MIT/LCS/TR-644

Mechanisms and Interfaces for Software-Extended Coherent Shared Memory

by

David L. Chaiken

Sc.B., Brown University (1986)
S.M., Massachusetts Institute of Technology (1990)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1994

Author . . . .
Department of Electrical Engineering and Computer Science
September 1, 1994

Certified by . . . .
Anant Agarwal
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Frederic R. Morgenthaler
Chair, Department Committee on Graduate Students

Mechanisms and Interfaces for
Software-Extended Coherent Shared Memory

by
David L. Chaiken

Abstract

Software-extended systems use a combination of hardware and software to implement shared memory on large-scale multiprocessors. Hardware mechanisms accelerate common-case accesses, while software handles exceptional events. In order to provide fast memory access, this design strategy requires appropriate hardware mechanisms including caches, location-independent addressing, limited directories, processor access to the network, and a memory-system interrupt. Software-extended systems benefit from the flexibility of software, but they require a well-designed interface between their hardware and software components to do so.

This dissertation proposes, designs, tests, measures, and models the novel software-extended memory system of Alewife, a large-scale multiprocessor architecture. A working Alewife machine validates the design, and detailed simulations of the architecture (with up to 256 processors) show the cost versus performance trade-offs involved in building distributed shared memory. The architecture with a five-pointer LimitLESS directory achieves between 71% and 100% of full-map directory performance at a constant cost per processing element.

A *worker-set* model uses a description of application behavior and architectural mechanisms to predict the performance of software-extended systems. The model shows that software-extended systems exhibit little sensitivity to trap latency and memory-system code efficiency, as long as they implement a minimum of one directory pointer in hardware. Low-cost, software-only directories with no hardware pointers are very sensitive to trap latency and code efficiency, even in systems that implement special optimizations for intranode accesses.

Alewife's *flexible coherence interface* facilitates the development of memory-system software and enables a smart memory system, which uses intelligence to help improve performance. This type of system uses information about applications' dynamic use of shared memory to optimize performance, with and without help from programmers. An automatic optimization technique transmits information about memory usage from the runtime system to the compiler. The compiler uses this information to optimize accesses to widely-shared, read-only data and improves one benchmark's performance by 22%. Other smart memory features include human-readable profiles of shared-memory accesses and protocols that adapt dynamically to memory reference patterns.

Keywords: multiprocessor, parallel processing, shared memory, cache coherence, computer architecture simulation, worker set, smart memory

Thesis Supervisor: Anant Agarwal
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

When I entered MIT, Anant Agarwal was a young, enthusiastic professor and I was a young, naïve graduate student. Six years later, Anant is still a young, enthusiastic professor. I feel privileged to have been one of his first students at MIT and attribute much of my success to his guidance.

Frans Kaashoek and Greg Papadopoulos served on my thesis committee, and gave helpful comments, criticism, and encouragement throughout the process. Their insistence that I understand the design space led me to develop the model in Chapter 7. Frans has helped me improve my understanding and documentation of the flexible coherence interface. Greg has voiced the concerns of industry that I attempt to address in this dissertation.

The members of the Alewife group have been work-mates and friends for the last six years. John Kubiatowicz and I worked head-to-head on the Alewife architecture for the majority of that time. Kubi designed and implemented the A-1000 CMMU, which is the heart of the LimitLESS system. As the compiler guru of the Alewife project, David Kranz taught me a lot about building the software for a large system. I thoroughly enjoyed hacking NWO and the Alewife kernel with David. Beng-Hong Lim has shared an office with me during our entire stay at MIT. He often provided the only calm voice in Alewife design meetings. I will miss Beng when we part ways. Dan Nussbaum tried his best to convey his system-building experience to the younger members of the group, and to restrain our over-enthusiastic ambitions. David, Beng-Hong, and Dan were responsible for the Sparcle simulator modules of ASIM and NWO.

Kirk Johnson and Chris Metcalf created a stable computation environment for the entire Computer Architecture Group. Since they provided this essential service, I can almost forgive them for getting me addicted to `netrek`. On many occasions, Kirk has given me helpful insight into designing systems and evaluating their performance.

Ken Mackenzie and Don Yeung designed Alewife's packaging and built the system's hardware. They humored me by allowing me to plug together some components every now and then. Gino Maa showed me the ropes at MIT and helped me start the early phases of this research. Anne McCarthy has run the Alewife group for several years. Her administration allows the rest of the group to concentrate on research rather than logistics. Ricardo Bianchini has provided useful suggestions from the point of view of Alewife's most prolific user. Sramana Mitra motivated me to improve the functionality in NWO and tested the simulator as she wrote the statistics interface for Alewife. Vijayaraghavan Soundararajan and Rajeev Barua shared the office with Beng and me: I thank them for putting up with me!

Kirk Johnson wrote and stabilized the programming environment for NWOP, the CM-5 version of NWO. Thanks go to Alan Mainwaring, Dave Douglas, and Thinking Machines Corporation for their generosity and assistance in porting NWO to the CM-5. Additional thanks to Thinking Machines Corporation (especially the folks who maintain the in-house machines) for allowing me to use many late-night CM-5 cycles during the results-generation phase of this research. Project Scout at MIT also provided a platform for some of the simulations.

A discussion with Johnny Piscitello during the summer of 1993 proved to be instrumental in the construction of the flexible coherence interface. Margo Seltzer suggested the comparison between Mach `pmaps` and the flexible coherence interface.

The notation for the transition state diagram in Chapter 4 borrows from the doctoral thesis of James Archibald at the University of Washington, Seattle, and from work done by Ingmar Vuong-Adlerberg at MIT.

It is no secret within the computer architecture community that similar, high-quality research is being conducted at Stanford University and the University of Wisconsin–Madison. The graduate students and professors in the research groups at these universities have kept me on my toes, reviewed my papers, given me good feedback on my work, and been fun companions at meals and baseball games.

