On the Use and Performance of Communication Primitives in Software Controlled Cache-coherent Cluster Architectures

by

Xiaohan Qin

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

1997

Approved by [illegible]
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree Computer Science & Engineering

Date 12/12/97

University of Washington

Abstract

On the Use and Performance of Communication Primitives in Software Controlled Cache-coherent Cluster Architectures

by Xiaohan Qin

Chairperson of Supervisory Committee: Professor Jean-Loup Baer
Computer Science and Engineering

Two recent trends are affecting the design of medium-scale shared-memory multiprocessors. The first is the use of nodes which themselves consist of clusters of processors. Clusters, already available as commodity parts, not only make powerful nodes, they also let the system scale up gracefully. The second trend is the use of programmable protocol processors and software for maintaining cache coherence to shorten the hardware design cycle and to provide flexibility and extensibility.

One problem arising from software cache coherence is that remote memory accesses suffer a longer latency than with a pure hardware scheme. Another issue raised by software schemes in cluster environments is that of contention on the protocol processor due to the high service demand for this device.

Our solution to the first problem offers users or compiler writers a set of explicit communication primitives to provide hints for moving data properly and promptly. The communication primitives, running on protocol processors, introduce a flavor of message-passing and permit protocol optimization. To the second issue, we investigate three architectural choices that strive to achieve resource balance: (1) selecting an appropriate cluster size to control resource sharing, (2) adding a remote cache (per node) to keep remote data in clusters, and (3) adding a forwarding logic to reduce the load on the protocol processor and to speed up the processing of simple messages.

This dissertation studies how the overhead of a software scheme and its contention on the protocol processor can be reduced by various combinations of the design options and how the software overhead can be further hidden by the communication primitives. In the absence of communication primitives, we employ an MVA-based analytical model to estimate the protocol processor's contention and overall performance for a fast turn-round. When communication primitives are present, we employ simulation method. We find that the software implementation supplemented with remote cache and forwarding logic can deliver a performance competitive with the rigid and pure hardware scheme. With the judicious use of communication primitives, the enhanced software scheme can improve performance beyond the limit of the hardware implementation. In addition, the software cache coherence is more flexible, scalable and easier to optimize.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Many people have contributed to the completion of this dissertation and my Ph.D education. Their kindness and open-mindedness have made the five years of graduate student life at the University of Washington an unforgettable experience.

First and foremost, I must thank my advisor, Jean-Loup Baer, for his guidance, encouragement, support, and patience. As a Ph.D student's advisor, Jean-Loup first listens carefully to what his student has to say. It's only after that he will give his own suggestions. To do that, he put up with my insufficient English. His enthusiasm and engagement in the research made it possible for me to carry out the study. When it came to writing, his instrumental assistance allowed me to publish. It was a special privilege and a great pleasure working with him for four years. Thank you Jean-Loup!

I am grateful to Susan Eggers, Hank Levy, and John Zahorjan for serving on my committee. Special thanks to John for taking time reading through the manuscript and providing technical and grammatical comments. Also thanks to Sandy Kaplan for editing this dissertation, which significantly improved its quality. I take responsibility for any remaining errors.

I would also like to thank fellow graduate students: Melanie Fulgham, Jack Lo, Terrance Goan, Kurt Partridge, Brendan Mummy, and Derrick Weathersby, for sharing frustrations, helping with presentations, and teaching me English.

My appreciation also goes to the technical staff of the Computer Science Laboratory and the administrative staff of the Computer Science and Engineering Department, most notably, Frankye Jones. Their assistance in the backstage has eased my use of the computer systems and my dealings with the University bureaucracy.

Finally, I want to thank my parents for their unconditional love and for inspiring me to come studying in America. Although at times I frowned upon their over-detailed plan for my career and future, I have to admit the outcome of the plan has always been rewarding.

The work presented in this dissertation was supported by grants from the National Science Foundation.

Xiaohan Qin
December, 1997

Chapter 1

INTRODUCTION

The design of medium-scale shared-memory multiprocessors has been influenced by two recent trends. The first is using clusters of processors as basic building blocks. The second trend is using programmable protocol processors and software for maintaining cache coherence. This thesis concerns with performance issues arising from employing software cache coherence in cluster environments. Our goal is to evaluate the overhead of a software scheme and investigate ways to reduce the software overhead through either simple hardware assists or software means.

1.1 Motivations

During the last decade, advances in semiconductor technology have contributed steadfastly to the shrinking of transistor size and the speeding of clock rates. The performance of microprocessors has benefited greatly from the resulting increase in real estate and clock speed. At the same time, the amount of data generated by personal computers, workstations, and other high-speed electronic devices has increased enormously. The flourishing internet is connecting more and more computers. Inexpensive and immediate accesses to the global network have motivated both consumers and producers to use more online processing and services, e.g., electronic newspaper/publishing and electronic commerce. To provide competitive products or high quality services, raw computing power is not sufficient. Commercial and government institutions must find ways to deal with the growing volume and use of data.

The demand for powerful and scalable enterprise-wide computing resources is at a premium.

In the enterprise server market, bus-based shared-memory multiprocessors, or multis [11], have already enjoyed a tremendous success. Many computer vendors ship their servers in this form, e.g., IBM's PowerPC-based RS/6000 J40 and J50 multiprocessor servers and HP's PA-RSIC-based 9000 K-class servers connecting up to six processors. Despite its success, the single-bus system has a major drawback: It can only accommodate a small number of processors because the bus soon becomes an overcommitted resource. This lack of extensibility due to bus contention is exacerbated by the fact that processor speed increases faster than the corresponding growth in bus bandwidth and reduction in memory latency.

One way to expand the number of processors without sacrificing the shared-memory paradigm is to consider each multi as a cluster node and to link nodes using an interconnection network such as a mesh [41] or a SCI ring [43]. The cluster architecture is more scalable than the single-bus multiprocessor for two reasons. First, the number of links or the aggregated bandwidth of the interconnection network can grow as cluster nodes are added to the system. Second, shared resources in each node, e.g., the bus, memory, and the I/O node, serve only a small number of processors; they are distributed in the system.

There exists a wide spectrum of implementations based on the shared-memory programming model, from tightly coupled hardware-based cache-coherent schemes (DASH [41], Alewife [2], Origin [39], and Exemplar [14]) to software-based shared-memory systems. Some of the latter rely on page management mechanisms in the operating system to fault on illegal accesses (IVY [42] and Treadmarks [4]). Others exploit compiler analysis techniques to detect stale data in cache [65, 17, 32]; another group modifies the program's executable to perform cache consistency checking before accessing each shared variable (Blizzard-S [57], Shasta [56]).

This dissertation studies a class of cluster architectures that employs communication processors and software to implement fine-grain cache coherence protocols, similar to those of DASH. In the DASH prototype, cache coherence is maintained at two levels. Within a cluster node, the caches of individual processors are kept coherent via a snoopy bus protocol. When a memory reference cannot be resolved by the intra-cluster snoopy protocol, it is intercepted by the communication/memory management unit (CMMU), which connects the cluster to the network and maintains inter-cluster cache coherence using a directory-based protocol.

Many systems with an interconnect [41, 2, 39, 14] implement the communication/memory controller in hardware as ASIC logic. The hardware-based CMMU usually includes a limited set of functions, e.g., a high-performance messaging mechanism (optimized for small messages), a fixed protocol, and, perhaps, some prefetching capability [41]. Most optimizations geared at reducing the communication cost, such as protocol improvements and weak consistency memory models, require non-trivial hardware modifications. Furthermore, the rigidity of the hardware scheme prevents the protocol policies and optimizations from being adapted to the needs of a given application.

Recent developments in shared-memory multiprocessor systems have led many computer architects to advocate using programmable protocol processors rather than ASIC to provide basic communication mechanisms and using software to enforce cache coherence policies. Efforts in that direction include Flash [38, 28] at Stanford, s3.mp [49, 48] at Sun Microsystems, Typhoon [55] at Wisconsin, and more recently NUMA-Q [43] at Sequent. The biggest advantage of software cache coherence is its flexibility and extensibility. Multiple protocols can be easily incorporated in one system. Additional functions may be obtained by programming the protocol processor. Furthermore, while a hardware full-map directory scheme limits the number of nodes in a system, a software scheme avoid such limitations, because the data structures for maintaining cache coherence are software-based.

This flexibility, however, comes at a cost in performance, since one can always build a hardwired controller that outperforms the software scheme for any given protocol. Software overhead can diminish performance in two ways. First, it directly contributes to the longer latency of some memory transactions (e.g., remote memory accesses) that require services from the protocol processor. Second, since the protocol processor is shared by several compute processors, the high service demand for the unit makes it a potential bottleneck of the cluster node.

This dissertation investigates ways to make use of the flexibility of protocol processors to provide users with some advantageous features of asynchronous message-passing, or explicit communication primitives, that hide memory latency with computation. Our study is performed in three phases. First, we study via trace-driven simulations the overhead of software cache coherence schemes and the performance gain of the programmer/compiler-controlled communication primitives in an architecture whose cluster node contains a single compute processor. This simplifying architectural assumption lets us isolate the effect of software overhead from that of shared resource contention. Second, we develop an analytical model to assess contention on shared resources such as the communication processor, the bus, memory, etc. The analytical approach allows us to search a large design space and identify quickly the architectural configurations worth further investigation. Finally, we evaluate in cluster environments the software implementation of the base cache coherence protocol as well as the software implementation enhanced by the communication primitives.

1.2 Contributions

This dissertation makes the following contributions to the study of cluster architectures that employ communication processors and software to maintain cache coherence.

- We proposed a set of explicit communication primitives to be executed on the protocol processors. These primitives can be used by the programmer or compiler writer to provide hints that enable the memory system to place data where they are needed. We demonstrated with examples and simulations the use and performance of the communication primitives in an architecture with a single compute processor per cluster node. Given the parameters that we chose for the communication processor, the memory latency of the software solution is at least 50% higher than that of the hardware implementation. With communication primitives, the optimized software scheme approximates, and sometimes exceeds, the hardware solution's performance in the simplified "cluster" architecture.

- We developed a model based on Mean Value Analysis (MVA) [40, 67] to assess the contention on cluster-shared resources and to estimate the impact of contention on the overall performance of normal cluster architectures. We used the model to evaluate a number of architectural choices, i.e., varying cluster size and adding simple hardware support (remote cache and forwarding logic) to ease contention. The analytical model indicates that the base software implementation of cluster architectures can outperform the single bus system. However, unless an application has good cluster locality, increasing cluster size often harms performance, due to the high software protocol processing overhead. The remote cache can improve performance significantly; it is most effective for applications suffering primarily from capacity or conflict misses. For applications that have a large portion of coherence misses, forwarding logic is more important. We validate the analytical model against simulation results and show that the accuracy of our model is within 5-30% of simulation results.

- We investigated the performance of communication primitives in cluster environments. We found results similar to those of uni-processor node implementation. However, when the cluster size increases, the memory overhead reduction brought by the communication primitives decreases because the growing native intra-cluster data sharing deceases the utility of the primitives. The remote cache is very important to communication primitives even when applications suffer mainly from coherence misses. The forwarding logic is crucial to communication primitives that require the processing of many small requests.

1.3 Organization of the Thesis

The remainder of this thesis is organized as follows. Chapter 2 describes the base architecture and its major components and discusses design and implementation issues. Chapter 3 introduces the communication primitives absent from a pure cache coherent shared-memory scheme. The goals of the chapter are to assess: (1) the overhead of software-based cache coherence protocol, and (2) the performance benefit of the communication primitives in an architecture where there is abundant hardware, hence little contention over shared resources. Chapter 4 develops an MVA-based analytical model for estimating the contention on shared resources. With the fast and reasonably accurate evaluation technique, we evaluate a number of architectural choices, i.e., varying cluster size and adding additional hardware. We also validate the model against simulation results. Chapter 5 studies the performance of communication primitives in cluster architectures. We then examine the effects of three architectural choices (discussed in Chapter 4) on the performance gain achieved by the communication primitives. Finally, we summarize the thesis and propose future work in Chapter 6.

Chapter 2

ARCHITECTURAL FRAMEWORK

This chapter, describes the baseline architecture, its major components, and the two-level cache coherence protocol. The cluster architecture employs two-split transaction devices: the interconnection network and the split transaction bus. These shared resources cause complications in implementing a cache-coherent shared-memory, which this chapter also addresses. Finally, we discuss issues involved in software implementation of a directory protocol.

2.1 Baseline Architecture and Components

Our study is based on an architecture consisting of *clusters* connected to each other by an interconnection network such as a mesh (Figure 2.1). Each cluster is a bus-based shared-memory multiprocessor augmented with a communication processor (CPP). The communication processor replaces the hardwired CMMU; it performs under software control the functions of the directory protocol. Every processor has a private cache (or cache hierarchy). Cache coherence is maintained at two levels. Within a cluster node, a snoopy protocol keeps the local caches coherent. Across cluster nodes, a directory-based protocol tracks the caching information and maintains coherence on a cache-line basis. To achieve a high bus bandwidth, we assume that data and addresses use separate split transaction buses. Our machine model is CC-NUMA, which has a global physical address space; each piece of data is assumed to live in only one of the memory modules, i.e., the data's home node. Furthermore, we assume that memory is sequentially consistent.

The remainder of this section examine three components of the cluster architectures: the communication processor, the memory and its access path, and the interconnect. Our interest in the network chiefly pertains to its performance.

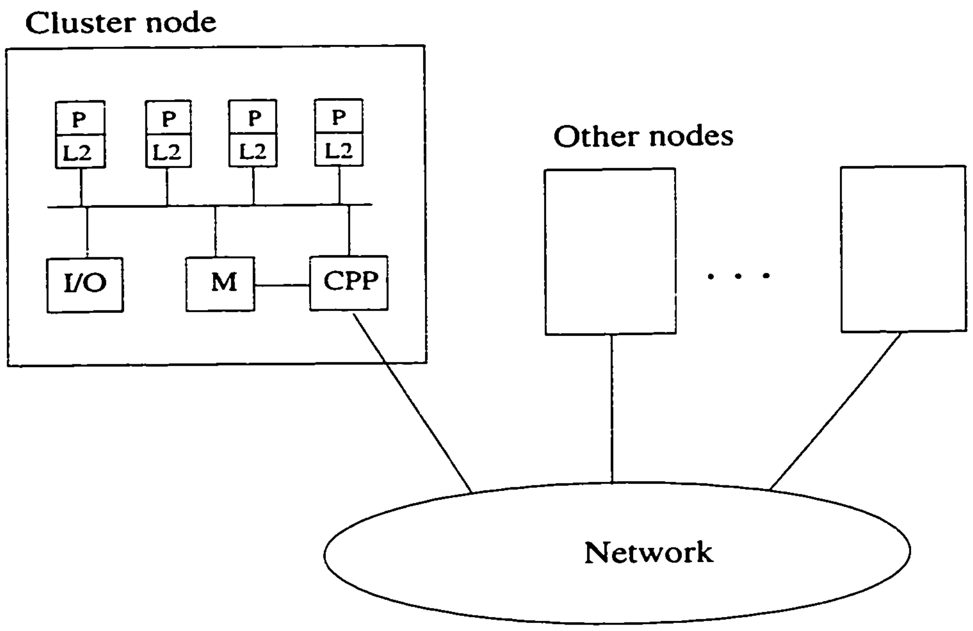


Figure 2.1: Model of the cluster architecture.

2.1.1 Communication Processor

The communication processor in the base architecture, Figure 2.2, contains an embedded protocol processor (PP) for running protocol handlers, a network interface (NI) for draining/pumping messages from/to the network, a bus interface (BI) for communicating with local processors, and a memory controller (MC) that lets the protocol processor access memory. The BI and NI each have two queues. The input queue of the BI stores requests and replies from the compute processors, e.g., miss requests and writeback data. It will also be used to save user-initiated communication primitives as we expand the functions of the protocol processor (discussed in Chapter

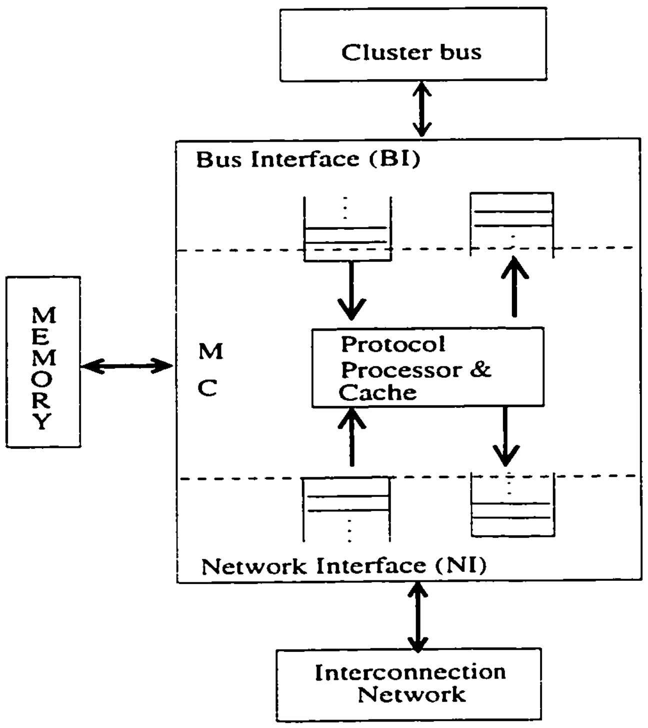


Figure 2.2: Model of the communication processor.

3). The output queue of the BI stores replying data and invalidation/writeback requests, etc. The input and output queues of the NI store the same types of messages from/to another node.

Handling small messages is one of the most critical functions of the communication processor. Therefore, we employ an integrated memory-mapped network interface for the CPP. We also believe that the messaging mechanism should be flexible enough to accommodate system evolutions. As a result, we chose to use interrupts to notify the embedded protocol processor of message arrivals and to schedule and dispatch messages in software based on their priorities. This is different from MAGIC, a customized communication co-processor in FLASH [38], which executes both message scheduling and dispatching in specialized hardware. The motivation for our choices is that the types and formats of messages may change as the system evolves. The versatile interrupt and software dispatching mechanisms let the system easily adjust to such changes. Interrupts were chosen over polling (the NI and BI) to free the communication co-processor from the overhead involved in periodical message checking. As we shall see in Section 3.2, this is important when we extend protocol processor functions to lower-priority execution of communication primitives.

When an arriving message interrupts the protocol processor, the interrupt handler dispatches the message to an appropriate message handler based on its type. The message handler either executes the message directly or moves the message from the BI or NI to a software message queue. To handle an interrupt efficiently, we require a processor architecture to have at least one hardware contexts dedicated to the message-receiving interrupt handler and cache coherence handlers. Thus, the service for interrupts and the cache coherence messages always avoids the overhead of register saving and restoring.

2.1.2 Memory and Its Access Path

Memory usually stores data and instructions for the compute processors. In the software cache coherence scheme, it also contains the local memory's directory accessed by the protocol processor exclusively. There are two possible ways to design an access path to memory. One is to attach the memory directly to the shared bus (Figure 2.3a). This design lets local memory references bypass the communication processor. On the other hand, it puts a stringent timing constraint on the communication processor: if a memory block is not in a desired state, the CPP must intervene in the ongoing memory operation by performing a fast lookup of the memory state. In hardware-based cache-coherent systems, such as DASH, the directory access cycle can be designed to match that of the main memory. For a software cache coherence scheme, however, it may not be possible to complete the memory state lookup (in software) in time to abort the illegal memory operation. A plausible solution, employed in STiNG [43], is to add SRAM that stores the memory state and performs snoopy operations in hardware. The SRAM controller can easily meet the timing constraint to detect and intercept requests to invalid memory blocks.

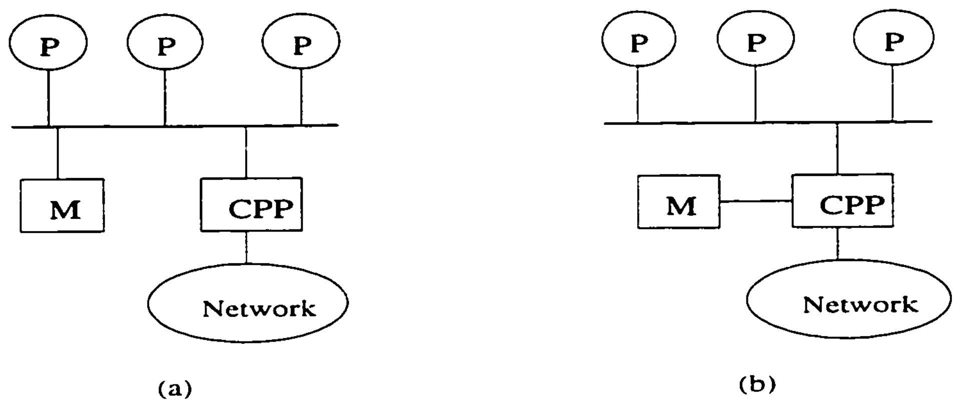


Figure 2.3: Choices of memory access path.

An alternative is to connect the memory to the communication processor (Figure 2.3b). In this case, all references that cannot be satisfied by the local caches go through the communication processor. Both hardware and software implementations of the directory protocol are feasible without the additional SRAM to store the memory state. For example, in Alewife [2], when a cache miss is encountered, the CMMU (hardware directory logic) first looks up the directory state; it then either issues a memory read if the block is in a clean state or a writeback if the data have been modified. The s3.mp [49] implementation is similar, except that both the state lookup and cache coherence are performed by microcode running on the protocol engine.

The memory design in our architectural model lies between these two alternatives (see Figure 2.1). We first assume that applications can distinguish shared data references from private data references. For private references, the memory can supply data without consulting the CPP on memory state. For this purpose, then, the memory module is directly connected to the shared bus. For shared references, the memory is prevented from answering the bus request directly; the communication processor is always involved in looking up the memory state and supplying data, if necessary. This design avoids using costly SRAM and slowing down private data references. It also saves on memory otherwise needed to store the protocol states for private data. However, if a thread migrates from one processor to another, its private pages should be purged from the cache. The protocol processor may need to access the memory to obtain directory information or data if they are not in its own cache. To avoid having the PP contend with compute processors for the shared bus, we employ a dedicated link between the CPP and memory.

Since private and shared data accesses can be distinguished, we require for private accesses that only read misses block the compute processor; write misses use write buffers. For shared references, we require that both read and write misses and write hits to shared lines block the compute processor. For write operations, the processor cannot proceed until all other nodes have acknowledged the invalidations sent on behalf of the write. Since private references are important to the issuing processors only, how they interleave with references from other processors dos not affect the execution of other processors. Therefore, the overall memory system is guaranteed to be sequentially consistent.

2.1.3 Interconnection Network

Two important metrics gauge the performance of interconnection networks: bandwidth and latency. One way to measure the aggregate bandwidth (B_{total}) is to consider the total data transfer rate of all channels[1]. By definition, it is a product of the number of links (L_{total}) and the link bandwidth (B_l). Assuming that data are transferred on only one of the clock edges[2], then $B_l = ClockRate \times BitWidth$. Ob-

[1] Alternatively, one can measure aggregate bandwidth by the bisection bandwidth. Bisection bandwidth is the maximum data rate between two halves of a network, which is a product of link bandwidth and the number of links that sever a network into two equal halves.

[2] Some networks can transfer data on both edges of the clock [44, 69].

viously, the wider the link, the larger the link bandwidth. For this reason, many high performance networks employ parallel data links [23, 61, 26, 13, 44]. Suppose an end-to-end message traverses h hops on average, the effective bandwidth available to a cluster node is approximately $B_{total}/(h \times n) = (B_l \times L_{total})/(h \times n) = B_l \times k/h$, where n is the total number of nodes in the system and $k = L_{total}/n$ is the number of links per node.

Examples of high-performance networks for tightly coupled parallel systems are Caltech's Mesh Routing System [23] (whose variations have been used in Paragon [31], DASH [41], and Alewife [2]) and IBM's Vulcan Switch [61] (used in SP1 and SP2 [70, 3]). Table 2.1 presents the performance data for two networks: DASH's 2-D Mesh Routing Chip (or MRC) and SP's MIN Vulcan Switch.

Table 2.1: Network performance data for DASH and SP2.

	Mesh Routing Chip used in DASH	**Vulcan used in SP2**
Nodes	16	16
Clock rate (MHz)	28.5	40
Bitwidth of data link (bits)	16	8
Link bandwidth (Mbits)	456	320
Aggregate bandwidth (Gbits)	58	41
Bandwidth per processor (Mbits)	910	853
Mean number of hops (nodes)	4	3
Switch and wire delay (ns)	50	75
Average network delay[3](ns)	200	225

[3] The average network latency given in Table 2.1 does not include network interface latency.

It is worth noting that although the high-performance networks used in prior parallel systems are proprietary, they are becoming commodity parts. For example, the NUMA-Q [43] employed SCI (Scalable Coherent Interface) [26] to connect multiprocessor cluster nodes. The basic topology of SCI is a ring, in which the output of one SCI node is connected to the input of the downstream node. In the ring topology, a SCI node performs very simple functions. It either removes incoming packets whose destination *id* matches the node *id* or passes packets to the next node. The simple functionality and point-to-point style connection make it possible to achieve a very high clock rate. For example, the SCI switches manufactured by Vitesse Semiconductor Corporation run at 500MHz. However, the bandwidth available to each processor is only $2 \times B_l/n$ ($k = 1$ and $h = n/2$). As the number of clusters (n) increases, the effective bandwidth per processor decreases quickly, and the average message latency increases linearly. To increase the bandwidth, multiple rings or higher dimension topologies may be necessary.

2.2 Cache Coherence Protocols

As mentioned earlier, cache coherence in a cluster architecture is maintained at two levels. Cache coherence within a cluster node is enforced via a snoopy protocol. Across clusters, cache coherence is maintained by a directory-based protocol. This section describes the detailed operations of these protocols and specifies the interface between them.

2.2.1 Intra-cluster Snoopy Protocol

In a snoopy cache coherence protocol, a cache line can be in one of five states[4] [62]:

[4] The EXCLUSIVE state suggests that the data live in one and only one cache. A write hit to an EXCLUSIVE line is performed without issuing any bus request. The OWNERSHIP state indicates that the memory does not have valid data for the block. The cache that owns the line

1. M (MODIFIED): Exclusive and owned
2. O (OWNERSHIP): Non-exclusive but owned
3. E (EXCLUSIVE): Exclusive but not owned
4. S (SHARED): Non-exclusive and not owned
5. I (INVALID)

If classified by cache state, the Write-once and Illinois protocols [6, 62] each have four states (M, E, S, and I) and therefore belong to the family of MESI protocols. The Berkeley protocol [6, 62], on the other hand, is a member of MOSI.

The EXCLUSIVE state in a stand-alone bus-based multiprocessor is easy to implement: all processors snoop on the shared bus, and each can observe all memory transactions. In a cluster architecture, this is no longer the case. When a processor requests a line to share, the request is seen on the bus of the requesting node and, possibly, on the bus of the home node. The home node must contact the node with an EXCLUSIVE cache line and request it to change the line to a SHARED state. Because of this extra overhead, we decided to use a MOSI protocol.

Our protocol is a variation of the Berkeley protocol. The main differences are twofold. First, our protocol enables more cache-to-cache transfers. For example, the Berkeley protocol only lets the owner of a line, i.e., the cache line in MODIFIED or OWNERSHIP state or the memory, supply the data. Our protocol lets SHARED lines respond as well. Second, our protocol defines the interface between intra-cluster and inter-cluster protocols. More specifically, when a memory request cannot be satisfied in a cluster node, the intra-cluster protocol generates a message of a specific type for the inter-cluster directory protocol. We describe the operations of our intra-cluster snoopy protocol as follows. The message's type is given in italics.

