HIGH-BANDWIDTH DATA MEMORY SYSTEMS FOR SUPERSCALAR PROCESSORS

by

Gurindar Sohi and Manoj Franklin

Computer Sciences Technical Report #968

September 1990

High-Bandwidth Data Memory Systems for Superscalar Processors*

Gurindar Sohi and Manoj Franklin

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706

Contact Address
G. S. Sohi
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706
(608)-262-7985
sohi@cs.wisc.edu

* This work was supported by National Science Foundation grants CCR-8706722 and CCR-8919635, and by an IBM Graduate Fellowship.

High-Bandwidth Data Memory Systems for Superscalar Processors

Abstract

This paper considers the design of a memory hierarchy, with a level 1 (L1) data cache at the top, to support the data bandwidth demands of a future-generation superscalar processor capable of issuing about ten instructions per clock cycle. It introduces the notion of *cache bandwidth* -- the bandwidth with which a cache can accept requests from the processor, and shows how the bandwidth of a standard, *blocking cache*, can degrade greatly because of its inability to overlap the service of misses. *Non-blocking* or *lockup-free* caches are discussed as a way of reducing the bandwidth degradation due to misses. To improve the data bandwidth to greater than 1 request per cycle, multi-port, interleaved caches are introduced. Simulation results from a cycle-by-cycle simulator, using the MIPS R2000 instruction set, suggest that memory hierarchies with standard, single-ported L1 caches will be unable to support the bandwidth demands of future-generation superscalar processors. *Multi-port, non-blocking (MPNB)* L1 caches introduced in this paper for the top of the data memory hierarchy appear to be capable of supporting the data bandwidth demands for several generations of superscalar processors.

1. Introduction

As technology advances allow more functionality to be put on a single chip, VLSI processor designers are looking for ways to exploit the available resources to enhance processor performance. One way of enhancing performance is to exploit fine-grain parallelism and issue multiple instructions in a clock cycle. By the middle of this decade, we expect processors that attempt to issue about ten instructions in a clock cycle to be within the realm of possibility[1].

Figure 1 presents our view of the overall organization of a circa 1993 high-performance superscalar processor chip that might have a peak instruction issue rate of perhaps ten instructions per clock cycle and a sustained issue rate of about 3-5 instructions per cycle. The CPU has functional units for computation, an instruction issue mechanism, an instruction cache to supply instructions to the instruction issue mechanism, a data cache for memory operands, and an interconnect that connects together the various components. There could be several functional units such as floating-point adders, floating-point multipliers, integer multipliers, integer adders, and adders for address calculation.

At the top level of the memory hierarchy, we expect there to be separate *level 1 (L1)* instruction and data caches as shown in Figure 1. These L1 caches are connected to a shared *level 2 (L2)* cache, via an *L1-L2 bus*. The L2 cache is in turn connected to the main memory, which may be shared by several other processors. It is also possible that, as technology advances, and multiple CPUs along with their L1 caches can be put on a chip, the L1-L2 bus and the L2 cache (which may be on-chip) may be shared by multiple CPUs[2].

To issue multiple instructions per cycle, an appropriate instruction issue mechanism is needed. Several mechanisms for issuing multiple instructions in a clock cycle have been pub-

---

[1] Attempts are already being made at 8 instructions per cycle [2].

[2] Technology projections have predicted a 100 million transistor processor chip by the end of the decade. Such a processor chip may have multiple superscalar CPUs, each connected to it own L1 cache (of the order of tens of kilobytes), and share a common L2 cache (of the order of a megabyte) [3].

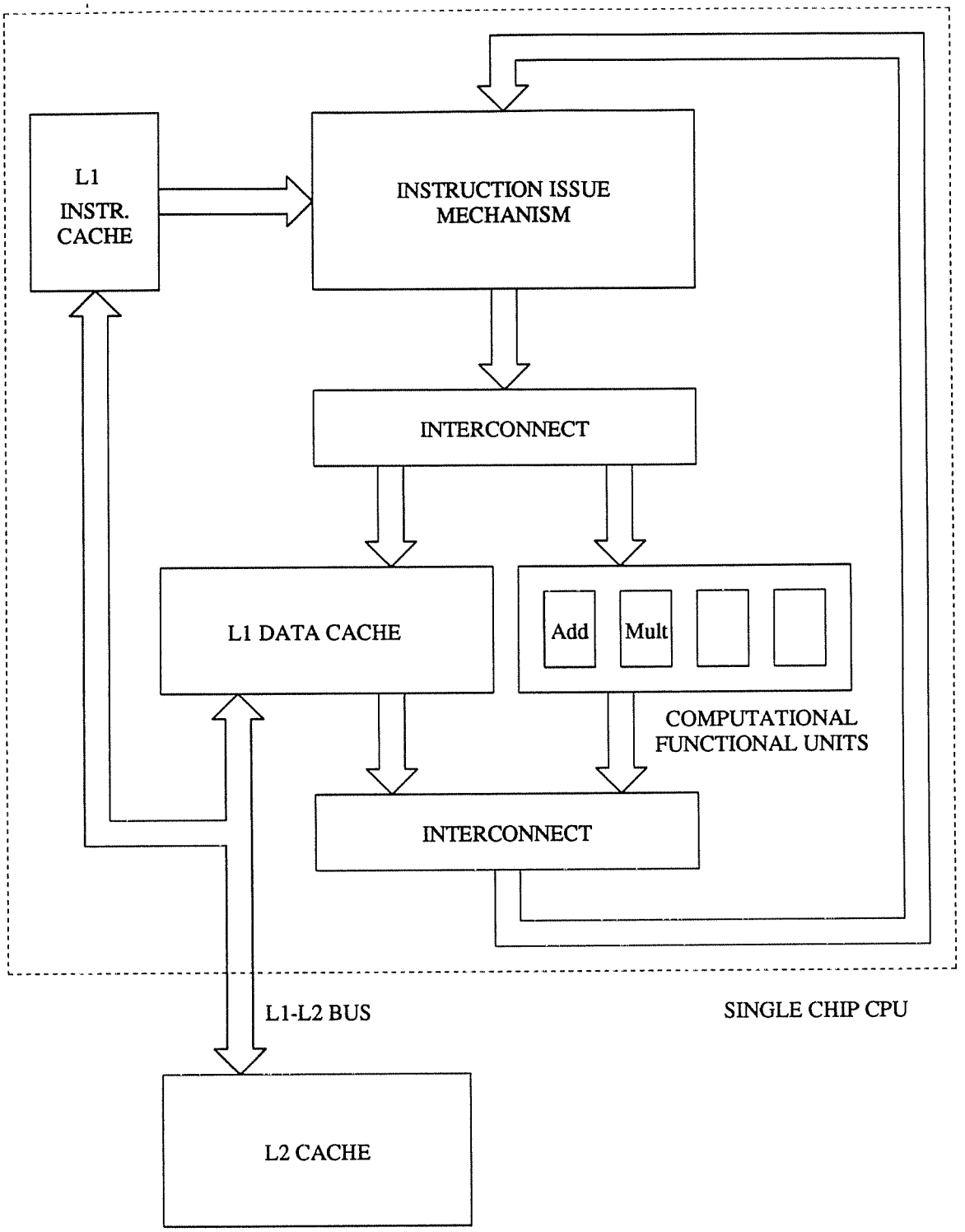


Figure 1: A Superscalar CPU

lished [1,5,7,8,11-14,18,20,21], and others are being investigated. We will not concern ourselves with the instruction issue mechanism of such a superscalar processor in this paper, nor with the design of the L1 instruction cache that can provide the appropriate instruction bandwidth. This is because the exact issue mechanism that might be used to issue about ten instructions in a clock cycle, in a superscalar fashion, and sustain an issue rate of 3-5 instructions is still the subject of research and debate [1,5,8,11-14,18,20]. Furthermore, because of the high hit ratios achievable for instruction caches [6,9,15,16,19], we feel that a small (few kilobytes) L1 instruction cache can be designed to support most instruction issue mechanisms, though the exact design of such a cache will be highly dependent on the instruction issue mechanism used.

Regardless of the instruction issue mechanism chosen, *an appropriate memory hierarchy is needed to support the data bandwidth demands of the instruction issue mechanism.* Our focus in this paper is to see how to provide a data memory system that can support the data bandwidth demands of a future instruction issue mechanism issuing about ten instructions per cycle.

1.1. The Importance of Data Memory Bandwidth

Why is a high-bandwidth data memory system critical? It is obvious that the bandwidth of the critical functional unit resource dictates the minimum number of clock cycles that it will take to execute a given program, regardless of the sophistication of the instruction issue mechanism. The best-case execution time is one in which the bandwidth demand of the program is equal to the bandwidth supply of the critical functional unit resource.

The data memory (L1 data cache, L2 cache, and main memory) is perhaps the most heavily demanded resource (with the possible exception of the adder used for generating memory addresses) and is likely to be the critical resource[3]. Accordingly, the peak instruction issue of our

[3]Even if the memory is not the critical resource and some other computational functional unit is, providing additional computational functional unit bandwidth is straightforward; all that we have to do is to provide more copies of the computational functional unit, and an enhanced interconnect.

superscalar processor will be limited to $\frac{BW_S}{f_M}$ instructions per cycle, where BW_S is the bandwidth that the data memory can supply, and f_M is the fraction of all instructions that are loads and stores. Clearly any improvements in the instruction issue strategy will be worthwhile only if they are accompanied by a commensurate increase in the data memory bandwidth.

