The Effect of Limited Network Bandwidth and its Utilization by Latency Hiding Techniques in Large-scale Shared Memory Systems

Sunil Kim
IBM
11400 Burnet Road
Austin, TX 78758
skim@austin.ibm.com

Alexander V. Veidenbaum
Department of EECS
University of Illinois at Chicago
Chicago, IL 60607-7053
alexv@eecs.uic.edu

Abstract

This paper addresses the use of two latency hiding techniques, prefetching and weak consistency, for large-scale shared memory multiprocessors with compiler-controlled cache coherence management and the interaction of latency hiding techniques and network bandwidth. The performance effect of latency hiding is evaluated and compared varying the network channel bandwidth. The interaction of reads, writes, and prefetches given a limited bandwidth is studied, and an approach to better network bandwidth utilization by limiting the number of outstanding requests in each node is investigated. Increasing network (channel) bandwidth helps both prefetching and non-prefetching systems, with the initial 2x increase in bandwidth giving the most improvement. The use of prefetching can deliver a much larger improvement than increasing network bandwidth for a 128 processor system for some benchmarks, even with the minimal bandwidth. Controlling bandwidth utilization is shown to be important when prefetch and write request rates are high.

1 Introduction

Memory latency is a major problem in high-performance systems. In uniprocessors it is primarily the result of a speed gap between processor cycle time and DRAM access time. In shared memory multiprocessors the addition of interconnection networks and contention for resources significantly increases the memory latency. The latency grows with the number of processors in the system making it an even bigger problem for large-scale systems. To achieve high performance it is crucial to minimize the memory latency.

Several approaches have been proposed to alleviate this problem including cache hierarchy, non-blocking caches, multithreading, prefetching, and relaxed consistency models. A cache hierarchy has been successfully used in uniprocessor systems and a great deal of effort has been aimed at optimizing it for multiprocessor systems. However, multiprocessor caches have coherence, true and false sharing misses in addition to cold start, conflict, and capacity misses seen in uniprocessors. Cache coherence management can further increase memory traffic and latency. Because of this a deep cache hierarchy is often not as effective in large-scale multiprocessors as in uniprocessors. Latency-hiding techniques such as multithreading, prefetching, and relaxed consistency models have to be utilized to overcome the increased memory latency.

This paper addresses several issues in the use of two latency hiding techniques in large-scale shared memory multiprocessors, prefetching and weak consistency, their interaction, and their memory and network bandwidth requirements under compiler-controlled cache coherence management. For multiprocessor systems without cache coherence hardware, such as BBN Butterfly [25], IBM RP3 [24], IBM SP-2 [27], and Cray T3D [18], a software-controlled cache coherence management is the only viable solution.

Data prefetching has been proposed as a mechanism to hide a large memory latency [3, 12, 14, 15, 21, 23, 26] In data prefetching, future data accesses are predicted, and data is moved to the upper levels of a memory hierarchy before it is referenced, thereby hiding memory access latency. Both hardware and software prefetching methods have been proposed. Several of the above references demonstrated the effectiveness of prefetching for systems with 32 or fewer processors using a hardware cache coherence. Data prefetching demands more memory bandwidth because it increases the memory request rate (reduces memory access stalls) and the total number of memory accesses (speculative prefetch). Therefore, data prefetching may or may not perform as well in larger systems as in the small systems. Data prefetching in systems with more than 64 nodes, especially with compiler-controlled cache coherence, has not been adequately explored.

A consistency model choice is another important issue in hiding memory latency. Sequential consistency makes it difficult to hide latency, while weak or release consistency make it easy. We will assume that weak consistency is used and allows all writes to be buffered and completed asynchronously to the processor. Processor reads are allowed to bypass previous writes and issue to memory out of order. Weak consistency and write buffering reduce processor stalls on memory accesses. This has an effect of demanding more memory bandwidth by increasing the memory request rate and is similar to prefetching in this sense. Also, write traffic under compiler-controlled cache coherence has different network/memory bandwidth requirements and usage patterns than the traffic under hardware cache coherence and has to be analyzed.

Read misses, prefetches, and buffered writes access main memory and compete for the already scarce memory and network bandwidth, and their interaction affects the overall performance. For example, prefetches may delay read misses and writes can delay prefetches and reads degrading the overall system performance. The limited network/memory bandwidth has to be allocated to these three different types of requests in a way consistent with their priorities, bandwidth requirements, and effect on the system performance. Understanding the interaction of different types of memory requests under a *given limited* network/memory bandwidth is especially important as the system size grows to hundreds of processors. Finally, the effect of *bandwidth increase* on prefetching and other types of memory requests and their interaction needs to be examined.

This paper addresses the above-mentioned issues and concentrates on the performance impact and interaction of latency hiding techniques and network bandwidth in large shared-memory systems under software cache coherence. The system uses a multistage network to connect 128 nodes each containing a processor, cache, and memory. Tagged prefetching and weak consistency are chosen as representative latency hiding techniques. Even though a simple sequential prefetching was shown to be very effective in [7], we do not claim to be making an optimal choice of latency hiding techniques. Rather, our main interest is the interaction of prefetching, weak consistency, and the network bandwidth. First, the performance effect of latency hiding is evaluated and compared while explicitly varying the network channel bandwidth. Next, the interaction of reads, writes, and prefetches given limited bandwidth is studied, and an approach to better network bandwidth utilization, *bandwidth allocation*, is investigated.

Bandwidth allocation refers to assigning a fraction of available bandwidth to a particular type of traffic. The difficulty in doing this is a lack of knowledge of and direct control over the available network bandwidth. Thus a distributed solution is required using only information local to a node. It has been shown [8, 28] that limiting a total number of outstanding requests in each node can help in reducing network and memory latency, primarily by reducing contention. This study evaluates the effect this approach has on performance and whether allocating bandwidth individually to each request type, such as prefetch and write, is worthwhile.

The paper is organized as follows. The next section presents the system architecture and the performance evaluation methodology. The third section describes performance metrics. The fourth section presents our results and their analysis. The last section presents conclusions.

2 Evaluation Methodology

This section outlines the programming model and cache coherence mechanism of the system we evaluate. Next, a set of benchmarks is described, followed by the performance evaluation methodology using trace-driven simulation. Finally, a description of the system organization and the performance metrics used are presented.