- Read miss: If the line is MODIFIED, OWNERSHIP, or SHARED in the cluster, one of the local caches supplies the data. A MODIFIED is changed to OWNERSHIP. If

is responsible for updating the memory, if necessary, when it replaces data.

none of the local caches has the data, a *ReadMiss* message is generated and entered into the communication processor. After the request is serviced, the block is loaded into the cache (of the requesting processor) in a SHARED state.

- Write hit: If the line is MODIFIED, the write proceeds with no delay. Otherwise, the line must be SHARED or OWNERSHIP[5]. If the cluster has the ownership, it is transferred to the requesting processor after the local copies are invalidated. Otherwise, an *ObtainOwnership* message is generated and entered into the communication processor. After the request is completed, the state of the cache line is set to MODIFIED.

- Write miss: If the line is MODIFIED or OWNERSHIP[5] in one of the local caches, the cache supplies the data and grants ownership after the local caches invalidate their copies. If the line is SHARED in the cluster, the local caches invalidate their copies, one of them supplies the data, and an *ObtainOwnership* is generated and entered into the communication processor. If none of the local caches has the block, a *WriteMiss* message is generated and entered into the communication processor. After the request is serviced, the block is loaded into the requesting processor's cache in a MODIFIED state.

2.2.2 Inter-cluster Directory Protocol

The intra-cluster protocol generates an explicit message for the inter-cluster protocol when a memory reference request cannot be satisfied by the former. In the directory protocol, a memory block can be in one of three states, MODIFIED, SHARED, and UNCACHED. In addition to the state, the directory keeps a complete list of

[5] When a cache line is in the OWNERSHIP state, the line may be shared by other caches in the same cluster, but not by caches in any other clusters. The memory block must be in the MODIFIED state in the directory (cf. Section 2.2.2). Therefore, no inter-cluster invalidation is generated.

nodes that may have a copy of the block. From the directory's viewpoint, nodes are clusters, not individual processors. When the protocol processor receives a memory reference request (i.e., a message of *ReadMiss* or *ObtainOwnership* or *WriteMiss*) from the bus interface, it first determines the home node based on the address, and, if appropriate, forwards the request to the home node. At the home node, the protocol processor looks up the state of the memory block and performs operations as follows.

- *ReadMiss*: If the memory block is SHARED or UNCACHED, the protocol processor reads data from memory and sends it to the requesting node. If the memory block is MODIFIED, the protocol processor looks up the node that modified the block and requests that the node write back the data. As the data arrives, it is forwarded to the requesting node and written back to memory. The directory state of the block is set to SHARED.

- *ObtainOwnership*: The memory block must be SHARED. The home node sends invalidations to all nodes sharing the line except the requesting node. After receiving all the acknowledgements, the home node grants ownership to the requesting node. The directory state is set to MODIFIED.

- *WriteMiss*: If the memory block is UNCACHED, data are read from memory and sent to the requesting node. If the memory block is SHARED, the home node sends invalidations to all nodes sharing the line. Also, data are read from memory. Upon receiving all the acknowledgments, the data and ownership are sent to the requesting node. In both cases, the state of the memory block becomes MODIFIED. If the block is MODIFIED, the protocol processor requests that the owner write back and invalidate the line. After the data arrives, the data and ownership are granted to the requesting node. The directory state of the block is still MODIFIED.

2.3 Cache Coherence and Split Transaction Devices

In a shared-memory system, two processors may request access to a piece of data at the same time. If both requests are reads, the order in which they are executed is unimportant. However, if one request is a read and the other is a write, the system must guarantee that one request is executed before the other for the sequential consistency model is to be maintained. In a system that relies heavily on split transaction devices, e.g. the split transaction bus and the interconnection network, a memory operation and the cache coherence protocol are carried out as a sequence of sub-operations. Maintaining the order of two memory operations can be tricky in certain circumstances. This section explores issues involved in maintaining memory operation ordering on split transaction devices. We then discuss the network properties required by directory-based protocols.

2.3.1 Memory Operation Ordering

Memory operations are executed as a sequence of sub-operations. Sub-operations such as cache snooping and response are executed in parallel by multiple devices. For the cache coherence protocol to function correctly, the sub-operations of two conflicting memory accesses should appear to be in the same order. In other words, the sub-operations of one memory request are executed in the same order with respect to the sub-operations of another memory request on each device. For example, suppose P_0 encounters a read miss R at the same time as P_3 encounters a write miss W to the same address. R is granted service first. Obviously, P_0 cannot perform the invalidation on behalf of P_3's write before transferring the data for the prior read. Figure 2.4 depicts the correct ordering of sub-operations for two conflicting memory requests:

In a stand-alone bus-based multiprocessor, memory sub-operation ordering can be enforced by the bus arbitrator and by the FIFO queue in each bus device

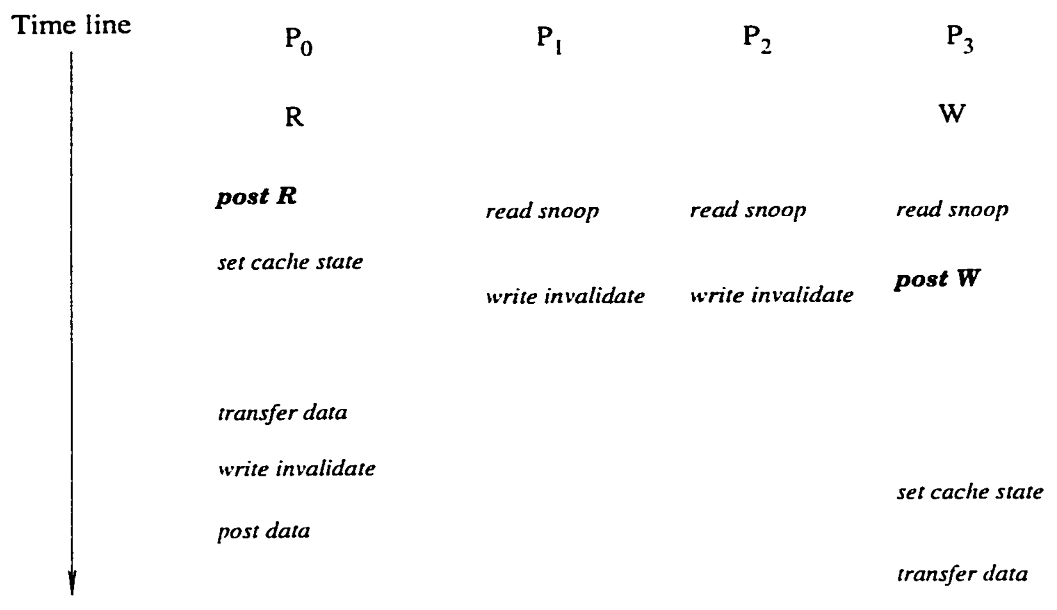


Figure 2.4: Memory sub-operation ordering.

that stores the coherent transactions received from the bus. If two processors raise concurrent requests, the requests' sub-operations enter the FIFO in the order granted by the bus arbitrator. The FIFO queue guarantees that the sub-operations of one request are executed before the sub-operations of the other request on each respective device. Furthermore, the scheme ensures that the two requests are completed in the same order as they were issued to the bus.

This approach is unnecessarily conservative: if two concurrent requests reference independent locations, their sub-operations need not follow the bus arbitration order strictly. A more sophisticated implementation that can relax the sub-operation ordering of two independent memory accesses is to remove long latency pending sub-operations (such as memory reads) from the FIFO queue and store them in a small fully-associative cache buffer. The subsequent sub-operations match their addresses to those of the pending sub-operations in the cache before being executed. If there is no conflict, they can proceed; otherwise, they can be retried later.

In a cluster architecture, the preceding optimization becomes more important, since a pending memory reference could take a few hundred cycles to complete. However, if the bus devices simply hold the addresses of pending requests, they may cause deadlock. To explain why deadlock can occur, let us look at the situation in Figure 2.5. In the scenario, P_0 in Node 0 issues a request to write (W_{p0}) a shared line A. The request is intercepted by the communication processor and forwarded to home Node 1. Before receiving W_{p0}, the home node receives another request from P_1 of Node 1 to write to the same line (W_{p1}). In servicing W_{p1}, the home node sends invalidations to clusters sharing the line, including Node 0. Because W_{p0} is still pending, if P_0 holds the invalidation of W_{p1}, a cyclic dependence is formed: W_{p1} cannot carry on because P_0 is holding the address for W_{p0}. Meanwhile, W_{p0} cannot continue because the home node is guarding the line for W_{p1}.

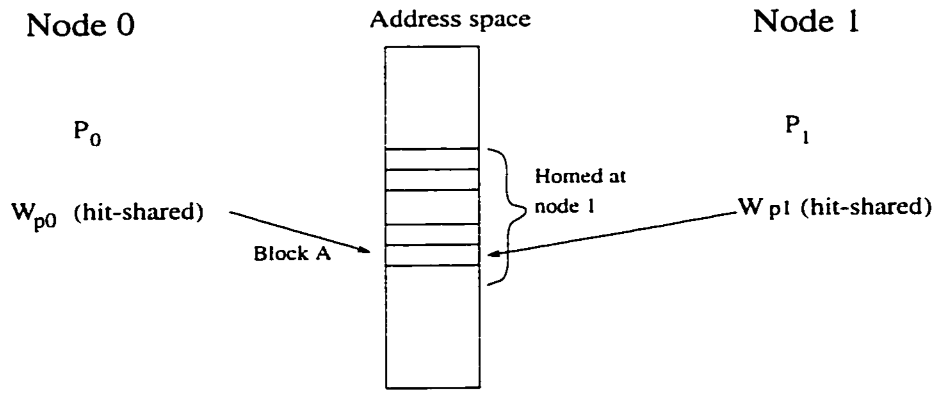


Figure 2.5: A deadlock-prone scenario.

We know that if a memory operation O cannot be resolved within a cluster C, it enters into the communication processor. While the request is pending, two kinds of conflicts can occur: (1) a local compute processor raises a read or write miss request (O_1) to the same address, or (2) a communication processor (home node) issues a writeback or invalidation request (O_2) to the same line. These two cases must be handled differently. In the former case, since the operation of O can affect the snoopy result of O_1, O_1 should retry. In the latter case, the fact that O_2 arrives at node C while O (from node C) is still pending implies that the home node issues O_2 prior to processing O; therefore, O_2 should take precedence, and O should retry.

2.3.2 Directory Protocol and Network Properties

Since the directory protocol frequently uses the interconnection network, the behavior of the network affects to a great extent the implementation of the directory protocol. This section lists the network properties required by a directory protocol and states our assumptions that simplify protocol implementation.

Reliability. Although the directory protocol can be implemented on unreliable interconnection networks, the implementation would be complicated and incur significant overhead if every outgoing message needed to be saved in order to recover from network errors. In most high-performance interconnects, however, data links usually operate under a well-controlled environment and span only a few meters. Therefore, the occurrences of bit errors or lost packets are extremely rare. Even if transmission errors do occur, networks such as Mercury [69, 44] and S-connect [47] provide reliable network service through hardware retransmission. Hence, in our study, we assume a reliable interconnection network layer.

In-order delivery. Under the sequential consistency memory model, correct operation of the directory-based protocol depends not only on a reliable network layer, but also on in-order message delivery. To elaborate, suppose node A issues a request to share a memory block, which is to be serviced by home node H. Meanwhile, node B issues a write request to the same line. At home node H, B's write arrives after A's read. As a result, H sends the replying data first to A, then an invalidation on behalf of B. If the invalidation arrives at A before the data does, node A would have an erroneous cache state. Even in networks that guarantee in-order message delivery, the split transaction style of memory accesses is still vulnerable to thrashing situations under extreme conditions, which can happen more easily in systems adopting multithreading processors. Kubiatowiz [37] illustrated several scenarios of "vulnerable windows" and discussed a range of solutions to the problems[6].

Deadlock. Interconnects such as Caltech's Mesh Routing Chip [23] are guaranteed to be deadlock free *if* messages are consumed at their destination. Problems arise when this condition is not satisfied due to the limited number of buffers in the network interface. To sink the messages and clear the network interface buffers as quickly as possible, designers usually prioritize messages based on their demand on network resources and their chances of causing deadlock. Acknowledgements or replying data are often given higher priorities, because processing them usually does not consume network resources, and their completion helps remove pending requests and release other resources. But message prioritization only reduces the likelihood of deadlock; it does not eliminate the problem. In case deadlock does occur, the system may have to drop messages in order to move forward. To avoid this complication, our study assumes that the network interface has sufficiently large input queues to sink messages.

2.4 Software Implementation of the Directory Protocol

We next discuss issues and details pertaining to the software implementation of a directory protocol. These include directory memory management, message handlers, and cache replacement.

2.4.1 Directory Memory Management

The coherence directory data structure we use is the dynamic pointer scheme [58, 38]. Each memory block has a directory header and a linked list of shared nodes. The

[6] These solutions are beyond the scope of our thesis, and the author refers you to [37] for further information.

directory header saves the state of the memory and the pointer to the shared node list. The directory headers of a (shared) page are organized as an array, an entry in which corresponds to one cache line-size memory block (Figure 2.6). The memory overhead of the directory header array is a function of the size of the directory header and the size of a cache line. Suppose the directory header is 8 bytes and the cache line size is 64 bytes, a 4KB page then needs a directory header array of 0.5KB, or 12.5% overhead.

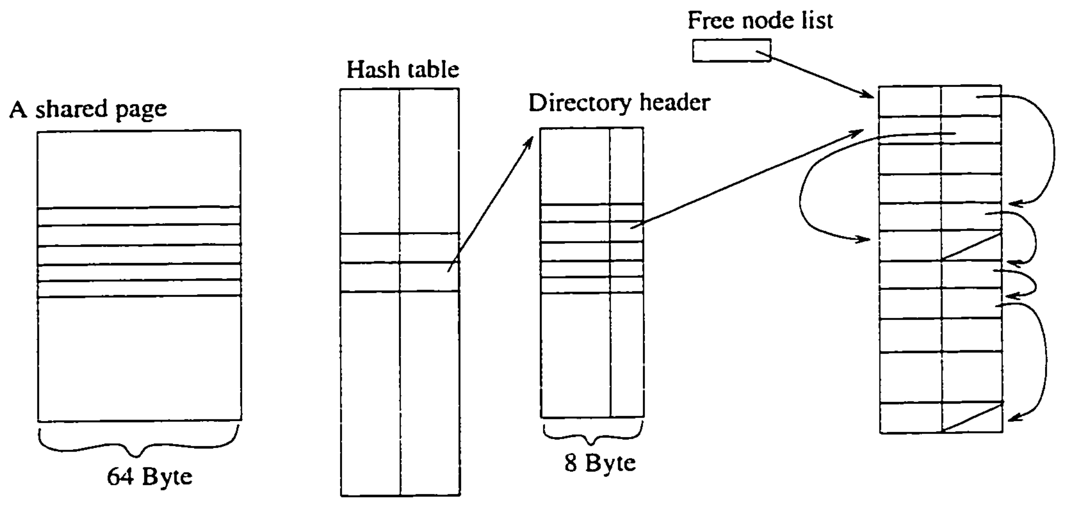


Figure 2.6: Coherence directory data structures.

Memory for the directory header array is allocated dynamically from a free directory-header-array pool. A hash table maps the physical address of a shared page to the address of a directory header array. When a shared page is referenced for the first time, the mapping is invalid. At that time, a directory header array is allocated, and a map is established for that page. Subsequent accesses to blocks in this page find the directory header array by a hash lookup and the directory header of a specific block by adding a displacement. Note that the communication processor does not recognize virtual addresses. Only physical addresses are passed to the communication processor.

Memory for the shared node lists is also dynamically allocated from a free node-list store. Initially, the OS assigns a certain number of physical pages (for the free directory-header-array pool and the free node-list store) to the communication processor based on the amount of memory applications request for shared data structures. This number can increase at runtime if the OS detects that it is allocating more shared pages than expected on a particular node.

2.4.2 Message Handlers

The protocol processor starts a message handler after receiving a message interrupt. The message handler can either process the message to completion if all coherence actions can be taken in the node or process the message partially if other nodes of the system are needed (e.g., write hit on a shared line). In the latter case, the handler needs to suspend itself, yielding control to other threads.

Before a message handler suspends itself, it sets the directory of the memory block to a transient state to prevent subsequent requests to the same line from being processed prematurely. In addition, the message handler creates a data structure to be used upon receiving the expected messages. This data structure contains the handler's name and state (indicating a point where the handler should continue once the expected messages arrive), a pointer to the directory header of the memory block in progress, and necessary bookkeeping information, e.g., the number of acknowledgements received. To resume the suspended handler appropriately, the replying messages carry a pointer to this data structure. During execution of the message handler, interrupts are disabled, as in Active Messages [68].

2.4.3 Cache Replacement

Cache replacement can cause instantaneous inconsistency between the directory information and the cache states. Inconsistencies occur between the time a node sends a request for replacing a cache line and the time the home node processes the request. Such inconsistency complicates implementation of the directory protocol. Consider the situation in which a home node requests a block to be written back from node C_1. The data are in MODIFIED or OWNERSHIP state in processor p_1 of C_1. Before p_1 receives the writeback request, it replaces the same line and sends the modified data to the home node while the home node is waiting for a reply to its writeback request. When the writeback request actually arrives at C_1, the data would no longer exist in the node. Consequently, C_1 sends a negative acknowledgement (Nack), containing no data, to the home node. When this happens, the home node can be certain that the modified data must have arrived earlier than the Nack, because the network provides in-order delivery service (cf. Section 2.3.2). The home node must preserve the data sent back by cache replacement so that it can retrieve the data when processing the Nack.

In case a line in the SHARED state is replaced, there are two options. The first is to replace the line without notifying its home node; the other is to report the replacement to the directory. The former case causes directory information to be inaccurate with respect to the state of the cache; the directory protocol can still function correctly, however. The penalty is that home nodes may generate needless invalidation messages. In the latter case, the processor that replaces the *last* shared copy in a cluster node needs to inform the home node. To do that, the bus signaling must have a capability similar to that of the snoopy protocol supporting the EXCLUSIVE state. Our implementation uses the second approach.

2.5 Summary

This chapter described the base cluster architecture and its major components, i.e., the communication processor, the memory, and the interconnection network. In addition, we outlined our intra- and inter-cluster cache coherence protocols. Since the cluster architecture relies heavily on split transaction devices, we also examined issues vital to both hardware and software implementations of a cache-coherent shared-memory. Finally, we discussed the issues unique to software implementation of the directory protocol.

Chapter 3

COMMUNICATION PRIMITIVES IN UNI-PROCESSOR NODE ARCHITECTURES

Although the use of communication processors and of software cache coherence permits protocol optimization and saves hardware design time, it increases the latencies of memory references that need the assistance of communication processors. This chapter addresses the memory latency problem. We propose a set of explicit communication primitives that exploit the flexibility of the programmable protocol processors. These primitives provide the user with the advantageous features of asynchronous message-passing while retaining the simplicity of the cache-coherent shared-memory paradigm.

Our goals are to assess the overhead of a software implementation of the base cache coherence protocol relative to an ideal hardware scheme and to measure performance gains when the proposed communication primitives are applied. For these purposes, we compare, via execution-based trace-driven simulation, a subset of the SPLASH-2 benchmark suite in four environments: (1) a PRAM model, (2) an idealized hardware cache coherence scheme, (3) a software scheme implementing only the basic cache coherence protocol, and (4) an optimized software solution supporting the additional communication primitives and running with applications annotated with those primitives. To isolate the effect of software overhead from that of shared resource contention, we assume in this chapter an architectural model in which each cluster node contains a single compute processor.

The rest of the chapter is organized as follows. Section 3.1 introduces the proposed communication primitives and their semantics. Section 3.2 discusses issues involved in implementing the primitives on the communication processor. We describe our experimental methodology in Section 3.3: design of experiments, selection of benchmarks, and the simulation environment and parameters. In Section 3.4, we show how the benchmarks were modified with the introduction of the communication primitives and present simulation results of the proposed scheme and the baseline architecture. We summarize related work in Section 3.5 and conclude the chapter in Section 3.6.

3.1 Explicit Communication Primitives

Under usual cache coherence policies, the fetching of data is performed on demand only, i.e., when a cache miss occurs. Similarly, the storing of data in memory is done only when a replacement is needed [1]. The primitives that we propose extend, under programmer or compiler control, these basic data movement operations. Table 3.1 shows the primitives and their semantics. Each primitive is a non-blocking operation. Furthermore, we require that global cache coherence be maintained for the requested data.

The first two primitives, $get(addr, size)$ and $getex(addr, size)$, let the programmer prefetch a set of consecutive cache lines and request that these lines be either in SHARED or EXCLUSIVE state. Granting ownership to the receiving processor can significantly reduce write latency in subsequent accesses to that data. Note that the *get* operations are not equivalent to page migration or DMA transfers since the data are transferred to the cache of the requester and global cache coherency is maintained.

While *get/getex* are consumer oriented, the next two primitives, $put(pid, addr, size)$ and $putex(pid, addr, size)$, are producer oriented. These operations are akin to an asynchronous send in message-passing; they can be used when the producer knows the identity of the consumer. By using a *put* operation, the producer process can

[1] Most of the lower-level caches in the memory hierarchy follow a write-back policy.

Table 3.1: Communication primitives. *addr* is the starting address of a block of data. *size* is its size in bytes. *pid* is a processor number. *pids* is a mask indicating a list of processors. *mode* is either write-back (*wb*) or write-through (*wt*).

Primitives	**Semantics**
$get(addr, size)$	fetch data into requesting processor's cache
$getex(addr, size)$	fetch data with ownership
$put(pid, addr, size)$	place data in the cache of processor *pid*
$putex(pid, addr, size)$	transfer data with ownership to processor *pid*
$multicast(pids, addr, size)$	disseminate data to a set of processors
$putmem(addr, size)$	return data to memory
$writemem(mode, addr, size)$	set the write policy

store a set of cache lines, in a given state, in the cache of the processor running the consumer process. When there is more than one consumer and their identities are known, the $multicast(pids, addr, size)$ primitive can be used.

When the producer knows that it will not use the data any longer, but does not know what process will consume it, the data can be stored in memory with the $putmem(addr, size)$ primitive. This may save half of the request-reply bandwidth in the network when the data are next used. A similar effect can be achieved on a word-per-word basis by changing the default write-back policy to a write-through one. The *writemem* primitive can restrict this policy to a range of addresses; if its parameters *addr* and *size* are both null, it can be applied to all write misses. By selecting the mode, the user has the choice of alternating between the two write policies.

When utilized appropriately, these primitives benefit the application for the following reasons:

1. *Overlap of communication with computation.* This overlap can be extensive since the operations dictated by the communication primitives are non-blocking, and multiple requests can be outstanding.

2. *Bulk data transfers.* The network can be better used by pipelining transmissions.

3. *Tailoring the cache coherence protocols.* For example, superfluous data transfers present in write-update protocols can be avoided, or early requests for ownership in write-invalidate protocols can reduce the number of control messages in migratory-like patterns.

However, there are dangers in using these primitives unwisely. These dangers resemble those that exist when using prefetching or post-storing too aggressively: cache pollution, increase in coherence traffic, and saturation of the network. Therefore, the programmer or compiler writers must insert the primitives with discretion, taking into account the cache configuration and usage of the application's data.

3.2 Implementation Issues

The communication primitives differ from cache miss requests in four significant ways. First, the compute processor issues communication primitives directly to the communication processor without checking the state of its own cache. Second, no local snoopy operations have been performed for the lines being requested. Third, the communication primitive requests are not limited to one cache line. Fourth, they do not block the issuing processor. This section discusses the effect these differences have on the hardware and protocols and outlines our implementation strategies.

3.2.1 Change in Cache Controllers

Since the compute processor issues primitives without checking its own cache, the cache controller is not aware of such pending requests. Therefore, a major change that the communication primitives require from the cache controller and the bus protocol is that they allow unsolicited data to be pushed into the cache.

3.2.2 Communication Primitive Protocols

Because the compute processor does not validate the requests of communication primitives with its own cache or the caches of other local processors, the protocols used for the primitives differ from those for cache miss requests. For instance, assume that a processor writes to a shared line, and the request cannot be satisfied in the local cluster; it is therefore forwarded to the home node. The home sends invalidations to all sharing nodes *excluding* the requesting node, because the requesting processor has already issued a local invalidation. On the other hand, if a processor issues a *getex* to a shared line, the home node sends invalidations to all sharing nodes *including* the requesting node.

One of the functions of the communication primitive protocols is to filter useless user requests. With the communication primitives, it is possible that a processor may request a piece of data that exists in one of the local processors. When this situation occurs, the communication primitive protocols can choose not to send the data, because the processor can get the data locally when referencing the block. The protocols can further require that the processor issuing *put*, *putex*, or *multicast* requests own the data; otherwise, no data are transferred upon such requests. In any case, since cache coherence policies are governed by software rather than hardware, they can be changed quite easily.

As can be seen, the communication primitive protocols rely entirely on directory information. Therefore, the accuracy of this information is crucial in determining efficient coherence policies for the protocols. In a simplified cluster architecture of one processor per node, we maintain an accurate list of the sharing nodes in the directory (cf. Section 2.4.3); we can therefore actually deduce the cache states of individual processors. The fact that the issuing processor does not validate the requests with the local caches does not create problems. However, in a general form of cluster architectures, the fact that there are multiple compute processors in a node does present challenges to the efficient implementation of the protocols. We discuss the issue further in Chapter 5.

3.2.3 Bulk Data Requests

In Section 2.4.1, we mentioned that the communication processors see physical addresses only. This implies that compute processors are responsible for all the address translation. If the size of the data being requested is larger than a page or not aligned to a page boundary, the compute processors also need to decompose the request into several smaller requests and align each to the page boundary.

Each communication primitive is forwarded to the home node based on the address of the request. At the home node, a bulk request is treated on a cache-line basis.

Message Priorities

Once the communication primitives are implemented, the communication processor receives and processes two kinds of messages: (1) mandatory messages required in the base cache coherence protocol, and (2) user communication primitives. Because the user primitives do not block the compute processors, we give them low priority, and their corresponding message handlers are interruptible by the mandatory messages, which have high priority.

High priority messages are executed as soon as the communication processor is not processing another high priority message. Low priority messages are removed from the BI and NI interfaces and stored in a software-managed message queue in memory, most likely in the communication processor's cache. When the communication processor becomes idle, it polls the (software) message queue to see if there are any pending messages. If so, the first one in the queue is processed in the hardware context for low priority messages.

Just as conflicts can happen among mandatory messages, they can also occur between miss requests and communication primitives. Each communication primitive is processed on a cache line basis. Therefore, the sub-operation ordering of two requests to the same line—one due to a cache miss request, the other to a communication primitive—can be resolved in the same way as it is for two conflicting mandatory messages. In other words, the order of the respective sub-operations is determined by the order in which the home node starts processing the conflicting requests.

3.3 Experimental Methodology

Generally speaking, the execution of an application running on a parallel system consists of computation time and communication and synchronization overhead. Even under a PRAM architectural model, where the overhead of communication reduces to zero, an application can still suffer from synchronization overhead due to serial sections or load imbalance. Therefore, we can divide the execution time[2] of a parallel application into four components:

1. Computation time (labeled *CPU busy* in the figures in Section 3.4)
2. Synchronization time under a PRAM model. This part is due to the intrinsic properties of the algorithm, e.g., producer-consumer relationships, serial sections of the program, load imbalance (labeled *Sync-algo*)

[2] In our measurements, we exclude the initialization time from execution time, because the initialization is often performed on one processor.

3. Communication time due to memory latency (*Mem latency*)

4. Extra synchronization time due to the effects of memory latency (*Sync-mem*)

These four components are not independent. For example, a mismatch between the partitioning of computation and partitioning of data can easily cause excessive communication overhead. Likewise, an inefficient memory system exacerbates load imbalance. Since our goal is to test the efficiency of the communication primitives with respect to a hardware implementation, we isolate the contributions of each component and compare the performance of the applications, via simulation, in four experimental environments.

