High-Bandwidth Data Memory Systems for Superscalar Processors

Gurindar S. Sohi and Manoj Franklin
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53705

Abstract

This paper considers the design of a data 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 blocking 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 such data bandwidth demands.

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 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

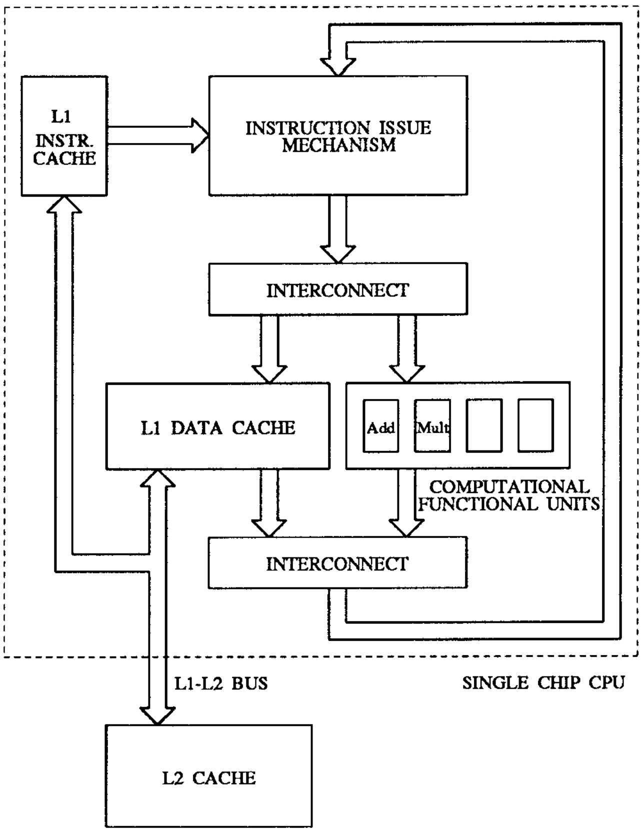

Figure 1: A Superscalar CPU

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

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 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 published [4-6,9-11,17,18,20], 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 [4,6,9-11,17,18]. Furthermore, because of the high hit ratios achievable for instruction caches [7,12,15], we feel that a small (few kilobytes) L1 instruction cache can be designed to support most instruction issue mechanisms, although the exact design of such a cache will be 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? The data memory (L1 data cache, L2 cache, and main memory) is perhaps the most heavily demanded resource and is likely to be a critical resource[3]. Accordingly, the peak instruction issue rate of our 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. As we shall see, 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, although 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 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 in section 4, we draw our conclusions.

2. Cache Bandwidth and High-Bandwidth Cache Designs

Most of the literature on cache memories [16] 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 [3], 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 load/store architecture such as the MIPS R2000) and, as

[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) [2].

[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.

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

we shall see, such a low average bandwidth could be squeezed out quite easily with most conventional 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, and possibly more if speculative execution is performed, 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.

Let us assume that the L1 cache is a writeback cache (a write-through cache can be analyzed similarly). 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 single-level blocking caches is well-understood [15, 16]; almost all computers built today have them. Multi-level blocking caches have also been investigated recently [13]. For our purposes, we note that 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 request at a time, 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 (of course, assuming that the L2 cache can sustain this service rate). Since the service of hits and misses can't be overlapped in a 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 minimum 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 $m=\dfrac{M}{H+M}$ 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 (which has 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 cache bandwidth if we hope to achieve a superscalar execution of more than a few instructions per clock cycle.

[5] If the CPU has a dynamic dependency resolution mechanism, as we expect it to, it can continue to issue instructions that have register-only operands, and need not 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 [7], if the L2 cache's access latency is sufficiently high, it may have to be sufficiently pipelined to handle multiple writeback requests.

Before proceeding further, from equation (1) we can also see why data 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 $1 \leq B \leq 4$.

To improve bandwidth, we have two options: (i) improve the bandwidth with which hits are serviced (by providing multiple ports to service hits) or (ii) reduce the bandwidth degradation due to misses. From equation (1), we can see that even if we provide an infinite bandwidth for cache hits, the bandwidth of a blocking cache can only be improved to $\frac{1}{m \times [T_m + B(1+d)]}$, 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, with a packet-switched bus in which the L1-L2 bus is not held for the duration of the memory request. In the best case, if all M miss requests can be overlapped perfectly, the time taken to service the misses can be reduced[8] to $M(1+d)B$ and the bandwidth of a single-ported cache can be improved to:

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

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. (Implicit in eqn. (2) is the assumption that the L2 cache has enough bandwidth to support the peak miss request bandwidth 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 [8]. 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.

