Memory-System Design Considerations for Dynamically-Scheduled Processors

Keith I. Farkas† farkas@eecg.toronto.edu

Paul Chow† pc@eecg.toronto.edu

Norman P. Jouppi‡ jouppi@pa.dec.com

Zvonko Vranesic† zvonko@eecg.toronto.edu

†Electrical and Computer Engineering
University of Toronto
10 Kings College Road
Toronto, Ontario M5S 3G4 Canada

‡Digital Equipment Corporation
Western Research Lab
250 University Avenue
Palo Alto, California 94301 USA

Abstract

In this paper, we identify performance trends and design relationships between the following components of the data memory hierarchy in a dynamically-scheduled processor: the register file, the lockup-free data cache, the stream buffers, and the interface between these components and the lower levels of the memory hierarchy. Similar performance was obtained from all systems having support for fewer than four in-flight misses, irrespective of the register-file size, the issue width of the processor, and the memory bandwidth. While providing support for more than four in-flight misses did increase system performance, the improvement was less than that obtained by increasing the number of registers. The addition of stream buffers to the investigated systems led to a significant performance increase, with the larger increases for systems having less in-flight-miss support, greater memory bandwidth, or more instruction issue capability. The performance of these systems was not significantly affected by the inclusion of traffic filters, dynamic-stride calculators, or the inclusion of the per-load non-unity stride-predictor *and the* incremental-prefetching *techniques, which we introduce. However, the incremental prefetching technique reduces the bandwidth consumed by stream buffers by 50% without a significant impact on performance.*

1 Introduction

Dynamically-scheduled processors offer much greater tolerance for data-cache misses than do statically-scheduled processors. This increased tolerance is provided by the ability to issue instructions in an order different from that in which they were fetched whenever a hazard prohibits in-order issue. Cache misses may induce data and structural hazards that involve the instructions that are waiting to be issued. The degree of tolerance represents a balance between the number instructions that are not affected by such hazards, and the time required to resolve the miss. This balance exists because the longer a cache miss takes to be resolved, the greater the number of instructions that are required to hide it.

The time required to resolve a miss is determined by the bandwidth of the memory interface that is situated between the data cache and the lower levels of the memory hierarchy, and by the time required to fetch a data item once a request for it has been issued. The ability to hide a cache miss is determined by the ability of the dynamic scheduler to issue unaffected instructions. The likelihood that such instructions are available to the scheduler is determined by a number of factors. Branch prediction is an important factor because its use allows the hardware to continue fetching instructions from beyond a branch for which the direction or destination (address) is not statically known. Speculative fetching of instructions provides the supply of (possibly) unaffected, new instructions. These instructions can be made available to the dynamic scheduler if the required hardware resources for processing the instructions are available.

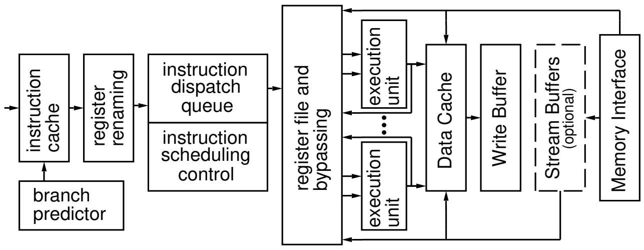


Figure 1: Overview of our dynamic scheduling implementation; only the data path is shown.

In our system model (Figure 1), the *dispatch queue* stores the instructions from which the dynamic scheduler chooses the instructions to issue next. To insert an instruction into the dispatch queue, there must be an available entry, and, if the instruction names an *architectural register* as a destination, there must be a *physical register* available to rename the named architectural register. If one of the required resources is not available, the process of inserting instructions must be stalled until a resource is available.

The availability of dispatch-queue entries and physical registers is affected by the number of miss-processing resources that are provided by the lockup-free data cache. The number of such resources places a limit on the number of cache misses that can be serviced concurrently. If a resource is required and none is available, the dynamic scheduler must stop issuing memory instructions until the required resource is available. Such stalls in the issuing of memory instructions can quickly lead to a full dispatch queue and a decrease in the rate of forward progress.

In this paper, our goal is to provide insight into the effectiveness of hardware-based techniques for reducing the apparent time cost of

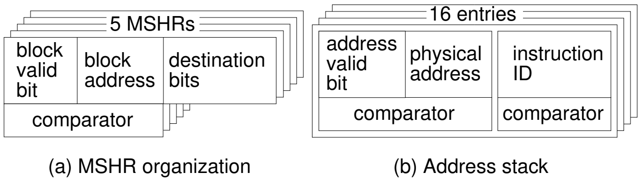


Figure 2: Functional block diagram of the proposed lockup-free cache implementation that supports 16 in-flight memory accesses and five in-flight cache-block fetches.

cache misses or increasing the tolerance for data-cache misses. The memory-system components that we consider are the register file, the lockup-free data cache, and the *stream buffers* [1], a technique for implementing hardware-based prefetching. We also examine the interface between these components and the lower levels of the memory hierarchy.

The presentation begins with a description of the lockup-free cache and stream buffer implementations we consider, followed by an overview in Section 3 of our system model and simulation methodology. Then, we examine the performance of various lockup-free data cache organizations in Section 4, and of various stream-buffer implementations in Section 5.

2 Hardware

This section describes, in the context of a dynamically-scheduled processor, the hardware that is required to implement a lockup-free data cache and the stream buffer implementations that we consider. A more complete discussion of these components is given in [2].

2.1 Lockup-free Cache

When the processor detects a data-cache miss, it must determine whether the cache-block containing the missing data is already being fetched, and it must resolve all pending cache misses when this cache-block is returned from the lower levels of the memory hierarchy. This functionality is provided by *miss status holding registers* (MSHRs) [3], but for dynamically-scheduled processors, additional functionality is required that is not present in statically-scheduled processors. This functionality allows selective cache misses to be suppressed while allowing others to be completed, thereby permitting the cancellation of speculatively executed memory instructions.

To provide the required functionality, we use a set of MSHRs and an *address stack*. The MSHRs are used to determine whether a cache block is already being fetched, whereas the address stack is used to resolve pending cache misses. As shown in Figure 2a, each MSHR has three fields. The "block valid bit" indicates whether the cache block with the address stored in the "block address" field is in the process of being fetched. When a data-cache miss is detected, the MSHRs are associatively searched to determine whether the cache block is already being fetched. If a match is detected, the cache miss is referred to as a *secondary miss*, whereas if no match is detected, the miss is referred to as a *primary miss* [3]. For a primary miss, a free MSHR is allocated, the block address is written, the "block valid bit" set, and a fetch request for the cache block is issued to the next level in the memory system. If there are no free MSHRs, no further memory requests can be issued until an MSHR is released by the return of previously-requested data; the load or store that caused this *structural-hazard-induced stall* will be replayed subsequent to the freeing of an MSHR.

When a cache-block is returned, an associative lookup of the MSHR structure is done to extract the "destination bits" (Figure 2a). This information indicates whether the cache block is to be returned to the data cache, the instruction cache, or perhaps neither because it is to be used to resolve an access to an uncached memory location. Some information must be returned with the data to facilitate this lookup. Since dynamically-scheduled processors often support cache consistency and thus have a mechanism for sending addresses to the processor from the memory system, a reasonable choice is to have the memory return the fetch-request address along with the data. After the destination bits are extracted, the cache-block is sent to the component that requested a copy.