1.2. Paper Objective and Outline

The goal of this paper is to consider the design of a data memory system to support the bandwidth demands of a future superscalar CPU capable of issuing possibly tens of instructions per clock cycle. Clearly this will require a data memory bandwidth of several requests per cycle[4]. We would ideally like to achieve this high bandwidth with the freedom and the flexibility that we have on the processor chip, i.e., with minimal additional demands on the off-chip components of the system, such as the L2 cache and the main memory. Accordingly, we will concentrate mainly on the top of the data memory hierarchy, i.e., the L1 data cache and the L1-L2 bus, though the results of our paper could easily be applied to the L2 cache and the L2-memory interface. Therefore, unless stated otherwise, all references to cache shall imply the L1 data cache. Furthermore, we shall also assume that all data references go through the L1 cache, i.e., the L1 cache can't be bypassed. Our results are easily extended if the L1 cache can be selectively bypassed.

In section 2, we consider the issue of cache bandwidth, and cache designs that can provide a high bandwidth. In section 3, we present simulation results, using a current instruction issue strategy, to illustrate how low-bandwidth cache designs can be a bottleneck to performance of future instruction issuing strategies, and we conclude in section 4.

[4]All future references to bandwidth shall be to the average bandwidth measured in requests per clock cycle.

2. Cache Bandwidth and High-Bandwidth Cache Designs

Most of the literature on cache memories [17] has concentrated on the latency with which memory requests can be serviced with a cache memory, and rarely has there been a discussion of the bandwidth of a cache. The possible exception to this is the literature on caches in shared-memory multiprocessors, starting with [4], that deal with how a cache can be used to reduce cache-memory bandwidth, but not specifically with how much *bandwidth a cache can provide to the CPU*. The reason for this, we believe, is that the bandwidth of caches is rarely a major concern for processors that issue a single instruction per cycle since such processors do not have a very high bandwidth demand (compared to superscalar processors). For example, to support a peak issue rate of a single instruction per cycle, a data cache with a bandwidth of f_M is sufficient (typically f_M is in the range of 0.25-0.4 for a RISC processor such as the MIPS R2000) and, as we shall see, such a low average bandwidth could be squeezed out quite easily with most standard cache designs. For processors capable of issuing multiple instructions per cycle, however, the data bandwidth demands are naturally much higher (at least the same number of references are made in fewer clock cycles) and therefore, the first step in designing an L1 data cache should be to evaluate the bandwidth that it can provide.

Without any loss of generality, we assume that the L1 cache is a writeback cache in this paper. A processor request to the L1 cache can either *hit* or *miss*. If the request hits, it is serviced by the L1 cache, without causing any actions on the L1-L2 bus. If it misses, the L1 cache creates a miss request, as well as a writeback request (if the replaced block is dirty) on the L1-L2 bus.

2.1. Blocking Caches

The most commonly used and studied caches are blocking caches. In such caches, the CPU can continue to issue instructions as long as the memory references it makes hit in the cache. However, when a miss occurs, the CPU stalls instruction issue[5] until the miss request has been

completed and the block has been fetched from the L2 cache to the L1 cache. Therefore, with a blocking cache, the CPU can have at most one miss request pending and, while a miss is pending, it can accept no other requests from the CPU, even though they might be hits.

The design of blocking caches is well-understood [16, 17]; almost all computers built today have them. With a blocking cache, the L1-L2 bus interface is straightforward. Since there is only one request from the L1 cache to the L2 cache at any time, the L1-L2 bus can be held, in a circuit-switched fashion, until the entire transaction has been carried out[6]. Finally, since the L2 cache has to handle only a single load request, its design is also straightforward[7].

The disadvantage of a blocking cache is the bandwidth degradation that can result because misses must be handled serially. Let us see how much bandwidth a standard single-ported, blocking L1 cache can supply and how much of a degradation in bandwidth can result because of the requirement of handling misses serially.

Suppose that a program makes $H+M$ memory requests, where H is the number of requests that hit in the L1 cache, and M is the number of misses. If there is a single cache port, the time taken to service H hits is H cycles. The L1 cache and the L1-L2 bus are busy for (T_m+B) and $[T_m+B(1+d)]$ cycles, respectively, for each miss that is serviced, where T_m is the miss time, i.e., the time taken by the L2 cache to respond with the first word of the block after the miss request is issued, d is the probability that the replaced block is dirty, and B is the number of cycles taken to transfer a block on the L1-L2 bus. Since the service of hits and misses can't be overlapped in a

---

[5]If hardware interlocks are used to enforce dependencies, the CPU can continue to execute instructions that have register-only operands, and does not have to stall instruction issue until the next load/store instruction is encountered. Our experience has shown little difference in performance if the CPU stalls instruction issue when the miss occurs or if it proceeds with instruction issue until the next load/store instruction. Therefore, we assume the standard practice of stalling instruction issue when the miss is encountered. In either case, instructions that are already in execution are not stalled.

[6]With a blocking cache, we have two choices of how to handle the writeback request. In either case, for getting a smaller miss latency, the miss request would be submitted to the L2 cache before the writeback request. The first alternative for handling the writeback request is to wait until the miss request has completed and then carry out the writeback request. The second alternative, which requires a more complicated L1-L2 bus design, is to release the L1-L2 bus after the miss request has been submitted, carry out the writeback request, and then grab the L1-L2 bus again to receive the response to the miss request. Since the former approach is the more commonly-used one, we shall assume it to be the way of handling writeback requests.

[7]As pointed out in [9], if its access latency is sufficiently high, it may have to be pipelined sufficiently to handle multiple writeback requests.

blocking L1 cache, the time taken by the L1 cache and the L1-L2 bus to service $H+M$ requests is $(H+M[T_m+B])$ and $M[T_m+B(1+d)]$ cycles, respectively. The upper-bound on the average bandwidth of the data memory system, assuming all data references go through the L1 cache, is simply the lower of the bandwidths of the L1 cache and the L1-L2 bus, i.e.,

$$Min\left[\frac{H+M}{H+M\times[T_m+B]}, \frac{H+M}{M\times[T_m+(1+d)\times B]}\right] =$$

$$Min\left[\frac{1}{1+m\times[T_m+B-1]}, \frac{1}{m\times[T_m+(1+d)\times B]}\right] \quad (1)$$

where $m=\frac{M}{H+M}$ is the miss ratio.

Figure 2 plots the bandwidth (requests per cycle) provided by a memory system with a standard, single-ported blocking L1 cache (with a maximum bandwidth of 1 request per cycle), obtained from equation (1), versus the miss ratio m, for some values of T_m, B, and d. As we can see from the figure, the bandwidth drops significantly as the miss ratio increases. For example, a cache with the optimistic parameters of $m=0.05$, $T_m=10$, $B=1$, and $d=0$, can achieve a bandwidth of only 0.67 requests per cycle. If we assume $f_M=0.4$, this implies that our superscalar processor with the above L1 data cache will be able to achieve a sustained issue rate of only 1.67 instructions per cycle, regardless of how many resources (other than those to improve memory bandwidth) we throw at it! It is clear that we must improve the bandwidth of the cache if we hope to achieve a superscalar execution of more than a few instructions per clock cycle.

Before proceeding further, from equation (1) we can also see why cache bandwidth has not been of much concern thus far. With a peak instruction issue rate of 1 per clock cycle, and with $f_M<0.4$, we require a bandwidth of less than 0.4 requests per cycle, and this can easily be achieved with $m<0.1$, if $T_m=10$ and $1\le B\le 4$.

To improve bandwidth, we have two options: i) provide multiple ports to service hits or ii) reduce the bandwidth degradation due to misses. From equation (1), we can see that even if we

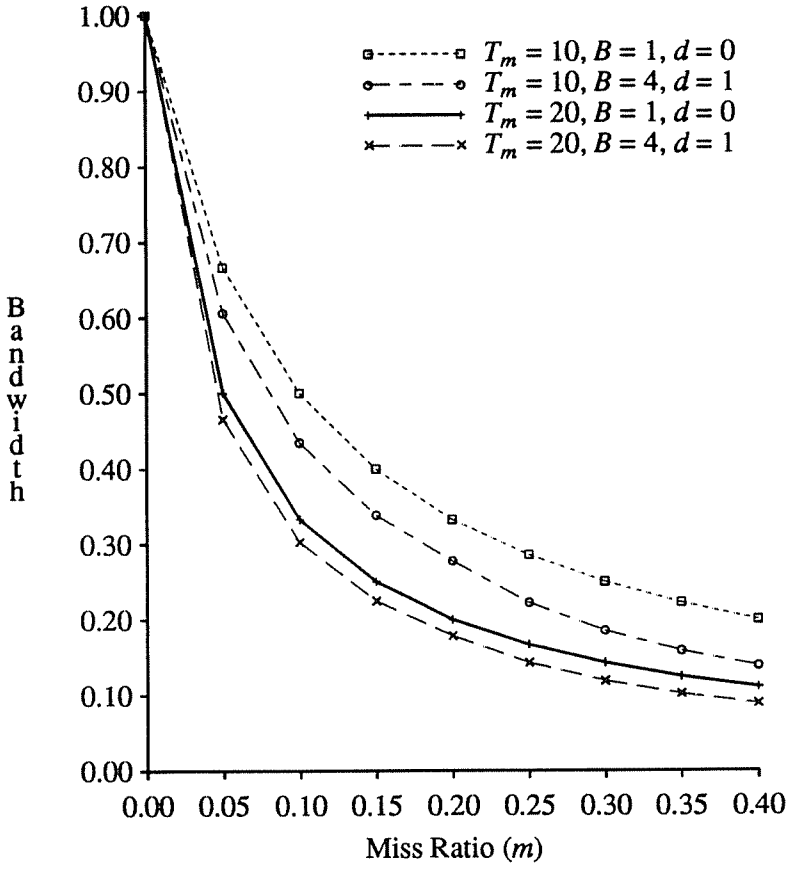


Figure 2: Data Bandwidth provided by a Blocking L1 Data Cache

provide an infinite bandwidth for cache hits, the bandwidth of a blocking cache can be improved to only $\frac{1}{m \times [T_m + B(1+d)]}$, a value that is dictated by the bandwidth with which misses can be serviced on the L1-L2 bus. If the bandwidth degradation due to misses is significant, as is likely to be the case unless both m and T_m are very small and $B = 1$, it is of primary importance to consider ways of decreasing it first before considering ways to improve the cache bandwidth for hits.