If there is any text in this document that I did not write, it is in Sections 2.1.3 and 3.2.2. Section 2.1.3 describes the notation for software-extended protocols. Anant and I agonized over this notation, and he eventually wrote the documentation. This section is probably the most controversial in the dissertation. One reviewer expressed concern that "future protocol designers will extend this scheme in bizarre ways to name new protocols." However, the notation is the best that we could produce. It is terse, convenient, and expressive. The ability to extend a language construct in bizarre ways is possibly a feature, not a bug.

Section 3.2.2 documents the applications used to experiment with Alewife's software-extended memory system. Two researchers deserve much credit for making these benchmarks available: Beng-Hong Lim of MIT gathered and documented the application suite for Alewife, and J.P. Singh of Stanford compiled the SPLASH benchmark suite. In addition, Dan Nussbaum wrote TSP; Kirk Johnson wrote AQ; Beng-Hong Lim and Dan Nussbaum wrote SMGRID; and Anshu Aggarwal wrote EVOLVE.

The workloads used in Chapter 4 came from different sources. Harold Stone and Kimming So helped obtain the SIMPLE and Weather traces. The post-mortem scheduler was implemented by Mathews Cherian with Kimming So at IBM. It was extended by Kiyoshi Kurihara, who found the hot-spot variable in Weather. Gino Maa wrote the ASIM network simulator. G.N.S. Prasanna wrote and analyzed Matexpr.

Dick Larson invited me to visit MIT in 1987 and deserves credit for attracting me to the Institute.

Thanks to Digital Equipment Corporation and the members of the Systems Research Center for giving me a great reason to get on with my life.

Ema and Aba sat me down in front of a key-punch at the age of 4, and I haven't stopped hacking since then. I would like to thank them for teaching me the value of an education and the persistence needed to pursue my goals.

My deepest gratitude to Ora for support, encouragement, friendship, and love.

Praised are You, the Eternal, our God, Ruler of the Universe,
who has graced us with life, sustained us, and allowed us to reach this season.

Machines used for simulations were provided by Digital Equipment Corporation, SUN Microsystems, and Thinking Machines Corporation. This research has been supported by NSF grant #MIP-9012773, ARPA grant #N00014-91-J-1698, and a fellowship from the Computer Measurement Group.

Contents

List of Figures

List of Tables

Chapter 1

Shared Memory Design

The search for inexpensive parallel computing continues. Computer manufacturers would like to tell consumers that processing is as easy to add to a system as memory. A system should be able to start with a single processor and grow arbitrarily large, as its owner's need for processing increases. The task of adding processors to a system should require only commodity components — packaged modularly — and perhaps some extra floor space in a computer room.

To a certain extent, this goal has already been achieved. Most workstation vendors market a product line that ranges from single-processor machines to high-end servers containing up to sixteen processors. The structure, or architecture, of these parallel systems keeps costs relatively low. Each processor's packaging looks much the same as in a standard workstation, and a bus (a set of up to 256 wires) connects the processors together. This combination of commodity processing elements and an inexpensive communication substrate keeps costs low.

Many of these multiprocessor systems offer a shared memory programming model that allows all processors to access the same linearly-addressed memory. Since this model is close to the one used for single-processor systems, it provides a convenient abstraction for writing multiprocessor applications: processors can store information for themselves or communicate information to others using a single mechanism.

Physical limitations restrict the ability of these computer architectures to scale past a small number of processors. Communication delay and bandwidth are the primary culprits. If every access to shared memory required interprocessor communication, the latency caused by reading and writing data would be prohibitive. Given the high rate of memory accesses required by today's fast processors, a bus simply can not move enough data to satisfy the number of requests generated by more than a few processors.

Cache memories solve part of the latency problem, just as they do in single-processor systems. By storing copies of frequently accessed data physically close to processors, caches provide fast average memory access times. In multiprocessors, caches also reduce the bandwidth that processors require from their communication substrate. The memory accesses that caches handle do not require round-trips on a system's bus.

Unfortunately, making copies of data causes the cache coherence problem: one processor's modifications of the data in shared memory must become visible to others. Small multiprocessors solve this problem by using the fact that all of the processors on a bus can receive all of the messages transmitted on it. This broadcast capability is at once liberating and constraining. By watching each other's transmissions, caches can keep themselves coherent. On the other hand, physical media that broadcast information have limited bandwidth. Even with caches, current technology limits inexpensive parallel computing to less than twenty high-performance processors.

Researchers who work beyond this limit strive instead for cost-effective parallel computing. A cost-effective system solves a customer's problem in the least expensive (and sometimes the only available) way, taking into account programming time, system cost, and computation time. The definition of a cost-effective system depends on each customer's application, but it typically requires a cost-efficient system: a cost-efficient system's price grows approximately linearly with the number of processors.

For the reasons discussed above, the decision to abandon a bus when designing a large, cost-efficient system is mandatory. The other choices involved in building a larger system are much more difficult. One possibility is to build a system without shared memory, and let programmers suffer the consequences. This strategy works for computing applications that consist of many large, independent tasks. Another possibility is to build a system without caches. This strategy requires an extremely high-bandwidth and expensive interconnection network, and might work well for applications that require so much computation that the system can effectively hide the latency of communication. A final possibility is to implement shared memory and caches without a broadcast mechanism.

An appropriate combination of hardware and software yields a cost-efficient system that provides coherent, fast, shared memory access for tens or hundreds of processors. The following chapters prove this thesis by demonstrating such a system and analyzing its performance.

There are two challenges involved in building coherent shared memory for a large-scale multiprocessor. First, the architecture requires an appropriate balance of hardware and software. The integrated systems approach provides the cost-management strategy for achieving this goal: implement common case operations in hardware to make them fast and relegate less common activities to software. Examining the access patterns of shared memory applications reveals that a typical *worker set* (the set of processors that simultaneously access a unit of data) tends to be small. In fact, most blocks of shared memory are accessed by exactly one processor; other blocks are shared by a few processors; and a small number of blocks are shared by many (or all) processors.