To resolve data-cache misses once a block is returned, the address stack is used. The address stack (Figure 2b) is implemented as a fully associative buffer. After the hardware issues a load or a store and calculates the physical address for the memory access, it writes this address into a free address-stack entry, if there is one available. If there is no free entry, then a structural-hazard-induced stall is mandated, and the instruction will be replayed subsequent to the freeing of an entry. Once an address-stack entry is allocated, should the required data not be found in the cache, the valid bit in the address stack for the instruction is set, and if necessary, a cache-block fetch is initiated. When this block is returned from the lower levels of the memory system, at same time as the hardware writes it into the cache, it does an associative lookup of all the entries in the address stack. For each match, the control logic notes that the corresponding memory instruction can now be replayed. In a complex and more costly design, the address-stack entry allocated to a memory instruction M is freed once M is completed. In a simpler and less costly design, the address-stack entry is freed when M is retired.

In the event that a mispredicted branch or an exception mandates the flushing of M from the processor as part of the flushing process, the address-stack entry held by M is invalidated. Thus, if the instruction was waiting for a cache block to be fetched, when the block is returned, it is guaranteed that the action indicated by M will not be performed, because there is no matching entry in the address stack. If M is a load, the named physical register will not be written, whereas if M is a store, the data will not be written. In this way, the instructions affected by the branch misprediction or exception are suppressed while those preceding the faulting instruction can be completed normally. This ability to easily suppress selective cache misses makes an address stack more attractive for dynamically-scheduled processors than the more conventional methods for recording information about primary and secondary misses [4].

Although not described in the literature, it is likely that lockup-free caches are implemented in this way by the MIPS R10000 [5] using the address queue and address stack, the PA-8000 [6] using the memory buffer and the address-reorder buffer, and in the PowerPC 604 [7] and 620 [8] using the load queue.

2.2 Stream Buffers

A stream buffer is a hardware-based prefetching technique that can be used to prefetch and store data that might be required to resolve future data cache misses. In this paper, we extend the model described in [9] by including the provision for: (1) a new prefetch strategy, called *incremental prefetching*, which reduces the memory traffic generated by stream buffers, and (2) a new method for dynamic calculation of strides, called the *per-load stride predictor*.

Memory-traffic filtering has been implemented using the *allocation filter* technique proposed by Palacharla and Kessler [10]. This filter prevented a stream buffer from being allocated until the second miss to a stream is detected. On the second miss, a stream buffer was allocated and it began prefetching the block subsequent to the one corresponding to the second miss. Our proposed *incremental prefetching* technique differs from the allocation-filter technique in that it limits the number of blocks fetched after a stream buffer has

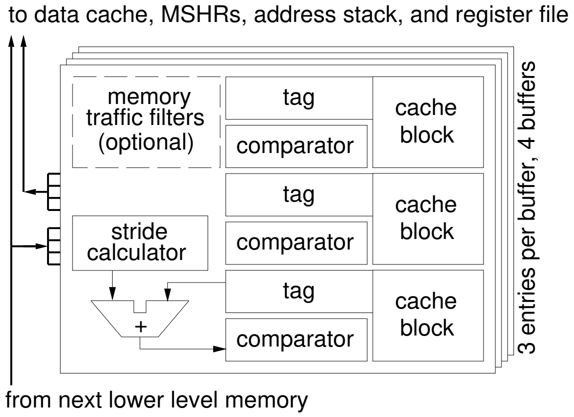


Figure 3: The enhanced stream-buffer model. In this example, there are four stream buffers with three entries each, and any one of the 12 entries can be used to supply the missing data for a cache miss.

been allocated until the stream is found to be useful. With incremental prefetching, when a stream buffer is first allocated, only one block is fetched. If the block is then used to service a cache miss, the next two blocks are fetched. This process of fetching the next $2 \times N$ blocks if one of the last N was used to resolve a cache miss continues until the stream buffer is reallocated.

Dynamic-stride prediction has been implemented using a scheme based on the *minimum delta* scheme proposed by Palacharla and Kessler [10]. With this scheme, on a stream-buffer miss, the allocation filter was applied to determine whether a unit-stride should be used. If there was a filter miss, then the minimum signed difference between the miss address and the last N miss addresses was determined. This minimum delta, which may be positive or negative, was the stride. However, if the stride S was smaller than the size of a cache block, then a unit stride was used with the same sign as S. This strategy guards against the prefetch stream of a stream buffer overlapping with itself. A stream buffer was allocated if the miss was the third miss in a series to blocks that were separated by this stride. The stride predictor is shown in Figure 3, which depicts the enhanced stream-buffer model.

We introduce the *per-load stride predictor*. It differs from the minimum-delta scheme in that a stride is determined for a load instruction L by considering only the previous miss addresses generated by L. This predictor is based on the scheme proposed by Fu et al. [11] for preloading the data cache and is similar to the data prefetching scheme of Chen and Baer [12]. Our predictor uses a fully associative buffer to record the last miss address for N static loads, along with the program-counter address of each load. Thus, a stride prediction is based only on the past memory behavior of the static load for which the prediction is being made. We also implement an enhancement to provide the functionality of an allocation filter. With this enhancement, a stream buffer is allocated with a unit stride the first time a memory instruction M is executed for which there is no entry in the miss-address table and the required data (with address A) is not in the data cache or a stream buffer. Subsequently, a stream buffer is allocated only if $S = A_{n-1} - A_{n-2} = A_n - A_{n-1}$, where A_x is the address of the data for the x^{th} stream-buffer miss generated by M. By default, a stream buffer is not allocated on the second miss. As before, if the stride S is smaller than the size of a cache block, then a unit stride is used with the same sign as S.

3 Simulation Methodology

This section describes our investigation methodology, the system model we assume, and our simulation framework. The methodology was selected to allow us to identify performance trends and design relationships, rather than to estimate the performance of specific system designs.

3.1 System Model

Our system model implements a RISC superscalar processor whose instruction set is based on the DEC Alpha instruction set. We assume that all instructions can be be speculatively executed, and that the processor can issue 4 or 8 instructions per cycle. These issue widths are representative of the current state-of-the art and future processors. The issue rules for the 4-way and 8-way issue processors are given in rows 1 and 2 of Table 1. The processor has a standard four-stage execution pipeline, and, with the exception of the execution stage (stage 3), all stages have a single-cycle latency. The functional unit latencies are given in row 3 of Table 1.

In a clock cycle, the number of instructions that can be inserted into the dispatch queue is equal to 1.5 times the maximum issue width of the processor, while the maximum number of instructions that can be retired is exactly twice the issue width of the processor. These values were chosen to reduce the possibility of either instruction insertion or instruction commitment being a significant bottleneck. Instructions are selected for issuing using a greedy algorithm that issues the earliest instructions in fetch order first. Hardware is included to dynamically disambiguate memory addresses so as to allow memory instructions to issue before those occurring earlier in the program order. The register file includes a configurable and equal number of integer and floating-point registers. The number of registers considered for each issue width are given in Table 2a; this table also specifies the number of dispatch-queue entries. The register file for the four-way issue processor has eight read ports and sufficient write ports to prevent any write-port conflicts arising when registers are filled on the resolution of a cache miss. For the eight-way issue processor, there is twice the number of ports.

The model implements precise exceptions, and uses a branch prediction scheme proposed by McFarling [13] that comprises a bimodal predictor, a global history predictor, and a mechanism to select between them. The prediction scheme is used to predict the direction of conditional branches; all other control flow instructions are assumed to be 100% predictable.