2.2. Reducing Bandwidth Degradation Due to Misses: Non-Blocking Caches

To reduce the bandwidth degradation due to misses, we must decrease the total time spent in servicing the misses. An obvious way to reduce the time spent in servicing misses is to

decrease m, T_m, or both. However, as we mentioned earlier, both m and T_m would have to decrease (so that their product is very small) for the bandwidth degradation to be inconsequential. This is counter to the current trend of increases in T_m because of increases in the processor clock speed.

From equation (1) we see that a major reduction in the degradation due to misses, and consequently a major improvement in bandwidth, can be made if we eliminate T_m from the equation entirely. This can be done if we allow the service of multiple miss requests to be overlapped, in a pipelined fashion. In the best case, if all M miss requests can be overlapped perfectly, the time taken to service the misses can be reduced to[8] $M(1+d)B$ and the bandwidth of a single-ported cache can be improved to:

$$Min\left[1, \frac{1}{m(1+d)B}\right] \tag{2}$$

The first term in the above equation corresponds to the bandwidth with which requests can be submitted to the cache, and the second term to the bandwidth with which the misses can be serviced on the L1-L2 bus.

2.2.1. Basic Non-Blocking Cache

To overlap miss requests, we consider a *non-blocking* or *lockup-free* cache organization first proposed by Kroft [10]. In Kroft's suggested implementation, registers called MSHRs (miss information/status holding registers) are used to hold the status information of the outstanding misses. One MSHR is associated with each outstanding miss. If there are N MSHRs, we have a *non-blocking(N)* cache. Therefore, in a non-blocking(N) cache, there can be up to N misses being serviced concurrently, and the service of hits can be overlapped with the service of misses.

[8] If the servicing of all misses is overlapped completely, in a pipelined fashion with a single port on the L1-L2 bus, the time taken to service M misses is $M(1+d)B + T_m$, which can be approximated by $M(1+d)B$.

The MSHRs have two major functions: (i) determining whether a *secondary miss* has occurred (a secondary miss is a miss to a block on which there is already a miss request pending) and (ii) routing the data supplied by the memory to the correct cache block and CPU register. For hits, a non-blocking cache is no different from a blocking cache. When a miss occurs, the MSHRs are checked (associatively) to see whether there is a pending miss to the cache block, i.e., whether the current miss is a *secondary* miss. If there is no pending miss to the same cache block, i.e., the current miss is a *primary* miss, a free MSHR is obtained (the cache stalls the processor if all MSHRs are being used). Information relevant to the servicing of the current miss, such as the cache block number and the CPU register to which the accessed data word must be routed, are entered in the MSHR and the miss request is submitted to memory. When the block is returned from memory, information in the appropriate MSHR is used (we will see how to access the "appropriate" MSHR shortly) to route the data both to the cache for further use, as well as to the CPU register. If a secondary miss occurs, the processor can continue, without a very complex MSHR design [10], unless the miss is to the same *word* to which there is a previous miss outstanding. Readers interested in more details of a single-ported, MSHR-based[9] non-blocking cache are referred to [10].

2.2.2. Additional Requirements of a Non-Blocking Cache

Let us consider what the additional requirements introduced by a non-blocking cache are with respect to a blocking cache. First, when a miss occurs, N MSHRs have to be searched associatively to determine whether the miss is a secondary miss or a primary miss, whereas no such associative search is needed in a blocking cache. Although it may be possible to design a non-blocking cache without penalizing the hit time, a wide associative search is still time-consuming in most cases, and every attempt should be made to keep the associative search confined to as few

[9]It is possible to have alternate designs that accomplish the same task as Kroft's design without limiting the number of MSHRs. The exact mechanisms that allow multiple outstanding misses to be handled is, in our opinion, highly dependent upon the particular situation, and is still an open question.

MSHRs as possible. Second, to allow the servicing of more than one miss to be overlapped, the L1-L2 interface must be pipelined, or packet-switched, whereas a circuit-switched L1-L2 bus is sufficient for a blocking cache. Third, if the L2 cache is handling more than one request concurrently, not only must it be pipelined to provide the bandwidth necessary to handle the requests, there must also be a way of routing return requests to the "appropriate" MSHR and from there to the requester in the CPU and to the correct cache block.

To match L2 cache responses with the appropriate L1 cache requester, we have two main options, both of which make demands of the L2 cache that are not made by a blocking L1 cache. The first option is to tag the miss request submitted to the L2 cache with the number of the MSHR of the L1 cache. When the L2 cache responds, it returns the tag along with the response, and the tag is used to access the correct MSHR and route the data. This option requires both the L1-L2 bus and the L2 cache to have special lines dedicated to the tags (bidirectional address lines could also be used as tags). The second option is for the L2 cache to return responses in the same order that it received the requests. In this case, the MSHRs can be managed as a queue, without the need for tags, and no additional lines are required on the L1-L2 bus. However, the burden is on the L2 cache to return the responses in the order that they were received.

2.3. Improving Bandwidth of Hits: More L1 Cache Ports

Having reduced the bandwidth degradation due to misses with a non-blocking cache, let us now consider how to improve the bandwidth to greater than 1 request per cycle by providing multiple ports to service hits. If we provide multiple ports for the L1 cache to service multiple hits simultaneously, with a single L1-L2 port, the bandwidth of the cache can be improved to:

$$Min\left[P_h, \frac{1}{m(1+d)B}\right] \tag{3}$$

where P_h is the number of ports from the CPU to the L1 cache. Let us now consider how we can provide multiple ports.

2.3.1. Duplicate Cache Banks

A straightforward way to implement multiple read ports is to provide multiple copies of the cache. For example, to provide 4 read ports for a 16Kbyte cache, we can have four 16Kbyte caches that have identical contents. We feel that this approach has a significant overhead in the amount of memory used, especially when considering an on-chip cache. Moreover, identical multiple copies allow only a single write port. Therefore, we do not consider a straightforward duplication of cache banks to be an adequate solution, if we need multiple read ports without a significant memory overhead and/or need multiple write ports.

2.3.2. Interleaved Banks

A better way to provide multiple cache ports is to interleave the cache blocks amongst multiple cache banks, much in the same way as an interleaved memory. Figure 3 shows how an interleaved L1 cache could be placed in the CPU and Figure 4 shows how the bits of a 32-bit address of a byte-addressable machine could be used to access a cache interleaved on the low-order index bits[10]. If there are M banks, and the cache stalls the processor on a miss, we have a *multi-port, blocking*, or *MPB(M)* cache, which can service up to M hits simultaneously (one to each bank), but only one miss at any time.

2.3.3. Multi-Port, Non-Blocking (MPNB) Caches

If each bank of a multi-port interleaved cache is a non-blocking(N) cache, i.e., has its own set of N MSHRs, with M banks we have an *MPNB(N, M) cache* that can collectively service up to M requests (each bank has a single read/write port) in a single clock cycle, as well as allow up to $N \times M$ misses to be overlapped simultaneously (with only N-way associative searchs) to reduce the bandwidth degradation due to misses.

[10]Using the low-order bits of the set to interleave the banks is analogous to the standard low-order interleaving used in memory systems. Other interleaving schemes could be used, and need further study.