- Case 1: A machine with a perfect memory system (PRAM model).
- Case 2: A hardware-based, cache-coherent (full directory [15]) system.
- Case 3: A system that uses a communication processor, with the embedded protocol processor implementing the coherence protocol (software implementation).
- Case 4: A system as in Case 3, with the addition of the communication processor able to handle user-based communication primitives (optimized software implementation).

The difference between Case 1, where only computation and synchronization time are taken into account, and Case 2 reveals the loss of parallel efficiency due to memory latency minimized as much as possible by the best hardware implementation. The difference between Cases 2 and 3 shows the additional overhead incurred with a software implementation. Case 4 shows the potential performance gains obtained by supporting communication optimization schemes on top of a flexible software infrastructure.

Table 3.2: Summary of benchmark data. Numbers are given in millions.

Application	Problem size	Total intsruction	Shared read	Shared write
FFT	64K points	33.32	6.00	5.73
LU	256x256 matrix, 16x16 block	64.20	23.11	11.19
RADIX	0.5M integers, 1024 radix	29.55	6.70	3.46
RAYTRACE[3]	teapot	375.93	54.60	0.22

3.3.1 Benchmarks

For our experiments, we selected three kernel applications—FFT, LU factorization, and RADIX sort—from the SPLASH-2 benchmark suite [71, 72]. These applications have been coded with a CC-NUMA system in mind. Thus, they already have some embedded communication optimizations. They also exhibit coarse-grain regular communication patterns that can be exploited by the proposed communication primitives. Table 3.2 summarizes some pertinent statistics about the applications: problem size, and number of instructions, number of read and write references to shared data.

3.3.2 Simulation Environment and Parameters

We chose Mint [64] as our simulation tool, since our interest is primarily in the performance aspects of the memory system. Mint is a software package that emulates multiprocessing execution environments and generates memory reference events that drive a memory system simulator. In addition to memory references, Mint provides an interface to trigger any user-specified event. We use this interface for specifying communication primitives in the applications. When encountering such primitives

[3] This benchmark is used in Chapter 4.

during execution, Mint generates special types of events that invoke the simulation back-end.

The systems we simulated had 16 processors. We considered five combinations of cache size and associativity: infinite cache, large (256KB) direct-mapped and 2-way set-associative caches, and small (32KB) direct-mapped and 2-way set-associative caches. The cache-line size was set at 32 bytes.

We assume perfect pipelining in the compute processor. This assumption affects only the *CPU busy* time, which remains essentially the same for the four environments, and thus does not bias the results. In addition to the cache effect, the memory system simulator models the operations of the communication processor in detail. This includes: the interrupt overhead of message reception; the overhead of moving the message from the BI and NI input queues either to be executed directly or to be stored in (software) queues; the time to package and write messages to the BI and NI output queues; and the manipulation of the directory data structure. Table 3.3 lists major architectural parameters specified to the simulator.

Table 3.3: Principal architectural parameters of the communication processor.

Parameter	**Value**
Interrupt and context switch	8 cycles
Bus interface (in/out)	2 cycles
Network latency (one way)	24 cycles
Network interface (in and out)	12 cycles
Retrieving data from memory (first word)	14 cycles
Size of buffers (BI, NI)	infinite
Max. number of pending requests	10

To illustrate the overhead of the software implementation, we show in Table 3.4

Table 3.4: Difference in timing between hardware and software implementations of a shared variable read miss. The directory for the block is in a home node different from the node requesting the data.

Action	**Resource**	**Software** (in cycles)	**Hardware** (in cycles)
Miss detection	Compute processor	6	6
BI processing (inbound)	BI (in)	2	2
Receiving/forwarding request	CPP/CMMU	17	4
(De)package message	NI (in and out)	12	12
Transfer message	Network	24	24
Home node handling	CPP/CMMU and memory	55	24
(De)package message	NI (in and out)	12	12
Transfer	Network	24	24
Processing data	CPP/CMMU	26	8
BI processing (outbound)	BI (out)	2	2
Totals		180	118

the timing differences between the hardware and software implementations of a shared read miss. The data is assumed to be in a non-exclusive state (i.e., one hop is sufficient). As can be seen, in ideal conditions (no contention) the software implementation is 54% slower. While this is the slow-down effect that we use in our simulations, it is in fact quite favorable to the hardware implementation, because we have not tried to optimize the communication processor as much as we could; we want to keep it a flexible, programmable resource. Furthermore, with current technological trends, network and memory latencies and bandwidths will not progress as fast as processor speed; the software implementation will suffer less from this widening gap. For example, doubling only the network latency would reduce the preceding 54% factor to 39%. Finally, the software implementation's latency will suffer less from expansion of the system; there will be no need to change the full directory structure into a partial one, since all directory look-ups are handled in software.

3.4 Performance Results

This section describes each application, emphasizing its communication aspects, i.e., the massaging of the major data structures. We then explain how to annotate the applications with the communication primitives proposed in Section 3.1. Finally, we present results of the simulation of the four architectures described in Section 3.3.

3.4.1 FFT

The application. The SPLASH-2 FFT algorithm is optimized for distributed or hierarchical memory systems [7]. The input data of FFT consists of a $\sqrt{n} \times \sqrt{n}$ matrix of complex numbers. The major data structures are the input matrix A, its transpose B, and another matrix of the same dimension for the "roots of unity". In the beginning, one processor performs initialization for the entire input matrix and the roots of unity. The post-initialization algorithm consists of six phases. (1) $B \leftarrow A^T$; (2) and (3) Compute and update B; (4) $A \leftarrow B^T$; (5) Compute and update A; (6) $B \leftarrow A^T$. Each processor is assigned to compute a set of consecutive rows. During a transpose phase, each processor reads a subblock of the source matrix and writes into the corresponding rows of the destination matrix. During a computation phase, each processor operates on the set of rows set up in the prior transpose phase. There are three synchronizing barriers, after phases 3, 5 and 6.

In a hardware invalidation-based protocol, communication occurs in the transpose phases. For example, in phase 4, when a processor reads a subblock of B, it generates read misses, since the data for that source matrix was generated on a subblock-by-subblock basis by other processors in phases 2 and 3. When a processor writes the data in A, it needs to generate invalidation messages, since the first transpose phase left the corresponding cache lines in a SHARED state.

We use the communication primitives to interleave computation with communication. More specifically, in phases 2 and 3 (likewise in phase 5), we insert *put* operations as soon as a processor (producer) has finished computing a row of B to disseminate the data to the other (consumer) processors. In addition, immediately after the *put*s, we insert *getex* calls to place in the correct state the cache lines corresponding to the rows of A that the processor overwrites in phases 4 and 6. Thus, when the processor finishes the computing phase, it has (or at least has requested) in its cache the needed data in the desired state to perform the transpose. Either the data have been *put* there by another processor, or they have been prefetched via a *getex*.

For this aggressive communication latency hiding scheme to be successful. the cache must be large enough to hold the working sets of both the computation and transpose phases. If it is not, cache pollution occurs because the *put* or *get* operations will have been performed too early. For small caches, we use *get* only and schedule the prefetching primitive right before the matrices are accessed in the transpose phases.

Simulation results. Figure 3.1 displays simulation results for FFT running on the four model architectures and the combinations of cache size/associativity described in Section 3.3. The leftmost bar is the result for the PRAM model. As mentioned above, the use of communication primitives was dependent on the size of the caches.

First, PRAM results show that the load balance is close to ideal. Second, a comparison between the PRAM and hardware implementation shows that communication, which occurs only during the transpose phases, is intensive. Execution time in the hardware implementation is more than double that of the PRAM when we in-

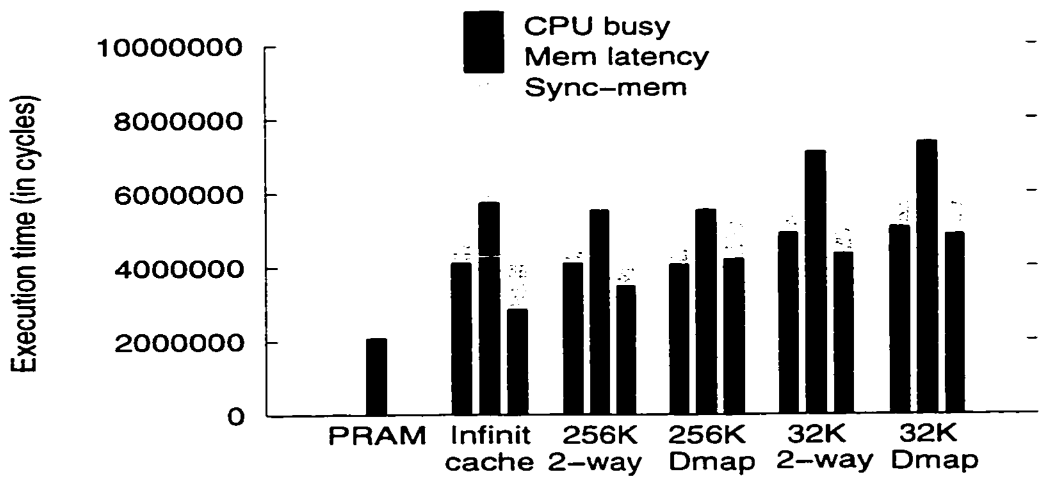


Figure 3.1: Execution times for FFT. "CPU busy" is the compute processor execution time. "Mem latency" is the amount of time the compute processor waits because of memory latency and cache coherence effects. "Sync-Mem" is the synchronization time, including load imbalance, due to the effect of memory latency. For each cache configuration, the three bars, from left to right, show the results of the hardware full-directory implementation, the software implementation, and the optimized software implementation, respectively.

clude a realistic memory latency and a fast but inflexible coherence protocol. Third, as already noticed in other studies [21], infinite caches can be worse than large finite caches: while there is less data traffic, there can be more coherence traffic or contention for data in a single cache. This phenomenon occurs here during phase 4 of the original program (phases 2 and 3 of the optimized program), since A is initialized by one processor.

Comparing the software and hardware implementations shows the price paid by adding (software) flexibility. Section 3.3 presented an example (cache read miss) in which the latency in the software implementation was 54% higher than that of the hardware implementation. This ratio is close to the 70-80% and 70-96% ratios of the *Mem latency* and *Sync-Mem* components of the execution time. The slightly higher ratio in the execution times can be explained by the two facts: (1) messages processed locally take relatively longer in the software implementation than in the hardware one, and (2) there is contention in the communication processor.

The insertion of communication primitives pays off handsomely. The optimized method is now much faster than the original software implementation. Unless conflict misses occur frequently (direct-mapped caches), it is even faster than the hardware implementation. Table 3.5 breaks down the phase-by-phase execution times of the two software schemes for the infinite cache case.

Notice first that memory latency in the optimized case is only 23% of the original. This gain outweighs overwhelmingly, in absolute numbers, the 22% loss in synchronization time. Introducing communication primitives has a negligible effect on computation time (*CPU Busy*). Overall, the optimized implementation is 40% faster than the original.

In the original implementation, almost all memory latency overhead is incurred in transpose phases 1[4], 4, and 6. This overhead is almost twice as high as it is for the processor's busy time. During the transpose phases, no computation takes place. Conversely, during the computation phases, there is almost no memory traffic (recall that we have an infinite cache).

After the optimization, communication is performed while computation is in progress, except during phase 1. In phase 1, the reduction of memory latency stems from the possibility of transferring of several cache lines with a single request. This eliminates the need for sending and processing multiple small messages, thus reducing

[4] The first transpose phase (phase 1) incurs less memory latency. When a processor first writes the transposed data into its own cache, no other processor has previously touched the data. Therefore, writes proceed almost twice as fast as in the other two transpose phases.

Table 3.5: Breakdown of execution times of FFT for original and optimized software implementation (infinite cache).

Phase	**Version**	**Instruction (in cycles)**	**Memory (in cycles)**	**Synchronization (in cycles)**
1	Original	53925	792100	0
	Optimized	61119	386036	0
2,3	Original	1004490	3691	140073
	Optimized	1010790	271857	1128018
4	Original	53910	1570010	0
	Optimized	53908	8482	0
5	Original	915967	2382	736366
	Optimized	922303	150690	51664
6	Original	53958	1271600	122772
	Optimized	53955	2670	16036
Total	Original	2082305	3641568	1026376
	Optimized	2102136	821539	1222850

network traffic (cf. Figure 3.2, which shows an appreciable decrease in the number of requests). This bulk transmission mode also enables the home node to pipeline data transfers, hiding communication latency even further. In phases 2, 3[5], and 5, the large amount of computation should be sufficient to overlap with the entire memory latency of phases 4 and 6, respectively. The residual memory latencies in

[5] The large synchronization time at the end of phase 3 of the optimized software implementation is caused by initializing the input matrix A on a single processor. A comparable corresponding synchronization time is found at the end of phase 5 of the original implementation.

those phases are caused by the fact that the simulation limits the number of pending communication operations to 10. This is less than the required number: namely, (*number of processors* $- 1$) *puts* and one *getex* for complete overlap with the computation.

The infinite cache case illustrates the potential performance gain of an aggressive communication and computation overlapping scheme. When the cache size is limited, the aggressive use of communication primitives raises the danger of polluting the cache. In FFT, the pollution can be caused by conflicts between the current working set (e.g., a set of consecutive *rows* of B in phases 2 and 3) and the future working set (e.g., a set of consecutive *columns* of B and corresponding rows of A in phase 4). In the case of the large 256KB cache, we see that the pollution effect is indeed present, since the memory latency time (in the optimized case) is twice as much as that of the infinite cache for 2-way set associativity and three times as much for direct-mapped. Even so, however, the performance is comparable to that of the hardware scheme, where there are almost no conflict misses: slightly better for the 2-way case, slightly worse for the direct-mapped. For the two small cache configurations, data are prefetched into the caches immediately prior to use. The results show that the conservative use of communication primitives also reduces the memory overhead significantly. Again, the results are comparable with those of the hardware scheme.

Figure 3.2 shows network traffic before and after optimization. As mentioned earlier, the optimized version results in fewer requests. Cache pollution, on the other hand, increases not only requests, but also data to be transferred, since data displaced by prematurely prefetched data must be refetched. This is particularly visible in the 256KB, direct-mapped cache. Finally, the number of control messages slightly decreases because of *getex* requests that place data in the right state.

Summary. For FFT, the use of communication primitives to prefetch data and to put cache lines in the correct state in advance yields execution times for the software optimization that are comparable, even slightly lower, than those of the hardware

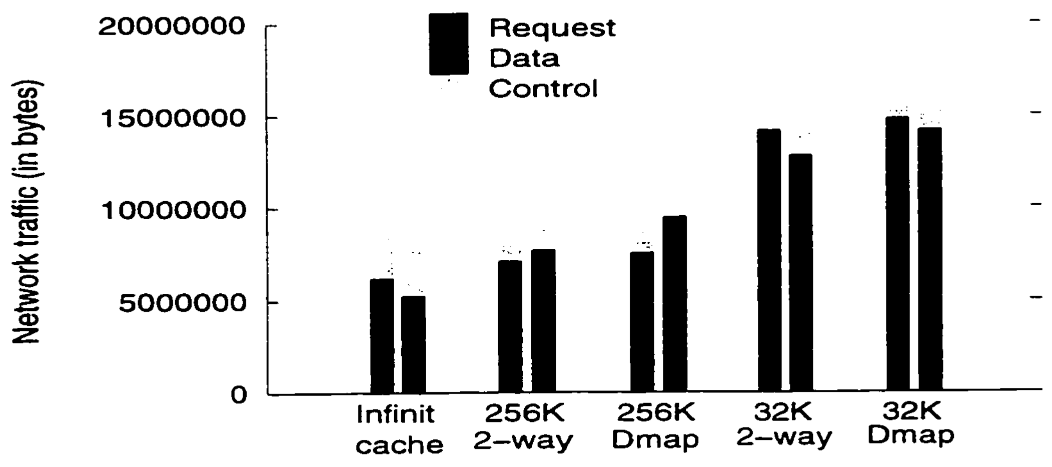


Figure 3.2: Network traffic for FFT. "Request" "Data", and "Control" are the network traffic for sending cache miss requests or communication primitives, data, and control messages (e.g., invalidation and acknowledgment), respectively. The left and right bars show the network traffic for unoptimized and optimized software implementations, respectively. Network traffic for the hardware implementation is omitted since it is equivalent to that of the software implementation.

implementation.

3.4.2 LU Factorization

The application. The SPLASH-2 parallel implementation of the LU factorization of a dense matrix has been optimized to exploit data locality [72]. Nonetheless, the serial sections of the application produce a fair amount of load imbalance. The input matrix is divided into submatrices, or blocks, which are assigned to processors in a two dimensional scatter decomposition fashion. In essence, every processor is responsible for factorizing the same number of blocks; this number diminishes almost uniformly for all processors during the computation. The algorithm iterates over the number of blocks along the diagonal. In the k-th iteration, the processor responsible for the block A_{kk} factorizes it. This is by necessity a serial part of the algorithm. The perimeter blocks A_{ik} and A_{kj} are then computed using A_{kk}, a phase that requires communication. This can be performed in parallel, but the processors owning the perimeter blocks all need to access data in the processor owning A_{kk}. Finally, the remaining interior blocks A_{ij} are updated by the processors responsible for them. This phase also requires communication between the updating processors and the processors to which the perimeter blocks have been assigned.

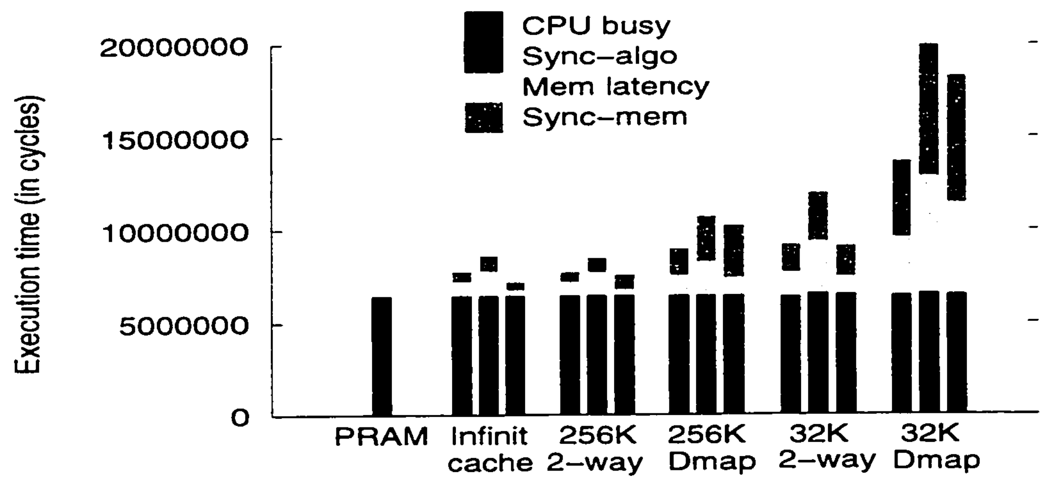


Figure 3.3: Execution times for LU. For a detailed explanation of the legend, refer to the caption of Figure 3.1. In addition, "Sync-algo" is the synchronization time due to load imbalance intrinsic to the algorithm.

Communication required in the first and second phases of each iteration provides opportunities for using communication primitives. Thus, we modified the algorithm in three ways. First, when a (producer) processor is factorizing a diagonal block A_{kk}, it sends (*multicast*) a row of data as soon as that row is computed to the (consumer) processors in charge of the perimeter blocks A_{ik} and A_{kj}. Second, after a (producer) processor finishes updating one row of a perimeter block, it sends (*multicast*) the data to the (consumer) processors that use the block for processing interior blocks. Finally, since an interior block is exclusively accessed by the processor in charge of its computation, we let the processor prefetch the data with ownership (*getex*) before updating the block. For infinite and large caches, prefetching is scheduled once at the first instance of phase 3. For small caches, prefetching is issued in each iteration.

Simulation results. Figure 3.3 displays execution times for the four environments. The PRAM simulation shows the effects of the serial sections. Processors are idle one third of the time, on average. This synchronization is exacerbated on a realistic system by the additional memory overhead, because busy processors take longer to complete assigned tasks, thus keeping idle processors waiting longer. On the other hand, memory overhead accounts for only 10% to 15% of the total execution time in the hardware implementation and 15% to 30% in the original software implementation. This leaves less room for optimization, however, reducing memory overhead is still desirable, because it helps reduce synchronization overhead as well.

The 256KB caches are large enough to accommodate the whole working set of LU for the problem size we simulated. When the caches are 2-way set-associative, the optimized solution is as efficient as the hardware implementation for both large and small caches. When the cache is direct-mapped, the large number of conflicts takes its toll; the effect is magnified in the software implementations, because of the increased latencies. The conflicts that occurred within the working set of phase 3, when each processor factorized interior blocks using perimeter blocks, cannot be avoided in the optimized software implementation. Thus, we see less significant improvement, especially in the case of the 32K direct-mapped cache.

Network traffic does not vary much among the three implementations (cf. Figure 3.4). The optimizations do not introduce severe cache pollution.

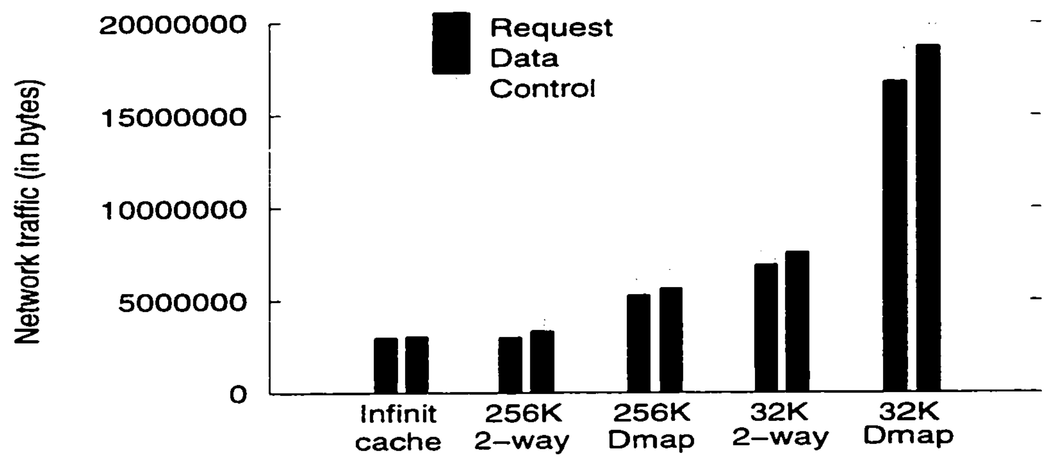


Figure 3.4: Network traffic for LU. For a detailed explanation of the legend, refer to the caption of Figure 3.2.

Summary. LU offers less opportunity to perform optimizations. However, the use of *multicast* and prefetching is worthwhile, mostly if the caches are not direct-mapped. In that case, the performance of the optimized software is as good as that of the hardware implementation, which is within 20% to 45% of the PRAM lower bound.

3.4.3 RADIX Sort

The application. RADIX sort is part of the NAS parallel benchmark [8]. It sorts k-bit integers by examining r bits ($r \leq k$) of the keys per iteration. In the parallel implementation, each processor is assigned an equal fraction of the keys. An iteration consists of three phases. In phase 1, each processor scans through its assigned keys and computes the local histogram and density of each radix value. In phase 2, prefix sums of the local histograms and the global densities (sums of local densities of all processors) are computed. These results are used to compute the new position of each key in the output order. In phase 3, the keys are permuted based on the new position computed in the prior step.

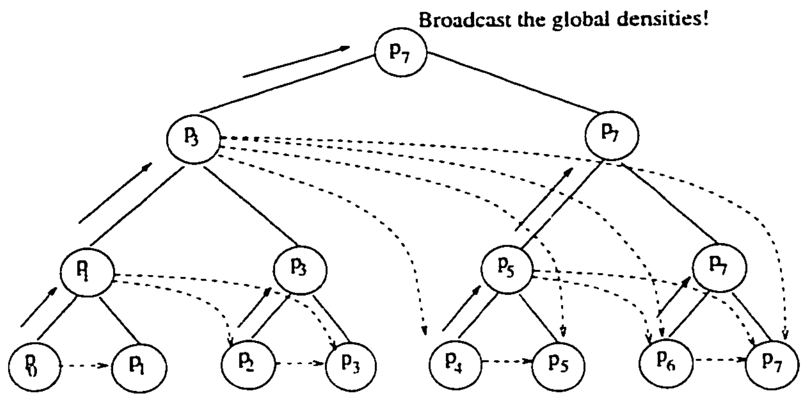


Figure 3.5: Communication patterns for computing the global densities and prefix sums of the local histograms. Each processor initially has local histograms and local densities in its cache. The solid arrows indicate the data transfer needed for both global densities and prefix sums. The dashed curves are for prefix sums only. The label in each node is the processor that computes the intermediate or final results.

The major data structures are two arrays that are used alternately for the input and sorted keys, and a binary tree of arrays to store the prefix sums of the local histograms and the (partial) results of the global densities. Communication occurs in every phase at each iteration, but with different patterns. In phase 1, each processor reads a portion of the keys, sorted and written in phase 3 of the previous iteration. Although the reading is sequential, the key order cannot be determined until phase 3 completes. Phase 2's communication patterns for computing the global densities and the prefix sums are illustrated in Figure 3.5 using eight processors. In phase 3, each processor determines the new order of the keys assigned to it, and the pattern is random. False sharing occurs frequently in this phase, since neighboring keys might be assigned to different processors.

Communication primitives can be used in each phase in different ways. In phase 1, the only possibility is to prefetch data as soon as phase 3 terminates. We insert *get* operations for prefetching keys before computing the local histograms and density functions. In phase 2, we insert *put* or *multicast* operations in the (producer) processors to send data to the (consumer) processors according to the regular pattern of communication flow shown in Figure 3.5. Neither *get* nor *put* primitives can be used in phase 3, because data locations are not known until run-time. Some false-sharing can be avoided if processors write memory directly (write-through) rather than cache data. Thus, a *writemem(wt)* call is inserted at the beginning of phase 3, and a corresponding *writemem(wb)* at the end.

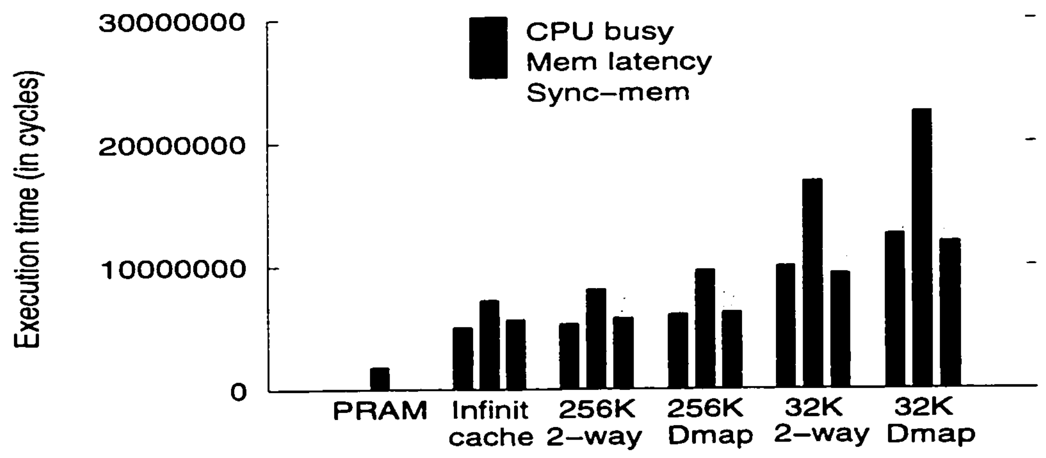


Figure 3.6: Execution times for RADIX. For a detailed explanation of the legend, refer to the caption of Figure 3.1.