The model includes separate instruction and data caches, and supports non-blocking loads and non-blocking stores. Stores are assumed to be implemented using write-around (i.e., no-write-allocate) and write-through policies with a write buffer situated between the data cache and lower levels in the data memory hierarchy. Since our goal was to include only the important features of a processor that affect the design of the register file and memory system, we assume that the servicing of instruction cache misses does not delay the servicing of data cache misses. Hence, the instruction cache has a fixed miss penalty. Furthermore, we assume that no memory bandwidth is required to retire stores in the write buffer. This assumption prevents any stalls due to a full write buffer and prevents stores from delaying the servicing of cache fetches.

The data cache is assumed to be a 64-KByte, two-way set associative cache that can be configured to be lockup, lockup-free, or perfect; the perfect organization assumes a 100% hit rate. These three organizations are listed in Table 2b. For the lockup-free cache, three types of in-flight miss restrictions are considered. The most restrictive type, designated m_x, limits the number of outstanding cache misses to be at most x, where x is an integer greater than zero. The second type, designated f_x is less restrictive because it imposes no limit on the number of secondary misses. Rather,

#		instruction types								
		total	integer			floating point			loads &	control
			total	multiply	other	total	divide	other	stores	flow
1	number issued	4	4	4	4	2	1	2	2	1
2	per cycle	8	8	8	8	4	2	4	4	2
3	latency in cycles			6	1		8/16	3	1†	1

Table 1: Instruction-issue rules and functional-unit latencies for the 4-way and 8-way issue processors. All functional units are fully pipelined with the exception of the floating-point divider. The divider is not pipelined and has an eight-cycle latency for 32-bit divides, and a 16-cycle latency for 64-bit divides. †There is a single load-delay slot.

(a) **processor details**	
issue width	# registers
4-way issue with 32-entry dispatch queue	48, 64, 96, or 128
8-way issue with 64-entry dispatch queue	64, 96, 128, or 256

(b) **memory details**		
cache type	abbrev.	fetch spacing
lockup	lk	0 8
lockup-free	m_x f_x i	
perfect	p	

(c)	**stream buffer details**			
#	stride predictor type	N_s	traffic filters	abbrev.
1	no stream buffers			N
2	unit		none (baseline)	S_b
3			incremental fetching	S_i
4			P&K allocation filters	S_f
5			P&K filters & incremental	S_{f+i}
6	min-delta	32	P&K allocation filters	A_d
7			P&K filters & incremental	A_{d+i}
8			none	P
9	per-load	10 or	stride filters	P_f
10		32	incremental fetching	P_i
11			incremental & stride filters	P_{f+i}

Table 2: System designs. A system is defined by selecting specific parameters from the processor, memory, and stream-buffer details tables. N_s in table (c) specifies the number of entries in the table of miss addresses.

it limits only the number of primary misses to be at most x. A cache with this functionality offers greater flexibility than one with the functionality offered by m_x, and therefore has a more aggressive implementation. Because a fetch request is required for each primary miss, the value of x for the f_x restriction also indicates the maximum number of outstanding cache-initiated fetch requests. The third type, designated i, is even less restrictive because it does not impose a limit on the number of primary or secondary misses. In real processors, the value of x for m_x might correspond to the number of address-stack entries, while the value of x for f_x might correspond to the number of MSHRs.

Requests for blocks of data are sent via the memory interface to the next level in the memory hierarchy. The memory interface returns the requested block in a constant number of cycles, called the *fetch latency*; we assume a 32-cycle fetch latency. The bandwidth of the interface is constrained by controlling the number of cycles between the launching of fetch requests. A *fetch spacing* of zero allows requests to be launched as soon as they are submitted, and thus, corresponds to an interface with a very large bandwidth. A *fetch spacing* of one allows the memory interface pipeline to be full whereas a spacing equal to the fetch latency allows at most one in-flight fetch. Thus, the time required to resolve a cache miss is not deterministic for non-zero fetch spacings but has a lower bound equal to the fetch latency; Table 2b specifies the two fetch spacings that we consider.

The 10 stream-buffer implementations (rows 2-11 in Table 2c) each comprise eight, four-entry stream buffers, and optional support for either memory-traffic filters, dynamic-stride prediction, or both. The implementation with none of this optional hardware is referred to as the *baseline* implementation, and for notational convenience it is designated as S_b. The abbreviations used for the other implementations are given in the table. We assume that one cycle is required to extract a block of data from a stream buffer.

When a block is returned to the cache, the cache line is written simultaneously with the writing of the appropriate words into all registers with loads outstanding to this block (updating all pending registers requires the multiple write ports mentioned above). Writing a register or a cache line is assumed to take one cycle.

Figure 4 presents an overview of the just-described system model.

3.2 Simulation Framework

This study is based on execution-driven simulations using an object code instrumentation system called ATOM [14], which is available for Alpha AXP workstations. The results presented correspond to simulations of seven benchmarks, six from the SPEC92 suite and the *appsp* benchmark from the NAS suite. The benchmarks are listed in Table 3 along with some run-time characteristics for the four-way and eight-way issue processors. For each of the six SPEC92 benchmarks, one of the official data sets was used (ref or small), and these are shown in Table 3 in column 2; for *appsp*, the data set that was used is described in the table caption. In all cases, the benchmarks were compiled using the Alpha native C compiler with the global ucode optimizer enabled, and the linker was directed to perform link-time optimizations.

The results in Table 3 are for a four-way issue processor and an eight-way issue processor. Both systems had a lockup-free data cache with no in-flight-miss restrictions, and an eight-cycle fetch spacing. The four-way issue processor had 64 registers (i.e., 64 integer and 64 floating point), while the eight-way issue processor had 96 registers.

Column 3 gives the number of instructions in the trace for each benchmark, which is equivalent to the number that commit. (An instruction is said to *commit* when it has completed *and* all the instructions preceding it in program order have completed.) The number of committed instructions does not necessarily equal the number of instructions that are executed due to mispredicted

bench-mark	data set	com-mit instr.	4-way issue						8-way issue					
			issue instr.		IPC		% load miss		issue instr.		IPC		% load miss	
			total	load	issue	commit	pri.	sec.	total	load	issue	commit	pri.	sec.
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)
appsp	note[a]	319	320	68	1.58	1.57	6.0	6.8	322	68	2.44	2.42	6.0	7.6
compress	ref	86	111	25	1.63	1.27	11.9	3.5	139	31	2.34	1.45	10.7	3.1
hydro2d	note[b]	237	238	54	1.45	1.44	12.6	21.1	240	55	2.52	2.49	12.5	36.4
mdljdp2	small	291	317	47	1.74	1.60	2.3	0.4	352	53	2.84	2.34	2.1	1.9
su2cor	small	417	432	106	1.88	1.81	8.4	8.1	445	109	2.63	2.46	8.2	14.3
swm256	ref	377	378	97	1.77	1.76	6.6	2.6	379	97	2.92	2.92	6.6	10.3
tomcatv	ref	910	911	247	1.63	1.63	11.2	21.1	911	248	2.40	2.40	11.2	28.4

Table 3: Dynamic statistics for each benchmark for both issue widths. Columns 3, 4, 5, 10, and 11 give instruction counts in millions; columns 8, 9, 14, and 15 give the primary and secondary data-cache miss rates for load instructions. Notes: (a) *appsp* was run for 50 iterations with a 12x12x12 grid; (b) *hydro2d* was run for 15 iterations rather than the 400 specified in the official "ref" data set.