2.1 A Programming Model and Cache Coherence

A shared-memory programming model based DOALL loops is used in this study [1] (*a DOALL programming model*). Under this programming model, a processor executes a subset of loop iterations within a DOALL as a single task. *COBEGIN and COEND* statements start and terminate a set of parallel tasks comprising a DOALL. A loop iteration may have private variables which are placed in local node's memory and their access by the processor executing the iteration does not require a network traversal. Shared memory is interleaved across all nodes and are accessed through the interconnect. The system uses weak consistency [11] and does not enforce cache coherence in hardware. Weak consistency allows memory requests to be issued and completed out of program order, when legal.

A software-initiated mechanism is used to maintain cache coherence [29]. Each processor's cache is invalidated and its write buffer is flushed to memory at the end of a parallel task (i.e. when it executes a *COEND* statement). The serial processor's cache is

[1] similar to FORALLs in HPF without requiring data partitioning

Benchmark	Subroutines traced	Global Read Access (M)	Global Write Access (M)	Read Message Generated(M)[2]	Write Message Generated(M)
ARC3D	STEP	2.675342	0.561258	0.438169	0.561258
FLO52	EFLUX	0.725123	0.263424	0.160802	0.263424
MDG	INTERF	4.550483	1.329371	1.098652	1.329371
MG3P	Whole Prog	3.355304	0.192632	0.161779	0.192632
TRFD	Whole Prog	7.628555	0.267484	0.353330	0.267484

Table 1: Benchmark Characteristics

flushed at the end of a serial task (i.e. when it executes a *COBEGIN* instruction). The invalidate/flush operation blocks the processor until it is completed, which includes performing writes in memory. This guarantees that 1) all the data is posted to memory before the next task starts, and 2) no stale data remains in the processor's cache. It has been shown that this simple software coherence scheme performs well on large scientific codes similar to the ones used in this study [5].

2.2 Benchmarks

The applications this study concentrates on are computationally-intensive numerical applications under the DOALL programming model. These applications usually exhibit large data set sizes and large amounts of data communication between computations. Most of them can be parallelized using DOALL parallel loops.

The benchmarks are selected among PERFECT [6] and NAS benchmarks [1] : ARC3D, FLO52, TRFD, MDG and MG3P. The first four codes solve 3-D CFD problems, 2-D CFD problems, two-electron integral transforms, and molecular dynamics, respectively. The last benchmark implements a 3-D multigrid algorithm.

The codes were profiled and subroutines which account for a large fraction of a benchmark's execution time were selected and traced. We started with a parallel version of the programs that ran on the CEDAR [13] multiprocessor. The benchmarks were further modified to increase the amount of parallelism and improve cache performance through the introduction of local variables, loop interchange and unrolling. All parallelism is expressed in terms of DOALL loops. Table 1 shows the characteristics of each benchmark. The number of read messages, which are generated for read misses, is shown for a system without prefetching.

The benchmarks were traced and trace-driven simulation was used to evaluate system performance. The trace size for some of the benchmarks may appear small. Consider the case of FLO52 where only one call to a subroutine EFLUX is traced. However, EFLUX accounted for over 30% of the serial execution time and is called 2514 times during the FLO52 execution and with the same four sets of parameters. The memory system is flushed by invalidating caches and waiting for request completion at the end of the last DOALL loop before EFLUX is called. Thus, each execution of EFLUX tends to exhibit the same cache, network and memory behavior. In such cases, we traced only one execution of a subroutine even though it generates what appears to be only a small trace.

2.3 Trace generation

Traces are obtained via instrumentation and execution of benchmark programs. Programs are instrumented by our MIPS tracing tool [4], compiled with the MIPS optimizing Fortran compiler and executed on a MIPS M/120 system. The instrumented code generates during its execution memory reference addresses and instruction execution time between such references, as well as information about parallel loop execution. The trace is then processed by a reordering tool which *schedules* loop iterations to emulate a static loop scheduling algorithm. The algorithm assigns iterations to processors as follows: the ith processor is assigned the ith, $(i + N)$th, $(i + 2 * N)$th .... loop iterations. All sequential execution is performed by processor 0.

2.4 System organization

The system consists of scalar processors, L1 caches, memories and an interconnection network as shown in Figure 1. These are described in more detail below starting with a base non-prefetching architecture followed by a system with prefetching.

2.4.1 Base Architecture

The base architecture uses a non-prefetching L1 data cache and a multistage network with a channel bandwidth of 1 byte per processor clock. A direct-

[2] for system size 128, 8K direct-mapped, write-through caches, line size 32 bytes.

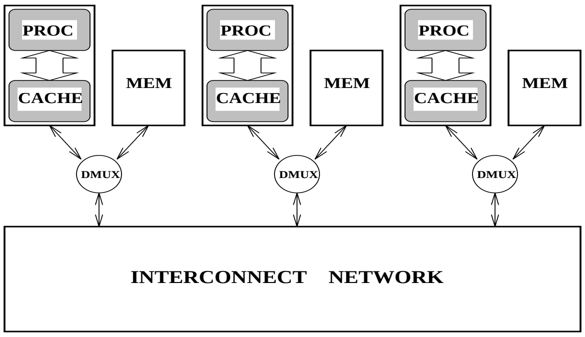

Figure 1: System Architecture

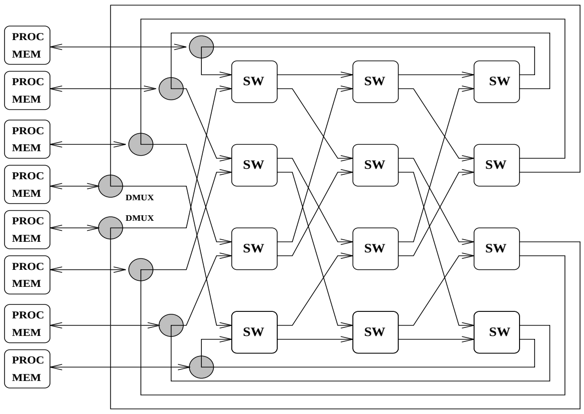

Figure 2: Multistage Shuffle-Exchange Network System

mapped, write-though, no-write-allocate cache is used. The size is 8 Kbytes, the line size is 4 words, and the word size is 8 bytes. A read hit takes 1 cycle and a read miss stalls a processor until the word is available. The L1 cache is thus typical of today's microprocessors. The L2 cache is not used because it was shown ineffective for this type of multiprocessor system. A 64-entry write buffer is used to exploit weak consistency and reduce the write-through traffic. A write takes 1 cycle if it can be stored in the write buffer, otherwise a processor stalls until there is buffer space. A write from the buffer is sent to memory when the network port is available.