This observation leads to a family of shared-memory systems that handle small worker sets in hardware and use software to handle more complex scenarios. Such *software-extended* memory systems require a number of architectural mechanisms. The shared memory hardware must handle as many requests as possible; it must be able to invoke extension software on the processor; and the processor must have complete access to the memory and network. The balancing act is tricky, because the architect must control hardware complexity and cost without sacrificing performance.

The second challenge involves designing the software component of a memory system. As in any software system, there is a trade-off between abstraction and raw performance. If the system is to perform well as a whole, the software must execute efficiently; if the system is to benefit from the flexibility of software, it must facilitate rapid code development. A software-extended system requires an interface between the memory-system hardware and software that achieves both of these goals.

The ability to extend the hardware provides the opportunity for software to take an active role in improving system performance. At a minimum, the software can measure its own performance. Appropriate data collection and information feedback techniques can allow the software to help programmers understand their applications, enable compilers to generate better code, and tune the system as it runs. Thus, not only does the software-extension approach lead to a cost-efficient machine, it enables the synergy between components of a multiprocessing system that high performance requires.

A software-extended memory system has been proposed, designed, tested, measured, and analyzed during the construction of Alewife [2], a shared-memory architecture that scales to 512 processors. Alewife serves as both a proof-of-concept for the software-extension approach and as a platform for investigating shared-memory design and programming. Since the study has been conducted in the context of a real system, it focuses on simple solutions to practical problems. This engineering process has resulted in a multiprocessor that is both scalable and programmable.

Experience with Alewife shows that software-extended memory systems achieve high performance at a per-processor cost that is close to standard workstations. A combination of minimal hardware and intelligent software realizes performance comparable to expensive shared memory implementations. By adopting this design strategy, computer vendors can build truly scalable product lines.

1.1 Contributions

This dissertation:

- Proposes the software-extension approach and develops algorithms for hybrid hardware/software cache coherence protocols.

- Measures and analyzes the performance of the first software-extended memory system, using two complete implementations of the Alewife multiprocessor: a physical machine proves the viability of the design, and a simulation system allows experimentation with a spectrum of memory systems.

- Describes a flexible coherence interface that expedites development of memory system software. Detailed performance measurements of the software show the trade-off between flexibility and performance.

- Constructs and validates an analytical model of software-extended systems. This model surveys the design space and predicts the performance impact of a variety of architectural mechanisms.

- Demonstrates examples of smart memory systems that use the flexibility of software to help improve system performance. One of these systems implements a novel technique for optimizing the use of shared memory.

1.2 Organization

The next chapter elaborates on the design of large-scale shared memory systems. It reviews the work related to this dissertation and defines the terms used in the following chapters. Chapter 3 describes the tools used to perform the study of software-extended systems: models, simulators, hardware, and applications. Chapter 4 summarizes the study that shows that large-scale multiprocessors need a software-extended memory system. The study also specifies the coherence algorithms used by Alewife's memory system hardware. Chapter 5 describes the software side of Alewife's memory system, including the flexible coherence interface. Chapter 6 presents the results of an empirical study of Alewife's software-extended system. The study examines the trade-offs between cost, flexibility, and performance. Chapter 7 uses the empirical study to validate a mathematical model of the performance of software-extended systems. The chapter then uses the model to explore the space of shared memory designs. Chapter 8 describes the user interfaces, implementations, and benefits of three smart memory systems. Finally, Chapter 9 concludes by specifying a minimal software-extended shared memory architecture.

Chapter 2

Distributed Shared Memory

Contemporary research on implementing scalable shared memory focuses on distributed shared memory architectures. Rather than partitioning memory and processors into separate modules, these architectures distribute memory throughout a machine along with the system's processors. A processing element — also called a node — contains a processor, a bank of memory, and usually a cache. Each processor can access memory in its own node or transmit a request through the interconnection substrate to access memory in a remote node. Intranode accesses proceed just as memory accesses do in a normal single-processor system; internode memory accesses require much more substantial architectural mechanisms.

A distributed shared memory architecture must service processors' internode requests for data and maintain the coherence of shared memory. Before the work on software-extended shared memory, systems used either software or hardware to implement protocols that perform these functions.

Software distributed shared memory architectures [20, 57, 24, 11, 9, 8, 41] implement truly inexpensive parallel computing. They provide a shared memory interface for workstations and networks, using only standard virtual memory and communication mechanisms. In order to reduce the effects of the overhead of software support and long communication delays, these systems transfer large blocks of data (typically thousands of bytes) and amortize the overhead over many memory accesses. Accordingly, applications achieve good performance if their subtasks share large data objects and communicate data in bursts.

Recent developments in software distributed shared memory use a combination of programmer annotations and protocol optimizations to allow systems to transmit kilobytes of data at a time, and only as much data as is needed to maintain coherence. These techniques use weaker memory consistency models [1] than sequential consistency [52] (a convenient model of shared memory), but provide synchronization mechanisms that allow programmers to write correct applications. In addition, some software memory systems make intelligent choices about data transfer and coherence policies. The latest software implementations of distributed shared memory exhibit good parallelism for small systems running large problem sizes.

Hardware implementations of distributed shared memory [55, 33, 45] attempt to provide cost-effective implementations of larger systems. These systems accommodate

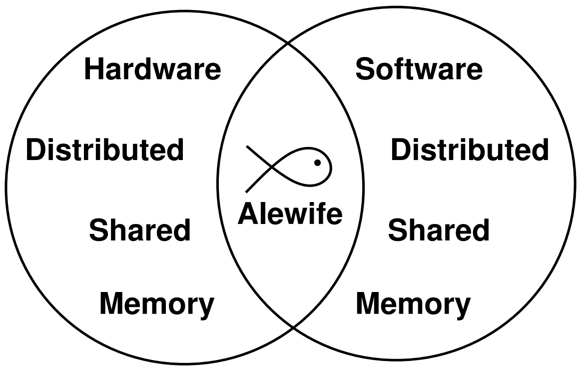


Figure 2-1: Alewife uses a hybrid approach to implement distributed shared memory.

applications with subtasks that share data in units of tens or hundreds of bytes, and make use of the high bandwidth and low latency provided by high-performance interconnection networks. Weak consistency models may also be applied to the design of these systems, but they are not absolutely essential for good performance: weak consistency is just one of a number of available latency-tolerance techniques [31]. Existing hardware implementations of distributed shared memory have expensive and complex memory systems. They are cost-effective solutions for applications when they are the only solution.