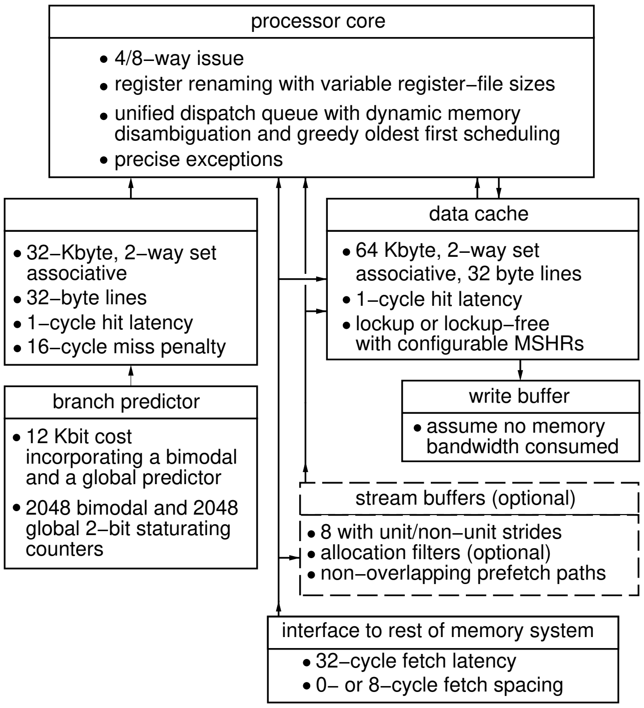


Figure 4: Overview of machine model.

branches (exceptions are not modeled). The number of executed instructions is given under columns 4 and 10 with columns 5 and 11 giving the number of loads. Both the number of committed instructions and the number of executed instructions are dynamic instruction counts.

The average number of instructions per cycle (IPC) for each benchmark and each issue width are given in columns 6, 7, 12, and 13. The *issue IPC*, given in columns 6 and 12, is the ratio of the number of instructions that are *issued* to the total (simulated) run time; the issue IPC measures the rate at which instructions are dispatched to the functional units. In our system model, the difference between the issue IPC and the maximum issue width is due to the dependences in the code and the number and type of functional units. The *commit IPC*, given in columns 7 and 13, is the ratio of the number of instructions that *commit* to the total run time. The difference between the issue IPC and the commit IPC is due to instructions that are incorrectly speculatively executed when following mispredicted branches.

The statistics presented in the table show that the benchmarks generate enough data-cache behavior to affect the performance of the memory-system implementations that we consider. First, the data presented in column 3 suggests that each trace contains a significant number of instructions, and second, the data presented in columns 4 to 5, 8 to 11, and 14 to 15 suggest that the benchmarks have significant data-cache behavior.

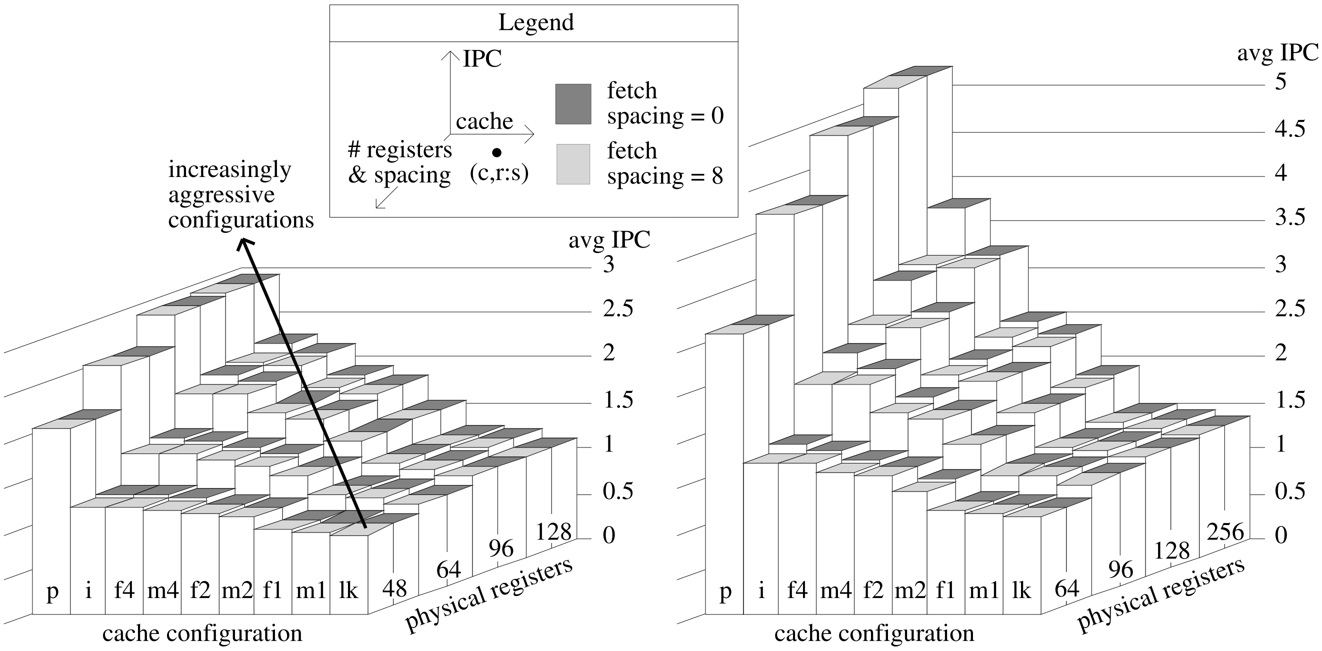
4 In-flight Cache Misses

To evaluate the design and performance implications of data-cache misses with different numbers of in-flight-miss resources, we evaluated performance for the system-design space listed in Tables 2a and 2b. Figures 5a and 5b respectively present the (overall) average commit IPC[1] obtained by the benchmarks on the systems with the four-way and eight-way issue processors. In both of these figures, coordinate $(c, r : s)$ gives the average commit IPC for a processor having a cache design c, a register-file size r and a fetch spacing s. The cache designs are mapped on the left-to-right axis with the less restrictive designs located to the left. The register-file sizes and fetch spacings are mapped on the front-to-back axis, with the larger register files located towards the back. For each register-file size, the IPC values for a zero cycle fetch spacing are behind those for an eight-cycle fetch spacing. Thus, the coordinates to the left and towards the back represent more aggressive system implementations.

Examination of the data presented in Figure 5 suggests a number of important relationships. These relationships are discussed below beginning with the relationship between performance, number of registers, and support for in-flight misses.

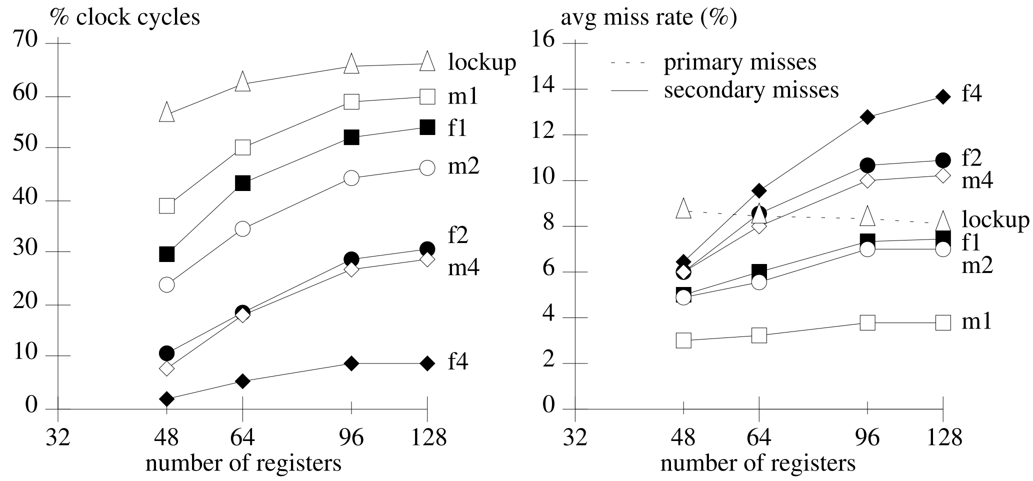
Consider the commit IPC values given in Figure 5a corresponding to the use of a lockup-free cache with no restrictions on in-flight misses, that is, coordinates $(c = \text{i}, r : s)$. Observe that the average commit IPC increases with the size of the register file. Furthermore, the rate of increase decreases at larger sizes. For instance, for a fetch spacing of eight cycles, doubling the size from 48 to 96 yields an improvement of 70%, while doubling the size from 64 to 128 yields an improvement of only 36%. In general, an increase in the number of registers permits more instructions to be in some stage of execution, thereby better utilizing the available hardware, and thus improving performance. However, this correlation between more registers and better performance is less pronounced for the systems with less support for in-flight misses. For example, if four in-flight fetches are permitted, $(c = f_4, r : s)$, doubling the size from 48 to 96 yields an improvement of 68% while doubling the size from 64 to 128 yields an improvement of 35%. If the support for misses is further reduced by allowing only four in-flight misses,