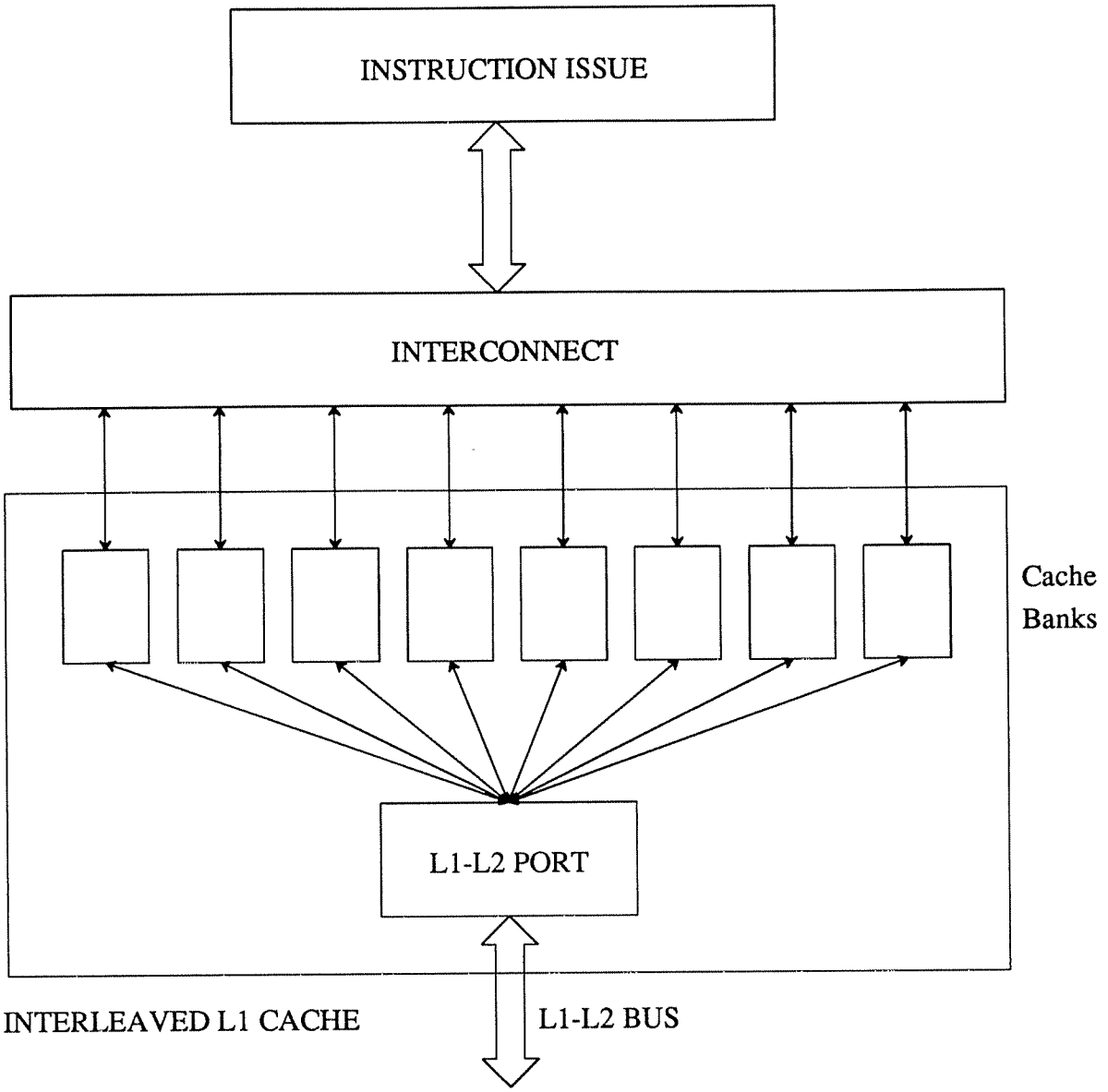
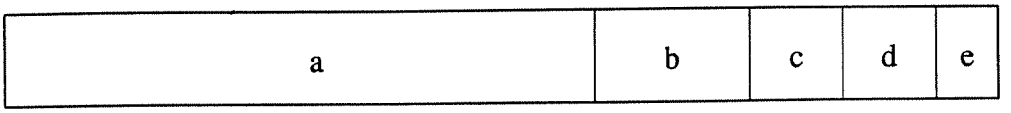


Figure 3: A Multi-Port Cache with Interleaved Banks



a – cache tag
b – set within a cache bank
c – cache bank
d – word within cache block
e – byte within word

Figure 4: Addressing a Multi-Port, Interleaved Cache

One potential drawback of an MPNB(N,M) cache design is the additional complexity and delay introduced by the crossbar from the instruction issue mechanism to the multiple cache banks (see Figure 3). Passing through this interconnect to get to the cache can potentially degrade the latency of cache hits. However, if we are to provide a bandwidth of greater than 1 request per cycle, we have no option but to provide such an interconnect between the instruction issue mechanism and the L1 cache. Moreover, the increase in the complexity of the interconnect may not be as bad as it sounds since we already have a $P\times(F+1)$ crossbar between the instruction issue mechanism and the functional units, where P is the peak number of instructions that can be issued per cycle and F is the number of computational functional units, and with M cache banks, the complexity is increased to $P\times(F+M)$. We can perhaps pipeline it so that passage through the interconnect is just an extra stage in the execution of an instruction. This potential latency loss for cache hits, in favor of increased bandwidth, needs further study.

2.4. Further Reduction in Bandwidth Degradation Due to Misses: More L1-L2 Ports

The techniques that we have considered so far for the L1 cache use mostly the on-chip hardware, and pose relatively few demands on the off-chip hardware (we only required the L1-L2 bus to be pipelined, the L2 cache to accept requests at a peak rate of 1 per cycle, and possibly return requests in order). We can reduce the bandwidth degradation due to misses even further by providing multiple ports on the L1-L2 bus. If we provide P_m ports on the L1-L2 bus, the

minimum time required to service M misses can be further reduced to $\frac{M(1+d)B}{P_m}$, and the peak bandwidth be improved to:

$$Min\left[P_h, \frac{P_m}{m(1+d)B}\right] \tag{4}$$

A multi-port L2 cache can be designed in ways similar to the ways proposed for a multi-port, non-blocking L1 cache. In fact, all of the design options for L1-L2 cache interactions could be applied to L2-memory interactions. Before going to multiple L1-L2 ports, however, we should first use the pin resources in the L1-L2 interface to maximize the bandwidth of the single port rather than to increase the number of ports. That is, we might use the additional pins to have larger block sizes that lower m, keeping $B = 1$. If m for an L1 cache can be made reasonably small (say 0.05-0.1), and the small m can be achieved with a small B (say 1), we can achieve a data bandwidth sufficient to support the issue of perhaps ten instructions per cycle with only a single L1-L2 port and an appropriate MPNB L1 cache, and therefore we do not expect multiple L1-L2 ports to be needed, at least for the next several generations of superscalar processors (though we would perhaps need a single, wider port so that $B = 1$).

3. Simulation Studies

In this section, we present some simulation studies to evaluate the potential and utility of MPNB caches. The simulation results are not meant to be exhaustive. Rather, they are intended to verify the observations of section 2 that blocking caches will be unable to support the data bandwidth requirements of future-generation superscalar processors, and that multi-ported, non-blocking caches are better able to do so.

3.1. Evaluation Environment

All our experiments are carried out with a detailed, cycle-by-cycle simulator that we have developed. The instruction set architecture for the simulator is that of the MIPS R2000; the simu-

lator accepts *a.out* files compiled for a DECstation 3100, and simulates their execution. Most aspects of the CPU and the memory system are modeled in detail (at the clock cycle level) by the simulator. The simulator is also detailed enough to handle the system calls (with traps to the OS) made by most programs. This allows benchmarks with file I/O, such as the SPEC benchmarks, to be simulated. By varying the parameters of the instruction issue mechanism, the memory system, and the resource architecture, we can simulate in detail the execution of an arbitrary program, along with its file I/O.

Because of the detail at which the simulation is carried out, and because the entire memory system is modeled, the simulator is slow. Depending upon the complexity of the instruction issue strategy, the resource architecture, and the memory system, we can simulate roughly 2,000-5,000 MIPS R2000 instructions per second on a DecStation 3100 hardware platform. This speed restricts our ability to explore the design space in great detail using substantial runs of large benchmark programs.

3.2. Baseline System

Our baseline system has a CPU with the instruction set architecture of a MIPS R2000, a 16Kbyte, direct-mapped L1 instruction cache and an L1-L2 bus that has separate address and data buses, each of which is 32 bits wide. With the above L1 instruction cache, we rarely encounter instruction cache misses for our benchmarks, and L1 instruction cache misses account for negligible traffic on the L1-L2 bus.

Since we are mainly interested in the L1 cache, we assume that all L1 misses hit in the L2 cache. The L2 cache is organized as a single-ported, interleaved memory, with 32 banks and a bank busy time of 4 clock cycles. Thus data can be transferred between the L1 and L2 cache at a peak rate of 4bytes per clock cycle, regardless of the latency of the L2 cache, if no L2 cache bank conflicts occur.

The baseline L1 data cache is 8Kbytes, direct mapped, virtually addressed, and has a hit time of 1 clock cycle. The blocking version is an 8-way interleaved (MPB(8)) cache, and the non-blocking version is an 8-way interleaved cache, with 4 MSHRs in each cache bank, i.e., an MPNB(4,8) cache (To have a uniform basis for comparison, we use the same basic organization throughout.)

3.3. Instruction Issue Strategy

For our simulations, ideally we would like to use instruction issue strategies that can issue about ten instructions per clock cycle, and perhaps sustain an issue rate of 3-5 instructions per cycle. Unfortunately, we are unaware of any known strategy that fits this model (though we are aware of several research efforts, including our own). Therefore, we will use a published instruction issue strategy, which can sustain a much smaller issue rate than what we expect to see in the future, and try to extrapolate the results.

The issue strategy that we use is the one implemented in the SIMP processor [12]. It uses dynamic dependency resolution and can issue up to 4 instructions per clock cycle. However, we do not do any branch prediction and speculative execution, i.e., we do not go beyond basic blocks to enhance instruction-level parallelism. Furthermore, we consider only a single floating point co-processor. Experiments with branch prediction and multiple floating point co-procerssors are the subject of our future studies.