Simulation results. Figure 3.6 shows performance results of the RADIX sort. As can be seen when comparing the PRAM and hardware implementation execution times, memory latency effects are the most important of the three applications. A very small load imbalance in the PRAM case occurs during phase 2. However, since phase 2 is short compared to phases 1 and 3 (cf. the instruction count in Table 3.6), the effect is not significant and cannot be seen given the scale of the figure.

Table 3.6 breaks down execution times for a 256KB, 2-way set-associative cache for the original and optimized software implementations. In the original implementation, and in the hardware implementation, more than half of the memory latency effects arise in phase 3. Unfortunately, this is where communication primitives are least applicable, since communication patterns are random. Nonetheless, the change of write policy in this phase yields a 20% improvement over the original implementation. The prefetching effects in phase 1 are quite significant, decreasing memory latency and associated load imbalance by a factor of two for the phase. Communication primitives also reduce memory latency in phase 2; however, this effect is less important, since phase 2 does not take much time to execute..

The gains of the optimized implementation are even more significant for small caches. The optimized software implementation performs almost as well as the hardware one.

Figure 3.7 displays the network traffic before and after the optimizations. Network traffic for RADIX was largely reduced due to the change in write policy. It is particularly striking for small caches, where false sharing and conflict misses are avoided by writing data through memory in phase 3.

Summary. Radix is the most memory-intensive of the three applications. It is also the least amenable to optimization. Nonetheless, a combination prefetching and of write policy change makes the optimized software implementation competitive with the hardware implementation.

Table 3.6: Breakdown of RADIX's execution times for the original and optimized software implementations given a 256KB 2-way set associative cache.

Phase	**Version**	**Instruction (in cycles)**	**Memory (in cycles)**	**Synchronization (in cycles)**
1	Original	704778	1783595	113057
	Optimized	704834	515701	310923
2	Original	139419	707990	1248630
	Optimized	139682	435627	1156961
3	Original	999610	3733430	248395
	Optimized	1048766	2905070	240463
Total	Original	1843872	6226998	1638077
	Optimized	1893347	3858511	1737689

3.5 *Related Work*

The communication primitives that instruct the system to perform efficient data transfers resemble: the asynchronous send/receive operations in message-passing interfaces [9, 51], prefetching [41, 16, 25] and poststore [34] commands, non-blocking (bulk) read (get) and write (put) operations in the split-phase assignment statement of Split-C [19], and explicit communication mechanisms [53]. The common idea is overlapping communication with computation. The differences are in policy with respect to message-passing systems, since the global addressing space paradigm and cache coherence is maintained; second, in the (extendible) set of primitives, since the communication primitives include both producer-consumer oriented operations and availability of bulk transfers; and third, in implementation, since the execution of the optional communication primitives are interruptible by mandatory shared memory

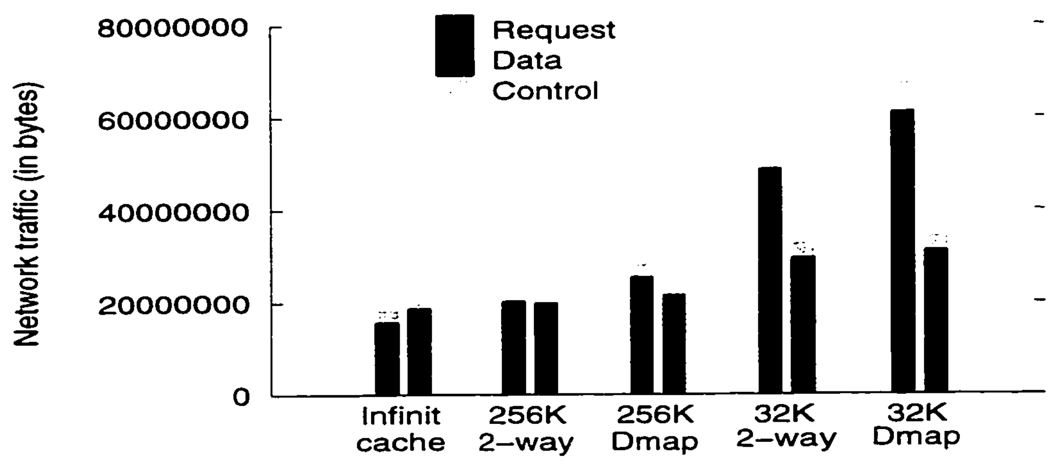


Figure 3.7: Network traffic for RADIX. For a detailed explanation of the legend, refer to the caption of Figure 3.2.

requests which are assigned a higher priority.

Tailoring the coherence protocol to the application can be done in several ways. Protocol modifications can be specified by the user at a coarse grain level [22] or in incremental fashion [29], can be dictated by the compiler [52, 20], or can be the result of hardware monitoring [18, 60]. In the applications we studied, we saw the need to apply different coherence strategies at the granularity of a computational phase on a data-structure-per-data-structure basis.

Combining message passing with shared memory to overcome the inefficiencies of cache coherence mechanisms was first proposed in the context of the Alewife project [35, 36] and further elaborated in Flash [27]. A number of important issues have been raised and discussed, e.g., user-level messaging and protection, and coherence strategy for bulk data transferring. Since our interest was mainly on performance-related issues, we concentrated on the latter, imposing a global coherence strategy for prefetching and bulk data transfers.

The design of our communication processor resembles that of Flash [38, 28]. Flash is a tightly coupled CC-NUMA system that uses a programmable processor (MAGIC) and software (running on MAGIC) to maintain cache coherence. The MAGIC chip is highly optimized. It includes special hardware to assist in message receiving, scheduling, dispatching, and directing outgoing messages to proper destination interface units. The processing of incoming messages, protocol handling, and preparation of outgoing messages are pipelined. In addition to its macro-pipelining architecture, MAGIC employs speculative memory access to overlap memory latency with protocol handling. As a result, the slowdown due to software overhead is minimized to only 2%-12% over that of an ideal hardware implementation. In contrast, our co-processor is more flexible, allowing more easily the introduction of new primitives or changes in protocols; however, our software overhead is on the order of 50%. The use of dedicated communication hardware can also be found in tightly coupled message-passing systems such as Intel Paragon [31] and has been proposed for networks of workstations [54].

3.6 Summary

This chapter proposed a set of communication primitives that can enhance the performance of cache coherent shared-memory multiprocessors. These primitives give the user some of the capabilities of message-passing systems, while maintaining the correctness and simplicity of the global address space paradigm. The primitives allow the prefetching and post-storing of blocks of data whose sizes are not limited to single cache lines; they also permit the tailoring of the cache coherence protocol to the needs of the application. We also described implementation strategies that let the communication processors handle in software not only these primitives, but all cache coherence transactions. This pure software approach incurs some overhead, but it adds flexibility by facilitating the introduction of new primitive and the implementation of variations in cache coherence protocols.

We selected three benchmarks from the SPLASH-2 suite for performance evaluation purposes. We then simulated a 16-processor system with five different cache configurations. We simulated four environments: an ideal PRAM model to gauge the effects of memory latency and cache coherence, a system where cache coherence was maintained with a full directory hardware scheme, and two implementations with a communication processor, one using the original benchmarks and one with communication primitives appropriately inserted. With the parameters that we chose for the communication processor, the memory latency and cache coherence overhead in the original software solution were at least 50% higher than they were in the hardware implementation. With communication primitives, the optimized software solution gave results comparable to those of the hardware solution.

These evaluation results are encouraging and point to the value of a software approach enhanced by user directives. Each of the communication primitives was exercised in at least one of the applications, with prefetching being used for all three applications. Their use resulted in significant performance improvements over the base software cache coherence scheme. The quantitative results, which show a performance comparable to the hardware solution, are conservative, since an increase in network latency or memory bandwidth would have a greater impact on the hardware implementation. The software solution is also more attractive because it is more scalable. For example, there is no difficulty in keeping a full directory, since it is a software data structure that is more amenable to change.

Chapter 4

MODELING CONTENTION ON SHARED RESOURCES

In cluster architectures, memory requests are satisfied either locally, i.e., within a cluster (intra-cluster), or externally, i.e., by another cluster (inter-cluster). The intra-cluster misses require use of the cluster's internal resources: the common bus, the local memory attached to the bus, the snooping caches of other processors in the cluster, and, on occasion, the communication processor. The inter-cluster misses require, in addition to the internal resources, use of the network and at least one remote communication processor.

An important difference between the two types of misses is that the latencies of intra-cluster misses are several times smaller than those of inter-cluster misses if contention on the bus or memory is not severe. Under this condition, obviously, the higher the proportion of intra-cluster misses, the better the performance, assuming that the total miss ratio is fixed. One way to achieve a high intra-cluster miss ratio is to incorporate as many processors as possible in a node. However, when there are too many compute processors sharing the bus or memory, intensive resource contention lengthens the time needed to resolve intra-cluster misses relative to contention-free inter-cluster misses. Therefore, a balanced system should be such that the number of intra-cluster misses must not saturate the cluster's bus or memory, while the number of inter-cluster misses must not be too large.

To estimate the performance of cluster architectures, we develop a Mean Value Analysis (MVA) [40, 67] based model. The model lets us efficiently assess the contention on major shared resources and its impact on the performance of the software cache coherence scheme. The model's input parameters include *architectural parame-*

ters, such as the cycle times of the sub-operations of memory reference requests, and *application-dependent parameters*, such as cache miss profiles. Application-dependent parameters are obtained via trace-driven simulation.

The rest of Chapter 4 is organized as follows. Section 4.1 discusses several design options that directly affect application behavior and resource usage, i.e., varying cluster size and adding simple hardware. Section 4.2 introduces the analytical model, notations, and formulae for estimating contention, cache miss latencies, and overall execution time. Section 4.3 describes our choice of architectural parameters, and how we obtained the application parameters for our evaluation. Section 4.4 presents the quantitative results by exercising the model. We highlight also where performance bottlenecks may arise and how performance responds to design alternatives. Section 4.5 compares analytical results with simulation results to validate the model. We present related work in Section 4.6 and summarize our results in Section 4.7.

4.1 Design Options

Many variables can be adjusted to change the performance of cluster architectures. In this chapter we keep constant most of the architectural parameters, such as the size of the processor's private cache and network latency. We discuss three design choices that either directly alter the intra-to-inter-cluster miss ratio or change the usage of highly demanded resources such as the protocol processor.

4.1.1 Cluster Size

Given a processor's private cache size, the number of cache misses it encounters is often independent of cluster size. However, the ratio of intra-to-inter-cluster misses does change when the cluster size varies. At one extreme is a cluster containing a single compute processor (the model used in Chapter 3). In this configuration, each processor may often need data on remote nodes. Thus, there may be many inter-cluster misses. When cluster size increases, the number of intra-cluster misses increases, and the number of inter-cluster misses decreases. At the other extreme is a single-bus system; all cache misses become intra-cluster.

On the other hand, when cluster size increases, more compute processors share the bus, the memory, and the protocol processor. Contention for these resources can be so high as to offset the benefit of data sharing within the cluster. Our goal is to determine the cluster size for which applications can enjoy the greatest benefit of sharing without incurring severe resource contention. Note that when we vary the cluster size, we keep constant the total number of processors.

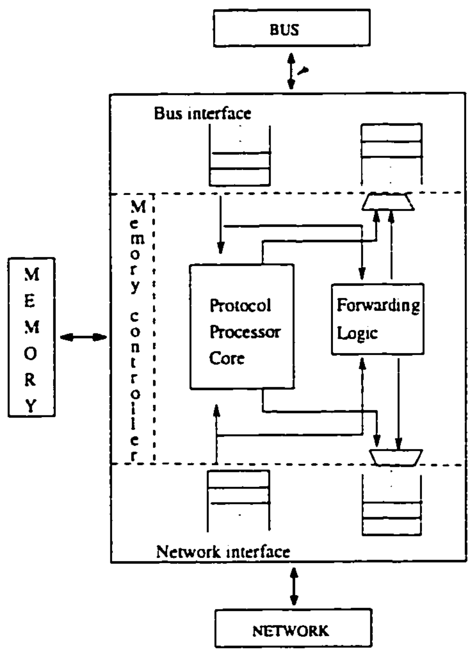
4.1.2 Remote Cache

The second design option to the base architecture is adding a remote cache shared by the processors in each node. The remote cache, often an order of magnitude larger than the processor's private cache, stores data homed at remote nodes. For applications that have a per-processor working set larger than the private cache, or for applications whose data cannot be statically partitioned across physical memories to exploit local memory access effectively, the remote cache can eliminate a good portion of inter-cluster misses. In this respect, the remote cache resembles the COMA model.

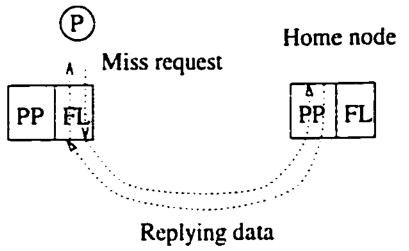
The remote cache affects cluster architecture performance in two ways. First, it increases the ratio of intra-to-inter-cluster cache misses. Second, it retains modified data in a node longer, increasing the number of 3-hops inter-cluster misses. Our goal is to evaluate the performance benefit of this additional hardware resource.

4.1.3 Forwarding Logic

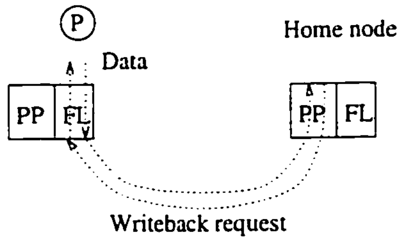
When several compute processors share a software-controlled communication processor, its protocol engine may quickly be overloaded. To decrease the load on the protocol engine, we propose adding special hardware, *a forwarding logic module* (Figure 4.1a). The module would allow the request that needs not access the coherence directory of a node to bypass the protocol engine of that node. For example, a read request for remote data would route through the forwarding module of the requesting node instead of using its protocol engine (Figure 4.1b). Similarly, a "writeback" from a home cluster would bypass the protocol engine of the node executing "writeback" and use the forwarding module (Figure 4.1c). With the forwarding logic, every cache miss or communication primitive uses only the protocol engine of the home node.



(a): The protocol processor



(b): A remote read routing through forwarding logic.



(c): A writeback request using forwarding logic

Figure 4.1: The communication processor augmented with forwarding logic.

In contrast to the logic of directory-based protocols, the forwarding logic is much simpler and can be easily implemented in hardware. In fact, such forwarding logic can be found in the previous generation of shared-memory machines such as RP3 [50] and BBN Butterfly [10]. In these systems, shared variables are not cacheable. The main function of the forwarding logic is to support a globally shared address space for distributed memories.

Table 4.1: Classification of read misses.

Miss	Resource	Protocol Actions Followed on a Miss
R1	Bus Cache	*Requested data is in a local cache.* Data transfer cache-to-cache.
R2	Bus,PP Mem	*Home node is local, data is clean in memory. "ReadMiss"* enters home, which issues a memory read and updates directory.
R3	Bus,PP Net,Mem Cache	*Home node is local, data owned remotely. "ReadMiss"* enters home, which asks owner to write back. As data returns, home sends it to requester, writes to memory, and updates dir.
R4	Bus,PP Net,Mem	*Home node is remote, data is clean in memory. "ReadMiss"* forwarded to home, which issues a memory read, then sends data to requester, and updates directory.
R5	Bus,PP Net,Mem Cache	*Home node is remote, data owned by home. "ReadMiss"* forwarded to home, which issues a writeback locally. When data returns, home forwards it to the requesting node, writes to memory, and updates directory.
R6	Bus,PP Net,Mem Cache	*Home is remote, data owned by a third cluster. "ReadMiss"* forwarded to home, which requests owner to write back. When data arrives, home forwards it to the requester, writes back to memory, and updates directory.

4.2 Analytical Model

We present our analytical model in a top-down fashion. We first express execution time in terms of memory access latencies. These latencies can increase when there is contention forcing the requests to wait before they are serviced. We thus develop a

Table 4.2: Classification of write misses.

Miss	Resource	Protocol Actions Followed on a Miss
W1	Bus Cache	*Requested data is owned by a local cache.* Data transferred cache-to-cache with ownership. Local copies are invalidated.
W2	Bus,PP Mem Cache	*Home is local, data is clean locally only. "ObtainOwnership"* enters home node. Data transferred from cache/mem/ . Local copies invalidated. Home grants ownership, updates dir.
W3	Bus,PP Net Cache	*Home node is local, data is owned remotely. "WriteMiss"* enters home, which asks owner to write back and invalidate. As data arrives, home sends it to the requester, updates dir.
W4	Bus,PP Net,Mem Cache	*Home is local, data is clean remotely too. "ObtainOwnership"* enters home node. Data transferred from cache/mem/ . Local copies invalidated. Home invalidates sharing clusters. After receiving acks, home grants ownership, updates dir.
W5	Bus, PP Net Cache	*Home is remote, data is owned by home. "Writemiss"* forwarded to home, which issues a writeback and invalidation. As data arrives, home sends it to the requester, updates dir.
W6	Bus,PP Net,Mem Cache	*Home is remote, data is clean locally only. "ObtainOwnership"* forwarded to home. Data transferred from cache/mem/ . Local copies invalidated. Home grants ownership, updates dir.
W7	Bus,PP Net Cache	*Home is remote, data owned by a third node. "Writemiss"* forwarded to home, which issues a writeback and invalidation. As data returns, home sends it to the requester, updates dir.
W8	Bus,PP Net,Mem Cache	*Home is remote, data clean at other nodes. "ObtainOwnership"* forwarded to home. Data transferred from cache/mem/ . Local copies invalidated. Home invalidates sharing clusters. After receiving acks, home grants ownership, updates dir.

MVA-based model to account for contention's effect on the memory latencies. The model's input includes application-dependent parameters and architectural parameters. The model's output provides the queue length and waiting time at each device, approximate miss latencies, and the execution time.

4.2.1 Estimating Execution Time

A program's execution time can be estimated as:

$$T_{execute} = T_{instrs} + \sum_{r} M_r \times L_r \tag{4.1}$$

where T_{instrs} is the time to execute the instructions on each processor (assuming good load balance), M_r is the number of type r cache misses, and L_r is the latency for type r misses. In Tables 4.1 and 4.2, we classify cache misses according to their service demand on shared resources and summarize the actions needed for each *type* of read and write miss. If there is no contention, L_r is the sum of service times of the resources used by a type r miss. If there *is* contention, L_r is the sum of service times *plus queuing times* on those resources:

$$L_r = \sum_{k}(W_{k,r} + S_{k,r}) \tag{4.2}$$

In Equation (4.2), $W_{k,r}$ is the total waiting time of a request of type r on a resource of type k. $S_{k,r}$ is the total service demand of a request of type r on a resource of type k. As shown in Tables 4.1 and 4.2, each miss request of a given type generates a specific sequence of sub-requests. These sub-requests, the resources they use, and their semantics are shown in Table 4.3. Tables 4.4 and 4.5 synthesize Tables 4.1, 4.2 and 4.3, showing in detail the sub-requests and associated resources for each miss type. For example, a read miss of type $R3$ first generates the following four sub-requests in its home cluster:

1. A miss request posted on a cluster's address bus (Areq on resource Abus)

2. A *ReadMiss* message inserted in the input queue of the bus interface (BreqI on BI(I))
3. Protocol processing operations (PPops on PP)
4. Request for a remote cluster to write back modified data (NreqO on NI(O))

When the remote cluster returns the data, the home node executes another sequence of sub-requests:

1. Data received at the network interface (NdatI on NI(I))
2. Protocol processing operations (PPops on PP)
3. Data inserted in the output queue of the bus interface (BdatO on BI(O))

4.2.2 Architectural Assumptions

As evidenced by previous discussion and by Tables 4.3 and 4.4, our architectural model is quite detailed. Nonetheless, we make a few assumptions and simplifications to keep the analysis computationally tractable and effective.

First, we consider only symmetric systems in which each cluster contains the same number of processors, the same amount of memory, and remote cache, if any. Second, we assume the following when looking at each component of a cluster.

- The instructions executed on the compute processor are perfectly pipelined. Therefore, the CPI is one under an ideal memory system.
- L2 cache misses block compute processors, which must wait for memory requests to complete before executing their next instructions.
- The L2 caches have dual-ported tags. Hence, snooping on the bus does not interfere with a processor's accessing its L2 cache if the processor and the snoop controller access different cache lines. We ignore contention at the L2 level, since it is extremely rare to have concurrent accesses to the same L2 cache line by the processor and by the snoop controller.

Table 4.3: Definitions of sub-requests and resources required. Only a subset of the protocol operations are displayed. Bus transactions are counted in bus clock cycles (marked with asterisks). The other operations are counted in CPU cycles. The latencies of data transactions are for the first 8 bytes.

Sub-requests	**Meaning**	**Resource**	**Cycles**
Areq	Request for data/writeback/ invalidation on address bus	Abus	2*
Xdat	Transfer data on data bus	Dbus	2*
Rl2	Read data from L2 cache	L2	4
Rmem/Wmem	Read/Write memory	Mem	14
Rrc/Wrc	Read/Write remote cache	RC	14
BmsgI	BI sends/receives msg: dat(data)	BI(I)	2
BmsgO	req(request), ack(acknowledgement)	BI(O)	2
NmsgI	NI receives msg	NI(I)	4
NmsgO	NI sends msg	NI(O)	8
Fmsg	Forwards msg	Fwd	3
PPops	Protocol processing (below)	PP	
PPsend/PPrecv	PP sends/receives messages	PP	3/12
PPsched	Schedule a protocol handler	PP	4
DIRstatus	Directory status lookup	PP	5
DIRadd	Add a sharing node	PP	6
DIRdel	Delete a sharing node	PP	6
DIRrtrv	Retrieve a sharing node	PP	3

Table 4.4: Service demands of read misses on shared resources. Rows of this table show the sub-requests and the resources needed (each resource has its own column) in a given cluster for a particular type of read miss (cf. Table 4.3 for the meanings of abbreviations). For misses involving multiple clusters, the first row shows service demands for the local cluster; the second and third rows (if present) are the service demands for the second and third clusters involved.

Miss Type	Sub-requests							
	Abus	Dbus	L2/Mem	BI(I)	BI(O)	NI(I)	NI(O)	Fwd/PP
R1	Areq	Xdat	Rl2					
R2	Areq	Xdat	Rmem	BreqI				PPops
R3	Areq	Xdat	Wmem	BreqI	BdatO	NdatI	NreqO	PPops
	Areq	Xdat	Rl2	BdatI	BreqO	NreqI	NdatO	Freq, Fdat
R4	Areq	Xdat		BreqI	BdatO	NdatI	NreqO	Freq, Fdat
			Rmem			NreqI	NdatO	PPops
R5	Areq	Xdat		BreqI	BdatO	NdatI	NreqO	Freq, Fdat
	Areq	Xdat	Rl2 Wmem	BdatI	BreqO	NreqI	NdatO	PPops
R6	Areq	Xdat		BreqI	BdatO	NdatI	NreqO	Freq, Fdat
			Wmem			NreqI NdatI	NreqO NDatO	PPops
	Areq	Xdat	Rl2	BdatI	BreqO	NreqI	NdatO	Freq, Fdat

Table 4.5: Service demands of write misses on shared resources. For a detailed explanation of the table, refer to the caption of Table 4.4.

Miss	Sub-requests							
Type	Abus	Dbus	L2/Mem	BI(I)	BI(O)	NI(I)	NI(O)	Fwd/PP
W1	Areq	Xdat	Rl2					
W2	Areq	[Xdat]	[Rl2] [Rmem]	BreqI	BownO			PPops
W3	Areq	Xdat		BreqI	BdatO	NdatI	NreqO	PPops
	Areq	Xdat	Rl2	BdatI	BreqO	NreqI	NdatO	Freq, Fdat
W4	Areq	[Xdat]	[Rl2] [Rmem]	BreqI	BownO	NackIs	NreqOs	PPops
	Areq			BackI	BreqO	NreqI	NackO	Freq, Fack
W5	Areq	Xdat		BreqI	BdatO	NdatI	NreqO	Freq, Fdat
	Areq	Xdat	Rl2	BdatI	BreqO	NreqI	NdatO	PPops
W6	Areq	[Xdat]	[Rl2]	BreqI	[BdatO]	[NdatI]	NreqO	Freq,[Fdat]
			[Rmem]			NreqI	[NdatO]	PPops
W7	Areq	Xdat		BreqI	BdatO	NdatI	NreqO	Freq, Fdat
						NreqI NdatI	NreqO NdatO	PPops
	Areq	Xdat	Rl2	BdatI	BreqO	NreqI	NdatO	Freq, Fdat
W8	Areq	[Xdat]	[Rl2]	BreqI	[BdatO]	[NdatI]	NreqO	Freq,[Fdat]
	[Areq]		[Rmem]	[BackI]	[BreqO]	NreqI NackIs	NreqOs [NdatO]	PPops
	Areq			BackI	BreqO	NreqI	NackO	Freq,Fack

- The cluster buses are split transaction [24]. The address and data buses are two separated resources that can be used simultaneously for different transactions.

Finally, we do not consider network contention for three reasons. First, the bandwidth provided by current network technology appears to be sufficient for the size of the systems investigated in this thesis, thus network contention is not an important factor. Second, many models have been developed to analyze the performance of various networks [1, 12]. Third, the MVA (Mean Value Analysis) technique we use cannot cope easily or directly with network contention. If necessary, results of existing contention models could be incorporated in the analysis (cf. Section 4.6).

4.2.3 Modeling Contention

Returning to Equation (4.2), $S_{k,r}$ and $W_{k,r}$ are sum totals of the service and waiting times of the sub-requests issued to k by a type r miss. While $S_{k,r}$ values are architectural parameters, contention is present in the determination of $W_{k,r}$ values. We use a closed queuing network to model the cluster architectures, with compute processors as customers, and buses, memories, and protocol processors as service centers. We use the MVA technique and the following notations to solve for $W_{k,r}$:

- N is the total number of processors in the system.
- N_c is the number of processors per cluster, i.e., the cluster size.
- I is the average number of instruction cycles between cache misses.
- R is the mean total time between cache misses.
- P_r is the probability that a miss is of type r.
- U_k is the utilization at a type k center.
- $Q_{k,loc}$ ($Q_{k,rmt}$) is the arrival queue length at k observed by a local (remote) request.
- $s_{k,r,i,loc}$ ($s_{k,r,i,rmt}$) is the service demand of the ith sub-request to k from a local (remote) r miss. Recall that a cache miss can issue multiple sub-requests to a

service center.

- s_k is the average sub-service demand per miss on center k.
- $m_{k,r,loc}$ ($m_{k,r,rmt}$) is the number of sub-requests to k issued by a local (remote) r miss. If r does not require a local (remote) k resource, $m_{k,r,loc} = 0$ ($m_{k,r,rmt} = 0$).
- $m_{k,r}$ is the number of local and remote sub-requests issued to a type k resource by a type r miss. $m_{k,r} = m_{k,r,loc} + m_{k,r,rmt}$. If r does not use type k resource at all, $m_{k,r} = 0$.
- $w_{k,loc}$ ($w_{k,rmt}$) is the waiting time at k for a sub-request issued by a local (remote) miss.