A cache miss generates a read request message to the network, and the memory responds by sending a reply message with a cache line back to the cache. A write access generates a write request message to the network and is acknowledged by a write reply message from the memory. The acknowledgment arrival shows that the write has been performed in memory.

A message is disassembled into flits, sent out to the network, and assembled at the other end. Read data is returned *requested word first*. Thus, the processor resumes its operation as soon as the first data word is assembled without waiting for an entire cache line to be assembled.

2.4.2 Memory Organization

The shared memory is physically distributed and directly addressable by each processor. The memory system is composed of interleaved and pipelined high-speed memory modules in each node. The memory modules are interleaved by a cache line size for storing shared data. All non-shared data and instructions are stored in a portion of node's memory called *local* memory. Thus, only memory references to shared data are accessed through the network.

A memory module operation is pipelined so that it can perform a memory access while accepting and assembling the flits of the next memory request message. The module produces the first flit of a reply message header to the network 4 processor cycles after it receives the request message. The message header generation proceeds in parallel with DRAM access. The module cycle time depends on the network data path width and the message length and is proportional to the length of a request message or a reply message, whichever is longer. A read request message (8 bytes) consists of only a message header. A write request message (16 bytes) has a message header and one data word. A write reply message (8 bytes) from a memory has only a message header. A read and prefetch reply message (40 bytes) has a message header and a cache line with 4 data words and starts with the word requested by a processor.

2.4.3 Scalar processor

A RISC processor based on MIPS R3000/3010 instruction execution is modeled to derive timing information for use in simulation. The major difference is that the word size, the address bus, and the data bus are assumed to be 64 instead of 32 bits. Instructions execute in one cycle with the exception of integer multiplication and division, floating-point addition, multiplication and division, and a few other instructions.

2.4.4 Interconnection Network

A multistage shuffle-exchange network (Figure 2) composed of 4x4 crossbar switches is used in the system. It is similar to networks in such systems as IBM SP-2 [27] and CEDAR [13]. Each node is connected through a multiplexor and demultiplexor (DMUX) to the first and last stages of the network. A tag-based distributed routing algorithm [22] with virtual cut-through flow control [17] is used.

The switch organization is shown in Figure 3. It uses 4 input queues to eliminate the head-of-

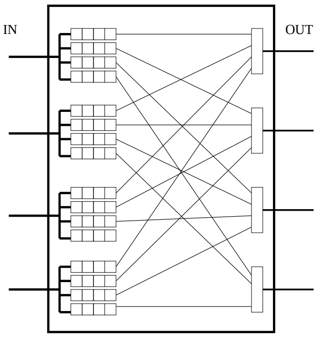

Figure 3: Non-Blocking Switch

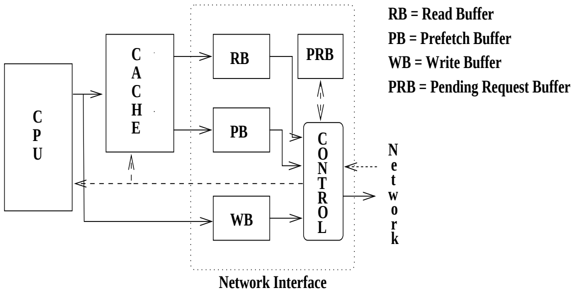

Figure 4: Network Interface

line (HOL) blocking [16, 20]. The first flit of a message contains its destination tag which selects an output port. Infinite queues are used since it has been shown that performance with a reasonable queue size is very close to that with infinite queues [2, 10]. Fixed priority, fair arbitration mechanism is used to resolve output port contention.

Base channel data path width is 1 byte, the switch is fully pipelined, and the channel bandwidth is 1 byte/clock. Switch latency without contention is 3 cycles, wire delay between switches is 1 cycle, and a DMUX latency is 1 cycle. The DMUX also has infinite queues. A switch implementation with such parameters using full-custom technology has been described in [9]. We assume that the channel data path width and the bandwidth can be increased by up to a factor of 8 without any effect on other timing parameters.

2.5 Tagged Cache and Network Interface

A tagged cache is very effective for short stride prefetching. Each cache line has a flag, the *prefetch bit*, to mark whether a read has accessed the line or not. The prefetch bit is initially set when the line is prefetched into a cache. On a read miss, the missed line and the next line are fetched. On a read hit, if the prefetch bit of the line is set, indicating that the line was prefetched and has not yet been accessed by a read, the next line is prefetched, and the prefetch bit is reset to 0. A cache line is not reserved for a pending prefetch, but allocated only when the prefetched line is returned from memory.

The network interface of a tagged cache consists of a control logic and four FIFO buffers: a read buffer, a write buffer, a prefetch buffer, and a pending request buffer as shown in Figure 4. Read miss, write, and prefetch requests are stored in a corresponding buffer. A full write buffer stalls the processor on either a read or a write until the write buffer has an empty entry. A full prefetch buffer causes additional prefetch requests to be discarded. 64-entry write and prefetch buffers are used to avoid this. The read buffer has only one entry because the processor stalls on a read miss. The pending request buffer is used to maintain memory access ordering among different memory requests by recording requests sent to the network but not completed. Non-prefetching systems use the same network interface except for the prefetch buffer and operations for prefetch requests.

When a node's network port becomes available, the control logic selects the next request to send using the following priority arbitration: reads have the highest priority, followed by prefetches, and writes have the lowest priority. The control logic also checks and enforces data dependencies between reads, writes, and prefetches, both buffered in the node and outstanding.

The pending request buffer size is initially set to 3 entries, one for each outstanding read, write, and prefetch request. Later it is changed to have more entries as described in the corresponding text.

3 Performance Metrics

The following metrics are used to compare the prefetching and non-prefetching systems.

- Hit ratio: Only lines present in the cache result in hits.
- Average read miss latency: This includes network delays and main memory access delay a read miss experiences.
- Average processor read latency: This is a read latency seen by a processor using a cache.
- Average memory stall time: The time spent waiting for memory accesses (see below).
- Completion time: The time when the last processor completes execution.