3.4. Benchmarks and Miss Ratios

We use 4 benchmarks for our experiments: `doduc`, `eqntott`, `matrix300` and `tomcatv`, all taken from the SPEC benchmark suite. The benchmarks are long programs, and take several minutes to run in their entirety on a DECstation 3100 hardware platform. Since the entire execution of the benchmarks for all the different configurations will take several months to simulate on our simulator, we simulate the execution of only the first `5 million` instructions that

occur for each benchmark (this perhaps captures only the initialization phases in most cases, but that is not important since we use the same portion of code for all the different cases and the trace is long enough for the cache size considered). We also carried out simulations for 100 million instructions and observed that the results are not significantly different from those with 5 million instructions. These results are presented in the appendix as Tables A.1 – A.4 and Figures A.5 – A.6, analogous to Tables 1 – 4 and Figures 5 – 6 presented in this section for 5 million instructions.

Table 1 presents the number of memory references (in millions) in the simulated portion of each benchmark, and the execution times (in millions of clock cycles) with a perfect memory system, i.e., a memory system in which all memory references are serviced in a single cycle, and the corresponding average data memory bandwidth demanded (BW_D). The data memory bandwidth demanded is simply the total number of data references divided by the execution time.

From Table 1 we can see that the issue strategy that we have considered is not aggressive enough since the average number of instructions executed per clock cycle ranges only from 1.15 for **`matrix300`** to 1.95 for **`eqntott`**, *even assuming a perfect memory system.* Moreover, the issue strategy does not make a very heavy demand on the data memory bandwidth (0.375-0.637 requests per cycle). As issue strategies become more sophisticated, and try to sustain an

Table 1: Benchmark Data

Benchmark	Memory References (millions)		Performance with Perfect Memory		
	Loads	Stores	Cycles (millions)	Issue Rate	BW_D
doduc	1.350	0.538	4.04	1.24	0.468
eqntott	0.936	0.361	2.56	1.95	0.507
matrix300	0.726	0.904	4.35	1.15	0.375
tomcatv	1.718	0.660	3.73	1.30	0.637

execution rate of 3-5 instructions per cycle, the demand for data bandwidth will increase because: i) fewer clock cycles are taken to execute the program and service the same number of "useful" data references and ii) additional data references may be generated that are not "useful", i.e., do not influence computation, because of speculative execution beyond a basic block.

In Table 2, we present the miss ratios obtained from simulation for various block sizes, and the corresponding bandwidth that a single-ported blocking L1 cache can supply (computed from equation (1)), assuming $T_m = 12$ and an L1-L2 bus width of 4 bytes. We will discuss the data of Table 2 shortly.

3.5. Experimental Results

3.5.1. Execution Times and Speedups

In Figure 5 we present the execution and processor cache stall times for nine memory configurations for each of the benchmarks, as obtained from our simulator. The execution time is the actual number of clock cycles taken to execute the first 5 million instructions, with the particular cache organization. The first set of 4 bars for each benchmark are for an 8Kbytes direct mapped MPB(8) cache, and the second set of 4 bars are for an 8Kbytes direct mapped MPNB(8) cache. The 4 bars of each set are for block sizes of 4, 8, 16, and 32 bytes, respectively. The last

Table 2: Miss Ratios and Bandwidth Supply with a Blocking Cache; $T_m = 12$

Benchmark	Block Size (Bytes)							
	4		8		16		32	
	m (%)	BW_S	m (%)	BW_S	m (%)	BW_S	m (%)	BW_S
doduc	12.88	0.393	6.81	0.530	5.05	0.569	4.80	0.523
eqntott	18.52	0.310	11.14	0.408	7.22	0.480	5.21	0.502
matrix300	62.04	0.115	31.08	0.198	15.76	0.297	8.12	0.393
tomcatv	30.90	0.212	15.47	0.332	10.18	0.396	7.84	0.401

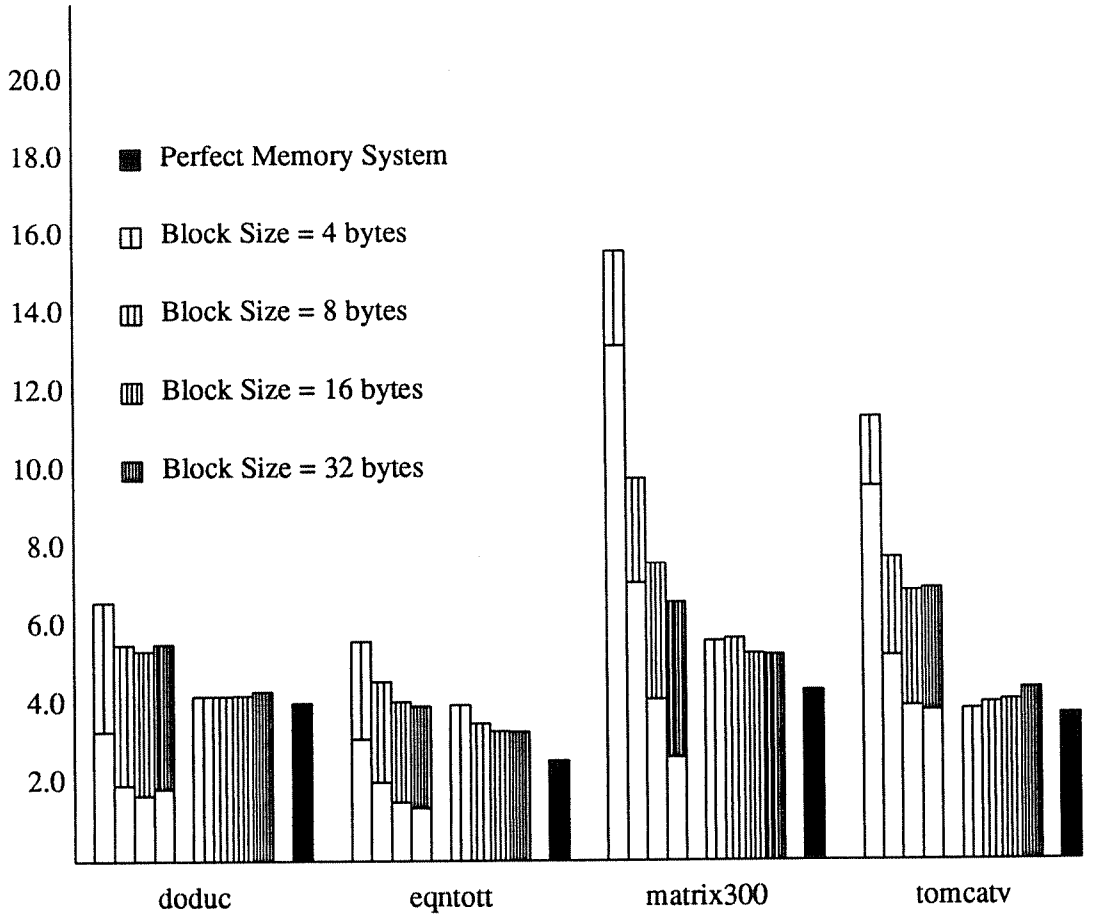


Fig. 5: Execution Times and Processor Cache Stall Times for Different Memory Configurations.

(dark shaded) bar for each benchmark is for a perfect memory system. The height of each bar is the total execution time, and the blank portion of the bar is the *processor cache stall time*, for that particular memory configuration. The processor cache stall time is the amount of time that the processor is blocked because the L1 cache cannot accept a request from it. The L1 cache will not accept processor requests when the limits of its abilities are reached with the miss requests that it is already servicing. The processor cache stall that occurs for a cache organization, therefore provides one lower bound on the execution time of the program with that cache organization.

Figure 6 explicitly shows the speedups obtained for each of the 4 block sizes for each benchmark, by presenting the percentage improvement in execution time while going from an MPB(4,8) cache to an MPNB(4,8) cache.

The first thing to notice from Figures 5 and 6 is that the execution time with an MPNB(4,8) cache is lower than that with an MPB(8) cache, even for the cases where an MPB(8) cache can provide sufficient bandwidth. For example, the best MPNB(4,8) configuration can improve the execution time by 27.3%, 20.1% and 25.5% for **doduc**, **eqntott** and **matrix300**, respectively. This is despite the fact that an MPB(8) cache provide adequate average bandwidth for our issue strategy (see Tables 1 and 2) in these cases. The execution time improves because, although an MPB(8) cache can meet the average bandwidth demand, it is unable to meet the peak miss bandwidth demand that arises when several misses occur close to each other, whereas an MPNB(4,8) cache can easily meet this demand. In other words, because of its higher "peak" bandwidth, an MPNB(4,8) cache allows some memory requests to be serviced earlier than they would with an MPB(8) cache, thereby allowing other instructions to be issued earlier, and consequently the total execution time to be reduced.