[1] The (overall) average commit IPC is calculated for each system configuration by first computing the average commit IPC for each benchmark for that system. The per-benchmark average commit IPC is equal to the total number of instructions that are committed when the benchmark is run divided by the total number of (simulated) clock cycles required to run it. Then, these per-benchmark averages are combined using an arithmetic average to obtain the reported (overall) average commit IPC. The commit IPC values for each benchmark are given in [15].



(a) 4–way issue processors (b) 8–way issue processors

Figure 5: Average commit IPC of all benchmarks for their (simulated) execution on the investigated systems.



(a) Percentage of (simulated) clock cycles in which a load-miss induced stall was in effect

(b) Primary and secondary load-miss rates. The primary miss rates for all lockup-free cache organizations are coincident with the curve for the lockup cache.

Figure 6: Statistics for load-induced cache misses for the four-way issue processors using an eight-cycle fetch spacing. Each figure contains a curve for each of the seven in-flight miss restrictions that were investigated.

$(c = m_4, r : s)$, the improvement percentages are 53% and 25%, respectively.

The data presented in Figure 5a also suggests that system performance is more heavily affected by increasing the number of registers than providing support beyond four in-flight misses. For example, consider the system $(c = m_4, r = 64 : s = 8)$, that is, one with support for four in-flight misses, 64 registers, and an eight-cycle fetch spacing. Compared to this baseline system, one having twice the number of registers $(c = m_4, r = 128 : s = 8)$ will perform 25% better, while one with no restrictions on in-flight misses $(c = i, r = 64 : s = 8)$ will perform only 5% better.

The above noted relationships between performance, number of registers, and support for in-flight misses are in part due to the increase in the number of structural-hazard-induced stalls that occur when the amount of in-flight miss support is reduced. To illustrate this increase in stalls, Figure 6a presents the percentage of (simulated) clock cycles in which such stalls occur in the systems using the four-way issue processors with an eight-cycle fetch spacing. In the figure, the percentages for each in-flight miss restriction are given by the seven curves as a function of the number of registers[2]. For instance, with 64 registers, the percentage for each in-flight miss restriction is: 5% for f_4, 18% for m_4, 19% for f_2, 34% for m_2, 43% for f_1, 49% for m_1, and 62% for a lockup cache.

The significant difference in the above percentages between f_x and m_x restrictions for the same value of x is due to the benchmarks requiring significant amounts of secondary-miss support. This requirement is suggested by the miss rates presented in Figure 6b. As can be observed, while the primary miss rates for all systems having a given number of registers are sufficiently similar that the data points are coincident, there are significant differences in the secondary-miss rates. With less support for in-flight misses, the load instructions that correspond to these secondary misses must instead be held in the dispatch queue. Moreover, because memory instructions cannot be issued during a structural-hazard-induced stall, the dispatch queue will tend to fill up with memory instructions for which the data is already in the data cache. Thus, the rate at which entries are freed and new instructions are inserted will decrease as the stall progresses. Together, these two effects reduce the demand for registers and the performance gains that accompany increasing the number of registers. As the data in Figure 6 suggests, this reduction is more significant with larger numbers of registers because the secondary miss rates are larger, and because structural-hazard-induced stalls occur for a greater percentage of the (simulated) run-time.

The trend of decreasing performance with less lockup-free support is more pronounced with the systems having eight-way issue processors. Data supporting this trend is presented in Figure 5. Observe that there is less variance in the commit IPC values when more restrictions are imposed on in-flight misses in the systems with four-way issue processors (Figure 5a) than in the systems with eight-way issue processors (Figure 5b). This trend occurs because the performance of systems with wider-issue processors is more sensitive to the design of the memory system. The cause of this increased sensitivity is the ability to issue more instructions per cycle, which tends to reduce the number of clock cycles between the issuing of load instructions. As a result, at any point in the execution of an application, there tends to be a greater number of in-flight fetches and secondary misses.

Additional insight into this difference between the two issue widths is suggested by the differential in the commit IPC values when the fetch spacing is reduced from 8 cycles to 0 cycles. As shown by the data in Figure 5, when the fetch spacing is reduced, there is greater variance in the commit IPC values for the eight-way issue processors, and in particular, for systems with greater support for in-flight misses and a larger number of physical registers.

In summary, a number of observations can be drawn from the above discussion, and these are presented in Section 6.

5 Stream Buffer Implications

To investigate the design and performance implications of stream buffers, the system model was augmented to include the stream-buffer models described in Section 2.2. The system model was then used to evaluate the behavior of the benchmarks on a number of systems that were chosen to capture the behavior at several representative points in the large design space.

5.1 Baseline Stream Buffers

For clarity in presenting key observations, consider first the performance implications of including the baseline stream-buffer implementation in 16 of the systems listed in Table 2. The performance of the benchmarks on the resulting 32 systems (16 without stream buffers and 16 with stream buffers) is presented in Figure 7. This figure gives the average commit IPC for each benchmark as a function of the issue width of the processor, the size of the register files, the type of data cache, the fetch spacing, and the optional use of stream buffers. The lockup-free cache had no restrictions on the number of in-flight misses (type i). The columns with N at their base (e.g., the column marked by $\boxed{1}$) give the commit IPC values for systems without stream buffers, while those with S_b at the base (e.g., the column marked with $\boxed{2}$) give the commit IPC for systems with stream buffers.

One important observation is that if stream buffers are included in a system, there is an overall performance gain, with this gain being bigger for systems having lockup caches and a larger number of registers. To illustrate this relationship, consider the columns marked with $\boxed{1}$ and $\boxed{2}$ in Figure 7 that correspond to the performance obtained on systems with 48 registers, a fetch spacing of eight cycles, and a four-way issue processor. Comparison of the relative position of the data points in these two columns shows that the data points are generally higher in the second column. The average increase in IPC for systems with the lockup-free cache (the filled circles) is 0.35, while for systems with the lockup cache (the filled squares), the average increase is 0.46. Expressed as speedup ratios, these increases correspond to a speedup of 1.29 for the systems with a lockup-free cache and a speedup of 1.51 for systems with a lockup cache.

