes larger than 64KB, since we are limiting our studies to first-level cache configurations which are feasible for on-chip implementation.

5.2. Variations in Cache Line Size

We have also investigated the effects of variations in the cache line size. One would expect that for larger line sizes, organizations that provide an unlimited number of secondary misses per line being fetched will do better in comparison to organizations that only support one miss per cache line. Similarly, for smaller cache line sizes, one would expect systems that support more primary misses at the expense of reduced support for secondary misses to perform relatively better. In our simulations, we have seen these effects, but their magnitude was smaller than we had anticipated.

For our comparisons we assumed a pipelined memory system with 14 cycles for the return of the first 16 bytes on a miss and 2 cycles per additional 16 bytes. Thus the miss penalty for systems with 16 byte lines was 14 cycles, and the miss penalty for systems with 32 byte lines was 16 cycles. Figure 5-2 shows the miss CPI for *doduc* with a 16 byte line size. Some differences can be seen between this figure and Figure 4-1, which uses 32 byte lines. First, the miss CPI increases slightly for all configurations using 16 byte lines because the 32 byte line size is a better choice given the pipelined memory system. However, the absolute values of the CPI should be ignored for the purposes of this comparison. Instead, looking at the relative performance of "mc=1", "mc=2", and "fc=1", in Figure 4-1, the "fc=1" case is about midway between the "mc=1" and "mc=2" cases. If 16 byte lines are used (Figure 5-2), the miss CPI incurred by "fc=1" moves closer to "mc=1" than "mc=2" (i.e., gets relatively worse). This is to be expected since the cache lines are smaller, so the benefit from supporting an unlimited number of secondary misses to a given cache line is less. In the limit as the cache line size is reduced to a single word, the "fc=1" organization will have the same miss CPI as the "mc=1" organization. We have seen this curve-movement effect when simulating the other benchmarks as well.

5.3. Variations in Miss Penalty

The above discussions have assumed a constant miss penalty of 16. Changing the miss penalty can affect the MCPI incurred by the benchmarks through two mechanisms. First, with longer miss penalties, there is likely to be a larger number of load instructions executed during the time required to service a miss, and thus there is a larger potential number of in-flight misses. With more in-flight misses, more structural stalls may occur. Second, longer miss penalties increase both the likelihood and the length of true-data-dependency stalls, while shorter latencies decrease both. To illustrate how the miss penalty affects the MCPI, we shall present some data for *tomcatv* which best illustrates the changes. Table 5-3 gives the miss CPI when using a scheduled load latency of 10 cycles. (Scheduling for load latencies greater than 10 has little effect, as Figure 4-8 shows.) The important thing to note is that for non-blocking organizations, the increase in miss

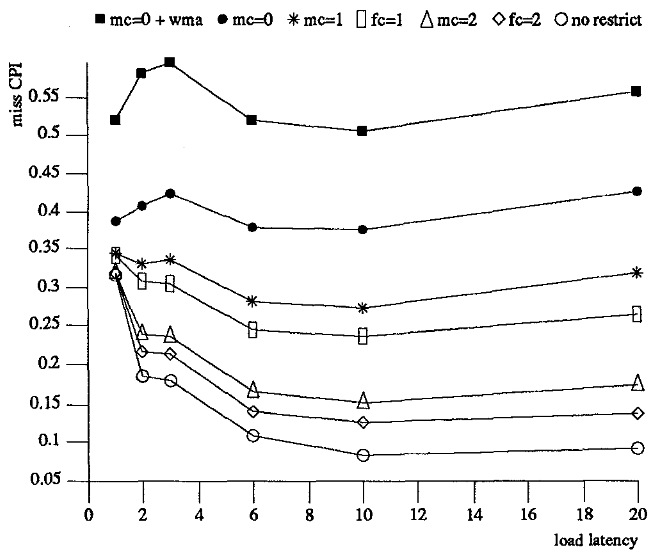

Figure 5-2: Miss CPI for *doduc* with 16 byte lines

CPI when moving from a small miss penalty to a large miss penalty is highly non-linear. This is especially true for the most aggressive implementations. For small miss penalties, virtually all of the miss penalty can be overlapped with computation, so the miss CPI remains very small. As the miss penalty is increased to large values, a higher and higher percentage of each miss penalty increase directly affects the miss CPI because the amount of possible overlap between misses and computation becomes exhausted. For example, for the unrestricted case when moving from a miss penalty of 16 cycles to 32 cycles (a factor of 2 increase), the miss CPI increases by almost a factor of five. In contrast, the blocking organization's (mc=0) miss CPI is strictly a linear function of the miss penalty.