In cases where the peak bandwidth of an MPB(8) cache is not sufficient to meet even the average bandwidth demands of our issue strategy, significant improvements in execution time result, in going from an MPB(8) cache to an MPNB(8) cache. For example, for **tomcatv**, an

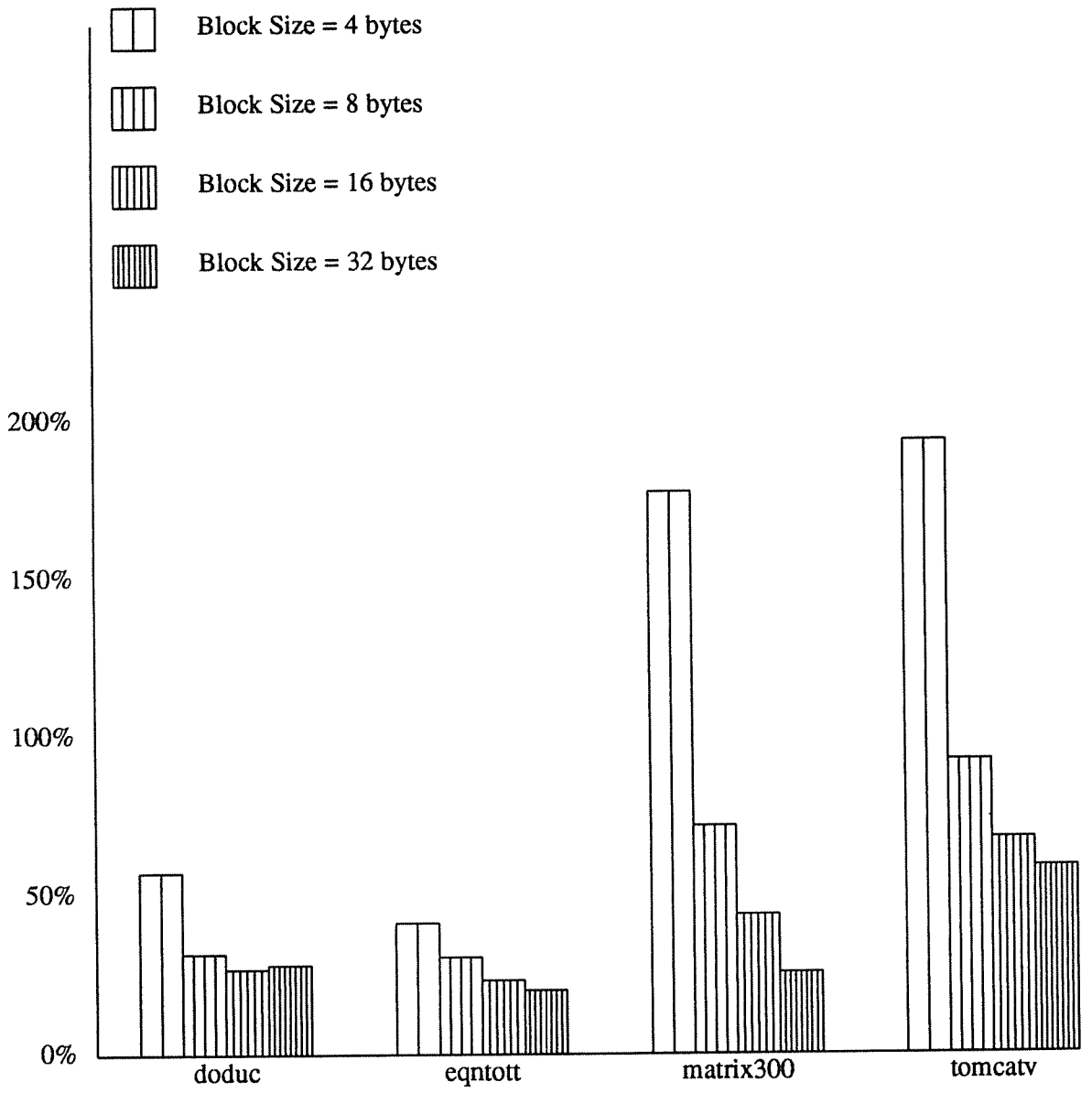


Fig. 6: Percentage Improvement in Execution Time with an MPNB(4,8) Cache over an MPB(8) Cache.

MPNB(4,8) cache is able to achieve an 80.1% performance improvement over the best MPB(8). For a block size of 4 bytes, where an MPB(8) cache does not have sufficient bandwidth to support the demands of the issue strategy for any of the benchmarks, execution time is improved by 57%, 41.2%, 177.6% and 193.5% for **doduc**, **eqntott**, **matrix300**, and **tomcatv**, respectively. Another point to note from Figure 5 is that in most cases, performance with an MPNB(4,8) cache is close to the performance with a perfect memory system for the issue strategy considered, indicating little room for further improvement in the memory system for the issue strategy.

For studying high-bandwidth caches for future issue strategies, more important than the program execution time that occurs with a cache organization and a particular issue strategy is the processor cache stall time that occurs because of the cache organization, since it presents one lower bound on performance. As we can see from Figure 5, the processor cache stall is a significant portion of the execution time with the blocking cache configurations, and is a negligible part with the non-blocking cache configurations. Therefore, with a blocking cache, we expect little room for further improvements in the issue strategy, whereas with an MPNB cache, we expect considerable freedom to support more sophisticated issue strategies. Table 3 reinforces this expectation by presenting the best-case execution times and instruction issue rates calculated for 3 cache organizations: i) a single-ported blocking cache, ii) an infinite-ported blocking cache (with one 32-bit L1-L2 port) and iii) a multi-ported, non-blocking cache (with one 32-bit L1-L2 port). In each case, the cache is 8Kbytes and is direct-mapped. The execution times are calculated as the minimum time to service the misses on a single, 4byte wide L1-L2 data bus, with the assumption that all hits can be overlapped perfectly with the service of misses (in the case of an MPB cache, the service of hits is actually overlapped with the service of the writeback request for the previous miss). Other relevant parameters are $d = 0.5$ and $T_m = 12$. As we can see by comparing Table 1 and 3, a blocking cache leaves little room for further improvements in the instruction

Table 3: Best-Case Execution Times and Instruction Issue Rates
8Kbyte, direct-mapped L1 Data Cache with 32-bit wide L1-L2 Data Bus

Benchmark	Single-Ported Blocking Cache		Infinite-Ported Blocking Cache		Multi-Ported Non-blocking Cache	
	Execution Time	Issue Rate	Execution Time	Issue Rate	Execution Time	Issue Rate
doduc	3.32	1.506	1.716	2.914	0.365	13.699
eqntott	2.58	1.938	1.621	3.085	0.360	13.889
matrix300	4.14	1.208	3.177	1.574	1.516	3.298
tomcatv	5.92	0.845	4.357	1.148	1.103	4.533

issue strategy (in the case of **tomcatv**, it can't even support the average bandwidth demands of our issue strategy!), even with infinite hit ports. There is still sufficient room for improvements in the issue strategy with an MPNB cache and our 8Kbyte, direct-mapped, MPNB(4,8) cache could possibly support instruction issue rates of 13.699, 13.889, 3.298 and 4.533 instruction per clock, for **doduc**, **eqntott**, **matrix300** and **tomcatv**, respectively, if we had instruction issue strategies capable of achieving these issue rates.

Table 4 presents the best-case execution times and instruction issue rates for the same parameters as in Table 3, but with a 128-bit wide L1-L2 bus instead of a 32-bit wide L1-L2 bus.

Table 4: Best-Case Execution Times and Instruction Issue Rates
8Kbyte, direct-mapped L1 Data Cache with 128-bit wide L1-L2 Data Bus

Benchmark	Single-Ported Blocking Cache		Infinite-Ported Blocking Cache		Multi-Ported Non-blocking Cache	
	Execution Time	Issue Rate	Execution Time	Issue Rate	Execution Time	Issue Rate
doduc	2.975	1.681	1.359	3.679	0.143	34.965
eqntott	2.175	2.299	1.014	4.931	0.140	35.714
matrix300	3.217	1.554	1.985	2.519	0.385	12.987
tomcatv	4.615	1.083	2.796	1.788	0.363	13.774

The wider L1-L2 bus has less impact on the peak performance that can be achieved with a single-ported, blocking cache (11.6%-28.6% improvement), and a somewhat bigger impact on peak performance with an infinite-ported, blocking cache (26.3%-60% improvement), but a tremendous impact on peak performance with a multi-ported, non-blocking cache (155%-294% improvement). In other words (as is also apparent from equations (1) and (3)), multi-ported, non-blocking caches are better able to make use of the additional pin-bandwidth that we expect to see in the future. With a single 128-bit wide L1-L2 port, a multi-ported, non-blocking cache can support an issue strategy with an issue rate of greater than 10 instructions per cycle for all our benchmarks.

3.5.2. Tolerance to Miss Ratio

Another point to notice from Figure 5 is that performance with an MPNB cache is not very sensitive to the miss ratio (the miss ratio varies greatly with the block size as shown in Table 2), whereas the performance with an MPNB cache is sensitive to the miss ratio. Clearly the total time taken to service misses is greater if the product of the number of misses (or the miss ratio) and the miss time is higher. However, unlike a blocking cache, where the time taken to service misses occurs serially, and is added to the time in which useful computation is carried out, a non-blocking cache allows the miss service time to be overlapped both with the service of other misses, as well as with useful computation, and therefore has a less significant impact on the total execution time.