The software-extension approach, illustrated in Figure 2-1, bridges the gap between these two design styles [16, 34, 81]. By implementing basic transfer and coherence functions in hardware, software-extended systems provide efficient access to small units of data. The software part of these systems helps manage costs by using intelligent transfer and coherence policies when the hard-wired mechanisms require assistance. For historical reasons, software-extended systems are viewed as hardware implementations of distributed shared memory with software support. Section 8.5 examines the design approach from the opposite viewpoint.

Two parallel architectures [79, 23] combine the hardware and software approaches in a different way. These hybrid systems use bus-based coherence protocols for small clusters of processors, and software distributed shared memory techniques to enforce coherence between clusters. Each individual cluster can handle subtasks that share small objects and synchronize often; but for these systems to perform well, tasks located in different clusters must share large data objects and synchronize infrequently.

2.1 Cache Coherence Protocols

There are two logical sides of a shared memory system: the processor (request) side and the memory (response) side. In distributed systems, each node contains its own part of both sides of the system. The request side fields memory accesses from a processor, determines whether each access can be satisfied within the local node, and transmits messages into the network to initiate internode memory requests. The memory side receives intranode and internode requests, transmits data to requesting processors, and maintains coherence.

The software-extended approach may be applied to either side of the memory system. While the processor side and the memory side of a memory system are inextricably linked in the design of a cache coherence protocol, they may be treated separately when considering a software-extension methodology. Each side of the protocol has its own, separate finite-state specification. Exactly which states and transitions happen to be implemented in software should be decided independently for each machine.

2.1.1 The Processor Side

Each node must provide support for location-independent addressing, which is a fundamental requirement of shared memory. Hardware support for location-independent addressing permits application software to issue an address that refers to an object without knowledge of where it is resident. This hardware support includes an associative matching mechanism to detect if the object is cached, a mechanism to translate the object address and identify its location if it is not cached, and a mechanism to issue a message to fetch the object from a remote location if it is not in local memory.

As nodes are added to a multiprocessor, the total of all of the processor-side structures in the system grows at most linearly. Curiously, it is possible to argue that the cost of the processor side might grow sublinearly with the number of nodes: while holding problem size constant, cache working sets decrease as the number of nodes increases. Thus, smaller caches might be used in large multiprocessors than in small ones.

Since this research concentrates on the problem of cost management, it does not address the processor side of cache coherence protocols. It is certainly possible that — for reasons of flexibility, if not cost — a system designer might choose to implement part of the processor-side state machine in software. While this decision process may benefit from the experience described in the following chapters, it will hinge on a different set of constraints, such as processor cycle time and cache hit rates.

In addition, there are a host of techniques used to tolerate the latency of communication between nodes, including multiple-context support, prefetching, and weak consistency models. Since these techniques primarily impact the design of the processor side of the interface, they are not critical in the decision to extend part of the memory-side state machine into software. Therefore, this dissertation does not study the cost or performance implications of these mechanisms.

2.1.2 The Memory Side

Implementing memory coherence for a large number of processors requires the system to keep track of the locations of cached copies of data. A *directory* is a structure that helps enforce the coherence of cached data by maintaining pointers to the locations of cached copies of each memory block. When one node modifies a block of data, the memory system uses the information stored in the directory to enforce a coherent view of the data. At least one company has designed a centralized directory for small, bus-based systems [26]. For larger systems, directories are not monolithic structures: they are distributed to the processing nodes along with a system's shared memory.

A *cache coherence protocol* consists of the directory states, the cache states, and the internode messages that maintain coherence: request messages ask for data; response messages transmit data; invalidation messages ask caches to destroy (or to return) copies of data; and acknowledgment messages indicate completed invalidations. Using a directory to implement cache coherence eliminates the need for a broadcast mechanism, which becomes a serious performance bottleneck for systems with more than sixteen processors.

The DASH multiprocessor [55] implements a distributed full-map directory [12, 73, 54], which uses hardware to track up to the maximum number of copies for every block of data in a system. Such an implementation becomes prohibitively expensive for systems with more than 64 processors. This strategy implicitly assumes that every block of data has a home location that contains both the block's data and an associated directory *entry*, which holds the *pointers* to cached copies.

Hierarchical directories [33, 45] implement the directory as a tree-like structure that is distributed throughout the system. This implementation strategy eliminates the need for a home location, and allows application data and directory information to migrate throughout the machine. This policy is both more intelligent and more complex than other directory schemes. Recent studies indicate that the increased data locality afforded by hierarchical directories improves performance marginally over lower-cost schemes [62, 37].

Another decentralized scheme [36, 7] uses home locations but distributes directory information to caches along with copies of data. Coherence information is stored in linked-lists; each link of a chain resides in a different cache. While these chained directories have reasonable storage requirements, they are relatively complicated and suffer from long write latencies.

It is also possible to design hardware to implement a cost-efficient limited directory [5], which only permits a small number of copies of any block of data. However, the performance of limited directories falls dramatically when many processors attempt to share a single block of data [14].

The software-extension approach leads to the family of LimitLESS protocols, which achieve close to the performance of a full-map directory with the hardware of a limited directory [16]. This scheme, described by Chapter 4 in detail, implements a small number of pointers in a hardware directory (zero through five in Alewife), so that the hardware can track a few copies of any memory block. When these pointers are exhausted, the memory system hardware interrupts a local processor, thereby requesting it to maintain correct shared memory behavior by extending the hardware directory with software.