The average memory stall time consists of the following non-overlapping components:

- Read stall time: Stall time for read misses which do not hit on pending prefetch lines.
- Prefetch stall time: Stall time waiting for a pending prefetch due to hits on pending prefetch lines.
- Write stall time: Stall time due to a full write buffer.

Note that the average processor read latency includes only the time taken for cache hits, read stall time, and prefetch stall time, whereas the average memory stall time contains not only read stall time and prefetch stall time but also write stall time.

4 Experimental Results and Analysis

To recapitulate, the baseline 128-processor system uses weak consistency, write buffering, 8-Kbyte caches, and a multistage shuffle-exchange network constructed from 4x4 crossbars with a channel bandwidth of 1 byte/clock. A total of three outstanding requests per node is allowed: one for reads, one for writes, and one for prefetches. We start by evaluating the effect of tagged prefetching on system performance and how it is affected by increasing the network channel bandwidth by a factor of 2x, 4x, and 8x. Read miss latency, processor read latency, and completion time are analyzed next, followed by the effect of write traffic and the utilization of network bandwidth.

4.1 Prefetching and Network Performance

We start by using the baseline architecture with and without tagged prefetching while varying the network bandwidth. The effect of prefetching on cache hit rates is measured first. Recall that only hits on lines present in the cache are counted, and hits on pending prefetches are counted as a miss. The hit rate depends on the available network bandwidth since additional bandwidth may allow prefetches to be initiated and completed earlier, which results in additional hits. The hit rate should not depend on bandwidth when prefetching is not used.

4.1.1 Cache Hit Ratios

Table 2 shows the cache hit ratios for each benchmark and 2x, 4x, and 8x increase in network channel bandwidth over the baseline architecture.

Prefetching improves the cache hit ratio of FLO52 and MG3P by up to 25 % and 10 %, respectively. ARC3D has only a small improvement because of a large number of long stride accesses for which tagged prefetching is not effective. MDG and TRFD show very small improvements. TRFD has a high cache hit ratio without prefetching, making further improvement from prefetching difficult. MDG has a relatively large number of stride 0 accesses and frequent synchronization that flushes the cache and causes unnecessary prefetches. In general, the cache hit ratio increases only slightly with additional network bandwidth, the biggest impact comes from using prefetching in the first place.

BW	1x	2x	4x	8x
ARC3D				
NPF	83.6	83.6	83.6	83.6
PF	86.3	86.4	86.4	86.4
FLO52				
NPF	77.8	77.8	77.8	77.8
PF	95.5	96.6	97.5	97.5
MDG				
NPF	75.3	75.3	75.3	75.3
PF	75.8	75.7	76.5	76.7
MG3P				
NPF	89.5	89.5	89.5	89.5
PF	98.2	98.2	98.3	98.3
TRFD				
NPF	97.9	97.9	97.9	97.9
PF	98.7	98.7	98.7	98.7

Table 2: Cache Hit Ratios. Legend: NPF = No Prefetching, PF = Prefetching, $BW = Nx$ (N times increase in Channel Bandwidth).

4.1.2 Read Miss Latency

The average read miss latency as a function of bandwidth is shown in Fig. 5. Write latency is not shown since it does not directly affect performance when write buffering is used. Prefetching increases the read miss latency in all the cases. The read miss latencies with and without prefetching differ by as much as 71 % for base channel bandwidth (1 byte/clock). But the difference is down to 28 % when the bandwidth is quadrupled, actually less than 10 % for four out of the five benchmarks.

Additional channel bandwidth helps to reduce the read miss latency for both prefetching systems and non-prefetching systems. The architecture without prefetching directly benefits from an increase in bandwidth which significantly reduces the latency. The largest drop in latency is observed when the base channel bandwidth is doubled.

4.1.3 Processor Read Latency

Prefetching increases the read miss latency but can reduce the average read latency seen by a processor. Successful prefetching increases the cache hit ratio and this compensates for the increased memory latency. The decrease in the average processor read latency

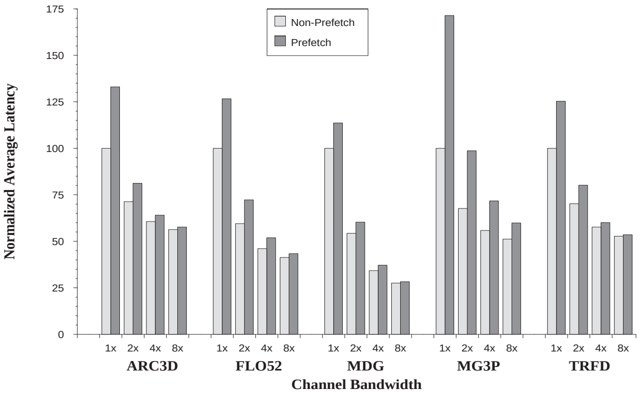

Figure 5: Average Read Miss Latency: normalized to that of non-prefetching systems with 1x channel bandwidth for each benchmark

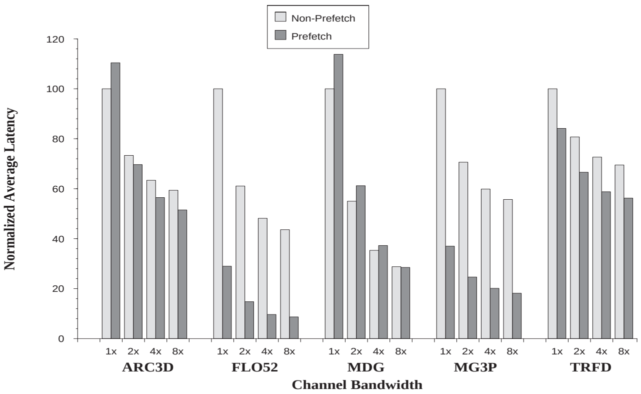

Figure 6: Average Processor Read Latency: normalized to that of non-prefetching systems with 1x channel bandwidth for each benchmark

(Fig. 6) due to prefetching and available bandwidth depends on a benchmark. With sufficient bandwidth all the benchmarks show an improvement in the average latency. Three out of five show an improvement even with the minimal bandwidth. Benchmarks with predominately stride-1 accesses, FLO52 and MG3P, show significant improvement for all channel widths, in spite of the increase in the read miss latency, because tagged prefetching makes a big difference in hit rates. ARC3D, has longer strides not picked by tagged prefetching and thus improvement is small. One can argue that with stride-directed prefetching it will show a significant improvement. TRFD also shows a small improvement but this is because its hit ratio is already high and only a small increase from prefetching is possible. In MDG prefetching increases the average processor read latency due to the presence of hot-spots. Overall, increased channel bandwidth and prefetching both improve the average processor read latency over the base non-prefetching system. Prefetching can reduce the average latency even with the smallest bandwidth more than with 8x increase in bandwidth when it is working well as in FLO52 and MG3P, but can also increase latency as in MDG. The performance improvement from prefetching is limited more by the nature of the tagged prefetching mechanism and the memory access behavior of benchmarks than by the network and memory bandwidth, as observed in the case of ARC3D and MDG. The results are additive but the difference becomes small under high bandwidth. Improvement from prefetching is very important since higher bandwidth is difficult to achieve in large systems.