Based on the assumption that a cache miss blocks the compute processor, each processor alternates between executing instructions and waiting for miss requests served by the memory system. Thus, the mean total time between cache misses can be expressed by the number of instruction cycles between two misses under an ideal memory system plus the time spent in the memory system:

$$R = I + \sum_r (P_r \times \sum_k (W_{k,r} + S_{k,r})) \tag{4.3}$$

where

$$W_{k,r} = \begin{cases} 0 & if\ k\ is\ L2\ cache\ or\ network \\ \\ m_{k,r,loc} \times w_{k,loc} + \\ m_{k,r,rmt} \times w_{k,rmt} & otherwise \end{cases} \tag{4.4}$$

$$S_{k,r} = \sum_{i=1}^{m_{k,r,loc}} s_{k,r,i,loc} + \sum_{j=1}^{m_{k,r,rmt}} s_{k,r,j,rmt} \tag{4.5}$$

Equations (4.4) and (4.5) express the total waiting time and total service time in terms of the waiting and service times of sub-requests, respectively. We now consider the queuing effect at each type of service center on the waiting time of each sub-request. Suppose a type r miss issued from cluster C requires resource k of the local cluster and the same type of resource k of a remote cluster. Since the system is symmetric and an application's memory accesses are often evenly distributed across memory modules, for each service performed at a remote cluster, resource k of C needs to perform the equivalent service on behalf of a type r request issued from a remote cluster. Therefore, for any request issued from the local cluster (C), the arrival queue length at k ($Q_{k,loc}$) is contributed by: (1) all the sub-requests issued by the other $N_c - 1$ local processors and (2) by the remote requests of the $N - N_c$ remote processors, with each request having a probability $\frac{N_c}{N-N_c}$ of requesting cluster C. Similarly, for any request issued from a remote cluster, the arrival queue length at k ($Q_{k,rmt}$) is contributed: (1) by the sub-requests issued by the local N_c processors, and (2) $\frac{N_c}{N-N_c}$ of the sub-requests issued by the other $N - N_c - 1$ remote processors. Hence, the arrival queue lengths at center k observed by a local or remote request are:

$$\begin{aligned} Q_{k,loc} &= \frac{\sum_r P_r \times (N_c - 1) \times \sum_{i=1}^{m_{k,r,loc}} (w_{k,loc} + s_{k,r,i,loc})}{R} \\ &+ \frac{\sum_r P_r \times N_c \times \sum_{j=1}^{m_{k,r,rmt}} (w_{k,rmt} + s_{k,r,j,rmt})}{R} \\ Q_{k,rmt} &= \frac{\sum_r P_r \times N_c \times \sum_{i=1}^{m_{k,r,loc}} (w_{k,loc} + s_{k,r,i,loc})}{R} + \\ &\frac{\sum_r P_r \times (N_c - \frac{N_c}{N-N_c}) \times \sum_{j=1}^{m_{k,r,rmt}} (w_{k,rmt} + s_{k,r,j,rmt})}{R} \end{aligned} \tag{4.6}$$

Finally, we can estimate the waiting time for each sub-request by the arrival queue lengths and utilization.

$$\begin{aligned} w_{k,loc} &= (Q_{k,loc} - U_k) \times s_k + U_k \times s_k/2 \\ w_{k,rmt} &= (Q_{k,rmt} - U_k) \times s_k + U_k \times s_k/2 \end{aligned} \tag{4.7}$$

where

$$s_k = (\sum_{r\ use\ k} P_r \times \frac{S_{k,r}}{m_{k,r}})/(\sum_{r\ use\ k} P_r) \tag{4.8}$$

$$U_k = N_c \times \frac{\sum_r P_r \times S_{k,r}}{R} \tag{4.9}$$

Since Equations (4.3) through (4.9) contain cyclic interdependencies, $W_{k,r}$ is solved iteratively with $w_{k,loc}$ and $w_{k,rmt}$ initialized to 0.

4.3 Application parameters

The preceding model takes architectural input parameters (the numbers of cycles needed for contention-free sub-requests) and application dependent parameters (the number of instruction cycles between misses and the cache miss profile). Most of the architectural parameters were presented in Table 4.3. The rest are shown in Table 4.6. This section describes how to obtain application parameters.

Table 4.6: Architectural parameters not included in Table 4.3.

Parameters	**Value**
Number of processors	16
L2 cache	size = 64KB, assoc = 1 line size = 64 bytes

Remote cache	size = 1MB x cluster size
	assoc =4, line size = 64 bytes

CPU clock/bus clock	2
Network latency	24

4.3.1 Methodology

In order to exercise our model properly, we need reasonably balanced workloads. We therefore selected RADIX, FFT, and RAYTRACE from the SPLASH-2 benchmark suite [71]. We have described the algorithms for RADIX in Section 3.4.3 and

FFT in Section 3.4.1. RAYTRACE is an image-processing program that renders a three-dimensional scene using ray tracing [59]. Recall that Table 3.2 (Section 3.3.1) summarizes pertinent statistics about the applications: problem size, and number of instructions, number of read and write references to shared data.

We collected application-dependent statistics by performing a trace-driven simulation using Mint. In the simulation, we assumed that cache misses were insensitive to the accuracy of timing information; hence, we use constant latencies for them. This assumption is reasonable for many applications for which false sharing is a rare case, such as FFT[1], or for those where memory accesses are randomized and evenly distributed across nodes such as RADIX and RAYTRACE. The application-dependent parameters of principal interest are the numbers of cache misses of each type and the probability that a replaced cache block is dirty and, if so, whether its home node is local or remote.

Among the three design options described in Section 4.1, only cluster size and the remote cache affect the cache miss profile. Therefore, we vary the number of processors in each cluster, from a single processor per cluster, to 2, 4, 8, and finally 16, i.e., a single cluster corresponding to a conventional shared-bus multiprocessor. In the alternative designs that employ remote caches, we keep constant the remote cache size per processor while altering the cluster size. This makes comparisons between various configurations as fair as possible. Parameters used in the simulation are shown in Table 4.6.

4.3.2 Cache Miss Profiles

Figures 4.2, 4.3, and 4.4 show the cache miss profiles of the three applications. The four and one half pairs of cache miss profiles correspond to architectures of cluster

[1] As mentioned earlier, the version we use has been optimized to eliminate false sharing and to facilitate bulk data transfer.

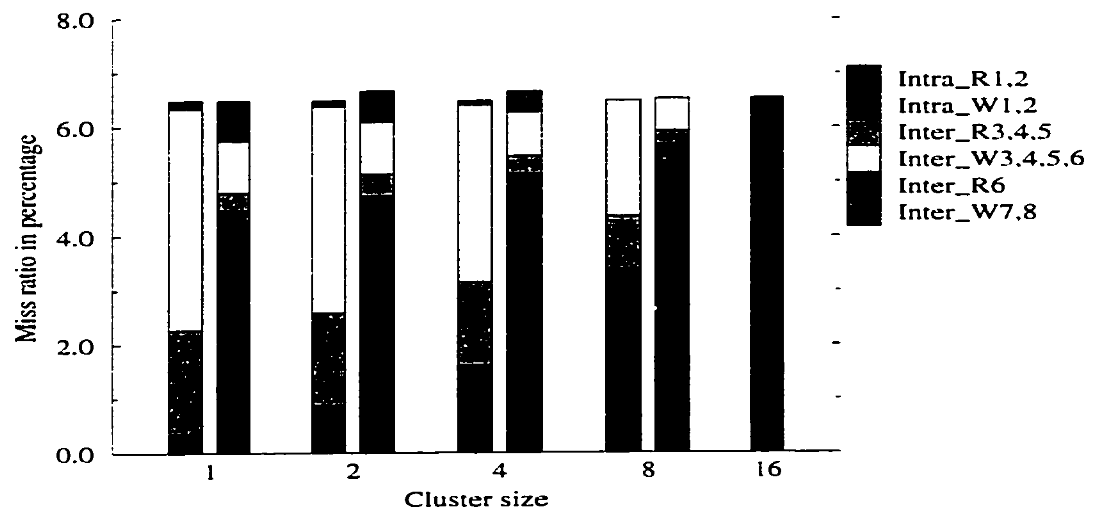


Figure 4.2: The cache miss profile of RADIX. The left (right) bar of each pair corresponds to an architecture not containing (containing) remote caches. $R1, R2$ and $W1, W2$ are intra-cluster misses. $R3, R4, R5$ and $W3, W4, W5, W6$ are inter-cluster misses that involve two clusters. $R6$ and $W7, W8$ are misses involving three clusters. Tables 4.1 and 4.2 provide a detailed explanation of each type of miss.

size 1, 2, 4, 8, and 16, respectively; the left (right) bar of each pair corresponds to an architecture not containing (containing) remote caches. Notice that the total cache miss ratio is independent of cluster size or the presence of remote caches. This number is governed by the configuration of the processor's private cache.

As expected, when cluster size increases, the number of intra-cluster shared reference misses rises. When the cluster size for RADIX and FFT doubles, the number of intra-cluster misses is doubles. A plausible explanation for this is that the data are uniformly distributed across cluster memories, and that the average number of processors sharing a piece of data is two in both applications. Hence, when cluster size halves, nearly half the intra-cluster misses have to cross the cluster boundary.

For RAYTRACE, the average number of processors sharing a piece of data is higher. As a result, when a cluster splits into two, a smaller portion of intra-cluster misses turn into inter-cluster misses. Intuitively, RAYTRACE has better cluster locality.

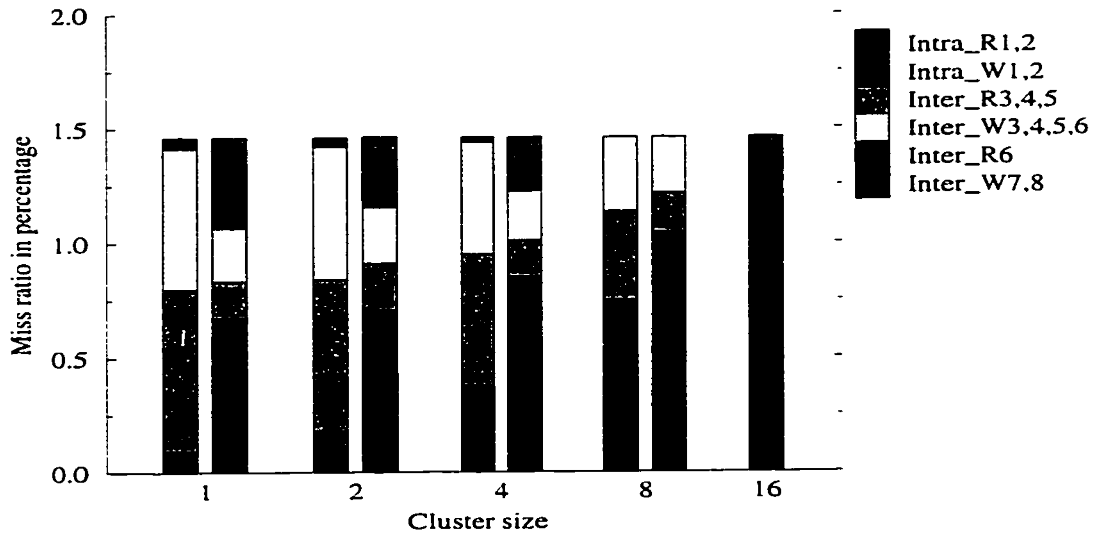


Figure 4.3: The cache miss profile of FFT. For a detailed explanation of this graph, refer to the caption of Figure 4.2.

Adding the remote cache significantly changes the cache miss profiles. The presence of the remote cache increases the retention of remote data. When a line whose home is in a remote cluster is replaced in an L2 cache, it can still exist in the remote cache because of the inclusion property. Thus, the remote cache can transform into intra-cluster misses some of the inter-cluster misses caused by conflict mapping or capacity limitation in the L2 caches. The extent of the reduction in inter-cluster misses is determined by the proportion of conflict and capacity misses in them. From [71], we know that when processor caches are of limited size, both RADIX and RAYTRACE encounter significant numbers of capacity misses. For these two applications, the remote cache is very effective in transforming the inter-cluster into intra-cluster misses. On the other hand, FFT has a smaller proportion of capacity misses. Therefore the remote cache is not as helpful.

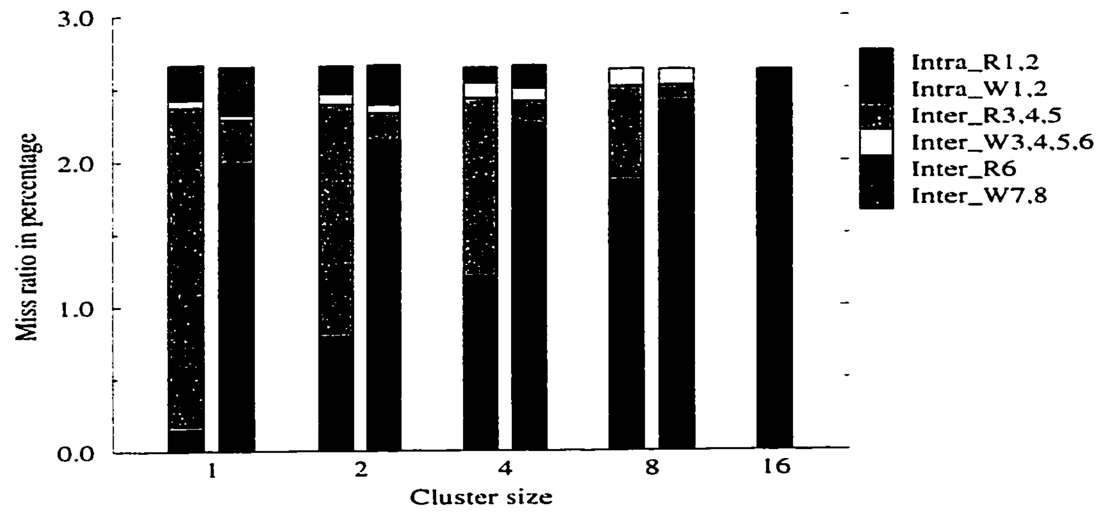


Figure 4.4: The cache miss profile of RAYTRACE. For a detailed explanation of this graph, refer to the caption of Figure 4.2.

When cluster size increases, the number of intra-cluster misses increases less significantly (compared to the architectures without remote caches). This is because the remote cache has already eliminated most of the inter-cluster capacity and conflict misses.

Finally, we note that the remote cache also brings a negative impact. Because a remote cache tends to retain data in remote clusters longer, it increases the number of 3-hop inter-cluster misses (misses involving three clusters). This can be observed most notably with FFT.

4.4 *Exercising the Model: Performance Evaluation*

Cache miss profiles alone are insufficient to assess performance. They are used as one of the inputs of the contention model introduced in Section 4.2.3. This section presents and discusses the analytical results. We start with a table displaying the service demands computed based on the cache miss profile of RADIX. The service demand table reveals where contention might exist. We then show how resource contention affect the cache miss latencies of this specific application. Finally, we present execution times for the three applications.

4.4.1 Average Service Demand

Tables 4.4 and 4.5 showed how each type of miss uses memory system resources. Based on these tables, the parameters given in Table 4.3, and the cache miss profile, we can compute for each resource an application's service demand (per processor). This is the average number of cycles a cache miss requires from a resource. Table 4.7 displays service demands for the RADIX application. As can be seen, of the 10 shared resources, the data bus (Dbus) and the protocol processor (PP) are the two resources with the heaviest demand. Hence, they are subject to the highest contention. This is true not only of RADIX, but also of FFT and RAYTRACE. Following discussion examines the effects of the three design options (cf. Section 4.1) on service demands, with a focus on the data bus and the protocol processor.

Cluster size. Recall from Section 4.3, when cluster size increases, the number of intra-cluster misses increases, and the number of inter-cluster misses decreases. The increase in the number of intra-cluster misses, however, does not imply an increase in service demand on the data bus. Independently of whether a miss is intra- or inter-cluster, the data have to transmit on the local cluster's bus. Intra-cluster misses use the bus only once. Some inter-cluster misses may use the bus twice if the data are owned by a cache of the remote cluster. Therefore, when the number of inter-cluster

Table 4.7: RADIX's average service demands. N_c is the cluster size.

N_c	Abus	Dbus	RC	Mem	BI(I)	BI(O)	NI(I)	NI(O)	Fwd	PP
	Average service demands (in CPU cycles)									
	No forwarding logic, No remote cache									
1	6.71	28.66	0.00	21.91	7.95	7.64	19.85	33.92	0.00	143.88
2	6.71	28.66	0.00	21.90	7.94	7.25	18.58	31.75	0.00	138.98
4	6.70	28.65	0.00	21.86	7.92	6.43	16.03	27.38	0.00	129.33
8	6.68	28.62	0.00	21.82	7.87	4.55	9.95	17.05	0.00	105.90
16	6.63	28.55	0.00	21.73	0.00	0.00	0.00	0.00	0.00	0.00
	With remote cache, No forwarding logic									
1	6.72	29.95	10.75	3.64	2.13	2.19	4.99	8.24	0.00	37.31
2	6.72	29.86	9.99	4.98	2.45	2.13	4.51	7.55	0.00	38.59
4	6.71	29.77	8.67	6.96	3.03	1.99	3.52	5.98	0.00	40.45
8	6.68	29.11	5.49	12.21	4.66	1.74	1.71	3.08	0.00	48.30
	With forwarding logic, No remote cache									
1	6.71	28.66	0.00	21.91	7.95	7.64	19.85	33.92	17.43	54.38
2	6.71	28.66	0.00	21.90	7.94	7.25	18.58	31.75	16.31	53.34
4	6.70	28.65	0.00	21.86	7.92	6.43	16.03	27.38	14.07	51.26
8	6.68	28.62	0.00	21.82	7.87	4.55	9.95	17.05	8.75	46.34
	With forwarding logic, With remote cache									
1	6.73	29.95	10.75	3.64	2.13	2.19	4.99	8.24	4.34	15.88
2	6.74	29.86	9.99	4.98	2.45	2.13	4.51	7.55	3.93	17.14
4	6.74	29.77	8.67	6.96	3.03	1.99	3.52	5.98	3.06	19.44
8	6.73	29.11	5.49	12.21	4.66	1.74	1.71	3.08	1.49	25.84

misses decreases, the overall demand on the data bus service center actually decrease. In our experiments, most remote requests were serviced by memory. Demand on the data bus remained practically unchanged, due to the dedicated link and memory controller between the protocol processor and memory.

For the protocol processor, a smaller inter-cluster miss ratio leads to a lower service demand. However, the change is limited. Note that service demands are computed on a per processor basis. If cluster size grows, demand per node rises monotonically.

Remote cache. When remote cache is employed, demand on the data bus barely changes, as discussed above. The slight increase is possibly due to the disturbance on the second level caches caused by the replacement in the remote cache. When a remote cache replaces a line, it has to issue an invalidation on the bus to purge data existing in the second-level caches in order to maintain the cache inclusion property.

With the remote cache, demand on the protocol processor drops most significantly, from 50-70%. Demands on other resources drops by various degrees. However, when cluster size increases, demand on the protocol processor surprisingly increases despite rising intra-cluster misses. It turns out that, with a bigger cluster size, a larger proportion of intra-cluster miss requests are homed at, hence served by, the local memory rather than the remote cache; they use the protocol processor as well.

Forwarding logic. Forwarding logic lessens the amount of time spent on protocol processor by performing some of its simple functions in hardware. As a result, forwarding logic is most useful to inter-cluster misses, which heavily rely on these processors. Unlike remote cache, forwarding logic does not change the cache miss distribution. Therefore, it also does not change the demands on any other resources. When both remote cache and forwarding logic are employed, the average service demand on the protocol processor drops even lower than that on the data bus.

4.4.2 Miss Latencies

We can use the contention model presented in Section 4.2.3 to estimate cache miss latencies. We compute the time a cache miss waits on each resource (cf. Equation (4.4) that has to be solved iteratively) and combine them with the service demands computed in the previous section. Figures 4.5 and 4.6 display the respective intra-cluster and inter-cluster cache miss latencies for RADIX.

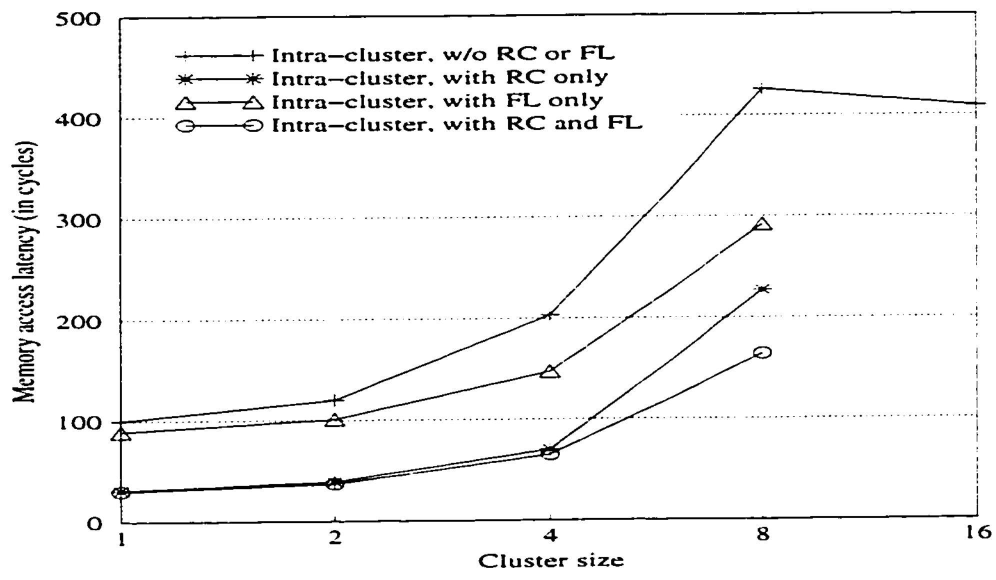


Figure 4.5: The average intra-cluster cache miss latencies for RADIX. RC and FL stand for remote cache and forwarding logic respectively.

Cluster size. Figures 4.5 and 4.6 show that both the intra- and inter-cluster miss latencies increase monotonically with cluster size. Because cluster size similarly affects both types of miss latencies, we only analyze the results for intra-cluster misses. When the cluster size is large, say 8, intra-cluster miss latencies are 4 to 10 times as long as those of respective architectures with one processor per node. The escalation of miss latencies is caused by contention on the data bus and on the protocol processor, because some of intra-cluster misses ($R2$ or $W2$) of the multi-cluster architectures

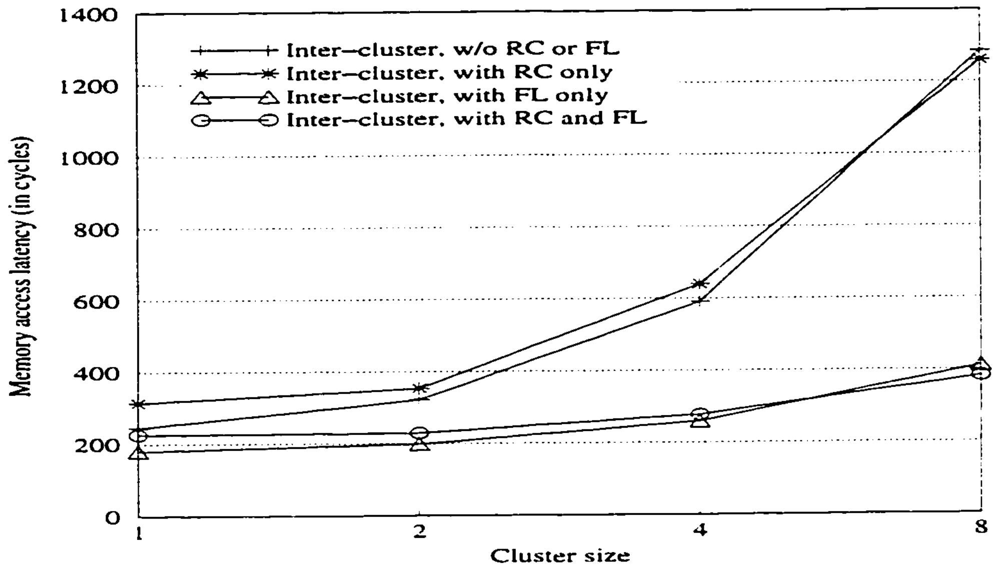


Figure 4.6: The average inter-cluster cache miss latencies for RADIX. RC and FL stand for remote cache and forwarding logic respectively.

also use the protocol processor. As we saw in the previous section, when cluster size increases, the service demand *per processor* on the data bus remains unchanged, and the service demand *(per processor)* on the protocol processor decreases slightly. However, the demands *per node* always increase with cluster size. So does the contention. For the single-bus system where a protocol processor is not needed, the major source of contention is the data bus; hence, the miss latency of the single-bus system is lower than the intra-cluster miss latency of the architecture that has 8 processors per node but no remote cache or forwarding logic. The miss latency of the single-bus system, however, is worse than the intra-cluster miss latencies of any other configuration.

Remote cache. Not only does the remote cache increase the proportion of intra-cluster misses, it also reduces the intra-cluster miss latency substantially (44-70%). This is because the intra-cluster misses serviced by the remote cache bypass the protocol processor; hence, they have a lower miss latency than those serviced by the local memory. However, the use of remote cache increases the inter-cluster cache miss latencies slightly in many cases. The reason is that the remote cache tends to retain modified data in a node longer, thus increase the number of 3-hop, inter-cluster misses (see Figure 4.2). As a result, the average inter-cluster miss latencies are higher than those of architectures without remote cache. At the cluster size of 8, however, this effect is offset by reduced contention on the protocol processor.

Forwarding logic. In contrast to remote cache, forwarding logic has little influence on intra-cluster miss latencies. In fact, no intra-cluster misses use the forwarding logic. Nevertheless, forwarding logic still reduces intra-cluster miss latencies by mitigating contention on the protocol processor. For inter-cluster misses, many transactions that formerly used the protocol processors can take advantage of the forwarding logic. Therefore, even at small cluster sizes (1 and 2) when contention is low, forwarding logic decreases inter-cluster miss latencies by 26-38%. At larger cluster sizes (4 and 8) when contention is high, the reduction is 50-70%.

4.4.3 Execution time

We apply Equation (4.1) to assess overall performance as measured by execution times. Results for the three applications are shown in Figures 4.7, 4.8, and 4.9. Execution times are normalized to the execution of a single-cluster system.

A parameter critical to the normalized execution time is the number of instructions executed between cache misses (I in Equation (4.3)). It is the reciprocal of the cache miss issuing rate. The smaller I is, the higher the contention. For our selected benchmarks, I is 43 (RADIX), 184 (FFT), and 145 (RAYTRACE). Consequently, RADIX suffers the highest contention in the single-cluster configuration and benefits the most from the cluster architectures. For the cluster architectures that do not have remote caches or forwarding logic, the performance of RADIX is about 36% and 22% faster than the single-bus system, reached at cluster sizes of 1 and 2 respectively. Since the proportion of inter-cluster misses in large clusters is still quite high, contention on

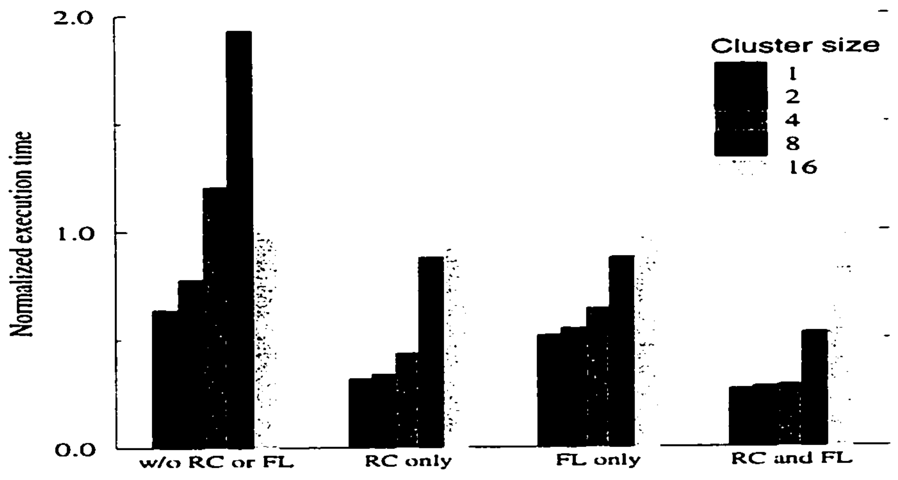


Figure 4.7: Normalized execution times for RADIX.