Another set of software-extended protocols (termed $\mathrm{Dir}_1\mathrm{SW}$) were proposed in [34] and [81]. These protocols use only one hardware pointer, rely on software to broadcast invalidates, and use hardware to accumulate the acknowledgments. In addition, they allow the programmer or compiler to insert Check-In/Check-Out (CICO) directives into programs to minimize the number of software traps.

All software-extended memory systems require a battery of architectural mechanisms to permit a designer to make the cost versus performance trade-off. First, the shared memory hardware must be able to invoke extension software on the processor, and the processor must have complete access to the memory and network hardware [16, 47, 81].

Second, the hardware must guarantee forward progress in the face of protocol thrashing scenarios and high-availability interrupts [48]. The system must also implement the processor-side support described in the previous section. Since these mechanisms comprise the bulk of the complexity of a software-extended system, it is important to note that the benefits of these mechanisms extend far beyond the implementation of shared memory [42, 29]: integrating message passing and shared memory promises to be an active topic of research in the coming years.

Alternative scalable approaches to implementing shared memory proposed in [61, 69, 77] use hardware mechanisms that allocate directory pointers dynamically. These schemes do not require some of the mechanisms listed above, but they lack the flexibility of protocol and application software design. Section 8.2 gives an example of how to incorporate scalable coherence algorithms into a software-extended system.

2.1.3 A Spectrum of Protocols

The number of directory pointers that are implemented in hardware is an important design decision involved in building a software-extended shared memory system. More hardware pointers mean fewer situations in which a system must rely on software to enforce coherence, thereby increasing performance. Having fewer hardware pointers means a lower implementation cost, at the expense of reduced performance. This tradeoff suggests a whole spectrum of protocols, ranging from zero pointers to n pointers, where n is the number of nodes in the system.

The n pointer protocol

The full-map protocol uses n pointers for every block of memory in the system and requires no software extension. Although this protocol permits an efficient implementation that uses only one bit for each pointer, the sheer number of pointers makes it extremely expensive for systems with large numbers of nodes. Even though the cost of the full-map protocol makes it impractical, it serves as a good performance goal for the software-extended schemes.

$2 \leftrightarrow (n-1)$ pointer protocols

There is a range of protocols that use a software-extended coherence scheme to implement shared memory. It is this range of protocols that allows the designer to balance hardware cost and system performance. From the point of view of implementation complexity, the protocols that implement between 2 and $n-1$ pointers in hardware are homogeneous. Of course, the $n-1$ pointer protocol would be even more expensive to implement than the full-map protocol, but it still requires exactly the same hardware and software mechanisms as the protocols at the other end of the spectrum.

In a load/store architecture, the software for protocol extension needs to service only two kinds of messages: read and write requests. It handles read requests by allocating an extended directory entry (if necessary), emptying all of the hardware pointers into the software structure, and recording a pointer to the node that caused the directory overflow.

Subsequent requests may be handled by the hardware until the next overflow occurs. For all of these protocols, the hardware can return the appropriate data to requesting nodes; the software only needs to record requests that overflow the hardware directory.

To handle write requests after an overflow, the software transmits invalidation messages to every node with a pointer in the hardware directory or in the software directory extension. The software then returns the hardware directory to a mode that collects one acknowledgment message for each transmitted invalidation.

Zero-pointer protocols

The zero-pointer LimitLESS scheme, termed a *software-only directory architecture,* provides an interesting design point between software and hardware distributed shared memory. It uses hardware to support the processor-side functions, but implements the entire memory side in software.

Since the software-only directory has no directory memory, it requires substantially different software than the $2 \leftrightarrow (n-1)$ range of protocols. This software must implement all coherence protocol state transitions for internode accesses [64].

One-pointer protocols

The one-pointer protocols are a hybrid of the protocols discussed above. Chapter 6 studies the performance of three variations of this class of protocols. All three use the same software routine to transmit data invalidations sequentially, but they differ in the way that they collect the messages that acknowledge receipt of the invalidations. The first variation handles the acknowledgments completely in software, requiring a trap from the hardware upon the receipt of each message. During the invalidation/acknowledgment process, the hardware pointer is unused.

The second protocol handles all but the last of a sequence of acknowledgments in hardware. If a node transmits 64 invalidations for a memory block, then the hardware will process the first 63 invalidations. This variation uses the hardware pointer to store a count of the number of acknowledgments that are still outstanding. Upon receiving the 64^{th} acknowledgment, the hardware invokes the software, which takes care of transmitting data to the requesting node.

The third protocol handles all acknowledgment messages in hardware. This protocol requires storage for two hardware pointers: one pointer to store the requesting node's identifier and another to count acknowledgments. Although a designer would always choose to implement a two-pointer protocol over this variation of the one-pointer protocol, it still provides a useful baseline for measuring the performance of the other two variations.

A notation for the spectrum

This section introduces a notation that clearly specifies the differences between various implementations and facilitates a precise cost comparison. The notation is derived from a nomenclature for directory-based coherence protocols introduced in [5]. In the previous notation, a protocol was represented as Dir_iX, where i represented the number of explicit copies tracked, and X was *B* or *NB* depending on whether or not the protocol issued broadcasts. Notice that this nomenclature does not distinguish between the functionality implemented in the software and in the hardware. The new notation attempts to capture the spectrum of features of software-extended protocols that have evolved over the past several years, and previously termed LimitLESS_1, LimitLESS_4, and others in [16], and Dir_1SW, $\text{Dir}_1\text{SW+}$, and others in [34, 81].

For both hardware and software, the new notation divides the mechanisms into two classes: those that dictate directory actions upon receipt of processor requests, and those that dictate directory actions for acknowledgments. Accordingly, the notation specifies a protocol as: $\mathbf{Dir_i H_X S_{Y,A}}$, where i is the number of explicit pointers recorded by the system — in hardware or in software — for a given block of data.