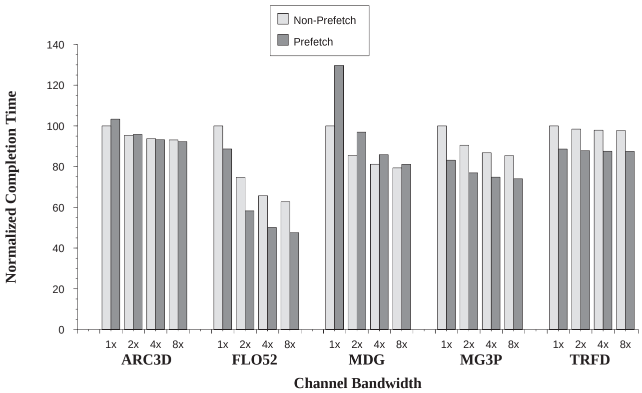

Figure 7: Completion Time: normalized to that of non-prefetching systems with 1x channel bandwidth for each benchmark

4.1.4 Completion Time

The program completion time (Fig. 7) for most benchmarks shows an improvement from prefetching and/or additional channel bandwidth. The general trend is similar to the average latency but is affected by the fraction of computation time in the completion time.

The overall completion time improvement is smaller than the decrease in the average latency would suggest. For instance, the contribution of memory stall time in FLO52 is high, close to 39 % of the average execution time. For a system with base channel bandwidth it shows a 71 % improvement in the average processor read latency when prefetching is used. However, the decrease in completion time is only 11 %. We find that write stalls significantly contribute to the completion time and examine this effect in more detail in the next section.

4.1.5 Write Stall Time

In FLO52 writes effect execution time due to *write buffer full* stalls. The effect of such stalls on average memory stall time and the contribution of other

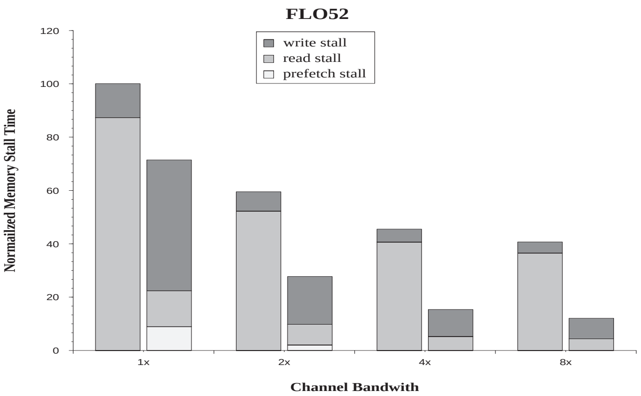

Figure 8: Normalized Memory Stall Time for 1x, 2x, 4x, 8x Channel Bandwidth

memory stalls is shown in Figure 8. The memory stall time is broken down into three non-overlapping components: read stalls, prefetch stalls, and write stalls. Stall cycles due to waiting on pending prefetches are shown as *prefetch stall*. The results are normalized to the average memory stall time for base channel bandwidth of 1 byte/clock. The other benchmarks are less affected by write stalls and we do not present their results.

In the base system without prefetching the read stalls are a major portion of the average memory stall time and the write stalls are relatively small. There is sufficient time and bandwidth to retire the writes from the write buffer since a processor stalls on read misses most of the time. When prefetching is used and improves cache hit rate, the trend changes. Prefetching greatly reduces the read stall time but a significant increase in the write stall time appears which now accounts for over 50 % of the average memory stall time. Read and write stall times are reduced with additional bandwidth but the write stall time dominates in all cases. The prefetch stall time is noticeable in systems with insufficient bandwidth but disappears as more bandwidth is provided.

The reason for increased write stall time with prefetching is that prefetch requests have higher priority in accessing the network. Prefetch requests are generated frequently and can block writes from accessing the network. The write buffer quickly fills up and stalls the processor. The actual network/memory bandwidth available to writes and prefetches is insufficient, in part due to a single outstanding request limit in the baseline architecture.

The results indicate that write and prefetch traffic should be managed so as to balance their sharing of the limited network and memory bandwidth in order to maximize the benefits of prefetching and weak consistency/write buffering. The next section discusses how to balance writes and prefetches in network and memory access to achieve the best possible improvement for a given network and memory bandwidth.

As we have indicated above, different types of memory requests, prefetches and writes, compete with each other for network and memory bandwidth. Reads are not a problem in this architecture but would also compete for the bandwidth if a non-blocking cache was used. Network and memory latency are an increasing function of the request rate because of contention [19], and the rate grows when prefetching and weak consistency are used. It has been shown [8, 28] that by keeping the requests out of the network locally at each node, the request rate and therefore the latency is reduced (although requests wait longer locally in an node). This allows the bandwidth to be reasonably equitably shared between processors while relying only on local information for the allocation algorithm. We propose to use a similar approach here. The conjecture is that by allowing more outstanding requests onto the network, but setting a limit of N outstanding request for prefetches and writes each, we can better utilize a given network bandwidth. We will refer to this method as *separate limit control*. The question we need to answer is how well it works in this type of system and how to allocate bandwidth fairly for reads, writes, and prefetches.

4.2 Managing Network/Memory Bandwidth Allocation

The results from previous studies mentioned above do not directly apply to this architecture and we start by investigating the effect of limiting a maximum number of outstanding prefetches and a maximum number of outstanding writes separately and searching for an optimal solution. We compare this type of allocation to another control scheme which sets a single limit on the total number of outstanding requests, a *single combined limit control*. Recall that the base architecture allowed only three outstanding requests per node. This may not be enough to cover the latency by prefetching or to retire writes from the write buffer fast enough to avoid stalls. But too many outstanding writes and prefetches may overload the network and memory, delay read requests, and eventually degrade the memory system performance. These two opposing trends imply there may be an optimal limit, separate or combined.