	miss penalty					
	4	8	16	32	64	128
mc=0+wma	0.483	0.967	1.934	3.868	7.736	15.472
mc=0	0.285	0.570	1.140	2.280	4.561	9.122
mc=1	0.127	0.300	0.714	1.596	3.494	7.469
fc=1	0.111	0.258	0.649	1.511	3.408	7.381
mc=2	0.030	0.097	0.310	0.803	1.939	4.376
fc=2	0.021	0.069	0.219	0.641	1.676	3.866
no restrict	0.001	0.013	0.066	0.300	0.928	2.226

Figure 5-3: MCPI as a function of the miss penalty for *tomcatv*.

6. Applying the Results to Specific Machines

The processor and memory models we employ (see Section 3.1) were chosen to isolate the performance obtainable with non-blocking hardware from other machine-specific issues. It is possible to interpret our results in the context of specific machines by scaling the input parameters and adjusting the resulting CPI.

For machines with a limited number of write ports, the MCPI values may need to be increased. With a limited number of write ports, more time may be required to complete a line fill once the block is returned to the cache. This additional time might affect performance by increasing the length of true data dependency stalls or may increase the number of structural hazard induced stalls. Such an increase could occur because the MSHR may be in use for longer amounts of time. The correction factor should take into account these effects and would be based on the ratio of secondary to primary misses. Note that this correction would only be a first-order approximation because the presence of other stalls changes the load-miss profile. In practice, this correction factor is probably not significant enough to be included. Our simulations of the 18 benchmarks have shown that most of the time there is only a few misses outstanding. This fact is illustrated by the in-flight miss and fetch histograms for *doduc* which are shown in Figure 4-2.

To interpret the results for superscalar machines, the simulation parameters can be scaled based on the average number of instructions issued per cycle (IPC) in the superscalar machine. The miss penalties of the superscalar machine should be multiplied by the average IPC of the superscalar machine to get the miss penalties corresponding to this study. Similarly, the latencies for which loads are scheduled for the superscalar machine should be multiplied by the average IPC to get load latencies corresponding to our results. Then the results we have presented for the scaled miss penalty and load latency can be used as a first-order approximation for the MCPI for the multi-issue machine.

To gauge the accuracy of this scaling, we compared simulations of the 18 benchmarks on a dual-issue machine[2] to those on a single-issue machine. This comparison was performed by first simulating the execution of each benchmark on the dual-issue machine using a load latency of 10 and a miss penalty of 16. These parameters were then scaled using the average IPC for each benchmark and a single-issue machine simulation was done using the new parameters. Because it was not convenient to compile the code for all values of the load latency, we used the load latency from the set {1,2,3,6,10,20} that was closest to the scale value; the miss penalty was rounded to the nearest whole number.

The results from this comparison for several of the benchmarks are presented in Table 6-1 for four non-blocking load implementations. In general, the scaling results in a good first-order approximation. This is especially true when considering the coarseness of the approximation.

[2]This processor could dual issue all instructions except for branches and all instructions had a single cycle latency.

	IPC	scaled		measured dual-issue MCPI and difference							
		load	miss	mc=0		mc=1		fc=2		no rest.	
		lat.	pen.	mcpi	%	mcpi	%	mcpi	%	mcpi	%
doduc	1.59	15.9	25.5	0.59	-10	0.50	-7	0.23	-6	0.20	-8
eqntott	1.16	11.7	18.6	0.13	-2	0.10	1	0.09	1	0.09	2
su2cor	1.76	17.6	28.1	2.21	1	2.00	5	0.82	10	0.22	21
tomcatv	1.82	18.2	29.1	2.08	1	1.48	4	0.59	7	0.27	28
xlisp	1.30	13.0	20.9	1.24	-14	1.21	-14	1.20	-14	1.20	-14
average	1.53	15.3	24.5	1.25	-5	1.06	-2	0.59	0	0.40	6

Figure 6-1: Dual and single issue MCPI scaling comparison.