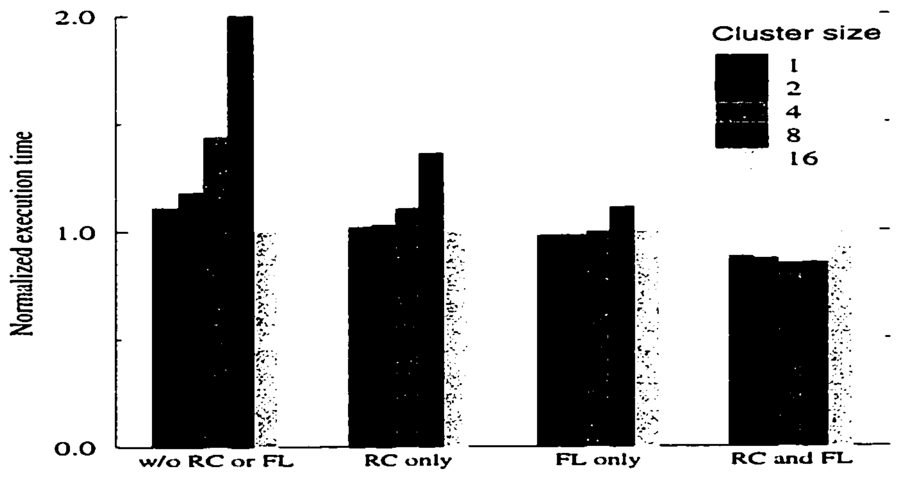


Figure 4.8: Normalized execution times for FFT.

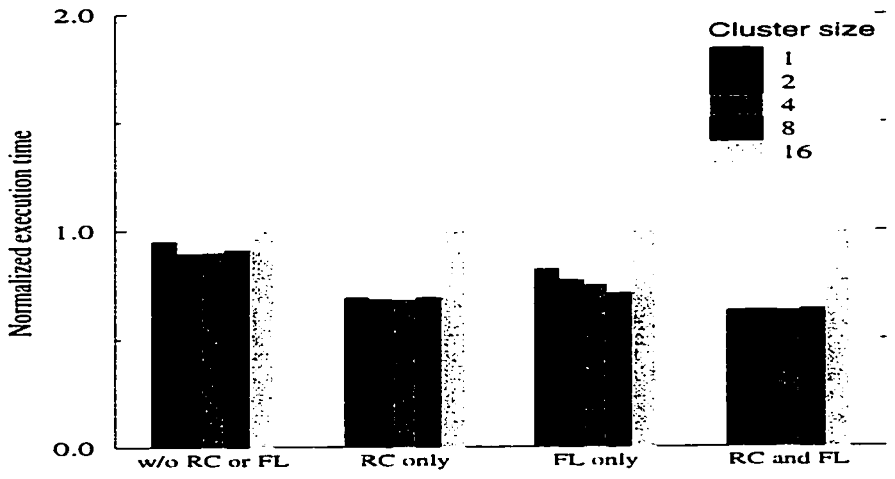


Figure 4.9: Normalized execution times for RAYTRACE.

the protocol processor takes its toll. This results in performance much worse than the single-bus system. The low cache miss issuing rate for FFT means the bus contention in the single cluster is not high enough to make the cluster architectures useful. For RAYTRACE, the improvement of cluster architectures over the single-bus system is moderate. A remarkable characteristic of this application is its good cluster locality: when cluster size increases, intra-cluster misses increase rapidly (see Figure 4.4) to cause the performance improvement.

When the remote cache is added, intra-cluster misses increase from the original 5-50% to 70-85% for RADIX. Its intra-cluster miss latency is lessened by 50% or more. As a result, the remote cache enhances the overall performance of cluster architectures by 50-60%. For FFT, the benefit of the remote cache is 8% at cluster size 1 and 36% at cluster size 8. The diminished value of the remote cache can be explained by the fact that FFT has a lower miss ratio (hence a smaller memory overhead) and a large portion of inter-cluster coherence misses that cannot be transformed by the remote cache. For RAYTRACE, the corresponding improvement is between 22-27%. Although at cluster size 1, the remote cache increase the number of intra-cluster misses significantly, its benefit is smaller than it is for RADIX because RAYTRACE has a lower miss ratio. At large cluster sizes, the remote cache does not help RAYTRACE as much as it does the other two applications because RAYTRACE has good cluster locality and the cluster architectures without remote cache are quite well balanced for this application.

Forwarding logic's main function is to reduce inter-cluster contention-free miss latency as well as contention on the protocol processor. It is most useful for architectures of large cluster sizes where contention is potentially high. For example, with forwarding logic, the normalized execution time of RADIX drops more than 50% at cluster size 8, but only 20% at cluster size 1. The same is true for FFT and RAYTRACE, to different degrees. Finally, since forwarding logic does not change the cache miss profile, it is often less effective than remote cache unless the application (e.g., FFT) has a large portion of coherence misses, for which the remote cache is useless.

4.5 Validation

Figures 4.10, 4.11, and 4.12 compare analytical results with simulation results. The analytical model predicts execution time trends quite well for RADIX and FFT. The errors range from 5% to 30%. The model yields optimistic results for two reasons. The first is that the algorithms of the two applications alternate between computational phases that have distinctive needs for communication. In one phase, the program performs communication intensive operations. In another phase, it performs CPU-intensive operations. When we compute mean values over the entire application, the model underestimates the contention effect for communication intensive phases. The second reason for optimistic results is that the MVA technique is incapable of

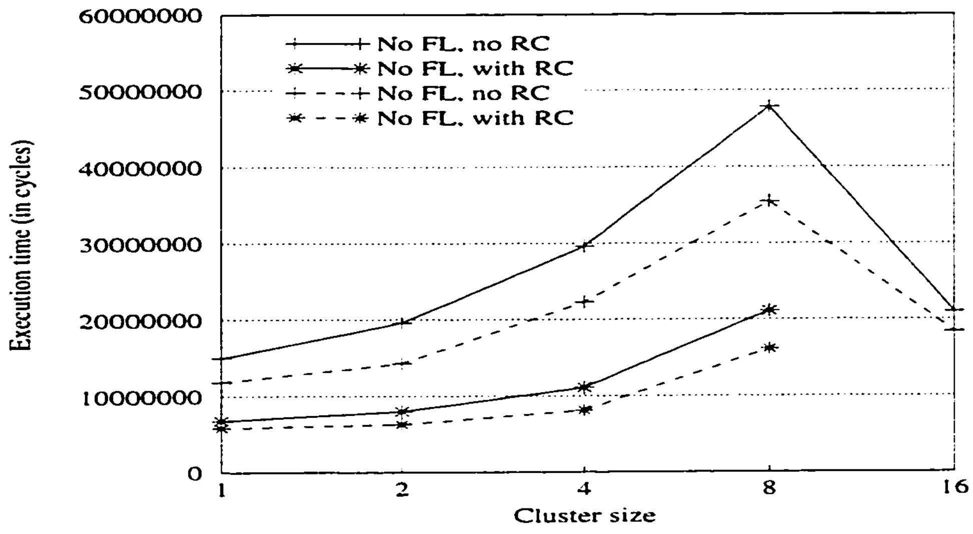


Figure 4.10: Comparison of analytical and simulation results for RADIX. Solid lines are simulation results. Dashed lines are the model's outputs.

handling synchronization overhead. This component is completely overlooked in the analytical model.

The MVA technique is also a poor predictor of the behavior of lock-intensive applications, such as RAYTRACE. We have selected RAYTRACE because it has no barrier synchronization overhead. Its overall synchronization overhead is also extremely low under an ideal (PRAM model) memory system [71]. However, as it turns out, this application is lock-rich. Under a realistic memory system, the memory overhead due to lock synchronization is quite large. As a result, the analytical model is not as accurate for RAYTRACE.

4.6 Related Work

The behaviors of service centers contained in our model generally satisfy the two assumptions required by the MVA technique, i.e., routing homogeneity and service

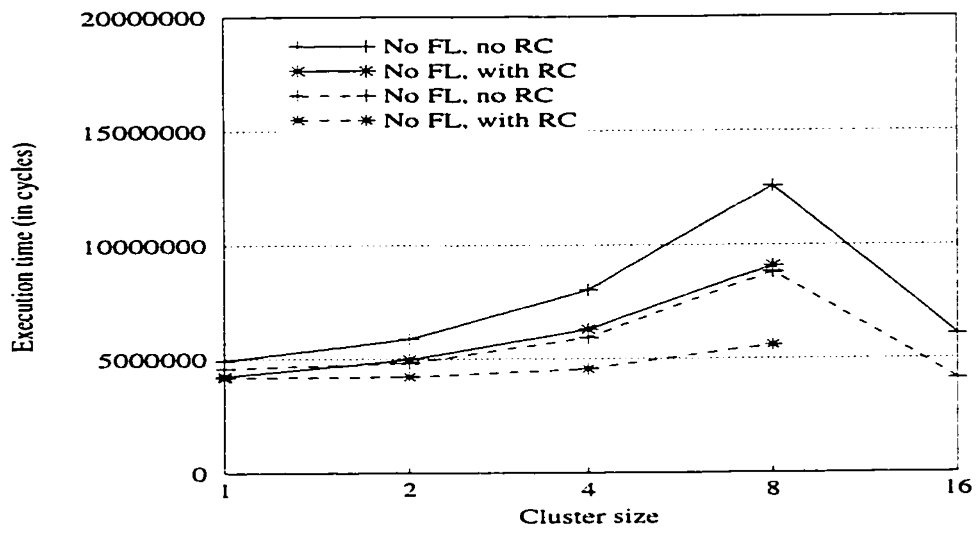


Figure 4.11: Comparison of analytical and simulation results for FFT. Solid lines are simulation results. Dashed lines are the model's outputs.

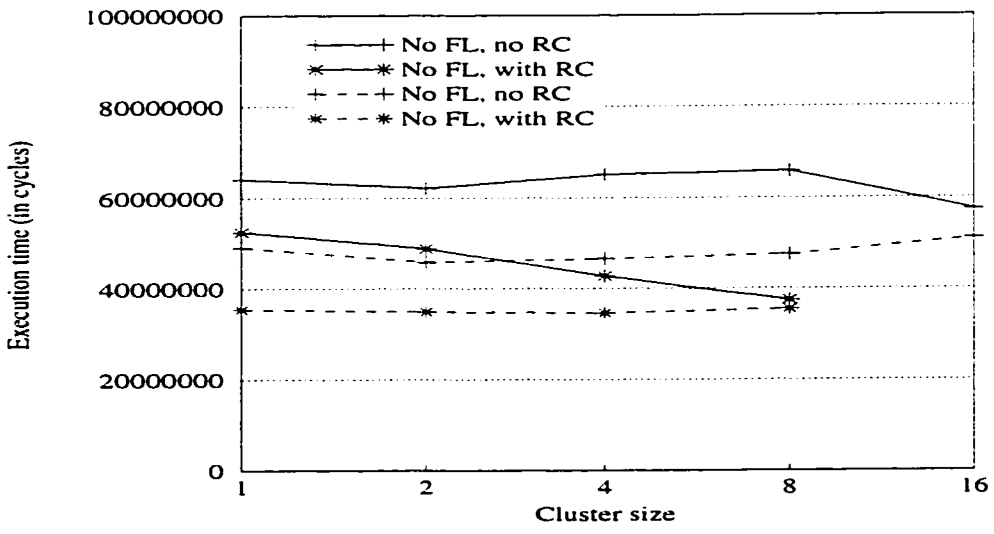


Figure 4.12: Comparison of analytical and simulation results for RAYTRACE. Solid lines are simulation results. Dashed lines are the model's outputs.

time homogeneity [40]. Vernon et al. have shown that the MVA technique applied to shared-bus multiprocessor systems is of remarkable accuracy [67]. They applied the same technique to studying the performance of purely bus-based hierarchical multiprocessors [66]. Our model and theirs share common elements because of the similarity in architectures. Torrellas, Hennessy and Weil also developed analytical models to investigate the impact of several architectural and application parameters in shared-memory processors based on DASH [63]. The main differences between their model and ours are that they used an open queuing network and their inter-cluster interface was hardwired [41]. They found that contention for the bus dominated contention on the cluster interface[2]. We show that this conclusion is no longer true in all cases when the design shifts from a hardwired controller to a protocol processor.

Holt et al. used simulation to study the effects of occupancy of the protocol processor and latency of the network on a Flash-like architecture with one processor per cluster [30]. They showed that the time consumed by the protocol processor had an impact on performance, especially with a fast network. However, it is unclear whether contention causes considerable delay in the protocol processor. Cluster-based architectures in which inter-cluster communication is provided by a (hierarchy of) bus and a shared cache(s) have been studied also via simulation by Nayfeh et al. [45, 46] and by Anderson [5]. Conclusions on the viability of clusters are mixed depending on the frequency of memory requests and their locality.

As several processors share a protocol processor, the latter is likely to become the performance bottleneck. Some researchers have looked at the possibility of using compute processors as protocol processors. The idea is that the compute processor may as well be used to resolve a cache miss that stalls a thread's execution. Karlsson and Stenstrom simulated such a scheme for a multiprocessor that used an ATM network to link clusters of bus-based multiprocessors [33]. Intra-cluster coherence was

[2] The number of processors per cluster was fixed at four.

maintained by cache snoopy hardware, while inter-cluster coherence was maintained at the page level by distributed virtual shared-memory software. With a network latency of 100us, the chance of finding a compute processor idle was quite high (52%-92% for 4 processors/cluster), even when a simple round-robin policy was used for scheduling the task of protocol processing. The bottleneck was in the ATM interface, and the software had to be tailored to the application for significant speed-ups to occur. The authors concluded that in the context of such an architecture and accompanying software, the presence of a separate protocol processor was not necessary.

Our study assumes that the interconnection network is of sufficient bandwidth and uses a contention-free network latency in the model. However, a network model can easily be incorporated in the evaluation if contention is not negligible. There exists a rich set of analytical models for various network topologies [1, 12]. These models usually take the request rate as an input parameter and compute the network latency taking contention into effect. This latency could be used in our model instead of the contention-free network latency to achieve a more accurate response time for inter-cluster misses. The response time would then in turn affect the request rate issued to the network. The two models would have to iterate until they converge.

4.7 Summary

In this chapter, we applied the Mean Value Analysis technique to assess the performance of cluster-based architectures. We characterized the service demand for shared resources by examining in detail the sub-requests involved in the resolution of cache misses. In addition to the overall system parameters and the service demands on shared resources, the analytical model needs parameters pertinent to applications, i.e., cache miss profiles. These parameters were obtained via trace-driven simulation for three applications.

We compared the performance of cluster-based architectures of various cluster sizes with and without remote cache or forwarding logic. Our analytical and simulation results shed light on three different aspects of cluster-architectures: the applications' cache miss profile, the service demand, and the overall performance (normalized to the single-bus system). We summarize these results as follows:

- Many existing parallel programs, and our example programs, are written for machines with a flat architectural model. When these applications run on the baseline cluster architectures, the portion of intra-cluster misses increases 50-100% as cluster size doubles. When large remote caches are present, they retain, as expected, much of the remote data in the cluster. In that case, increasing cluster size appears less effective in increasing the portion of intra-cluster misses (compared to the baseline architectures).
- In the cluster-based architectures that employ protocol processors and maintain cache coherence in software, the protocol processor and the data bus are in the greatest service demand. The service demand (per processor) on the data bus remains approximately the same independent of cluster size. Service demand on the protocol processor decreases as cluster size increases. However, the demands per node always increase with cluster size. The remote cache significantly reduces service demand on the protocol processor significantly, but has little impact on service demand on the data bus. This is also the case for forwarding logic.
- Given the size of the processor's private cache, the network latency, and other architectural parameters, the performance of cluster architectures is governed by (1) the proportion of intra-cluster and inter-cluster misses, and (2) the contention on shared resources. The former is affected by cluster size and the presence of remote cache, while the latter is determined by the miss issuing

rate and service demands. Service demands are further determined by cluster size and the presence of remote cache or forwarding logic.

- The performance of the baseline architectures is mixed. For two of the three applications, cluster architectures perform 5-36% better than the single-cluster system. However, increasing cluster size often harm performance due to high protocol processing overhead unless the application has good cluster locality. The remote cache improves performance by 8%-60%; it is most effective for applications suffering primarily from capacity and conflict misses. The forwarding logic improves performance by 12-54%; it is more important than the remote cache for applications that have many coherence misses. When both remote cache and forwarding logic are used, the cluster architectures consistently outperform the single-bus system.

- When validated with simulation results, the analytical model captures the performance trend of cluster architectures reasonably well, with errors ranging from 5% to 30%. However, the MVA-based model is not suitable for applications that have high synchronization overhead.

Chapter 5

COMMUNICATION PRIMITIVES IN CLUSTER ARCHITECTURES

The previous two chapters studied the use and performance of communication primitives in a simplified "cluster" architecture with a single processor per node. They also used an MVA-based analytical model to explore the issue of contention over highly demanded resources in general cluster environments with multiple processors per node. The contention discussion excluded the use of communication primitives: the analytical model was founded on a set of assumptions used in the theory of stochastic processes (such as that jobs are stochastically independent), while the effectiveness of communication primitives depends upon the exact time requests are issued or processed.

In this chapter, we study communication primitives in the context of general cluster architectures. Our evaluation is via trace-driven simulation. We established in Chapter 4 that in the absence of communication primitives, contention on shared resources, especially the protocol processor, is quite high in the baseline architectures with a cluster size of 4 or 8. The use of communication primitives may have positive as well as negative effects on shared resource contention. On one hand, communication primitives consume fewer protocol processor cycles per cache line by reducing the need for processing many small cacheline-based requests. Additionally, communication primitives can be issued ahead of time, for example, in computation phases when the communication processor would otherwise be idle. In this situation, communication primitives better utilize the protocol processor and other resources. On the other hand, communication primitives may be superfluous. By the time requests are processed, they may no longer be useful. In this case, although the protocol processor is kept busier, its actual utility decreases.

In general, how would communication primitives perform in cluster environments? Would they ease or exacerbate the contention on the protocol processor? Would communication primitives hide memory latency as effectively as they did in the uniprocessor node implementation? How much would the additional hardware (remote cache or forwarding logic) improve the performance of the software scheme enhanced by communication primitives? We answer these questions in this chapter.

The rest of the chapter is organized as follows. Section 5.1 examines implications of adding remote cache and forwarding logic to the cache coherence protocols and issues in implementing the communication primitives in the cluster architectures. Section 5.2 briefly describes the experimental methodology, similar to that used in Chapter 3. Section 5.3 presents simulation results. Finally, we summarize the findings in Section 5.4.

5.1 Implementation Issues

This section describes the necessary changes on the cache cache protocols when adding remote caches and forwarding logic to the baseline architecture. It also examines the complications of incorporating multiprocessor nodes on the communication primitive protocols.

5.1.1 Remote Cache and Forwarding Logic

The remote cache stores data homed in remote memories, thus reducing inter-cluster miss requests. The forwarding logic reduces the load on the protocol processor and speeds up the processing of requests or replies that do not need coherence directory information. Despite significant functional differences, these two hardware units share a common characteristic, namely the address of a request or reply determines whether the remote cache or the forwarding logic participates in service. If the address of the request is homed in *local* memory and no local caches respond to the request, it is inserted into the bus interface's input queue and processed by the protocol processor on that node. If the address of the request is homed in a *remote* memory, the remote cache performs a cache lookup in parallel with the local caches. In case the request cannot be satisfied within the node, it also enters the BI's input queue and routes through the forwarding logic.

Using the remote cache introduces a new layer of memory hierarchy for remote data. It requires a new state for the remote cache. If a cache line is MODIFIED in a local cache, the corresponding line in the remote cache is set to the new state GONE, meaning the data are not valid. If the remote cache chooses to replace that line, however, it needs to issue a writeback/invalidation request on the local bus to enforce the inclusion property.

The default write policy between the second-level cache and the remote cache is writeback. With the remote cache, the replacement of remote data in the local caches is simplified. If the replaced line has been modified, it is written back to the remote cache. If the replaced line is clean, no action is necessary. Note that the replacement of a cache line homed in the local memory does not change. That is, if the replaced line is dirty, the processor issues a writeback request, which is inserted into the BI's input queue. If the replaced line is clean and the directory keeps an accurate sharing node list, the requesting processor issues a "dropping a line" notification. Whether the notification enters the communication processor depends on whether the line is present only in the replacing cache or not.

5.1.2 Data Placement for Communication Primitives

Under the NUMA model, data can be duplicated only in caches. In order to minimize cache pollution, we let the communication primitives place data in the cache at the level farthest away from the processor. Hence, for systems without remote caches, the communication primitives place the data in the requesting processor's private cache. For systems with shared remote caches, the primitives place data in the remote cache of the requesting node. Recall that the remote cache is used to store remote data only. Thus, if the data requested by a processor residing in local memory, they cannot be pushed into the remote cache of that node, they remain in local memory. When the requesting processor later accesses the data, local memory supplies it at the cost of an intra-cluster miss latency.

5.1.3 Review of Communication Primitive Protocols

We mentioned in Chapter 3 that communication primitives are issued before the status of local caches is validated. As a result, primitives could be issued for data already in the requesting processor's cache. In the simplified cluster architecture, the communication processor can deduce the cache status of individual processors based on directory information, and thereby filter out superfluous data transfer. In a general cluster architecture, this is not possible because the directory keeps only the summary states of several caches in a cluster. The directory's ignorance of the exact individual cache states makes it difficult for the home node to perform intra-cluster operations intelligently.

Suppose a processor initiates a primitive to acquire a block that is already in the node of the requesting processor. The home node cannot tell whether the data are in the cache of the requesting processor. In such a situation, the home node has two options: (1) always transfer the data, or (2) do nothing. If the request is to read share a block via a *get*, for example, choosing the first option implys a higher protocol overhead and a waste of network bandwidth; choosing the second option possibly results in a future cache miss that will likely be satisfied by a local cache. On the other hand, if a processor requests a block of data with ownership via a *getex*, for example, option 2 causes the requesting processor to suffer an inter-cluster miss later. To avoid this, we let the home node invalidate all sharing nodes (including the node of the requesting processor) and transfer the data into the cache of the requesting processor.

In general, we require that the communication primitive protocols execute requests that can eliminate future inter-cluster misses. The protocols can avoid servicing requests that will result in future intra-cluster misses, because resolving those misses on demand incurs a short intra-cluster miss latency.

5.1.4 Buffers in the Communication Processor

Buffers in the bus and network interfaces are crucial to accommodate a steady flow of pipelining operations between compute processors and the protocol processor. If a protocol processor wants to issue a request on the shared bus and that bus is busy, it can store the request in the output queue of the bus interface, if not full, and process another request. In the absence of communication primitives, the maximum number of messages that can accumulate in the queues is bounded by the number of outstanding requests per processor, the cluster size, and the number of processors in the system. We can estimate upper bounds for the buffer sizes by blocking the channel through which buffered messages are drained.

We illustrate this method by determining the maximum size for the output queue of the network interface (NI). Assume that each processor has p outstanding memory operations and that the total number of processors in the system is N. Suppose that the output channel of a node is blocked due to network congestion; however, assume the input channel is still open, and all other nodes forward their requests to this node. The node receives at most $p \times N$ requests. If the request is a read, the node needs to supply the data. If the request is a write, in the worst case the node needs to send out $N/N_c - 1$ invalidations, where N_c is the cluster size. In addition, the node may need to store one additional data message for each pending miss request issued from a local processor in case a dirty line has been replaced. Therefore, the maximum buffer size needed for NI's output queue is $pN \times max((N/N_c - 1)S_{inv}, S_{data}) + pN_cS_{data}$, where S_{inv} and S_{data} are the sizes for invalidation and data messages respectively. For the input queue of the network interface, in the worst case the node may receive $p \times N$ replaced dirty lines. Hence, the maximum buffer size for NI's input queue is $pN \times max((N/N_c - 1)S_{ack}, S_{data}) + pNS_{data}$. In the same way, we can estimate the maximum sizes for BI's output queue $[pN_cS_{data} + p(N - N_c)S_{req}]$ and input queue $[pN_cS_{req} + pN_cS_{data} + p(N - N_c)S_{data} = pN_cS_{req} + pNS_{data}]$ respectively.

When communication primitives are added to the system, we could use the same analysis by increasing p since each primitive is a non-blocking operation, and each may involve hundreds of cache lines.

It may not be cost effective to implement a buffer size that can accommodate the volume of messages accumulated in the worst case. However if, due to the limited buffer space, one or more message queues are filled up, the whole system may not function efficiently: it is deadlock-prone, and messages may have to be dropped to make room for higher priority messages so that the system can move forward. To avoid this, we implemented a simple flow control mechanism. When the number of messages in a queue reaches a threshold, the network interface raises a warning flag. Any communication primitives that heavily use the resource are temporarily blocked to let other requests that require different resources be processed.

5.2 Experimental Methodology

The methodology we adopt in this chapter is the same as that used in Chapter 3, namely using Mint to simulate the memory sub-system. We study the performance of communication primitives by comparing three sets of simulation results (as in Case 2, 3, and 4 in Section 3.3):

1. A hardware-based cache coherent system in which the coherence protocol is hardwired into circuitry.

2. A software implementation that uses the communication processor, with its embedded protocol engine performing cache coherency operations in software.

3. Another software implementation, with the additional communication primitives protocols available. Applications are annotated with these primitives.

For each simulation environment, we evaluate the effect of cluster size as well as the importance of the remote cache. In addition, for the two software implementations, we examine the impact of adding forwarding logic in the communication processor.

In simulations, we use the same latency parameters as those found in Tables 3.3 and 4.3. We fix the size of second-level caches (64KB per processor), the size of the remote cache (1 MB x cluster size), and other cache parameters as presented in Table 4.6. At cluster size 1, the machines without remote caches or forwarding logic are comparable to the machines (studied in Chapter 3) that have smaller processor caches (32KB). The machines with remote caches but no forwarding logic are comparable to those that have larger processor caches (256KB).

The applications we use in this chapter are also the same as those in Chapter 3, i.e., FFT, LU, and RADIX, as they have already been annotated with communication primitives. We mentioned earlier that optimizations depend upon the cache configuration. Optimizations applied to systems with small caches are more conservative than those applied to systems with large caches. Correspondingly, in this chapter we apply aggressive optimizations (the version used for 256KB caches in Chapter 3) to the architectures with remote caches; we apply conservative optimizations (the version used for 32KB caches in Chapter 3) to the architectures without remote caches.

5.3 Performance Results

This section presents performance results for the three applications. For each application, we present ten sets of results: two hardware implementations (with the choices of adding the remote cache), four software implementations (with the choices of adding the remote cache, the forwarding logic, or both), and four corresponding enhanced software implementations. In each implementation, we vary cluster size. To make a complete comparison, we place in one graph the ten set results showing execution time vs. cluster size. In the course of our analysis, we supply additional graphs, which contain subsets of results, to improve readability and highlight major points.

In the graphs, hardware implementations are drawn in black lines with circles; software implementations appear in light gray lines with squares; and optimized software implementations are in dark gray and marked with plusses. The types of lines indicate what hardware has been employed. Solid lines indicate that neither the remote cache nor the forwarding logic has been used in the implementations. Dotted lines mean that the remote cache has been employed. Dashed lines imply that the forwarding logic is utilized. Finally, dotted-dashed lines indicate that both remote cache and forwarding logic are present.

5.3.1 FFT

The results of FFT are shown in Figures 5.1, 5.2, and 5.3. Figures 5.2 and 5.3 are subsets of Figure 5.1.