4.2.1 Optimal Limits on the Outstanding Requests

The base architecture with the minimal network channel bandwidth of 1 byte/clock is used throughout

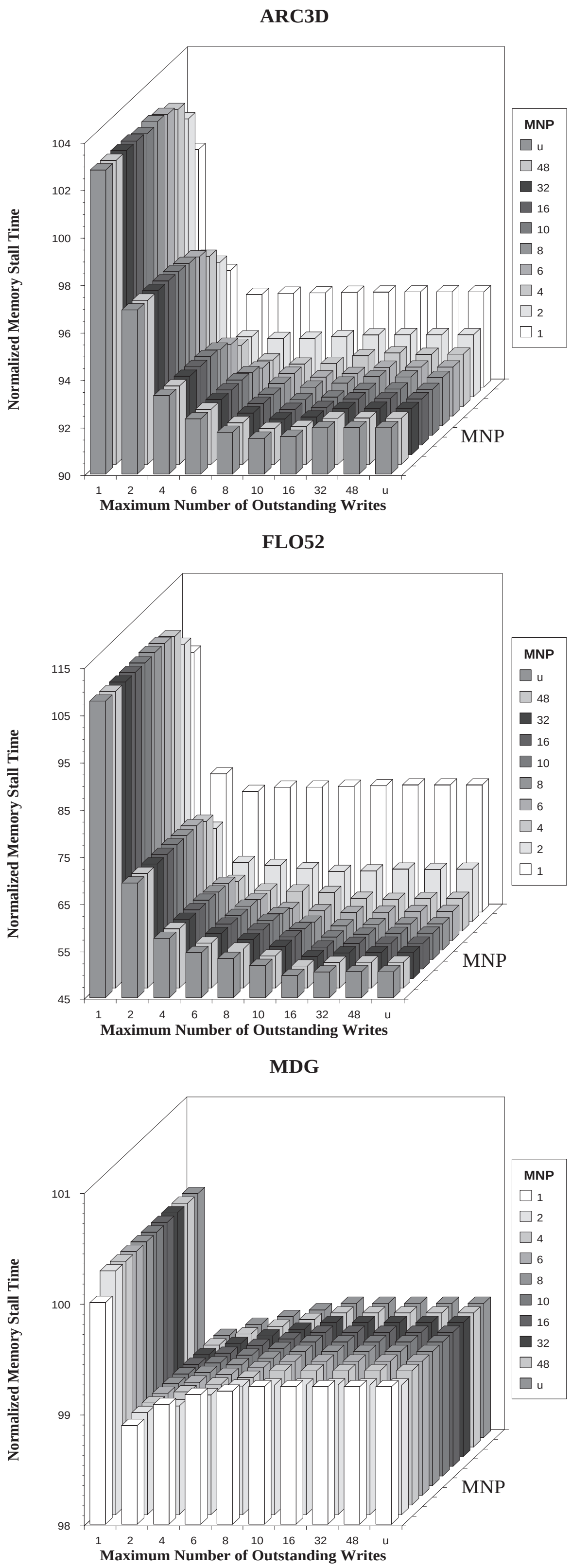

Figure 9: Average Memory Stall Time as a function of the Maximum Number of Outstanding Prefetches (MNP) and Writes.

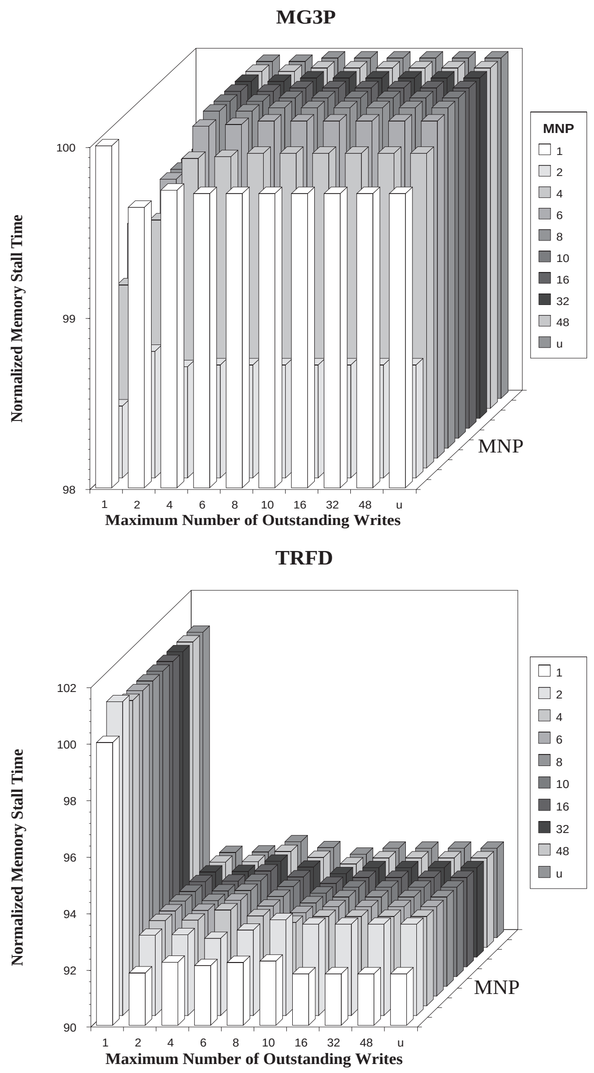

Figure 10: Average Memory Stall Time as a function of the Maximum Number of Outstanding Prefetches (MNP) and Writes.

this section since it needs bandwidth management the most. Figures 9 and 10 show the normalized memory stall time for each benchmark as a function of the maximum number of outstanding writes and the maximum number of outstanding prefetches. The memory stall time is normalized to that of the system allowing only one outstanding request for each prefetch and write. The limits are varied separately and range from 1 to 48 requests for each type plus an unlimited number. The **u** label in the legend's X and Y axis refers to the unlimited case. In this latter case, the total number of outstanding requests is limited by the pending request buffer size of 128 entries.