7. Conclusions

We have studied a wide range of techniques for implementing non-blocking loads. These techniques have ranged from organizations that allow only one outstanding miss to organizations which allow as many misses as there are possible load destinations in the machine. Non-blocking loads are a very powerful technique for tolerating cache miss latency. In our baseline system configuration with a 8KB direct-mapped data cache, 32 byte lines, and a 16 cycle miss penalty, non-blocking load implementations can reduce the miss stall CPI of integer benchmarks by up to a factor of two, and can reduce the miss stall CPI of many numeric benchmarks by a factor of 4 to 10.

For integer benchmarks, a simple hit-under-miss organization is the most cost effective as it achieves a performance comparable to organizations that allow an unbounded number of in-flight misses. On the other hand, the most cost-effective organizations for many of the numeric benchmarks are those that permit several in-flight primary and secondary misses. For some benchmarks, when using a direct-mapped cache it is worthwhile to support multiple misses to different addresses which map to the same cache set.

Surprisingly, we see very similar relative improvements with the addition of non-blocking loads to larger caches. Even though the miss rates may be significantly reduced by using larger caches, the remaining misses are still clustered enough that supporting many simultaneously outstanding misses results in large proportional changes in the miss stall CPI for numeric programs.

As we expected, with 16 byte cache lines rather than 32 byte lines, more benefit is obtained from supporting a greater number of primary misses than secondary misses; the opposite is true for cache lines larger than 32 bytes. Regardless of the cache line size, for lockup-free caches, the miss stall CPI varies non-linearly with the miss penalty.

Finally, our results point out the importance in non-blocking systems of scheduling load instructions wherever possible for cache misses instead of cache hits. An aggressive compiler that uses trace-scheduling and/or other techniques for increasing instruction-level parallelism is crucial to getting enough flexibility to schedule for the longer cache miss latencies.

Acknowledgements

We thank Joel Emer, Bob Nix and David Web for their guidance and answers to numerous questions as we modified and used the simulation infrastructure they developed. We also thank Jeff Mogul, Joel McCormack, Annie Warren and the other WRL-ites for both helping out and putting up with the simulations. Finally, we thank Paul Chow, Corinna Lee, Zvonko Vranesic and the anonymous reviewers for their useful comments.

References

1. Tom Asprey, et. al. "Performance Features of the PA7100 Microprocessor". *IEEE Micro 13*, 3 (June 1993), 22-35.

2. Tien-Fu Chen and Jean-Loup Baer. Reducing Memory Latency via Non-blocking and Prefetching Caches. Fifth ASPLOS Conference, October, 1992, pp. 51-61.

3. Keith Diefendorf and Michael Allen. "Organization of the Motorola 88110 Superscalar RISC Microprocessor". *IEEE Micro 12*, 2 (April 1992), 40-63.

4. Manoj Franklin and Gurindar Sohi. *Non-Blocking Caches for High-Performance Processors.* Unpublished, 1991.

5. Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Hiding Memory Latency using Dynamic Scheduling in Shared-Memory Multiprocessors. The 19th Intl. Symp. on Computer Architecture, May, 1992, pp. 22-33.

6. Norman P. Jouppi. Cache Write Policies and Performance. The 20th Intl. Symp. on Computer Architecture, May, 1993, pp. 191-201.

7. David Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. The 8th Intl. Symp. on Computer Architecture, May, 1981, pp. 81-87.

8. P. Geoffrey Lowney et al. "The Multiflow Trace Scheduling Compiler". *Journal Of Supercomputing 7*, 1-2 (May 1993), 51-142.

9. Edward McLellan. "The Alpha AXP Architecture and 21064 Processor". *IEEE Micro 13*, 3 (June 1993), 36-47.

10. Sunil Mirapuri, Michael Woodacre, and Nader Vasseghi. "The MIPS R4000 Processor". *IEEE Micro 12*, 2 (April 1992), pages 10-22.

11. Anne Rogers and Kai Li. Software Support for Speculative Loads. Fifth ASPLOS Conference, October, 1992, pp. 38-50.

12. Gurindar Sohi and Manoj Franklin. High-Bandwidth Data Memory Systems for Superscalar Processors. Fourth ASPLOS Conference, April, 1991, pp. 53-62.