The parameter X is the number of pointers recorded in a hardware directory when a software extension exists. X is *NB* if the number of hardware pointers is i and no more than i shared copies are allowed, and is B if the number of hardware pointers is i and broadcasts are used when more than i shared copies exist. Thus the full-map protocol in DASH [55] is termed $Dir_n H_{NB} S_{-}$.

The parameter Y is *NB* if the hardware-software combination records i explicit pointers and allows no more than i copies. Y is B if the software resorts to a broadcast when more than i copies exist.

The A parameter is *ACK* if a software trap is invoked on *every* acknowledgment. A missing A field implies that the hardware keeps an updated count of acknowledgments received. Finally, the A parameter is *LACK* if a software trap is invoked only on the *last* acknowledgment.

According to this notation, the LimitLESS_1 protocol defined in [16] is termed $Dir_n H_1 S_{NB}$, denoting that it records n pointers, of which only one is in hardware. The hardware handles all acknowledgments and the software issues invalidations to shared copies when a write request occurs after an overflow. In the new notation, the three one-pointer protocols defined above are $Dir_n H_1 S_{NB,ACK}$, $Dir_n H_1 S_{NB,LACK}$, and $Dir_n H_1 S_{NB}$, respectively.

The set of software-extended protocols introduced in [34] and [81] can also be expressed in terms of the notation. The Dir_1SW protocol maintains one pointer in hardware, resorts to software broadcasts when more than one copy exists, and counts acknowledgments in hardware. In addition, the protocol traps into software on the last acknowledgment [80]. In the notation, this protocol is represented as $Dir_1 H_1 S_{B,LACK}$. This protocol is different from $Dir_n H_1 S_{NB,LACK}$ in that $Dir_1 H_1 S_{B,LACK}$ maintains only one explicit pointer, while $Dir_n H_1 S_{NB,LACK}$ maintains one pointer in hardware and extends the directory to n pointers in software. An important consequence of this difference is that $Dir_n H_1 S_{NB,LACK}$ potentially interrupts processors on read requests, while $Dir_1 H_1 S_{B,LACK}$ does not. Unlike $Dir_n H_1 S_{NB,LACK}$, $Dir_1 H_1 S_{B,LACK}$ must issue broadcasts on write requests to memory blocks that are cached by multiple nodes.

2.2 Flexible and Smart Memory Systems

Virtual memory designers know well the benefits of flexible interfaces in memory systems. The CMU Mach Operating System uses an interface that separates the machine-dependent and machine-independent parts of a virtual memory implementation [65]. This `pmap` interface allows the Mach memory system to run efficiently on a wide range of architectures. Some of the features of this interface are remarkably similar to Alewife's flexible coherence interface, which separates the hardware and software components of the software-extended memory system.

Flexible virtual memory systems can implement a number of mechanisms that are useful to higher-level software [6]. In fact, software distributed shared memory architectures use such mechanisms to implement smart memory systems, which use intelligence to optimize memory performance. DUnX [53] uses heuristics to place and to migrate pages of virtual memory on the BBN GP1000. Munin [11] allows programmers to annotate code with data types and handles each type of data differently. Orca [8] uses a compiler to analyze access to data objects statically. Based on the analysis, the compiler selects an appropriate memory for each object.

Similar optimization techniques work for software-extended implementations of distributed shared memory. Lilja and Yew demonstrate a compiler annotation scheme for optimizing the performance of protocols that dynamically allocate directory pointers [58]. Hill, Wood, and others propose and evaluate a programmer-directed method for improving the performance of software-extended shared memory [34, 81]. The studies show that given appropriate annotations, a large class of applications can perform well on $Dir_1H_1S_{B,LACK}$. Cachier [21] takes this methodology one step further by using dynamic information about a program's behavior to produce new annotations, thereby improving performance. Cachier requires the programmer to label all shared data structures and to run the program on a simulator.

The profile-detect-optimize technique implemented for Alewife (in Section 8.4) is similar to Cachier, except that it runs entirely on a real software-extended system and requires no simulation. This technique was inspired by the Multiflow trace-scheduling compiler [22].

The LimitLESS profiler in Section 8.3 and Mtool [30] both attempt to provide information about shared memory usage. The two systems' implementations are very different, however. Mtool instruments code on the processor side and presents information in terms of memory access overhead in procedures. The LimitLESS profiler instruments the memory side and reports the worker-set behavior of important data objects.

FLASH [51] and Typhoon [66] explore alternative methods for building flexible memory systems. These architectures dedicate processors to implement scalable shared memory. Section 7.4.3 explores the differences in performance between these architectures and Alewife. Hand-crafted protocols that can optimize access for specific types of data have been developed for FLASH and Typhoon [18, 66].

2.3 Understanding Memory System Performance

A worker set is the set of processors that access a block of memory between subsequent modifications of the block's data. The notion of a worker set is loosely based on working sets, which specify the portions of memory accessed by a single processor over an interval of time. In the working-set definition, the time interval is artificially defined for the sake of measurement. Conversely, worker sets specify the processors that access a block of data over an interval of time. The worker-set time interval is defined implicitly by the reference pattern of each object.

Denning [27] writes that "...the working set is not a model for programs, but for a class of procedures that measure the memory demands of a program." This model predicts the appropriate amount of memory or cache required by each process using a virtual memory system. Similarly, the worker-set behavior of shared memory programs specifies the demands on a software-extended system. The worker-set model predicts the appropriate mix of hardware and software for running a multiprocessor application.

Measuring access patterns in terms of worker sets is certainly not new. The worker-set invalidation profiles in Figure 7-3 are virtually identical to the ones studied by Weber in [78]. Simoni and Horowitz used stochastic models to predict similar invalidation patterns, cache miss statistics, and transaction latencies for directory-based protocols [70]. Simoni completed this work with an analysis of scalable hardware directory schemes [68]. He also presented a simple analysis that translated data sharing patterns into processor utilization. Tsai and Agarwal derived sharing patterns and cache miss ratios directly from parallel algorithms [75].