Hardware implementations. When the directory coherence protocol is implemented in hardware, we observe that all cluster architectures outperform the single-bus system, which corresponds to the system with a cluster of size 16. (This is the right-most point in Figure 5.2.) However, performance decreases slightly as cluster size increases. This is because the application has not been optimized for the cluster architecture: each processor works on independent data sets with an all-to-all communication pattern. When cluster size increases, the number of intra-cluster misses increases (cf. Figure 4.3 in Chapter 4). Contention over shared resources is also intensified. The latter slightly dominates the performance, resulting in longer execution times. Nevertheless, the performance of the 8-processor node is still comparable to

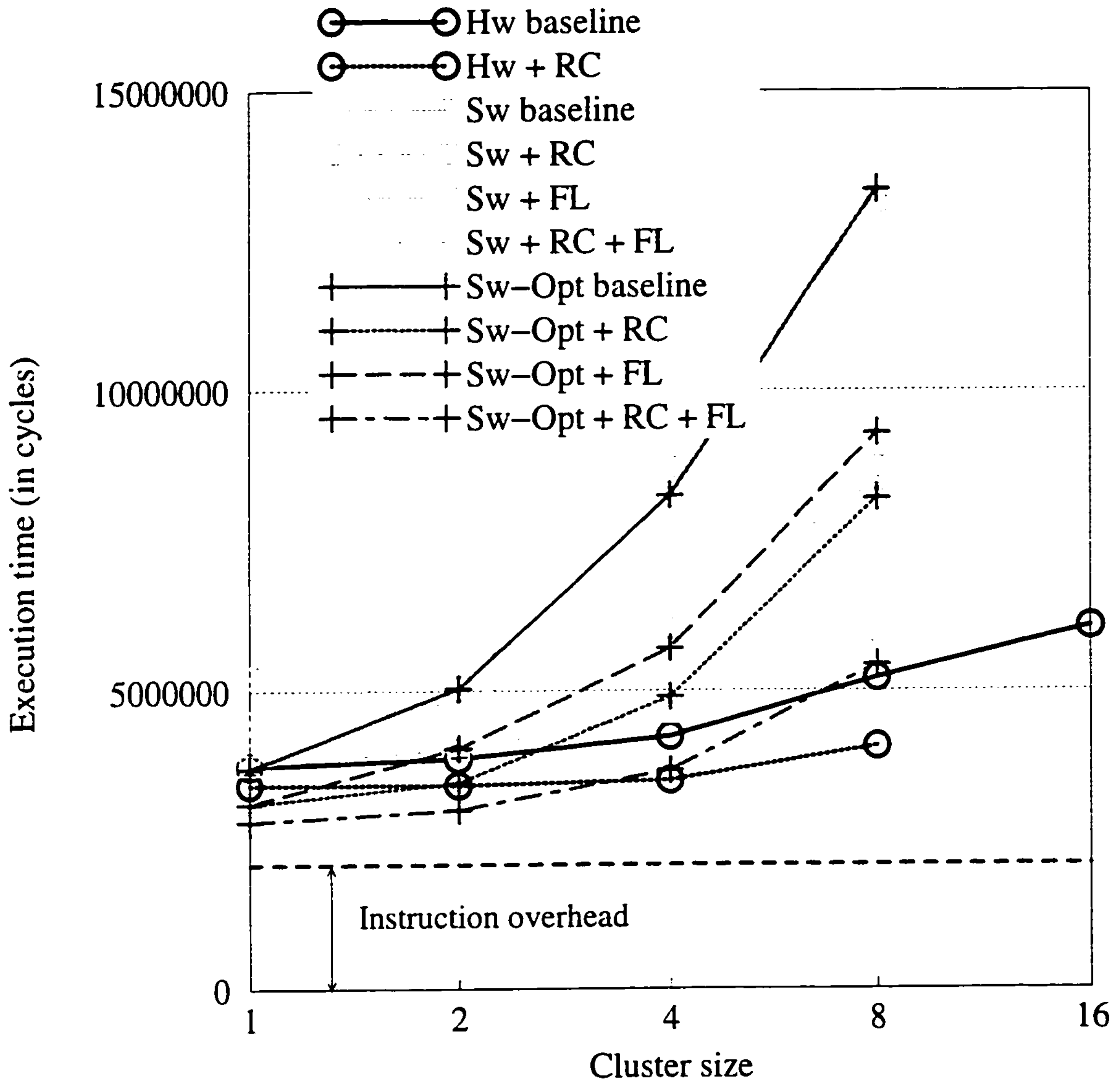


Figure 5.1: Execution times for FFT.

that of smaller cluster sizes. However, the amount of hardware needed to connect the cluster nodes is reduced by half when cluster size doubles.

When the remote cache is added, the execution time is reduced by 10-17% because most inter-cluster capacity or conflict misses are converted into intra-cluster misses. However, there remains a significant amount of memory overhead (the difference between the straight dashed line and the dotted black line), mainly due to inter-cluster coherence misses, which is the real communication cost that cannot be eliminated by the remote cache.

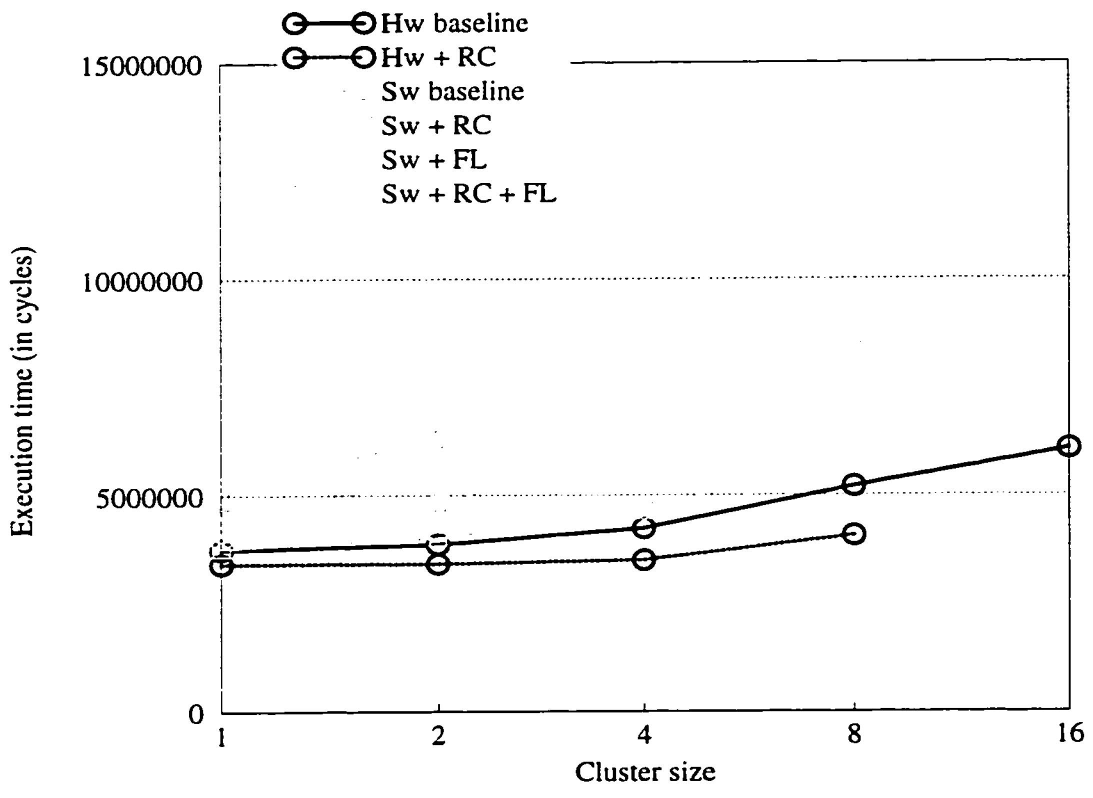


Figure 5.2: Execution times for FFT (hardware and software without communication primitive implementations).

Software implementations. Compared to the hardware implementation, the software implementation in the base architecture increases execution time by 33% at cluster size of 1 and by 150% at cluster size 8 (cf. Figure 5.2, also shown in Figure 3.1 in Chapter 3). The performance degradation due to the increasing cluster size is much more severe in the software implementation than in the hardware implementation. This is caused by contention on the protocol processor considering that the other parts of the two implementations are equivalent. With quad- or octa-nodes, the protocol processor's utilization is always above 80%.

The use of remote cache reduces memory overhead by 19-42% since it retains most of the capacity and conflict misses within the cluster. The use of forwarding logic reduces memory overhead by 30-45% for it reduces the contention-free inter-cluster miss latency and the load on the protocol processor. For FFT, forwarding logic is more effective than remote cache (also predicted by the model in Figure 4.8) because the majority of the misses are coherence misses. This type of miss can take advantage of the simple and fast message-passing mechanism provided by the forwarding logic, but it cannot use the remote cache.

When both remote cache and forwarding logic are applied, the performance of the software implementations is very close to that of the hardware implementations. This suggests that the software implementations with appropriate hardware support are practical alternatives to pure hardware implementations.

Enhancement of communication primitives. We now examine the performance gains of communication primitives. To be fair, we compare the two software implementations, i.e., with and without communication primitives, under equivalent hardware. In the figures, the type of line indicates what specific hardware, remote cache or forwarding logic, has been employed. This translates into comparing lines of the same kind, i.e., measuring the difference between a solid line and another solid line or that between a dotted line and another dotted line, etc.

In the baseline architecture, communication primitives improve the performance

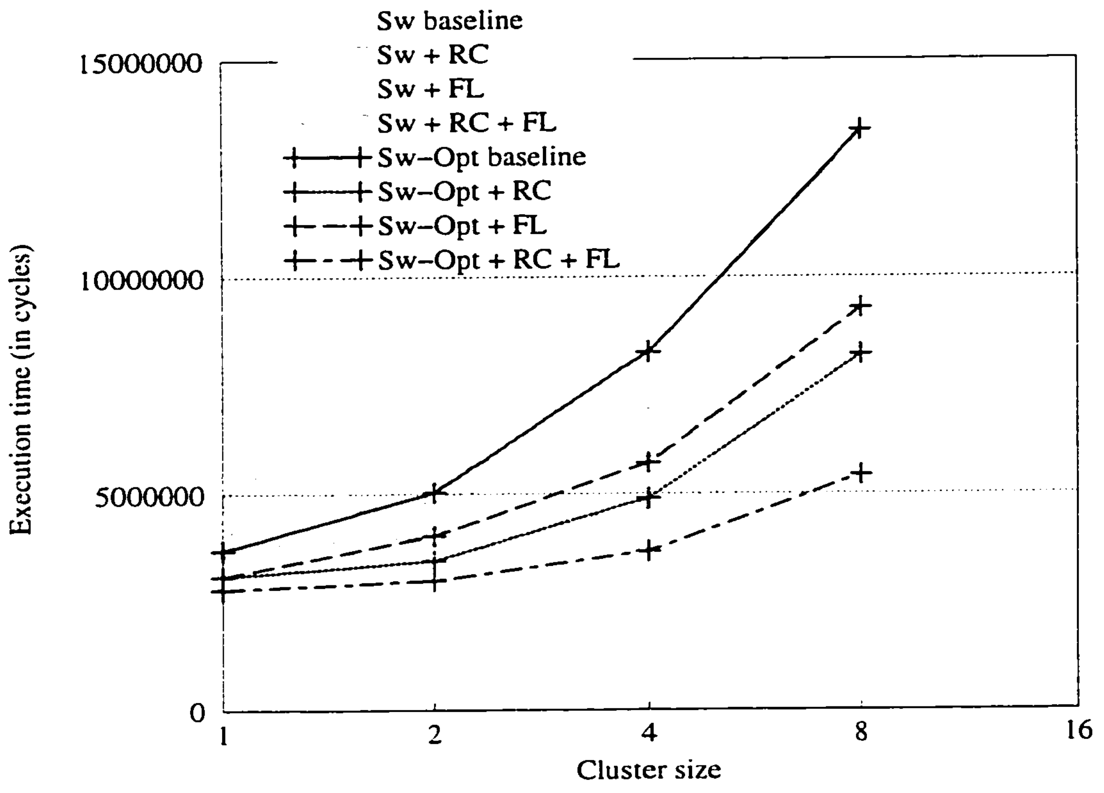


Figure 5.3: Execution times for FFT (software and enhanced software implementations).

significantly at the cluster size 1 (by 30%). When cluster size increases, the benefit of the communication primitives diminishes. They eventually become harmful at cluster size of 8. This is because, as cluster size grows, more cache misses can be satisfied in clusters. Primitives attempting to acquire the intra-cluster data will be canceled on-the-fly by the protocol processor. However they still consume a certain amount of the protocol processor cycles. Therefore, the overall memory overhead improvements by the primitives decrease with the increasing cluster size.

When remote cache is used, communication primitives improve performance by approximately 35% (cf. Figure 5.3, the difference between the dotted lines). In comparison, when forwarding logic is employed, the corresponding improvement is about 22% (the difference between the dashed lines). The main reason that the communication primitives perform better with the remote cache is that the optimizations applied to machines with the remote cache are more aggressive than those applied to machines without it. Without the remote cache, only the consumer-initiated primitives (*get* and *getex*) are issued immediately before the data are needed because of the modest cache sizes. With the remote cache, as soon as a (producer) processor finishes computing partial results, it disseminates the data to its (consumer) processors. Also, during the computation phase, each processor prefetches the data it uses in the next phase. As a result, communication largely overlaps with computation.

Finally, we point out that the performance of the enhanced software scheme is not bounded by the best fixed hardware implementations. For FFT, the enhanced software implementations with the assistance of remote cache or forwarding logic or both approximate or out-perform the corresponding hardware implementations at cluster sizes 1, 2, and 4.

5.3.2 LU

The results for LU decomposition are presented in Figure 5.4. For a detailed explanation of the graphs, please refer to Section 5.3. In addition, the two lines at the bottom show the application's instruction overhead and synchronization overhead under the PRAM model respectively. The significant synchronization overhead in the PRAM model reveals a very unbalanced workload.

Hardware implementations. LU decomposition suffers severely from conflict misses when cache size is small, which is the case for machines without remote caches. The data that are frequently replaced are shared by the processors in charge of data in the same row or column. When cluster size increases, intra-cluster data sharing transforms a part of the inter-cluster conflict misses into intra-cluster misses. Besides,

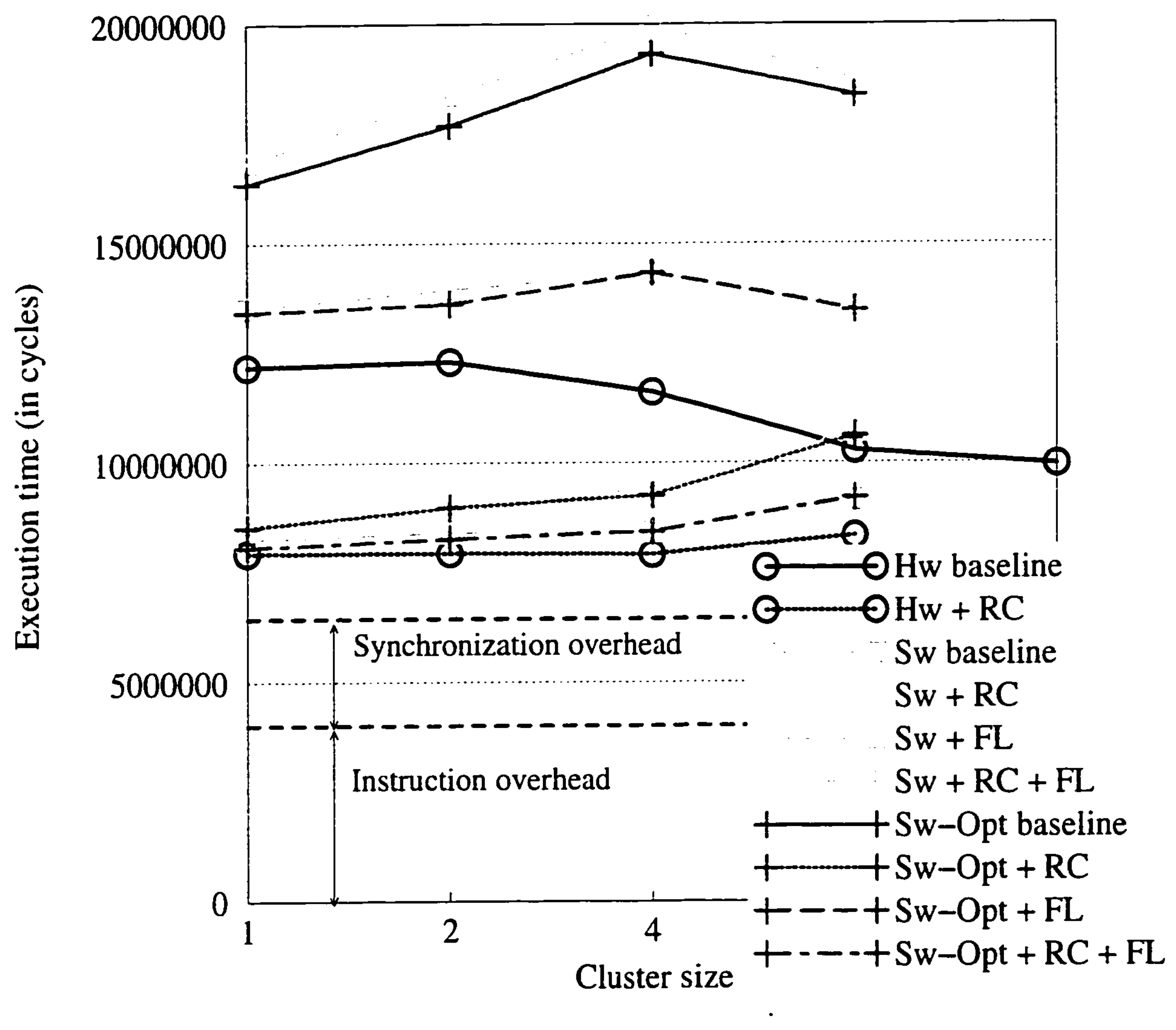


Figure 5.4: Execution times for LU. The dashed line at the bottom represents the processor busy time and the line above it is the synchronization overhead under the PRAM model.

the number of local memory accesses increases with cluster size. As a result, clustering benefits the application despite the increasing resource sharing.

The remote cache improves performance by 17-33%. However, performance degrades as cluster size increases. This is because the remote cache retains a large part of the working data set within a cluster regardless of cluster size. Therefore, when cluster size increases, the number of intra-cluster misses may not grow by much. However resource sharing and contention take effect, causing performance degradation. Even though a multiprocessor node implementation does not win performance-wise, it reduces the cost.

Software implementations. In the software implementations, LU exhibits inconsistent clustering behavior. Two factors counteract each other. On one hand, as is the case for the hardware implementations, clustering eliminates some inter-cluster conflict misses and increases the local memory accesses. On the other hand, contention on shared resources, especially the protocol processor, takes its toll. The latter dominates the performance, resulting in an increase in execution time until cluster size reaches 4. With 8 processors per node, intra-cluster data sharing and local memory accesses become more prevalent; consequently, the execution time starts to decrease.

Adding the remote cache improves performance substantially, by 45-50%. This is because the majority of cache misses in LU are conflict and capacity misses. The remaining memory overhead is mostly due to cold misses. In comparison, forwarding logic reduces execution time by 25%. With the forwarding logic, the amount of inter-cluster misses remains the same. Only the latency of the inter-cluster misses is lessened, yet it remains higher than that of intra-cluster misses.

Enhancement of communication primitives. Improvements resulting from communication primitives are negligible or small in all cases. A number of reasons account for this fact. The most important is that a significant portion of memory overhead consists of barrier synchronization time (cf. Figure 5.5). When there are

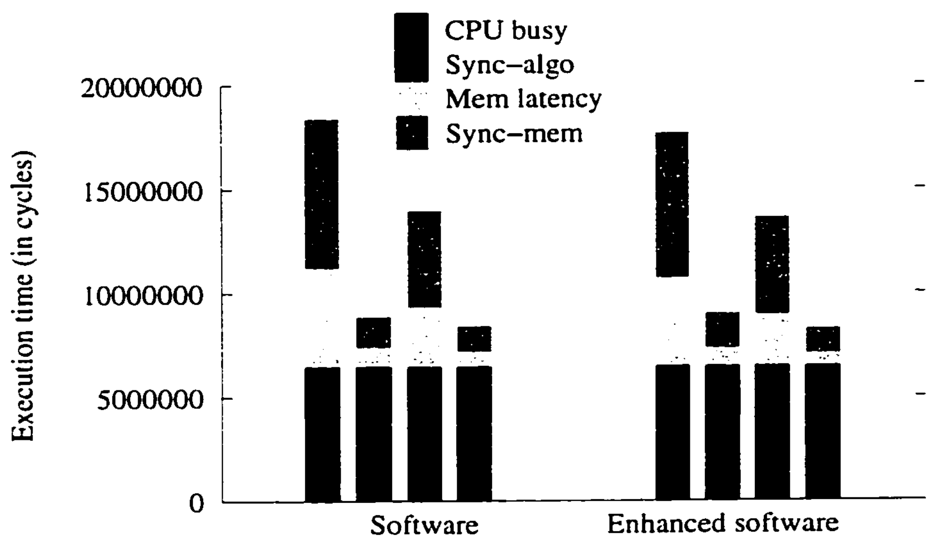


Figure 5.5: Execution times of LU: software and enhanced software implementation at the cluster size 2. In each group, the four bars from left to right correspond to the baseline architecture, the remote cache implementation, the forwarding logic implementation, and the implementation with both remote cache and forwarding logic respectively. For a detailed explanation of the legend, refer to the caption of Figure 3.3

no remote caches, the major components of the cache misses are conflict misses and capacity misses. Communication primitives are useless in reducing conflict misses due to their unpredictable nature. The modest improvement that is achieved can be attributed to the conservative prefetching primitive targeting for the data encountering capacity misses. When there are remote caches, the overall memory overhead is insignificant compared to the synchronization overhead. Thus, the slight gain of the communication primitives is overshadowed by the fluctuating of the synchronization overhead.

In summary, although the communication primitives do not appear to be of much use for LU decomposition, the software implementation assisted with remote cache and forwarding logic does present a performance competitive with that of the hardware scheme.

5.3.3 RADIX

Figures 5.6, 5.7, and 5.8 show performance results for RADIX. For a detailed explanation of the graphs, refer to Section 5.3.

Hardware implementations. The results of RADIX resemble those of FFT in that execution time is best at smaller cluster sizes. But the performance degradation due to the increasing cluster size is more pronounced for RADIX (cf. Figures 5.7 and 5.2) because of its higher miss ratio.

The remote cache improves performance by 30-42%. The corresponding number for FFT is 10-17%. The remote cache is more effective for RADIX because it has a large working data set and a higher miss ratio, the majority of which are capacity misses. By contrast, FFT has a lower cache miss ratio; a significant proportion of cache misses are coherence misses, which cannot be dealt with by the remote cache.

Software implementations. In the baseline cluster architecture, execution time of the software implementation doubles that of the hardware implementation at cluster size 1 and almost triples at cluster size 8. (We reported similar results, 33% and 150%, for FFT). The sharp increase in execution time at the large cluster size is caused by contention on the protocol processor.

Adding remote cache enhances performance of the software implementation by over 50%. Adding forwarding logic enhances performance by 26-38% (cf. Figure 5.7). Obviously, for RADIX, the remote cache is more useful than the forwarding logic. Recall that the reverse was true for FFT (cf. Section 5.3.1). The explanation also lies in the components of cache misses: The majority of the RADIX cache misses

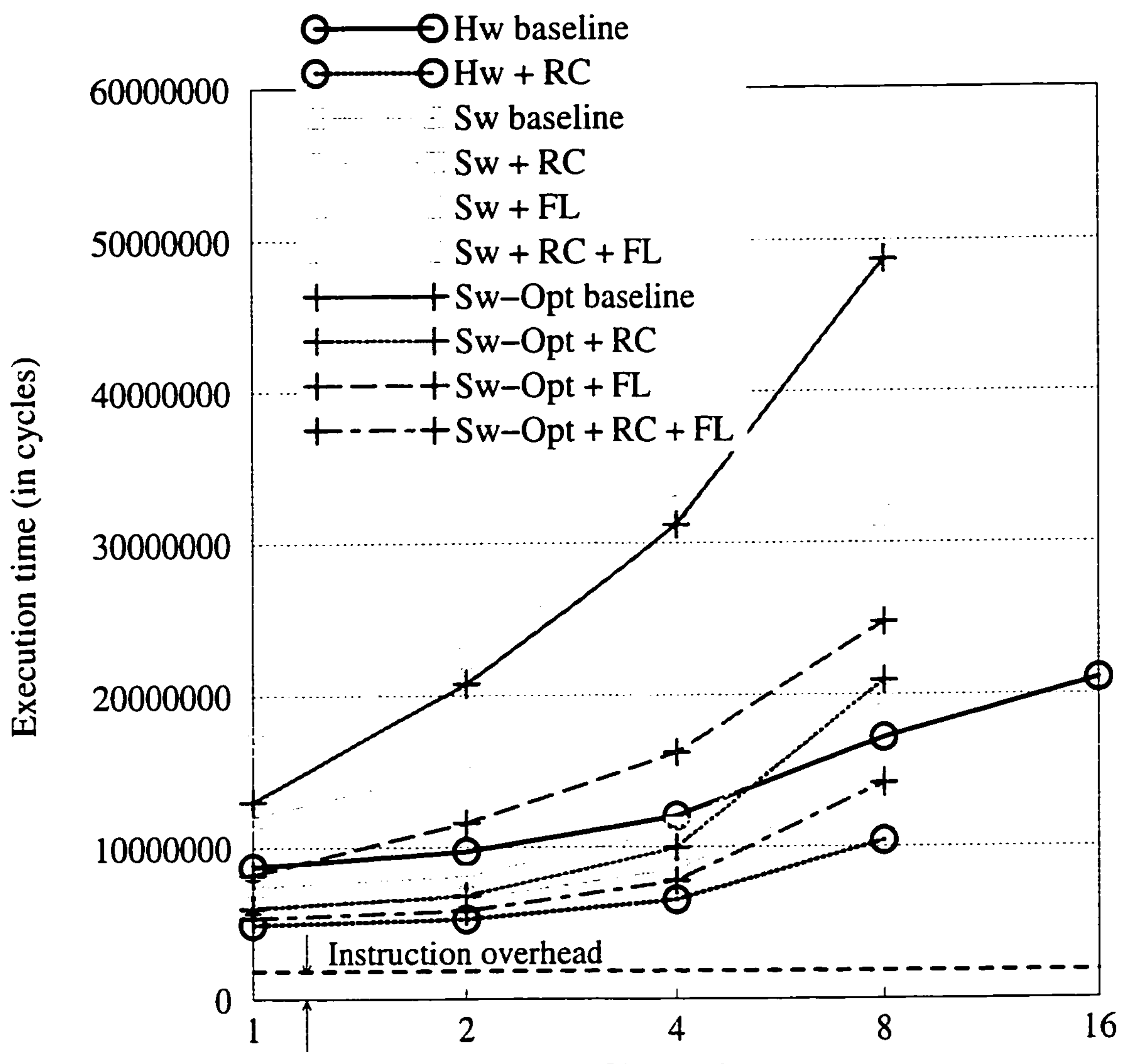


Figure 5.6: Execution times for RADIX.