As seen in Figures 9 and 10, there is an optimal value for each limit resulting in the best system per-

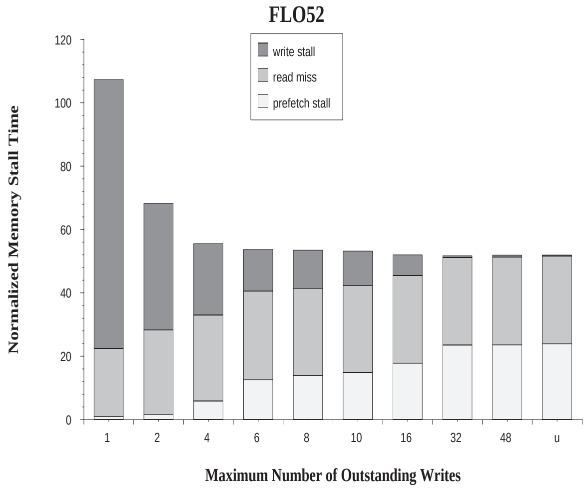

Figure 11: Stall Time Components: 4 Max. Outstanding Prefetches.

formance but the values depend on the benchmark. In some cases the optimal value does not change the performance much but in others it has a significant effect. The number of outstanding writes affects the average memory stall time more than the number of outstanding prefetches. In other words, writes need more memory bandwidth than prefetches. With the exception of FLO52, only a small number of outstanding writes and prefetches, such as 4, is sufficient to obtain the performance close to the optimal. Increasing the limit from 1 to 2 yields the biggest improvement for all benchmarks. For some benchmarks, increasing the limit beyond 2 for each request type degrades the performance slightly.

Figures 11 shows the memory stall time components for FLO52 as more outstanding writes are allowed. The maximum number of outstanding prefetches is fixed at 4 here. The write stall time is very large with only one outstanding write, dominating the total. The write stall time is reduced quickly as more outstanding writes are allowed. However, as the maximum number of outstanding writes increases beyond 4, the prefetch stall increases noticeably. At 32 outstanding writes the write stall time is minimized, but the prefetch stall time now accounts for a large portion of the total memory access time. It appears difficult to reduce the write stall time and the prefetch stall time simultaneously under limited network bandwidth. This implies that prefetch requests need to be generated earlier when more outstanding writes are allowed.

4.2.2 A Single Combined Limit

The separate limits investigated above did not provide an optimal value for all benchmarks, and too large a limit was shown to degrade performance. Another approach to allocating bandwidth is to have a single combined limit per node. Allocating bandwidth regardless of request types has an advantage, when one type of requests dominates, by allowing it to utilize more bandwidth. The approach may have a problem when several request types demand the bandwidth simultaneously but have different priorities. In such a case a higher priority request type can monopolize the network access.

MOP	maximum N of outstanding writes					
	1	2	4	8	16	32
ARC3D						
1	102	99.5	100	102	102	102
2	106	102	99.5	101	101	101
4	109	103	100	100	100	100
8	111	105	102	100	100	100
16	112	105	102	99.9	99.7	100
32	111	105	101	99.8	99.6	100
FLO52						
1	115	110	118	128	138	140
2	149	104	100	105	110	111
4	174	116	99.9	100	104	103
8	187	125	106	102	101	99.8
16	200	133	113	106	99.3	100
32	203	134	112	105	98.6	100
MDG						
1	101	100	100	100	100	100
2	102	100	99.9	100	100	100
4	101	99.8	99.9	99.9	100	100
8	101	99.8	99.8	99.9	100	100
16	101	99.7	99.8	99.9	100	100
32	101	99.7	99.8	99.9	100	100
MG3P						
1	101	100	100	99.9	99.9	99.9
2	99.3	99.3	99	99	99	99
4	99.4	99.6	99.9	99.9	99.9	99.9
8	99.4	99.7	100	100	100	100
16	99.4	99.7	100	100	100	100
32	99.4	99.7	100	100	100	100
TRFD						
1	104	99.5	99.6	99.5	99.2	99.2
2	105	100	99.8	100	100	100
4	105	99.9	100	100	100	100
8	105	100	100	100	100	100
16	105	99.9	99.9	100	100	100
32	105	99.9	99.9	100	100	100

Table 3: Average Memory Stall Time Ratios: Separate Limit over Single Combined Limit, MOP = the Maximum Number of Outstanding Prefetches.

The comparison of single combined vs. separate limits is shown in Table 3 as a ratio of memory stall time under the separate limit over that under the

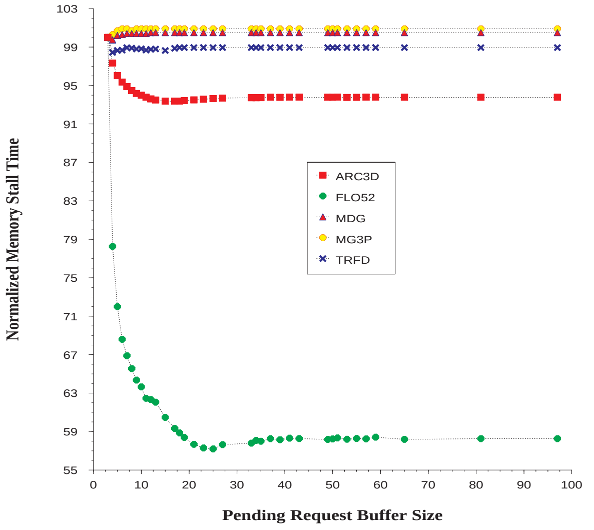

Figure 12: Average Memory Stall Time vs. Pending Request Limit (Normalized to the Case of 3 Total Outstanding Requests).

single combined limit. Each entry represents a different maximum number of outstanding writes and prefetches and is compared with the single limit maximum equal to the sum of the maximum number of outstanding writes and prefetches. The single combined limit shows better or almost equal performance for all parameters, although the difference is large only when the total number of outstanding requests is very small. This is because for a small number of outstanding requests sharing the limit utilizes the bandwidth more effectively and becomes more crucial. Once a large limit on each type of request is reached both types of requests can get enough bandwidth as well and there is no significant performance difference between the control schemes.