The inputs to the worker-set model in Chapter 7 are similar to the outputs of the preceding studies: it predicts the performance of software-extended systems from worker-set behavior and cache access statistics. The model is validated by an empirical study of Alewife's software-extended memory system and its performance, which is the subject of the next four chapters.

Chapter 3

An Architect's Workbench

A primary goal of the research on software-extended shared memory was to build a large, working multiprocessor system with a convenient programming interface. This chapter describes the methodology for achieving this goal and the various tools used during the life-cycle of the Alewife project.

The life-cycle began with initial studies using mathematical models and simple simulation techniques that determined the basic features of the multiprocessor's architecture. These studies investigated a wide range of design options, including the programming model, the practical number of processors, the balance of communication and computation, cache coherence protocols, and the type of interconnection network.

The tools used during this stage of the project did not model the intricate details of the architecture, and therefore could be constructed quickly. In practice, the models and simulators either produced their results instantaneously or simulated long executions in a relatively short amount of time. Due to the lack of detail, the tools measured quantities such as component utilization, transaction latency, and system throughput. While these metrics did not indicate end-to-end performance, they measured the relative benefits of different design alternatives.

The next stage consisted of a preliminary system design at the functional level. The Alewife simulator, called ASIM, modeled large system components and their internal structure at the level of state machines, data repositories, and objects communicated between components. This simulator took considerable effort to build, because it implemented the cross-product of a wide range of design options, at a more detailed level than the tools of the previous stage. The additional detail supported the development of the first software for Alewife, including compilers and task schedulers. It also allowed performance to be measured directly in terms of the number of cycles required to run a program, and the speedup when using multiple processors instead of a sequential system.

While ASIM could still measure utilization, latency, and throughput, these quantities typically were used only to explain behavior observed with the higher-level metrics. ASIM simulated about 20,000 cycles per second. Given some patience, this speed permitted the development of small applications and the investigation of design trade-offs. The preliminary evaluation of software-extended shared memory took place at this stage of the project.

The first two stages provided the foundation for the next — and longest — phase of development: the implementation of the architecture, dubbed the A-1000. The design of Alewife's hardware and software required two complementary simulation systems. One simulator, provided by LSI Logic, directly executed the gate-level specification of the hardware implementation. Such a simulation was integral to the hardware fabrication process, but it modeled the system at such a low level of detail that it barely reached speeds of one clock cycle per second.

The other simulator, called NWO, modeled the system at a high enough level that it reached speeds of 3,000 clock cycles per second but could still run all of the same programs as the A-1000. NWOP, a parallel version of NWO, executed tens of thousands of cycles per second on the CM-5, a parallel computer built by Thinking Machines. The dual LSI/NWO simulation approach decoupled Alewife's hardware and software efforts, allowing the development of compilers, operating systems, and applications to advance in parallel with the physical system's design, test, layout, and fabrication.

During this phase of development, the standard metrics used in previous stages helped refine the design and fine-tune the performance of the system as a whole. At this stage, the correct operation of the system was just as important as the projected speed. The rate that hardware bugs were found (and removed) served as a measure of the stability of the system and gated the decision to fabricate the components.

The development phase also included the finalization of the hardware-software interface and the design of abstractions within the the software system. A qualitative metric used during this process was the time required to write code for the system. There were two types of programmers who had to be satisfied: the application programmer, who works for the multiprocessor's end-user; and the operating system programmer, who works closely with architects and compiler writers. The flexible coherence interface grew out of the need for a rich environment that reduced the effort required for operating system programmers to write modules of the memory-system software.

Fabricating the A-1000 culminated the development phase of the project. While the simulators from the previous stage remained important, the abstract workbench turned into a physical one with the real tools required to assemble Alewife machines. The working machines validated the simulation and modeling techniques used to define the architecture. Hardware running at millions of cycles per second enabled research using applications with real data sets and operating systems with a full complement of functionality.

The final stage of the project reexamined the design space in light of the lessons learned from designing and implementing the architecture. The tools used for the analysis were the same as those used in the initial studies: mathematical models quantified the importance of features of the architecture with utilization, latency, and throughput. It was somewhat satisfying to use the metrics from the preliminary studies of the Alewife memory system. The analysis with these metrics closed the research life-cycle by using the specific implementation to help evaluate a wide range of design options.

The next section describes the Alewife architecture, ASIM, and the two implementations of Alewife: the A-1000 and NWO. This chapter also describes a synthetic workload and the benchmarks that drive the performance analysis.

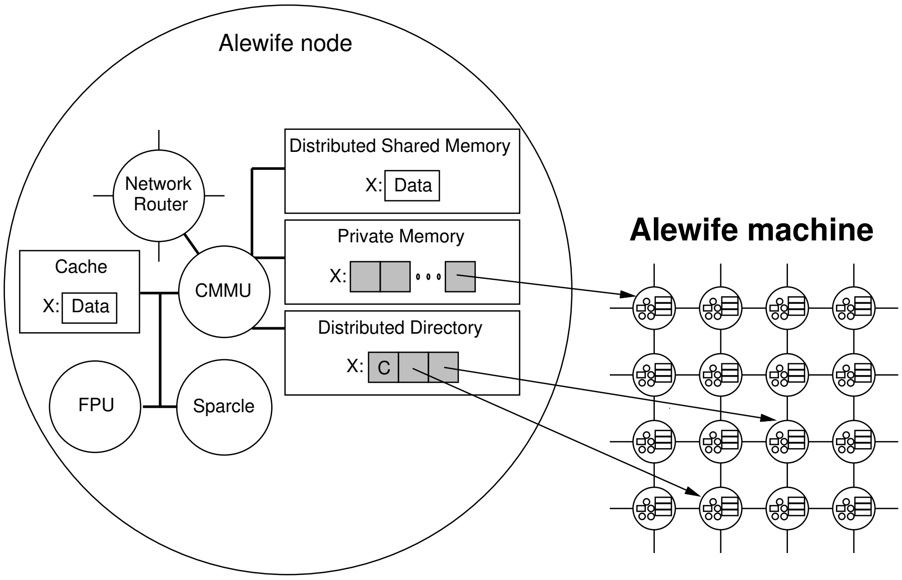


Figure 3-1: Alewife node, with a $Dir_nH_2S_{NB}$ memory block.