The more significant speedup for systems with a lockup cache is a consequence of the hardware being blocked from issuing memory instructions during stalls induced by structural hazards. For systems with this type of cache, stream buffers enhance the performance because they significantly reduce the effective cache-miss penalty. Systems with lockup-free caches, however, can issue unrelated memory instructions during such stalls, and, thus, are both less sensitive to the effective cache-miss penalty, and more tolerant of cache misses. The amount of tolerance such systems have is a function of the number of physical registers, because the number of registers determines how many instructions can be simultaneously in execution. Thus, with more registers, the speedup obtained with stream buffers will decrease. As illustrated in Figure 7, when the number of physical registers is increased from 48 to 64 (columns $\boxed{3}$ and $\boxed{4}$), the speedup for systems with lockup-free caches drops to 1.25, while the speedup for systems with lockup caches rises to 1.58.

The above observations suggest that if a system can support a large number of in-flight misses, it is more beneficial to increase the number of registers than it is to include stream buffers. To illustrate this phenomenon, consider again the same four system configurations. When a lockup-free cache is used, a 32% performance improvement is obtained by increasing the number of registers from 48 to 64, but an improvement of only 29% is obtained by using stream buffers. On the other hand, when a lockup cache is used, a 14% performance improvement is obtained by increasing the number of registers from 48 to 64, but an improvement of 51% is obtained by using stream buffers.

The performance gains achieved with stream buffers are a result of the buffers prefetching a significant number of cache blocks that are used to resolve cache misses. Table 4 presents data to quantify the frequency at which cache misses were anticipated. This table provides: in column 2, the run-time speedup due to the use of the stream buffers; in column 3, the primary data-cache miss rate; in column 4, the percentage of primary misses resolved using data either present in a stream buffer, or for which a fetch request had already been launched; and in column 5, the number of cache blocks used to resolve one of these misses as a percentage of the total number of cache blocks returned by the memory system in response to a fetch request from a stream buffer. Ignoring *compress*

[2] These percentages represent the average of the percentages for each benchmark.

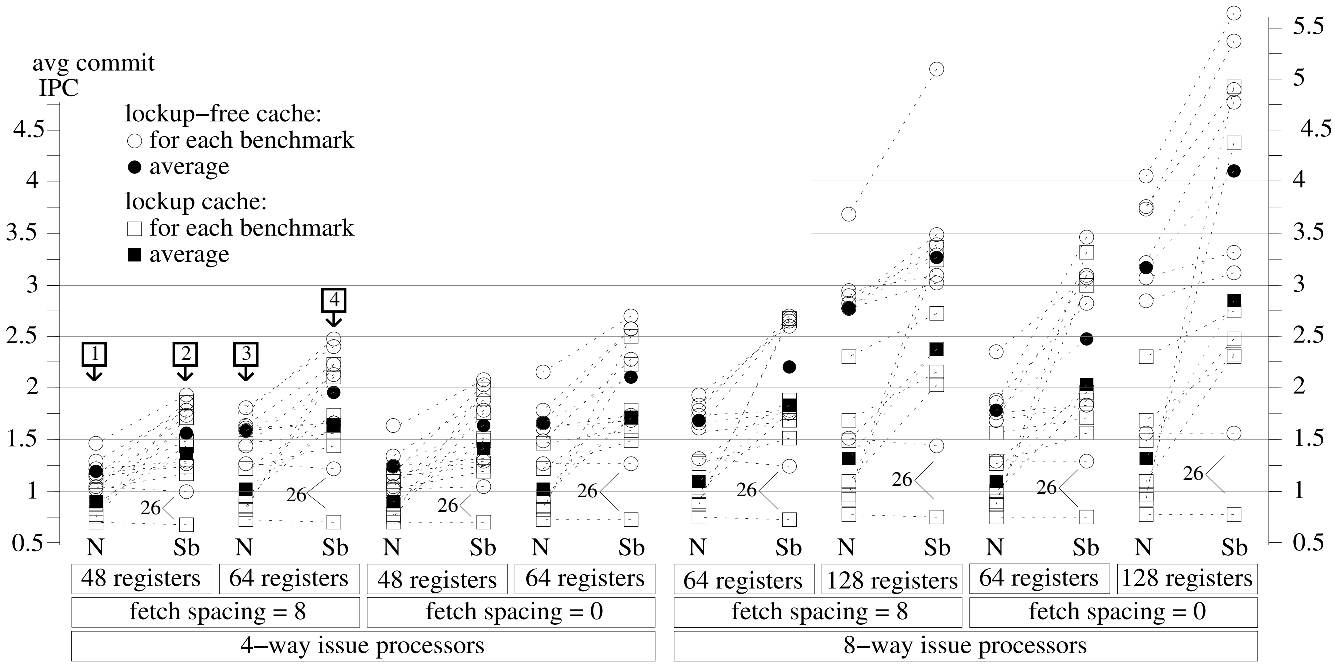


Figure 7: Average commit IPC for systems using either a four-way or an eight-way issue processor and either no stream buffers (N) or the baseline stream-buffer implementation (S_b). The dotted lines connect the data points of each benchmark for each cache implementation. The unfilled circles indicate the IPC for each benchmark obtained with the use of a lockup-free cache having no in-flight miss restrictions, while the filled circles indicate the overall average IPC for all benchmarks. The squares indicate similar information but with the use of the lockup cache.

for the moment, observe that 18% to 98% of the blocks prefetched by the stream buffers (column 5) are used to resolve 29% to 77% of primary misses (column 4).

The statistics presented in the table are sensitive to the number of physical registers. If the number is increased, there is little change in both the number of primary misses, and the number of times data is fetched in response to a stream buffer request. But, there is a significant drop in the number of misses for which the missing data is actually present in a stream buffer. This drop is largely due to the increase in the issue IPC that accompanies the increase in the number of registers. This increase in the issue IPC reduces the time between successive cache misses, and thus reduces the time available for prefetching. *Compress*, however, as already noted, does not benefit from stream buffers. Rather, as shown in Figure 7 by the curves labeled "26", for a fetch spacing of 8 cycles and a four-way issue processor with 48 registers, its performance decreases by

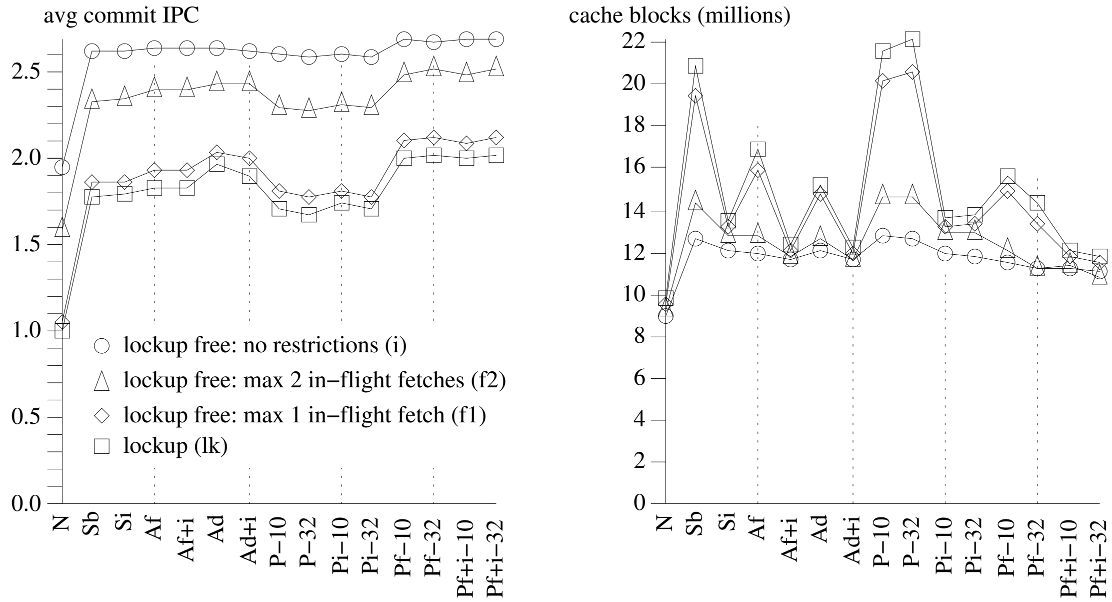
bench-mark	speed-up	primary misses		SB
		%	% resolved by SB	% blocks used
(1)	(2)	(3)	(4)	(5)
appsp	1.09	6.0	28.9	18.1
compress	0.95	12.9	0.2	0.1
hydro2d	1.70	12.6	77.3	97.7
mdljdp2	1.06	2.4	60.6	48.5
su2cor	1.32	8.7	53.1	63.8
swm256	1.36	6.7	44.7	29.3
tomcatv	1.64	11.2	71.6	98.4