4.2.3 Pending Request Buffer Size

The outstanding requests issued to the network need to be tracked to maintain memory access ordering. The baseline architecture uses the pending request buffer to keep the address and request type of each outstanding requests. The size of this buffer serves as limit on the total number of simultaneously outstanding reads, writes, and prefetches. This buffer is unlikely to be very large and should be carefully selected to optimize performance. We now examine the effect of the pending request buffer size as a limit on the total number of requests. Any number of outstanding write and prefetch requests is allowed as long as there is room in the pending request buffer. Figure 12 shows how the average memory stall time vs. the buffer size for each benchmark, using the single combined limit control. The memory stall time is normalized to the case of 3 total outstanding requests allowed for reads, writes and prefetches. Two programs do not need more than the initial 3 request limit and have no change to speak of. The other 3 programs show a clear minimum, although only ARC3D and FLO52 need a higher limit to achieve it, 15 and 25 requests, respectively. The memory access time of ARC3D and FLO52 decreases noticeably until the minimal value and increases slightly as the buffer size further increased.

5 Conclusions

The interaction of two latency hiding techniques, prefetching and write buffering under weak consistency, with the limited bandwidth in a large multiprocessor was investigated.

The results showed that even simple prefetching can improve the overall memory system performance, although it increases the average network latency. Increasing network (channel) bandwidth helps both prefetching and non-prefetching systems, but slightly more for prefetching systems. Prefetching can deliver a much larger improvement than increasing network bandwidth when prefetching is working well. The performance improvement from prefetching is limited more by the nature of the tagged prefetching mechanism and the memory access behavior of benchmarks than by the network and memory bandwidth, as observed in ARC3D and MDG. Other prefetching mechanisms can do even better.

The results also indicate that write and prefetch traffic should be managed so as to balance their sharing of the limited network and memory bandwidth. Controlling write traffic seems especially important in large-scale shared memory systems using software-controlled cache coherence because the traffic can become significant. We proposed and investigated balancing bandwidth utilization by limiting the number of outstanding requests. The mechanism works well and is very important when a request rate is high, e.g. for intensive prefetching or write traffic. A single combined limit performed better than separate limits for each request type. An optimal value for the limit exists but is program dependent. This suggests that the mechanism needs to be applied adaptively for each different application.

References

[1] D. Bailey, H. Simon, J. Barton, and T.Lasinski. The NAS parallel benchmarks. Technical Report RNR-91-02, NASA Ames Research Center, 1991.

[2] P-Y. Chen, D. H. Lawrie, P-C. Yew, and D. A. Padua. Interconnection networks using shuffles. *IEEE Com-*

puters, pages 55–64, 1981.

[3] Tien-Fu Chen. *Data Prefetching for High-Performance Processors*. PhD thesis, University of Washington, 1993.

[4] Yung-Chin Chen. *Cache Design and Performance in a Large-Scale Shared-Memory Multiprocessor System*. PhD thesis, University of Illinois at Urbana-Champaign, 1993.

[5] Yung-Chin Chen and Alexander Veidenbaum. Comparison and analysis of software and directory coherence schemes. In *Supercomputing91*, pages 818–829, 1991.

[6] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer performance evaluation and the perfect benchmarks. In *International Conference on Supercomputing*, 1990.

[7] Fredrik Dahlgren and Per Stenstrom. Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors. In *International Symposium on High Performance Computer Architecture*, January 1995.

[8] W. J. Dally and H. Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *IEEE Transactions on Parallel and Distributed Systems*, 4:466–475, April 1993.

[9] Andre DeHone, Frederic Chong, Matthew Becker, Eran Egozy, Henry Minsky, Samuel Peretz, and Jr. Thomas F. Knight. Metro: A router architecture for high-performance, short-haul routing networks. In *International Symposium on Computer Architecture*, pages 266–277, April 1994.

[10] Daniel M. Dias and J. Robert Jump. Analysis and simulation of buffered delta networks. *IEEE Transactions on Computers*, C-30(4):331–340, April 1981.

[11] M. Dubois, Scheurich C, and F. Briggs. Memory access buffering in multiprocessors. In *International Symposium on Computer Architecture*, pages 434–442, 1986.

[12] John W.C. Fu and Janak H. Patel. Data prefetching in multiprocessor vector cache memories. In *International Symposium on Computer Architecture*, pages 54–63, May 1991.

[13] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh. Cedar - a large scale multiprocessor. In *International Conference on Parallel Processing*, pages 524–529, 1983.

[14] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *International Conference on Supercomputing*, pages 354–368, June 1990.

[15] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *International Symposium on Computer Architecture*, pages 364–373, May 1990.

[16] Mark J. Karol, Michael G. Hluchyj, and Samuel P. Morgan. Input versus output queueing on a space-division packet switch. *IEEE Transactions on Communications*, COM-35(12):1347–1356, December 1987.

[17] Parviz Kermani and Leonard Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3:267–286, 1979.

[18] R. Kent Koeninger, Mark Furtney, and Martin Walker. A shared memory MPP from Cray research. *Digital Technical Journal*, 6(2):8–21, Spring 1994.

[19] Clyde P. Kruskal and Marc Snir. The performance of multistage interconnection networks for multiprocessors. *IEEE Transactions on Computers*, C-32(12):1091–1098, December 1983.

[20] Manoj Kumar and J. R. Jump. Performance enhancement in buffered delta networks using crossbar switches and multiple links. *Journal of Parallel and Distributed Computing*, (1):81–103, 1984.

[21] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, pages 87–106, June 1991.

[22] Krishnan Padmanabhan. Design and analysis of even-sized binary shuffle-exchange networks for multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):385–397, October 1991.

[23] Subbarao Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *International Symposium on Computer Architecture*, pages 24–33, May 1994.

[24] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The ibm research parallel processor prototype(rp3): Introduction and architecture. In *International Conference on Parallel Processing*, pages 764–771, 1985.

[25] G. E. Schmidt. The butterfly parallel processor. In *International Conference on Supercomputing*, pages 362–365, 1987.

[26] Alan Jay Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[27] Craig B. Stunkel, Dennis G. Shea, Don G. Grice, Peter H. Hochschild, and Michael Tsao. The SP1 high-performance switch. In *Proceedings of the Scalable High Performance Computing Conference*, pages 150–157, May 1994.

[28] Stephen W. Turner and Alexander V. Veidenbaum. Scalability of the Cedar system. In *Supercomputing*, November 1994.

[29] Alexander Veidenbaum. A compiler-assisted cache coherence solution for multiprocessors. In *International Conference on Parallel Processing*, pages 1029–1036, 1986.