3.5.3. Tolerance to Miss Time

As a final point, let us consider the ability of an MPNB cache to tolerate a large miss time for our issue strategy. Figure 7 presents the execution and processor cache stall time for 8Kbyte, direct mapped MPB(8) and MPNB(4,8) caches, with a block size of 32 bytes, as T_m is varied, for our issue strategy. The first set of four bars for each benchmark are for an MPB(8) cache (8Kbytes, direct mapped with block size of 32 bytes), for miss times of 10, 15, 20 and 25 clock

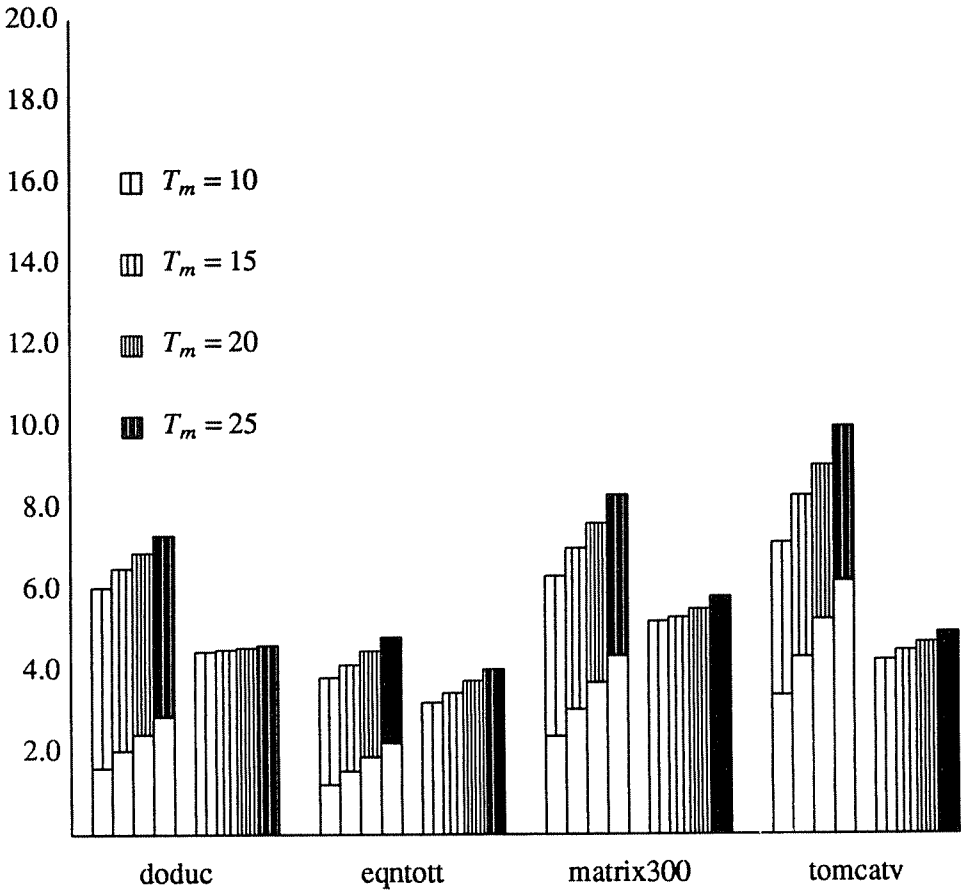


Fig. 7: Execution Times and Processor Cache Stall Times with Different Memory Latencies.

cycles, respectively and the second set of four bars for each benchmark are the same for an MPNB(4,8) cache (8Kbytes, direct mapped with block size of 32 bytes). From the figure, we can see that the execution time with an MPNB(4,8) cache is less sensitive to increases in T_m than that with an MPB(8) cache. Furthermore, except in the case of **eqntott**, an MPNB(4,8) cache with $T_m = 25$ actually allows better performance than an MPB(8) cache with $T_m = 10$!

4. Conclusions

As the instruction issuing capabilities of processors are improved, allowing the issue of several instructions per clock cycle, the bandwidth of the data memory system must be improved commensurately. We have considered ways of providing a high-bandwidth data memory hierarchy in this paper, with a level 1 data cache at the top of the hierarchy, using the flexibility in the use of on-chip real estate that might be provided by a future-generation, single-chip processor, and without requiring many demands of the off-chip components. To the best of our knowledge, this is the first paper that considers the bandwidth that a cache-based memory system can provide to the CPU — a metric that ultimately dictates the performance that the processor can achieve.

We saw that unless both the miss ratio and the miss time for the L1 cache are very low, its bandwidth can suffer greatly if it is a standard blocking cache because of the serial service of misses. To reduce this bandwidth degradation, we considered non-blocking caches and saw how they would impact other components of the system. To further improve the bandwidth of the memory system to more than one request per cycle, we proposed interleaving the L1 cache to create a multi-ported cache. Our proposed multi-port, non-blocking (MPNB) cache design allows multiple memory requests to be serviced in a single cycle, with only a single port to the off-chip memory.

We also presented results of a detailed cycle-by-cycle simulation for 4 benchmarks, compiled for a DecStation 3100. Our simulation results suggest that the proposed MPNB caches are a good choice for meeting the high data bandwidth demands of future-generation superscalar processors.

The work presented in this paper addresses only a few of the multitude of issues in the design of an adequate-bandwidth data memory system for superscalar processors. We expect that many of the design tradeoffs that have typically been studied in the context of blocking cache designs, may not be applicable to MPNB cache designs. For example, we saw that an increase in

both the miss ratio and the miss time had little impact on the performance with an MPNB cache for our benchmarks, whereas it had a significant impact on the performance with a multi-port blocking cache. Much work remains to be done in the area of multi-ported, non-blocking cache designs -- not only on which designs are better than others, but also on metrics to evaluate the design — and evaluation techniques that are computationally less expensive than a cycle-by-cycle simulation of the entire system.

5. Acknowledgments

The authors would like to thank Jim Goodman, Mark Hill, Jim Smith and David Wood for their comments on an earlier draft of this paper. The authors would also like to thank Jean-Loup Baer for his comments on a previous version of this paper. Financial support for Manoj Franklin was provided by an IBM Graduate Fellowship, and for Gurindar Sohi by NSF Grants CCR-8706722 and CCR-8919635.

References

[1] R. D. Acosta, J. Kjelstrup, and H. C. Torng, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors," *IEEE Trans. on Computers*, vol. C-35, pp. 815-828, September 1986.

[2] H. O. Bugge, E. H. Kristiansen, and B. O. Bakka, "Trace-Driven Simulations for a Two-Level Cache Design in Open Bus Systems," in *Proc. 17th Annual Symposium on Computer Architecture*, Seattle, WA, pp. 250-259, May 1990.

[3] P. P. Gelsinger, P. A. Gargini, G. H. Parker, and A. Y. C. Yu, "Microprocessors circa 2000," *IEEE Spectrum*, vol. 26, pp. 43-47, October 1989.

[4] J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *Proc. 10th Annual Symposium on Computer Architecture*, pp. 124-131, June 1983.

[5] G. F. Grohoski, "Machine Organization of the IBM RISC System/6000 processor," *IBM Journal of Research and Development*, vol. 34, pp. 37-58, January 1990.

[6] M. D. Hill, "Aspects of Cache Memory and Instruction Buffer Performance," Technical Report UCB/CSD 87/381, University of California at Berkeley, Berkeley, CA, November 1987.

[7] P. Y. T. Hsu and E. S. Davidson, "Highly Concurrent Scalar Processing," *Proc. 13th Annual Symposium on Computer Architecture*, pp. 386-395, June 1986.

[8] N. P. Jouppi and D. W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," in *Proc. ASPLOS III*, Boston, MA, pp. 272-282, April 1989.

[9] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," in *Proc. 17th Annual Symposium on Computer Architecture*, Seattle, WA, pp. 364-373, May 1990.

[10] D. Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," *Proc. 8th International Symposium on Computer Architecture*, pp. 81-87, May 1981.

[11] S. W. Melvin, M. C. Shebanow, and Y. N. Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines," in *Proc. 21st Annual Workshop on Microprogramming and Microarchitecture*, San Diego, CA, November 1988.

[12] K. Murakami, N. Irie, M. Kuga, and S. Tomita, "SIMP (Single Instruction Stream / Multiple Instruction Pipelining): A Novel High-Speed Single-Processor Architecture," in *Proc. 16th Annual Symposium on Computer Architecture*, Jerusalem, Israel, pp. 78-85, May 1989.

[13] R. R. Oehler and R. D. Groves, "IBM RISC System/6000 Processor Architecture," *IBM Journal of Research and Development*, vol. 34, pp. 23-36, January 1990.

[14] Y. N. Patt, W. W. Hwu, and M. Shebanow, "HPS, A New Microarchitecture: Rationale and Introduction," in *Proc. 18th Annual Workshop on Microprogramming*, Pacific Grove, CA, pp. 103-108, December 1985.

[15] D. N. Pnevmatikatos and M. D. Hill, "Cache Performance of the Integer SPEC Benchmarks on a RISC," *ACM SIGARCH Computer Architecture News*, vol. 18, pp. 53-68, June 1990.

[16] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 14, pp. 473-530, September 1982.

[17] A. J. Smith, "Bibliography and Readings on CPU Cache Memories and Related Topics," *ACM SIGARCH Computer Architecture News*, vol. 14, pp. 22-42, January 1986.

[18] J. E. Smith, "Decoupled Access/Execute Architectures," *Proc. 9th Annual Symposium on Computer Architecture*, pp. 112-119, April 1982.

[19] J. E. Smith and J. R. Goodman, "A Study of Instruction Cache Organizations and Replacement Policies," *Proc. 10th Annual Symposium on Computer Architecture*, pp. 117-123, June 1983.

[20] M. D. Smith, M. S. Lam, and M. A. Horowitz, "Boosting Beyond Static Scheduling in a Superscalar Processor," in *Proc. 17th Annual Symposium on Computer Architecture*, Seattle, WA, pp. 344-354, May 1990.

[21] G. S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," *IEEE Trans. on Computers*, vol. 39, pp. 349-359, March 1990.

Appendix

The results obtained with 100 million instructions are not significantly different from those with 5 million instructions. These results, presented below in Tables A.1 – A.4 and Figures A.5 – A.6, correspond to the tables and figures of Section 3.