Table 4: Effectiveness of the baseline stream-buffer implementation when used in a 4-way issue processor with 48 registers and an 8-cycle fetch spacing.

4% when run on the systems having the lockup cache, and by 5% when run on the systems having the lockup-free cache. This performance decrease is a result of the stream buffers prefetching mostly un-needed cache blocks, thereby delaying the launching of fetch requests that are needed to service cache misses. On these systems, less than 0.2% of primary cache misses are resolved using prefetched cache blocks. The percentages are much higher for the other benchmarks, as shown in Table 4 by the data given in column 4.

A second important relationship that is suggested by the data of Figure 7 is the performance insensitivity of the systems to the bandwidth of the interface. When the fetch spacing is reduced from 8 cycles to 0 cycles, with the exception of *compress*, the commit IPC values change relatively little. This observation suggests that even with a fetch spacing of 8 cycles, contention for the memory interface is not significant. One result of this lack of contention is that with an 8-cycle fetch spacing, when a stream buffer is re-allocated, 95 times out of 100, all of its entries contain valid data. Consequently, when the fetch spacing is reduced to 0 cycles, there will not be a large increase in the amount of prefetched data, and hence, only small performance gains are likely. When the fetch spacing is changed to 0 cycles, the least significant change in the speedup ratios is an increase from 1.29 to 1.33 with 48 registers and a lockup-free cache, and the most significant change is an increase from 1.58 to 1.67 with 64 registers and a lockup cache. These speedup ratios are given in Table 5 under the column heading "4-way". Like the other benchmarks, *compress* performs better with a fetch spacing of 0 cycles, but unlike the other benchmarks, it does not achieve a speedup greater than one. This phenomenon is shown in Figure 7 by the curves near the bottom whose right-most data points are labeled with "26". Observe that when a fetch spacing of 0 cycles is used, the curves are nearly horizontal.

When the baseline stream buffers are used in systems with an



(a) performance (i.e., average commit IPC) (b) total number of cache blocks

Figure 8: Performance of *su2cor* and the total number of cache blocks supplied by the memory interface when it was run on systems using one of the 15 stream-buffer implementations.

cache type	fetch spacing	4-way # registers		8-way # registers	
		48	64	64	128
lockup	8	1.51	1.58	1.70	1.87
	0	1.57	1.67	1.80	2.15
lockup-free	8	1.29	1.24	1.30	1.18
	0	1.33	1.28	1.38	1.29

Table 5: Average speedup in (simulated) run-time for all benchmarks due to the use of the baseline stream-buffer configurations.

eight-way issue processor and 64 dispatch queue entries, the same trends as those noted for the four-way issue processors occur, but the differences tend to be greater. The increased significance is a consequence of the eight-way issue processor issuing and committing more instructions per cycle thereby resulting in a compression of time. This temporal compression not only reduces the time between the occurrence of cache misses, but also increases the number of secondary cache misses and the instruction cost of servicing all cache misses.

5.2 Memory Traffic and Dynamic Strides

Although the baseline stream-buffer implementation improves the performance of six of the benchmarks, the basic design can be augmented to reduce the possible negative impact of excess memory traffic, unit strides, and the possibility of flushing useful data from a stream buffer when it is reallocated. To counter the negative impact of stream buffers, the enhancements listed in Table 2c can be employed.

The results obtained from the simulated execution of the benchmarks on these systems show a number of common relationships, and these will be illustrated using the *su2cor* benchmark and its execution on 60 systems. These 60 systems had an eight-way issue processor, 64 physical registers and a fetch spacing of 8 cycles. Figure 8a presents the average commit IPC values for these executions, while Figure 8b presents the total number of cache blocks that were supplied by the memory interface. In both figures, each of the curves for the four data-cache configurations (i, f_2, f_1, and k) includes a data point for the 15 stream-buffer implementations. The stream-buffer implementations are listed along the horizontal axes using the abbreviations given in Table 2c. However, the abbreviations used for the designs based on the per-load scheme (e.g., P_f) also include a suffix (e.g., 10) to indicate the number of entries in the missed-load buffer.

An important relationship suggested by the data presented in Figure 8a is that the enhancements to the baseline stream-buffer implementation have at best a small effect on performance, with this effect being more significant for the systems with less aggressive cache configurations. For example, when the implementations are used with the lockup cache (type lk), there are pronounced variations in commit IPC values, but when the implementations are used with the lockup-free cache having no in-flight miss restrictions (type i), the commit IPC values vary very little. The increased performance sensitivity of the more restrictive cache configurations to the stream-buffer implementation is due to two factors. First, systems with more support for in-flight misses have greater tolerance for cache misses. Thus, whether a stream-buffer implementation improves performance by correctly prefetching data that is missing from the cache, or degrades performance by tying up the memory interface with misfetched data, the overall performance impact, be it positive or negative, will be smaller.

Second, systems with more support for in-flight misses tend to require fewer clock cycles to execute an application. Hence, there are fewer clock cycles available for prefetching data, and there is a reduction in the number of clock cycles between the detection of cache misses for which the missing data has not been prefetched. As a result of these two effects, systems with less restrictive cache configurations tend to fetch fewer cache blocks from the memory interface. This trend is suggested by the relative positions of the curves in Figure 8b. The performance differential between the stream-buffer implementations is a reflection of the ability of an implementation to correctly prefetch data and to hold on to the data until it is required. Thus, the implementations with some form of filtering give better performance (Figure 8a) and generate less memory traffic (Figure 8b).

When the number of physical registers is increased and/or the bandwidth-limit on the memory interface is removed, data similar to that presented in Figure 8 is obtained. Due to space constraints, this data is not presented, but the following two relationships are nonetheless noted; the corresponding data is presented in [15].

First, concerning the number of physical registers and the amount of support for in-flight misses, for systems using the type i lockup-free cache, the performance impact of the stream-buffer implementation is even smaller if the number of physical registers is increased. Increasing the number of registers allows more instructions to be simultaneously in some stage of execution, which increases the tolerance for cache misses, and decreases the number of clock cycles required to run the application. Thus, when the number of registers is increased, the commit IPC varies less for the various stream buffer implementations, and the number of blocks fetched from the memory system decreases. However, for the systems with restrictive cache configurations, increasing the number of registers increases the performance sensitivity to the stream buffer implementations. A larger number of registers increases the percentage of clock cycles in which memory instructions cannot be issued (Figure 6a), thus increasing the performance sensitivity to the effective cache-miss latency. Because the performance impact of such stalls is a function of their duration, stream-buffer implementations that are better at lowering the effective cache-miss latency give better performance. Furthermore, the implementations that perform better generate less memory traffic because there is less time to prefetch the data.