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 another miss request is already 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 same 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

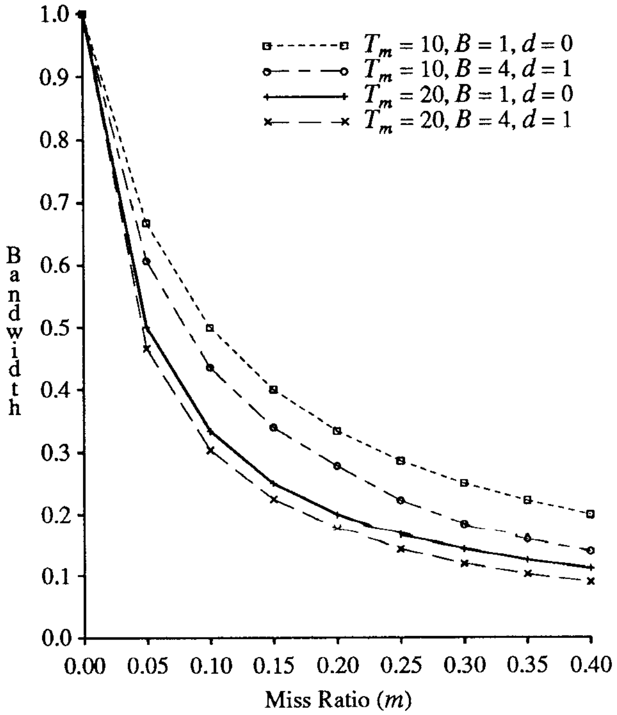

Figure 2: Data Bandwidth provided by a Blocking L1 Data Cache

[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 $M(1+d)B$.

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 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 shortly see how to access the "appropriate" MSHR) 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 [8], 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 [8].

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 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 designed 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. The order in which the L2 cache services requests and returns them to the L1 cache can be different from the order in which they were submitted to it because of L2 cache misses and L2 cache interleaving.

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 L1 cache's MSHR number. 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 for 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 — a task that can be complicated by L2 cache misses.

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, 4 read ports can be provided to a 16Kbyte cache by having 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[10], with a cache block present entirely in one cache bank. Figure 3 shows how an interleaved L1 cache could be placed in the CPU. If there are C banks, and the cache stalls the processor on a miss, we have a *multi-port, blocking*, or *MPB(C)* cache, which can service up to C hits simultaneously (one to each bank), but only one miss at any time.

[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.

[10] The straightforward way to interleave the banks is to use standard low-order interleaving. Other interleaving schemes, such as those described in [14, 19] could be used, and need further study.

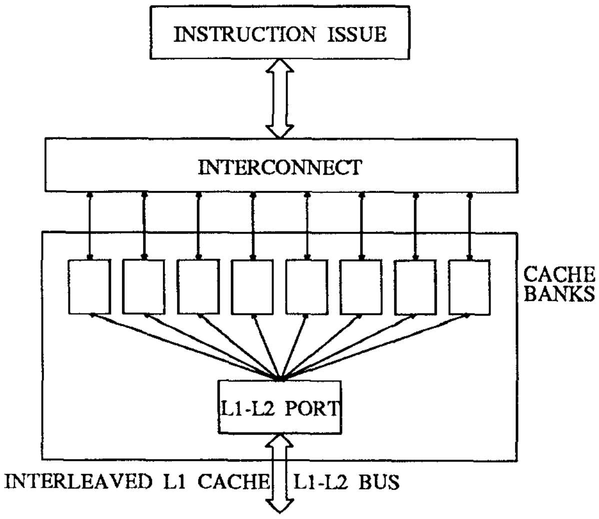

Figure 3: A Multi-Port Cache with Interleaved Banks

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

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

One potential drawback of an MPNB(N, C) 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, 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 degradation 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:

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

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 could 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 for while (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 support these requirements.

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 simulator 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.

Because of the detail at which the simulation is carried out, and because the entire memory system is modeled, the simulator is slow. 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 16Kword 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 data 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 4 bytes 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 simulation, 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 (although we are aware of several research efforts, including our own). Therefore, we will use a published instruction issue strategy, which sustains a much smaller issue rate than what we expect to see in the future.

The issue strategy that we use is the one implemented in the SIMP processor [10]. It uses dynamic dependency resolution and branch prediction, and can issue up to 4 instructions per clock cycle. We do not implement branch prediction and speculative execution, however, i.e., we do not go beyond basic blocks to enhance instruction-level parallelism.

3.4. Benchmarks and Miss Ratios

We use 4 benchmarks for our experiments: **doduc**, **eqntott**, **matrix300** and **tomcatv**, 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. Due to resource constraints, we simulate the execution of only the first **100 million** instructions that occur for each benchmark.