Table A.1: Benchmark Data

Benchmark	Memory References (millions)		Performance with Perfect Memory		
	Loads	Stores	Cycles (millions)	Issue Rate	BW_D
doduc	28.003	10.106	89.775	1.114	0.424
eqntott	21.702	3.238	55.529	1.801	0.449
matrix300	17.971	9.527	98.945	1.011	0.278
tomcatv	34.755	11.740	68.181	1.467	0.682

Table A.2: Miss Ratios and Bandwidth Supply with a Blocking Cache; $T_m = 12$

Benchmark	Block Size (Bytes)							
	4		8		16		32	
	m (%)	BW_S	m (%)	BW_S	m (%)	BW_S	m (%)	BW_S
doduc	13.67	0.379	7.26	0.514	5.45	0.550	4.90	0.518
eqntott	17.72	0.320	10.50	0.423	6.81	0.495	5.17	0.504
matrix300	44.43	0.158	22.29	0.257	11.37	0.370	5.89	0.472
tomcatv	39.39	0.175	19.78	0.280	13.53	0.330	11.53	0.310

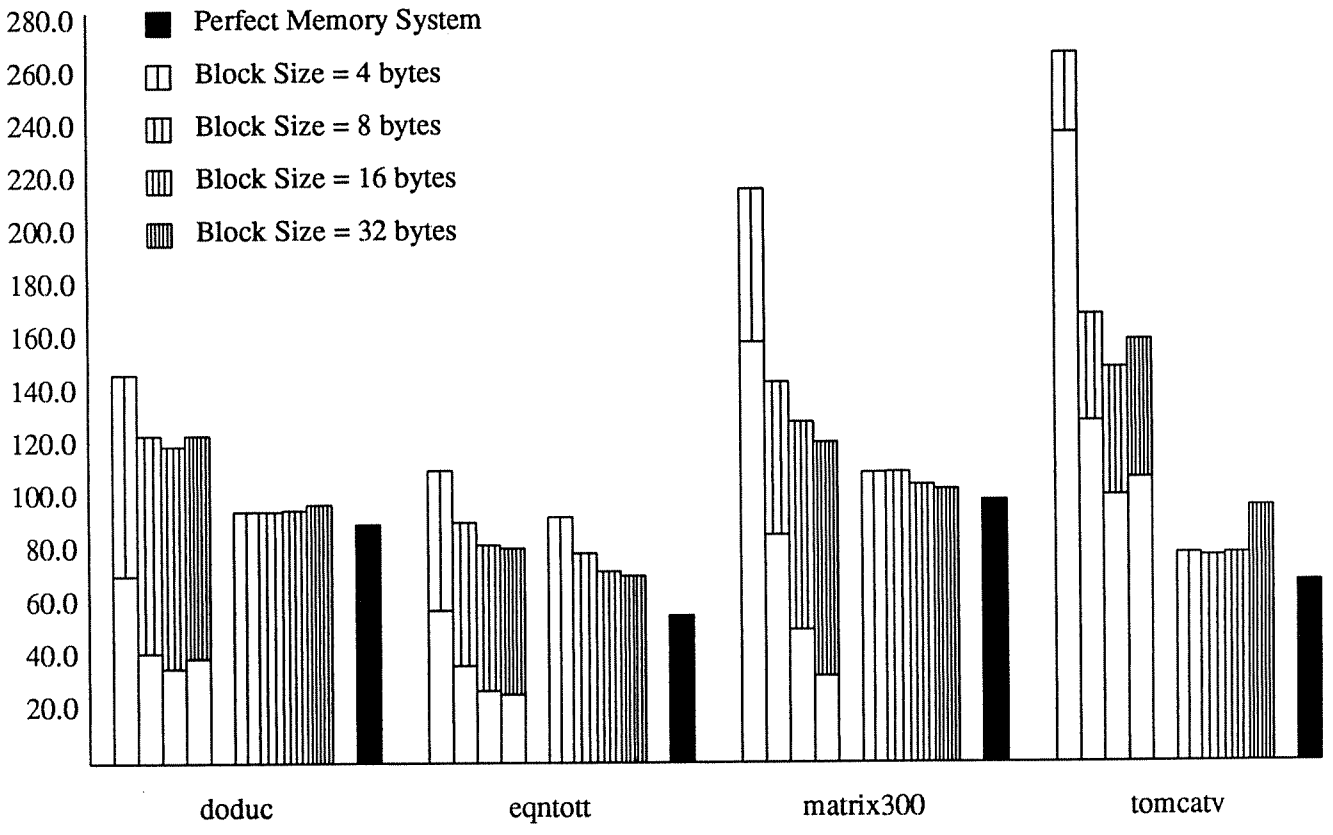


Fig. A.5: Execution Times and Processor Cache Stall Times for Different Memory Configurations.

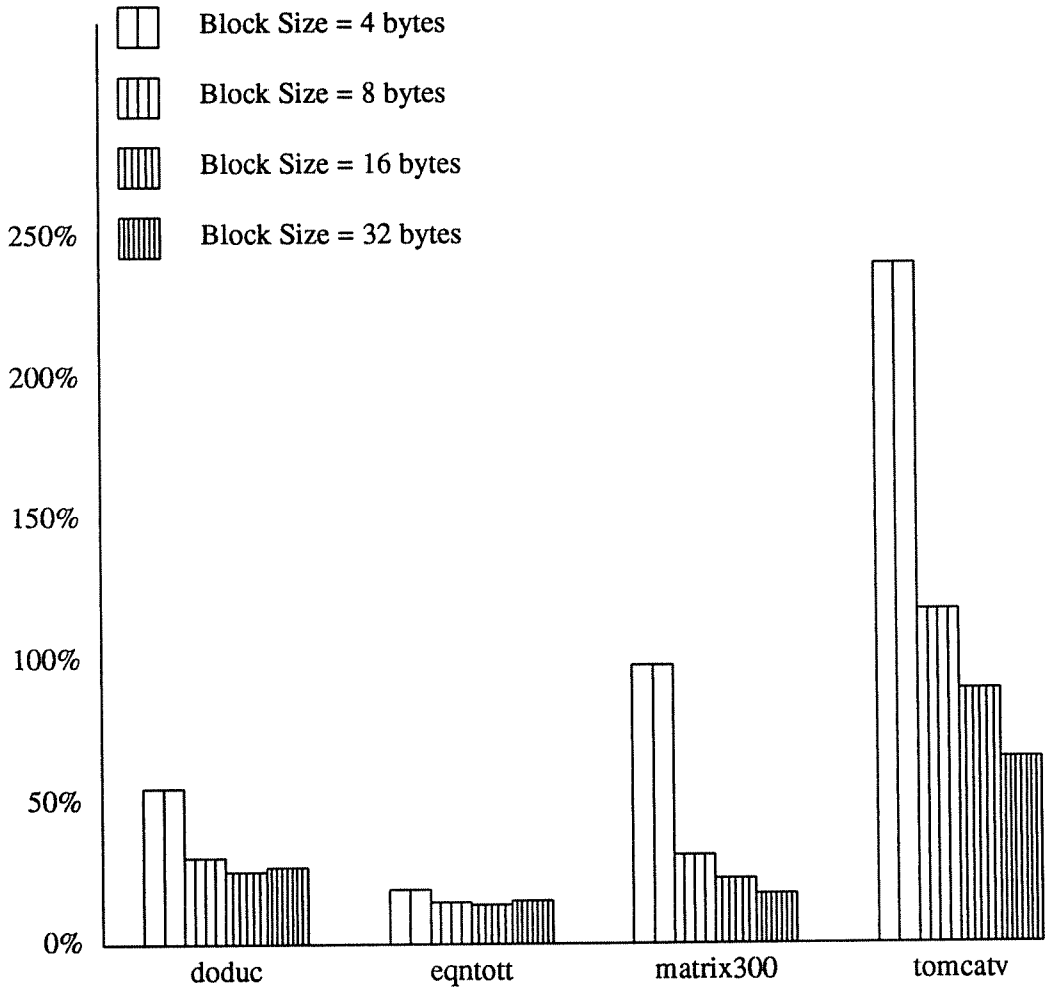


Fig. A.6: Percentage Improvement in Execution Time with an MPNB(4,8) Cache over an MPB(8) Cache.

Table A.3: Best-Case Execution Times and Instruction Issue Rates
8Kbyte, direct-mapped L1 Data Cache with 32-bit wide L1-L2 Data Bus

Benchmark	Single-Ported Blocking Cache		Infinite-Ported Blocking Cache		Multi-Ported Non-blocking Cache	
	Execution Time	Issue Rate	Execution Time	Issue Rate	Execution Time	Issue Rate
doduc	69.263	1.444	37.385	2.675	7.814	12.798
eqntott	49.439	2.023	30.571	3.271	6.629	15.085
matrix300	58.271	1.716	38.871	2.573	18.326	5.457
tomcatv	140.857	0.710	113.234	0.883	27.472	3.640

Table A.4: Best-Case Execution Times and Instruction Issue Rates
8Kbyte, direct-mapped L1 Data Cache with 128-bit wide L1-L2 Data Bus

Benchmark	Single-Ported Blocking Cache		Infinite-Ported Blocking Cache		Multi-Ported Non-blocking Cache	
	Execution Time	Issue Rate	Execution Time	Issue Rate	Execution Time	Issue Rate
doduc	62.384	1.603	28.010	3.570	3.115	32.103
eqntott	41.702	2.398	19.341	5.170	2.548	39.246
matrix300	48.553	2.060	24.294	4.116	4.690	21.322
tomcatv	116.186	0.861	80.413	1.244	9.436	10.598