Second, concerning the bandwidth of the memory interface, removal of the bandwidth limit leads to a more pronounced difference in the relative performance obtained with the stream-buffer implementations. When memory bandwidth is not limited, the stream-buffer implementations that are more beneficial are those that are better at prefetching and holding onto data that is subsequently required to resolve a cache miss. Thus, while the techniques for filtering stream-buffer allocations have a positive performance impact, the incremental prefetching technique, which serves only to reduce memory bandwidth requirements, has no significant performance impact. However, incremental prefetching reduces the bandwidth consumed by stream buffers by 50%.

Finally, concerning all the benchmarks[3], several comments should be made. On systems with a type i lockup-free cache, the stream-buffer enhancements had less than a 5% impact on performance with the exception of *swm256*. Furthermore, these enhancements had little impact on the performance of *mdljdp2* and *hydro2d* irrespective of the cache organization, while they significantly degraded the performance of *tomcatv* if in-flight miss restrictions existed. *Appsp*, *su2cor*, and *swm256* performed best if the system included one of the following two sets of mutually exclusive enhancements: (1) the PK-allocation filters, and optionally, the min-delta stride predictor; or (2) the per-load stride predictor with filtering, and optionally, incremental prefetching. Finally, for all the systems considered and all benchmarks, the stream-buffer implementations with PK-allocation filters or stride filters generated between 1.5 and 4.2 times less memory traffic. In most cases, the per-load stride predictor with stride filtering generated the least amount of memory traffic.

6 Conclusions

In this paper, we have presented an investigation and analysis of the design of the register file and the other levels of the data memory hierarchy. This analysis has focused on identifying performance trends and design relationships. The components we considered affect the apparent time-cost of servicing cache misses and the tolerance for data-cache misses. The following conclusions can be drawn from the analysis.

First, similar performance was obtained from all systems having support for fewer than four in-flight misses, irrespective of the register-file size, the issue width, and the memory bandwidth. While increasing the hardware support for in-flight misses beyond this point did increase system performance, for the configurations considered, the improvement was less than that obtained by increasing the number of registers.

Second, systems with a greater amount of support for in-flight misses require a greater proportion of the support to be for secondary misses, since the secondary miss rate tends to increase as the amount of in-flight-miss support is increased. Additional registers should also be provided to offset the increase in the average register lifetime that is a result of the ability to support a larger number of in-flight misses.

Third, system performance is relatively unaffected by the bandwidth of the memory interface if the processor can issue a maximum of only four instructions per cycle. But, when the issue width is increased to eight, the bandwidth of the memory interface has a more significant impact on performance, especially in systems with support for at least four in-flight fetches. While the performance sensitivity to the bandwidth is small for all but the most aggressive systems, the reported percentages represent a lower bound due to the assumption that neither the instruction cache nor the write buffer use the memory interface. If these two components were to compete for the bandwidth of the interface, contention would increase, which would increase the performance sensitivity to the bandwidth. The rate of this increase would be greater for systems with the eight-way issue processor since wider issue processors generate more traffic in a given time period than narrower issue processors. Nonetheless, if stream buffers are not included in the system, the bandwidth is unlikely to be a significant factor for the less aggressive systems and the benchmarks discussed in this paper.

Fourth, the addition of stream buffers to a system leads to a more significant performance increase for systems having either more restrictive lockup-free caches, more memory bandwidth, or more instruction issue capability. For the systems investigated, the design of the lockup-free cache had the greatest impact on performance, and the bandwidth of the memory interface had the least impact. Increasing the number of registers results in a more significant performance increase with systems having a lockup cache but a less significant increase with systems having a lockup-free cache. The increase is larger for the systems with the eight-way issue processors.

Fifth, system performance is not significantly affected when stream buffers are used that have traffic filters and dynamic-stride calculators. This observation is not surprising since the performance of the benchmarks is at best 16% better when the memory interface has an infinite bandwidth. In other words, contention for the interface is not a significant problem. This observation also suggests that the address stream generated by cache misses when each benchmark is run is dominated by interleaved unit-stride streams. Thus, the cost of supporting dynamic-stride calculation is not warranted for these seven benchmarks. However, in spite of the similar performance of many of the enhanced stream-buffer implementations, the cost of those that have traffic filters is probably warranted since the system model does not take into account all possible sources of memory traffic. The exact type of filter would have to be determined when a specific system is being considered. A larger set of benchmarks would also be required, as well as including all of the traffic effects into the model.

Finally, the incremental-prefetching technique we introduce reduces the bandwidth consumed by stream buffers by half with little

[3] The data for these benchmarks is given in [15].

performance loss.

Acknowledgments

The research described in this paper has been partially funded by the Natural Sciences and Engineering Research Council of Canada and by Digital Equipment Corporation. We thank Brad Calder and Alan Eustace for helping us with the ATOM simulation infrastructure, Annie Warren and Jason Wold for logistical support while our simulations ran, and the other WRL-ites for putting up with these simulations. Finally, we thank Digital Equipment Corporation for providing us with the Alpha AXP workstations.

References

[1] Norm Jouppi. Improving Direct Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers. Technical Report TN-15, Digital Equipment Corporation Western Research Lab, March 1990.

[2] Keith I. Farkas. *Memory-system Design Considerations for Dynamically-scheduled Microprocessors*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, Ontario, Canada, January 1997. (URL: http://www.eecg.toronto.edu/~farkas/thesis_phd.html).

[3] David Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. *In the Proceedings of the Eighth International Symposium on Computer Architecture*, pages 81–87, May 1981.

[4] Keith I. Farkas and Norman P. Jouppi. Complexity/Performance Tradeoffs with Non-Blocking Loads. *In the Proceedings of the 21st International Symposium on Computer Architecture*, pages 211–222, 1994.

[5] Kenneth C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, 1996.

[6] Linley Gwennap. PA-8000 Combines Complexity and Speed. *Microprocessor Reports*, 8(15):1,6–11, 1994.

[7] S. Peter Song, Marvin Denman, and Joe Chang. The PowerPC 604 RISC Microprocessor. *IEEE Micro*, 14(5):8–17, 1994.

[8] IBM Microelectronics. *PowerPC 620 RISC Microprocessor Technical Summary*, 10 1994. document number: MPR620TSU-01.

[9] Keith I. Farkas, Norman P. Jouppi, and Paul Chow. How Useful Are Non-blocking Loads, Stream Buffers and Speculative Execution in Multiple Issue Processors? *In the Proceedings of the First International Symposium on High Performance Computer Architecture*, pages 78–89, 1995.

[10] Subbarao Palacharla and R. E. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. *In the Proceedings of the 21st International Symposium on Computer Architecture*, pages 24–33, 1994.

[11] John W. Fu, Janak H. Patel, and Bob L. Janssens. Stride Directed Prefetching in Scalar Processors. *In the Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 102–110, 1992.

[12] Tien-Fu Chen and Jean-Loup Baer. Reducing Memory Latency via Non-blocking and Prefetching Caches. *In the Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, October 1992.

[13] Scott McFarling. Combining Branch Predictors. *Digital Equipment Corporation Western Research Lab Technical Note TN-36*, 1993.

[14] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. *In the Proceedings of the ACM SIGPLAN '94 Conference on Programming Languages*, March 1994.

[15] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic. Memory-system Design Considerations for Dynamically-scheduled Processors. Technical Report 1, Digital Equipment Corporation Western Research Lab, 1997. (URL: http://www.research.digital.com/wrl/techreports).