Table 1 presents the number of memory references (in millions) in the simulated portion of each benchmark, 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 average data memory bandwidth demanded (BW_D) during the execution of the program with the issue strategy considered. The data memory bandwidth demanded is calculated as the 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 1.011 for **matrix300** to 1.801 for **eqntott**, *even assuming a perfect memory system.* Moreover, the issue strategy does not make a very heavy demand on the data memory bandwidth (0.278-0.682 requests per cycle). As issue strategies become more sophisticated, and attempt to sustain an execution rate of 3-5 instructions per cycle (assuming, of course, that sufficient parallelism exists in the programs to support this issue rate), 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 by substituting these miss ratios in equation (1) and 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

In Figure 4 we present the execution and processor cache stall times obtained from our simulator for 9 memory configurations for each of the benchmarks. The execution time is the actual number of clock cycles taken to execute the first 100 million instructions, with the particular cache organization. The first group of 4 bars for each benchmark are for 8Kbytes direct mapped MPB(8) caches, and the second group of 4 bars are for 8Kbytes direct mapped MPNB(4, 8) caches. The 4 bars of each group correspond to block sizes of 4, 8, 16, and 32 bytes (block sizes of greater than 32 bytes are not considered because we feel that they are not a good design point for our system since they saturate our 32-bit L1-L2 bus). The last (unshaded) bar for each benchmark is the execution time with a perfect memory system. The height of each bar is the total execution time, and the lightly shaded bottom portion (if present) of each bar is the *processor cache stall time*, for that particular cache configuration. The processor cache stall time is the amount of time the processor is blocked from issuing instructions because the limits of the request handling abilities of the L1 cache have been reached, i.e., the cache is busy servicing the peak number of miss requests that it can handle.

The first thing to notice from Figure 4 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 26.3%, 15.1% and 17.4% for **doduc**, **eqntott** and **matrix300**, respectively. This is despite the fact that an MPB(8) cache provides adequate average bandwidth for our issue strategy (see Tables 1 and

Table 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 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

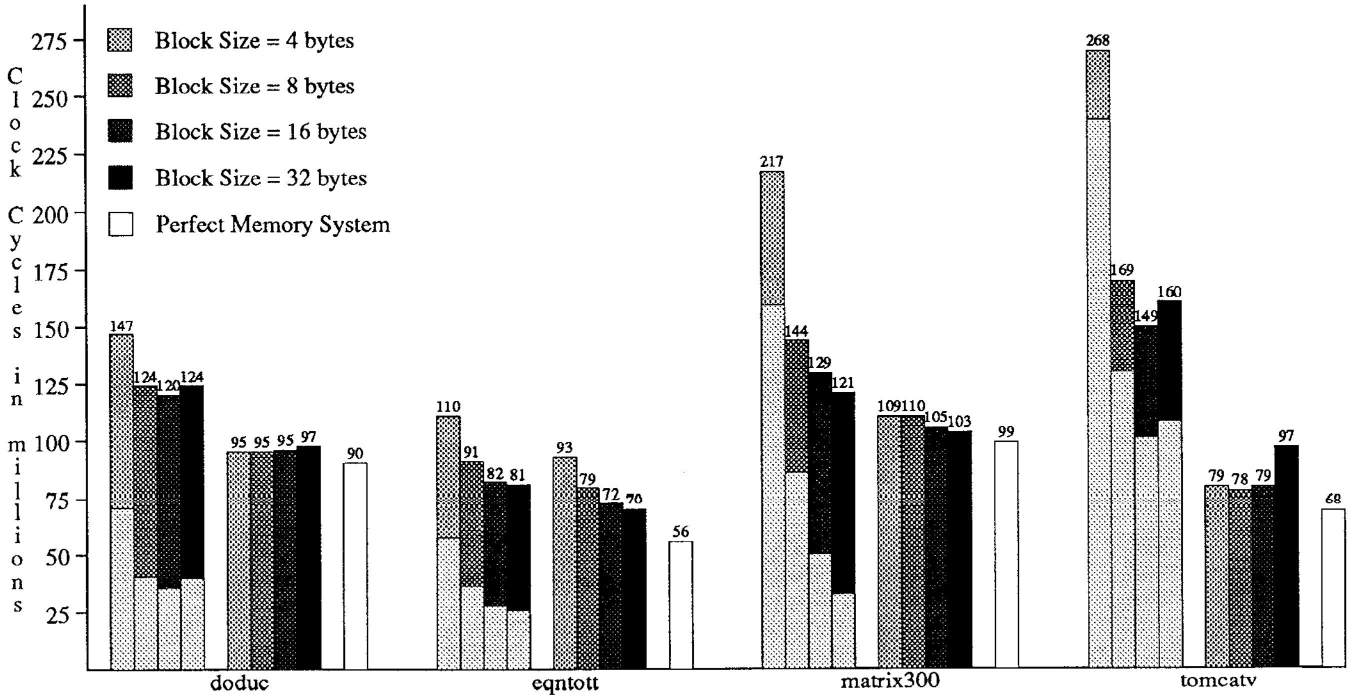

Figure 4: Execution Times and Processor Cache Stall Times for Different Cache Configurations