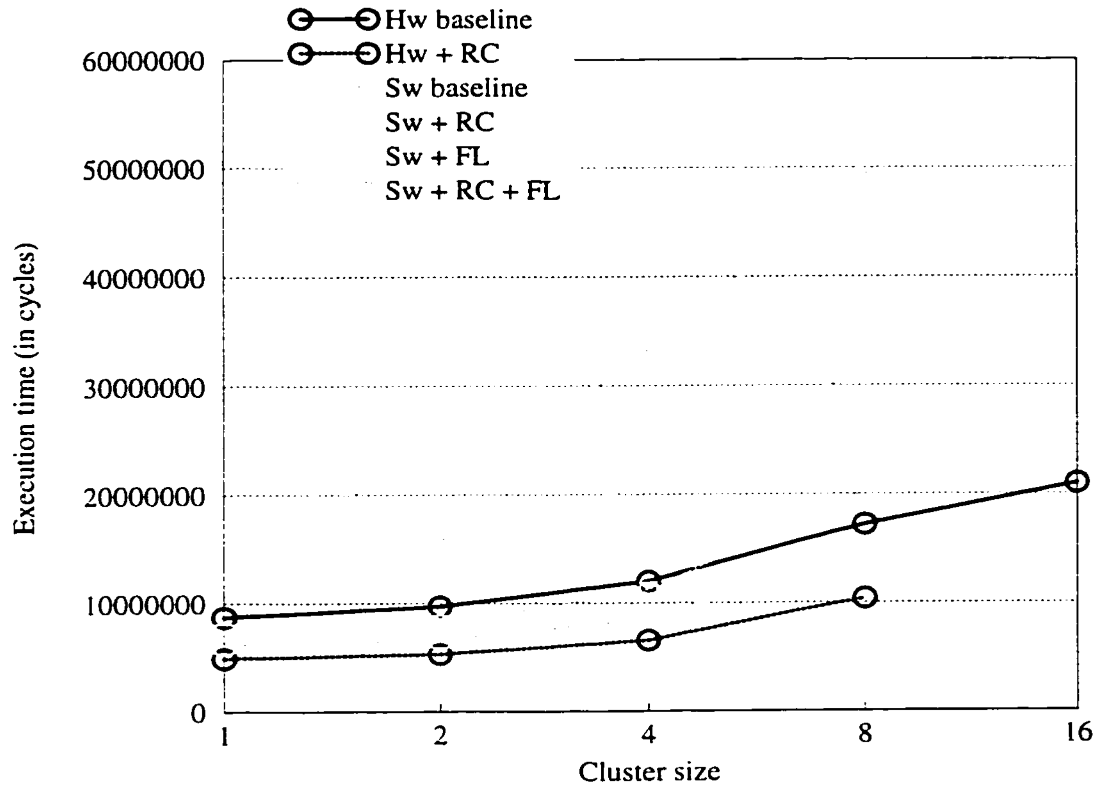


Figure 5.7: Execution times for RADIX (hardware and software without communication primitives implementations).

are capacity misses; the majority of the FFT cache misses are coherence misses. The remote cache is beneficial to inter-cluster capacity misses only.

As can be seen, as long as the remote cache is present, the software protocol processing overhead is not substantial since most cache misses are resolved by the cluster's internal hardware and protocol.

Enhancement of communication primitives. We applied two communication primitives to RADIX: *get* and *writemem*. The former prefetches the partial array to be sorted (in the first phase of the computation). The latter sets the write policy to

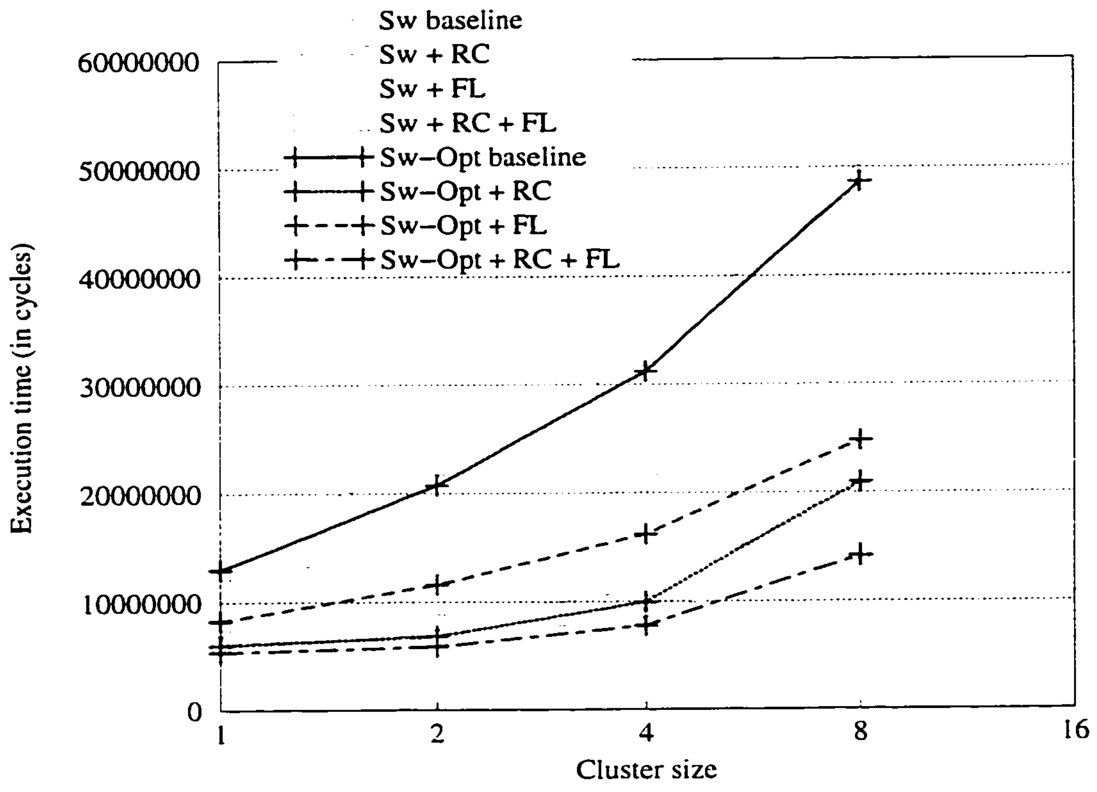


Figure 5.8: Execution times for RADIX (software and enhanced software implementations).

write-through when each processor writes the keys into new locations based on their "global ranks" (in the third phase of the computation).

In the baseline architecture, communication primitives improve performance by 4-25% (cf. Figure 5.8). The enhancement comes mostly from using the write-through policy; the prefetching primitive, although used in a very conservative manner, still caused cache pollution in the small caches and hence was not as useful as expected. Under the write-back policy, a processor encountering a write miss is blocked until the request is serviced completely, and it needs to place the acquired data in its cache.

Because RADIX has a very large working set and random data access patterns, cache lines are often replaced before they can be re-used when the cache is small (the case for machines without remote caches). In contrast, if the write-through policy is used, the processor writes into memory directly (instead of bringing data into cache). In addition, the processor can continue computation as soon as the write request is inserted into the write buffer[1]. However, the gain of the write-through policy narrows when cluster size increases (cf. Figure 5.8). This is because some inter-cluster capacity misses are converted to intra-cluster misses by intra-cluster data sharing under the write-back policy.

When forwarding logic is employed, the write-through policy reduces software memory overhead by 28-44%. This significant improvement suggests that forwarding logic is important to the write-through policy. The reason is that the write-through policy generates many small requests, a large portion of which is merely passing data or requests from one node to another. The forwarding logic can effectively speed up the processing of these requests. The write-through policy, however, is not worthwhile when the remote cache is present because the write-back policy enable processors to reuse data that are kept within a cluster by the remote cache, while the write-through policy foregoes such intra-cluster locality and forces data in the remote caches to be written back to memory. Therefore, when the remote cache is used, we exercise only the prefetching primitive, which gives an improvement of 10% at cluster sizes 1 and 2.

Once again, RADIX's results show that the software cache coherence scheme supplemented with forwarding logic and remote cache performs nearly as well as the hardware implementation. With the careful use of communication primitives, the enhanced software scheme assisted by simple hardware can out-perform the pure hardware scheme.

[1] When the write-through policy is used, the memory system is no longer sequentially consistent.

5.4 *Summary*

This chapter continued the study of communication primitives introduced in Chapter 3. We had previously restricted ourselves to the implementation and performance impact of these primitives in single-processor nodes. This chapter expanded the study to multiprocessor nodes.

The implementation of communication primitives in multiprocessor nodes presents additional challenges. In particular, the protocols must now deal with imprecise cache states since coherence directories are kept on a cluster-per-cluster rather than on a local-cache-per-local-cache basis. The solution we proposed lets the communication primitive protocols deal with inter-cluster operations and leaves intra-cluster operations for the default cache coherence protocol to resolve on demand. This approach avoids transferring data already in the requesting processor's node. However, data transfer cannot be avoided if a processor requests data with exclusive ownership unless there is appropriate hardware support.

We evaluated the performance of communication primitives in the cluster environment via trace-driven simulation. Given the results obtained from Chapter 4, we examine the impact of three architectural choices—cluster size, remote cache, and forwarding logic—on the performance of the communication primitives. A summary of results follows.

- *Cluster size.* When cluster size increases, the improvements by the communication primitives decrease. The diminishing value of the communication primitives is caused by the increasing intra-cluster data sharing that decreases the utility of the primitives.

- *Remote cache.* Chapter 4 demonstrated that the remote cache can significantly improve the performance of applications that suffer severely from capacity or conflict misses. This chapter shows that remote cache is also very important for applications (such as FFT) that suffer mainly from coherence misses because communication primitives can be exercised more aggressively.

- *Forwarding logic.* Forwarding logic is crucial to communication primitives that require the processing of many small requests, e.g., the write-through protocol, because pure software implementations carry a high overhead of receiving and processing small messages. The use of forwarding logic reduces the load on the protocol processor, hence its contention, as well as the overall cache miss latency.

The judicious use of communication primitives improves the performance of software implementations. However, this does not imply that the use of communication primitive always makes software implementations competitive with hardware implementations. When the cluster size for a particular application is too large, software overhead and contention on communication processors still outweigh the gains from implementing the primitives. On the other hand, for appropriate cluster sizes, software implementations supported by remote caches and forwarding logic yield execution times comparable to those of hardware implementations with remote caches. The overall conclusion is the same as that reached in Chapter 3, namely: the enhanced software solution is flexible, scalable, and yields competitive performance.

Chapter 6

CONCLUSION

This thesis is motivated by the need to build high-performance and cost-effective shared-memory systems, which call for efficient communication and graceful scalability. There exists a wide spectrum of techniques to implement the shared-memory paradigm. They vary in the amount of hardware or software support, the communication granularity, the flexibility with which systems can adapt to future needs, and the price/performance range. This thesis investigates the performance of fine-grain cache-coherent cluster architectures that employ communication processors and software to maintain cache coherence and are expected to scale better than bus-based shared-memory.

The use of communication processors and a software controlled cache coherence scheme provides a flexible way to enhance communication in shared-memory systems. To exploit this flexibility, we proposed a set of communication primitives that execute on a protocol processor embedded in a communication processor. These primitives give the user opportunities to optimize the communication cost conveniently and progressively. They allow the prefetching and post-storing of blocks of data whose sizes are not limited to single cache lines and permit the cache coherence protocol be tailored to specific application needs

6.1 Summary

Our study on the use and performance of communication primitives has been performed in three phases. We first assessed via trace-driven simulation the overhead of implementing the base protocol in software and the performance gains of the communication primitives in a simplified "cluster" architecture with one processor per node in which there is not much contention over the shared resources. Given the parameters we chose, the software implementation increased memory overhead by more than 50% relative to the hardware implementation. Nonetheless, when the software implementation is enhanced with a judicious use of communication primitives in the application, its performance is competitive with, and occasionally, exceeds, that of the hardware implementation.

The second phase of our study examines the issue of shared resource contention in the cluster environment, focusing on the resources of highest demand such as the protocol processor and data bus. The methodology we used is an MVA-based analytical model. The model lets us evaluate promptly a number of architectural choices, namely, cluster size and the use of remote cache and/or the use of forward logic. The model's input parameters include application-dependent parameters, such as the number of cache misses and the proportion of intra- and inter-cluster misses, as well as architectural parameters. The former were obtained via trace-driven simulation.

Three main conclusions were obtained by exercising the model. First, cluster size affects performance in two ways. On one hand, increasing cluster size decreases the demand (per compute processor) on the communication processor and does not change the demand (per computer processor) on the data bus. On the other hand, increasing cluster size increases resource sharing, and hence contention. Overall, increasing cluster size often degrades performance unless the application has good cluster locality. Second, employing the remote cache increases intra-cluster misses and significantly reduces protocol processor contention. Overall, it improves performance by 8-60% on the three applications we modeled. Third, the forwarding logic improves performance by 12-54%. Comparing remote cache with forwarding logic, we find that adding remote cache is more effective in applications suffering mostly from capacity or conflicts misses. Adding forwarding logic is more important to applications suffering mainly from coherence misses.

The final phase of our study evaluated communication primitives in multiprocessor cluster architectures. Following the methodology used in the first phase, we compared simulation results obtained under three environments: a hardware cache coherence scheme, a software scheme implementing only the basic cache coherence protocol, and an enhanced software scheme supporting additional communication primitives. We found that the memory overhead improvements decreases as cluster size increases. This is because a larger cluster size promotes native intra-cluster data sharing and decreases the utility of the primitives. Furthermore, our experiments demonstrated that remote cache is important not only to the applications suffering primarily from capacity and conflict misses but also to applications suffering mostly from coherence misses because the remote cache lets the primitives be inserted aggressively, hiding the latency of inter-cluster misses more effectively. We also found that, forwarding logic is crucial to communication primitives that require the processing of many small requests.

Our experiments assumed that the network has sufficient bandwidth and a fixed short latency. A slower network will increase the execution times of all three implementations. However, a longer network latency will harm the performance of the software implementation less than it does that of the hardware implementation. Its effect on the enhanced software scheme is even smaller because the communication primitives can pipeline the data transferring and overlap communication with computation. Also, we note that our estimation for the software protocol processing overhead is aggressive. If the software overhead is greater, due to less efficient interrupt handling or messaging mechanism, the performance of the software scheme will certainly deteriorate. However, its impact on the software implementation enhanced with the communication primitives will be less pronounced.

In summary, given the architectural parameters we chose and the estimated software directory protocol processing overhead, the software implementation supplemented with remote cache and forwarding logic can deliver a performance competitive with the rigid and pure hardware scheme. Also, with the judicious use of communication primitives, the enhanced software scheme can improve performance beyond the limit of the hardware implementation. In addition, the software cache coherence is more flexible, scalable and easier to optimize.

6.2 Future Work

In the future, we expect that parallel computers will be able to support applications written in both the shared-memory and message-passing programming models. If the message-passing model is chosen, an application's communication will be coded explicitly in calls to a message-passing library. Execution of message-passing code should be allowed to use a simple protocol that keeps cached data consistent with respect to the local memory, instead of a full-fledged shared-memory global cache-coherence protocol.

In case the shared-memory model is chosen, it will be helpful if the programmer is aware of communication costs and provides hints to the system for placing the data where they will be needed ahead of time. However, it may not be convenient or possible for the user to perform this task, due perhaps to the nature of the application or the lack of knowledge about source code. If source code is available, compiler analysis will be useful. For programs written in HPF (High Performance Fortran), there exists compiler techniques that detect inter-processor communication and generate functions calls to message-passing libraries. These techniques could be applied to shared-memory systems that support communication primitives. Since compiler analyses are based on static data and computation partitioning information, they are limited to scheduling communication statically.

If source code of the application is not available, or if the application has dynamic and complicated communication patterns, it would be desirable to have either hardware or software mechanisms that can monitor the application's memory access behavior and place the data speculatively. This would be an interesting avenue to explore further.

Future parallel computers will be used not only to execute SPLASH-like applications, but also to provide the horsepower for running web servers, database servers, graphics and media servers. Much work needs to be done to characterize the behavior of these commercial, possibly I/O intensive, applications. Studying commercial applications is usually difficult for a number of reasons. First, they are often released without source code. Second, some of the applications frequently use locks to protect shared meta- or database data. Consequently, a trace-based simulator may produce skewed results with respect to synchronization behavior, and an execution-driven simulator may be necessary. If the applications require many disk operations, the simulator also needs to capture the operating system's kernel activities. Despite the difficulties, this type of work is becoming increasingly important and providing fertile ground for future research.

BIBLIOGRAPHY

[1] A. Agarwal. Limits on interconnection network performance. *IEEE Trans. on Parellel and Distributed Systems*, 2(4):398–412, April 1991.

[2] A. Agarwal et al. The MIT Alewife machine: architecture and performance. In *Proc. of the 22nd Int. Symp. on Comp. Architecture*, pages 2–13, 1995.

[3] T. Agerwala et al. SP2 system archtecture. *IBM Systems Journal*, 34(2):152–84, 1995.

[4] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, and H. Lu. Treadmarks: shared memory computing on networks of workstations. *Computer*, 29(2):18–28, Feb. 1996.

[5] C. S. Anderson. *Improving Performance of Bus-Based Multiprocessors.* PhD thesis, Dept. of Computer Science, Univ. of Washington, 1995.

[6] J. Archibald and J.L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Trans. on Computer Systems*, 4(4):273–298, Nov. 1986.

[7] D. H. Bailey. FFT in external or hierarchical memory. *J. of Supercomputing*, 4(1):23–35, March 1990.

[8] D. H. Bailey et al. The NAS parallel benchmarks. *Int. J. of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[9] V. Bala et al. The IBM external user interface for scalable parallel systems. *Parallel Computing*, 20(4):445–462, April 1994.

[10] BBN Advanced Computer Inc. *Butterfly CP1000 switch tutorial*, 1989.

[11] G. Bell. Multis: A new class of multiprocessor computers. *Science*, pages 462–467, April 1985.

[12] L. N. Bhuyan, Q. Yang, and D. P. Agrawal. Performance of multiprocessor interconnection networks. *Computer*, 22(2):25–37, February 1989.

[13] N. J. Boden et al. Myrinet: a gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[14] T. Brewer and G. Astfalk. The evolution of the HP/Convex Exemplar. In *Digest of Papers, COMPCOM Spring 97*, pages 23–26, 1997.

[15] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Trans. on Computers*, C-27(12):1112–1118, Dec. 1978.

[16] T.-F. Chen and J.-L. Baer. A performance study of software and hardware data prefetching schemes. In *Proc. of the 21st Int. Symp. on Comp. Architecture*, pages 223–32, 1994.

[17] H. Cheong and A. V. Veidenbaum. A cache coherence scheme with fast selective invalidation. In *Proc. of the 15th Int. Symp. on Comp. Architecture*, pages 299–307, 1988.

[18] A. L. Cox and R. J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proc. of the 20th Int. Symp. on Comp. Architecture*, pages 98–108, 1993.

[19] D. E. Culler et al. Parallel programming in Split-C. In *Proc. of Supercomputing '93*, pages 262–73, 1993.

[20] R. Cytron, S. Karlovsky, and K. P. McAuliffe. Automatic management of programmable caches. In *Proc. of Int. Conf. on Parallel Processing*, pages 229–238, 1988.

[21] S. Eggers and R. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proc. of the 15th Int. Symp. on Comp. Architecture*, pages 373–382, 1988.

[22] B. Falsafi et al. Application-specific protocols for user-level shared memory. In *Proc. of Supercomputing '94*, pages 380–9, 1994.

[23] C. Flaig. VLSI mesh routing systems. Technical report, California Institute of Technology, 1987. Technical report 5241:TR:87.

[24] M. Galles and E. Williams. Performance optimizations, implementation, and verification of the SGI challenge multiprocessor. In *Proc. of the 27th Hawaii Int. Conf. on System Sciences. Vol. I: Architecture*, pages 134–43, 1994.

[25] E. Gornish, E. Granston, and A. Veidenbaum. Compiler-directed data prefetching in multiprocessor with memory hierarchies. In *Proc. of Int. Conf. on Supercomputing*, pages 354–368, 1990.

[26] D. B. Gustavson and Q. Li. The scalable coherent interface (SCI). *IEEE Communications Magazine*, 34(8):52–63, Aug 1996.

[27] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of message passing and shared memory in the Stanford FLASH multiprocessor. In *Proc.*

of the 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, pages 38–50, 1994.

[28] M. Heinrich et al. The performance impact of flexibility in the Stanford FLASH multiprocessor. In *Proc. of the 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 274–285, 1994.

[29] M. Hill, J. Larus, S. Reinhardt, and D. Wood. Cooperative shared memory: software and hardware for scalable multiprocessors. In *Proc. of the 5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 262–273, 1992.

[30] C. Holt. The effects of latency, occupancy, and bandwidth in distributed shared memory multiprocessors. Technical report, Dept. of Computer Science, Stanford Univ., 1995. CSL-TR-95-660.

[31] Intel Corporation. *Intel Paragon(tm) Supercomputer Product Brochure.* http://www.ssd.intel.com/paragon.html#system.

[32] David J.Lilja and Pen-Chung Yew. A compiler-assisted directory-based cache coherence scheme. Technical report, CSRD, Univ. of Illinois at Urbana-Champaign, 1990. CSRD report no. 990.

[33] M. Karlsson and P. Stenstrom. Performance evaluation of a cluster-based multiprocessor built from ATM switches and bus-based multiprocessor servers. In *Proc. of the 2nd Conf. on High-Performance Computer Architecture*, pages 4–13, 1996.

[34] Kendall Square Research Corporation. *KSR1 technical summary*, 1992.

[35] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.H. Lim. Integrating message-passing and shared-memory: early experience. In *Proc. of the 4th ACM Symp. on Principles and Practice of Parallel Programming*, pages 54–63, 1993.

[36] J. Kubiatowicz and A. Agarwal. Anatomy of a message in the Alewife multiprocessor. In *Proc. of the 7th ACM Int. Conf. on Supercomputing*, 1993.

[37] J. Kubiatowicz, D. Chaiken, and A. Agarwal. Closing the window of vulnerability in multiphase memory transactions. In *Proc. of the 5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 274–284, 1992.

[38] J. Kuskin et al. The Stanford FLASH multiprocessor. In *Proc. of the 21st Int. Symp. on Comp. Architecture*, pages 302–313, 1994.

[39] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scable server. In *Proc. of the 24th Int. Symp. on Comp. Architecture*, pages 241–251, 1997.

[40] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance.* Prentice-Hall, Inc., 1984.

[41] D. Lenoski et al. The Stanford DASH multiprocessor. *Computer*, 25(3):63–79, March 1992.

[42] Kai Li. Ivy: A shared virtual memory system for parallel computing. In *Proc. of Int. Conf. on Parallel Processing*, pages 94–101, 1988.

[43] T. Lovett and R. Clapp. STiNG: A CC-NUMA computer system for the commercial marketplace. In *Proc. of the 23rd Int. Symp. on Comp. Architecture*, pages 308–317, 1996.

[44] A. Muet al. A 9.6 GigaByte/s throughput plesiochronous routing chip. In *Digest of Papers. COMPCON '96*, pages 261–6, 1996.

[45] B. A. Nayfeh and K. Olukotun. Exploring the design space for a shared-cache multiprocessor. In *Proc. of the 21st Int. Symp. on Comp. Architecture*, pages 166–175, 1994.

[46] B. A. Nayfeh, K. Olukotun, and J.P. Singh. The impact of shared-cache clustering in small-scale shared-memory multiprocessors. In *Proc. of the 2nd Int. Symp. on High-Performance Computer Architecture*, pages 74–84, 1996.

[47] A. G. Nowatzyk, B. C. Browne, E. J. Kelly, and M. Parkin. S-Connect: from networks of workstations to supercomputer performance. In *Proc. of the 2nd Int. Symp. on Comp. Architecture*, pages 71–82, 1995.

[48] A. Nowatzyk et al. Exploiting parallelism in cache coherency protocol engines. In *Proceedings of EURO-PAR '95 Parallel Processing*, pages 269–86, 1995.

[49] A. Nowatzyk et al. The S3.mp scalable shared memory multiprocessor. In *Proc. of Int. Conf. on Parallel Processing*, pages 1–10, 1995.

[50] G.F. Pfister, W. C. Brantley, D. A. George, and S. L. Harvey. The IBM research parallel processor prototype (RP3): introduction and architecture. In *Proc. of Int. Conf. on Parallel Processing*, pages 764–71, 1985.

[51] P. Pierce. The NX message passing interface. *Parallel Computing*, 20(4):463–480, April 1994.

[52] D. K. Poulsend and P.-C. Yew. Integrating fine-grained message passing in cache coherent shared memory multiprocessors. *J. of Parallel and Distributed Computing*, 33(2):172–188, March 1996.

[53] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, and I Yanasak. Architectural mechanisms for explicit communication in shared memory multiprocessors. In *Proc. of Supercomputing '95*, 1995.

[54] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled hardware support for distributed shared memory. In *Proc. of the 24th Int. Symp. on Comp. Architecture*, pages 34–43, 1996.

[55] S.K. Reinhardt, J.R. Larus, and D.A. Wood. Tempest and Typhoon: User-level shared memory. In *Proc. of the 21st Int. Symp. on Comp. Architecture*, pages 325–326, 1994.

[56] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: a low overhead, software-only approach for supporting fine-grain shared memory. In *Proc. of the 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 174–85, 1996.

[57] I. Schoinas et al. Fine-grain access control for distributed shared memory. In *Proc. of 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, 1994.

[58] R. Simoni. *Cache coherence directories for scalable multiprocessors.* PhD thesis, Stanford University, 1992. TR-CSL-TR-92-550.

[59] J. P. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: performance and architecture implications. *IEEE Computer*, 27(7):45–55, July 1994.

[60] P. Stenstrom, M. Brorsson, and L. Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proc. of the 20th Int. Symp. on Comp. Architecture*, pages 109–118, 1993.

[61] C. B. Stunkel et al. Architecture and implementation of vulcan. In *Proc. of the 8th Int. Parallel Processing Symp.*, pages 268–74, 1994.

[62] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocol and their support by the IEEE Futurebus. In *Proc. of the 13th Int. Symp. on Comp. Architecture*, pages 414–23, 1986.

[63] J. Torrellas, J. Hennessy, and T. Weil. Analysis of critical architectural and program parameters in a hierarchical shared-memory multiprocessor. In *Proc. of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 163–72, 1990.

[64] J. E. Veenstra and R. J. Fowler. MINT: a front end for efficient simulation of shared-memory multiprocessors. In *Proc. of the 2nd Int. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 201–7, 1994.

[65] A. V. Veidenbaum. A compiler-assisted cache coherence solution for multiprocessors. In *Proc. of Int. Conf. on Parallel Processing*, pages 1029–1036, 1986.

[66] M. Vernon, R. Jog, and G. S. Sohi. Performance analysis of hierarchical cache-consistent multiprocessors. In *Proc. of the Int. Seminar on Performance of Distributed and Parallel Systems*, pages 111–126, 1988.

[67] M. K. Vernon, E. D. Lazowska, and J. Zahorjan. An accurate and efficient performance analysis technique for multiprocessor snooping cache-consistency protocols. In *Proc. of the 15th Int. Symp. on Comp. Architecture*, pages 308–315, 1988.

[68] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. K. Schauser. Active messages: a mechanism for intergrated communication and computation. In *Proc. of the 19th Int. Symp. on Comp. Architecture*, pages 256–66, 1992.

[69] W.-D. Weber et al. The Mercury interconnect architecture: A cost effective infrastructure for high-performance servers. In *Proc. of the 23rd Int. Symp. on Comp. Architecture*, 1997.

[70] B. I. Wladawsky. The power and promise of parallel computing. *IBM Systems Journal*, 34(2):146–51, 1995.

[71] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. of the 22nd Int. Symp. on Comp. Architecture*, pages 24–36, 1995.

[72] S. C. Woo, J. P. Singh, and J. L. Hennessy. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In *Proc. of the 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 219–229, 1994.

Vita

Xiaohan Qin was born in Shanghai on December 23, 1964. She attended Fudan University, from which she received a Bachelor of Science and Master of Science in 1985 and 1988 respectively. After that, she joined the faculty of Department of Computer Science, also in Fudan University. In 1990, She came to United States for further education. She received another Master of Science from University of Texas at San Antonio in 1991. She started her doctoral studies at the University of Washington in 1992, which she completed in 1997. She is now a research staff member at IBM TJ Watson Research Center.