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 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(4, 8) cache. For example, for **tomcatv**, an MPNB(4, 8) cache is able to achieve a 91.9% 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 55.0%, 19.2%, 98.1% and 240.5% for **doduc**, **eqntott**, **matrix300**, and **tomcatv**, respectively. Another point to note from Figure 4 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.

To see if more sophisticated instruction issue strategies could be supported for our benchmarks, with the cache organizations considered, we consider the processor cache stall. Recall that this stall is the amount of time that instruction issue is blocked because the limits of the request handling abilities of the cache have been reached. Therefore, the processor cache stall time is one lower bound on the total instruction issue (and total program execution) time. As we can see from Figure 4, the processor cache stall is a significant portion of the execution time with the blocking cache configurations[11], indicating little room for further improvement in performance, even with more sophisticated issue strategies. That is, even if an instruction issue strategy, that is more sophisticated than the one we have considered, could overlap the execution of instructions to achieve high performance, and enough parallelism exists in the programs to allow a high degree of overlap, a blocking cache will prevent an improvement in performance by preventing the issue strategy from overlapping instruction issue with the service of misses, and the minimum amount of time taken to issue all program instructions will be bound by the time taken to service all the misses serially. Alternately, a blocking cache will not be able to supply the bandwidth demanded by a more sophisticated issue strategy.

For the non-blocking caches, the processor cache stall time is negligible[12]. This indicates that a non-blocking cache is not artificially constraining the issue strategy by preventing it from overlapping the service of misses with instruction issue and is therefore better able to support more sophisticated issue strategies.

4. Conclusions

As the instruction issuing capabilities of processors are improved to allow the issue of several instructions per clock cycle, among other things, the bandwidth of the data memory system must be improved commensurately. We have considered in this paper ways of providing a high-bandwidth data memory hierarchy, 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 — one of the most important metrics that ultimately dictate 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, the bandwidth can suffer greatly if the L1 cache 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, and that blocking caches are unlikely to be able to meet this demand.

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. 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

[11] For a blocking cache, since only one miss request can be serviced at a time, and since instruction issue is blocked on a miss, the processor cache stall time is simple the time taken to service the misses serially.

[12] In a non-blocking cache, if the cache is always able to accept a request from the processor, the processor cache stall time is zero. However, instruction issue can be stalled because of a dependency on a memory operand. With a non-blocking cache, this stall is a part of the processor execution time, and not of the processor cache stall. With a blocking cache, the processor stall due to a dependence on a memory operand (that missed in the cache) is overlapped completely with the processor cache stall, and counts as a part of the processor cache stall. Consequently, the non-processor-cache-stall execution time is not the same, with and without a non-blocking cache, in Figure 4.

evaluate the design — and evaluation techniques that are computationally less expensive than a cycle-by-cycle simulation of the entire system.

Acknowledgments

Financial support for Manoj Franklin was provided by an IBM Graduate Fellowship, and for Gurindar Sohi by NSF Grants CCR-8706722 and CCR-8919635. We would like to thank Hakon Bugge, Jim Goodman, Mark Hill, Jim Smith and David Wood for their comments on an earlier draft of this paper and Jean-Loup Baer for his comments on a previous version of this paper. We would also like to thank the five anonymous referees for their thorough and insightful comments. Their reviews have been very helpful to us in preparing this version of the paper.

References

[1] 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.

[2] 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.

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

[4] 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.

[5] P. Y. T. Hsu and E. S. Davidson, ''Highly Concurrent Scalar Processing,'' *Proc. 13th Annual Symposium on Computer Architecture*, pp. 386-395, June 1986.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] S. A. Przybylski, *Cache and Memory Hierarchy Design: A Performance Directed Approach.* San Mateo, California: Morgan Kaufmann, 1990.

[14] B. R. Rau, M. S. Schlansker, and D. W. L. Yen, ''The Cydra^{TM} 5 Stride-Insensitive Memory System,'' in *1989 Int. Conference on Parallel Processing*, St. Charles, IL, August 1989.

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

[16] 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.

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

[18] 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.

[19] G. S. Sohi, ''High-Bandwidth Interleaved Memories for Vector Processors - A Simulation Study,'' Computer Sciences Technical Report #790, University of Wisconsin-Madison, Madison, WI 53706, September 1988.

[20] 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.