3.1 The Alewife Architecture

Alewife provides a proof-of-concept for software-extended memory systems and a platform for experimenting with many aspects of multiprocessor design and programming. The Alewife architecture scales up to 512 processing elements with distributed shared memory. It uses caches to implement fast memory access, and a software-extended memory system to enforce sequential consistency.

Figure 3-1 shows an enlarged view of a node in the Alewife machine. The processor on each node consists of a Sparcle processor [3], which is based on the 32-bit SPARC architecture [72], and a floating-point coprocessor. The nodes communicate via messages through a direct network [67] with a mesh topology. Alewife's memory system includes a cache, a bank of private memory, a portion of the globally-shared main memory, and the corresponding portion of the hardware limited directory. A single-chip communications and memory management (CMMU) on each node holds the cache tags and implements the memory coherence protocol by synthesizing messages to other nodes.

The 512 node restriction is primarily a function of Sparcle's 32-bit address, which contains enough space for 22 bits (4 Mbytes) of offset, a 9-bit node identifier (512), and one bit to differentiate shared addresses from private addresses.

In addition to the standard hardware directory pointers, Alewife implements a special one-bit pointer for the node that is local to the directory. Several simulations show that this extra pointer improves performance by only about 2%. Its main benefit lies in reducing the complexity of the protocol hardware and software by eliminating the possibility that a node will cause its local hardware directory to overflow.

Alewife also includes a mechanism that allows a zero-pointer protocol to be optimized. This optimization may be described as follows[1]: the zero-pointer protocol uses

[1]The actual implementation in Alewife is slightly different.

one extra bit per memory block to optimize the performance of purely intranode accesses. The bit indicates whether the associated memory block has been accessed at any time by a remote node. When the bit is clear (the default value), all memory accesses from the local processor are serviced without software traps, just as in a uniprocessor. When an internode request arrives, the bit is set and the extension software flushes the block from the local cache. Once the bit is set, all subsequent accesses — including intranode requests — are handled by software extension.

All of the performance results in Chapter 6 measure protocol implementations that use these one-bit optimizations. Sections 7.4.3 and 7.4.5 use the analytical model to evaluate the contribution of this architectural mechanism to the performance of the system.

There are two other features of the Alewife architecture that are relevant to the memory system, the most important of which is integrated shared memory and message passing [42, 47]. This mechanism enables software-extended memory by allowing the processor to transmit and receive any kind of network packet, including the ones that are part of the cache coherence protocol. Part of this mechanism is a direct-memory-access facility. As a result, much of the interprocessor communication — especially in the operating system, the task scheduler, and the synchronization library — uses message-passing semantics rather than shared memory. This policy improves performance and relieves the memory system of the burden of handling data accesses that would be hindered, rather than helped, by caches.

The Sparcle processor uses the SPARC register windows to implement up to four hardware contexts, each of which holds the state of an independent thread of computation. This implementation allows the processor to switch quickly between the different threads of execution, an action which is typically performed upon a remote memory access. While the context-switching mechanism is intended to help the system tolerate the latency of remote memory accesses [4, 50, 31], it also accelerates the software-extended memory system: when the processor receives an interrupt from the CMMU, it can use the registers in an empty context to process the interrupt, rather than saving and restoring state. Section 7.4.1 uses the analytical model to determine how this fast trap mechanism affects performance.

There are currently two implementations of the Alewife architecture: a working hardware prototype (A-1000) and a deterministic simulator (NWO). Both of these implementations run exactly the same binary code, including all of the operating system[2] and application code. In contrast, the simulation system used for preliminary studies (ASIM) implements a wide range of architectural options but can not run actual Alewife programs.

3.1.1 ASIM: the Preliminary Simulator

ASIM models each component of the Alewife machine — from the multiprocessor software to the switches in the interconnection network — at the functional level. This simulator, which does not represent the final definition of the Alewife architecture,

[2]NWO does not run a few of the hardware diagnostics.

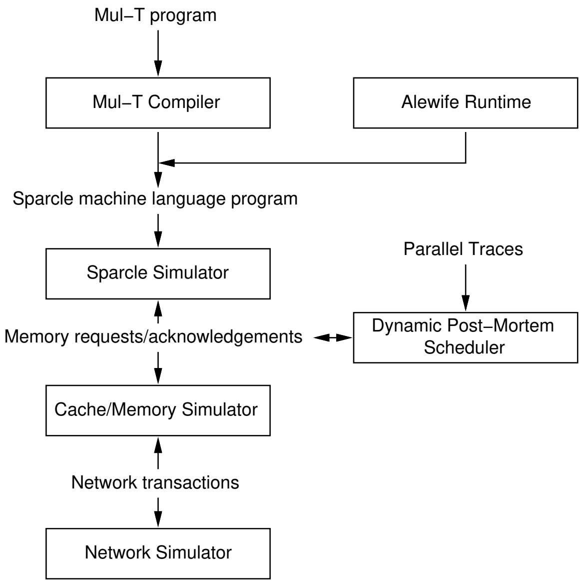


Figure 3-2: Diagram of ASIM, the Alewife system simulator.

became operational in 1990. The simulator runs programs that are written in the Mul-T language [44], optimized by the ORBIT compiler [43], and linked with a runtime system that implements both static work distribution and dynamic task partitioning and scheduling. The code generated by this process runs on a simulator consisting of processor, cache/memory, and network modules.

Although the memory accesses in ASIM are usually derived from applications running on the Sparcle processor, ASIM can alternatively derive its input from a dynamic post-mortem trace scheduler, shown on the right side of Figure 3-2. Post-mortem scheduling is a technique that generates a parallel trace from a uniprocessor execution trace with embedded synchronization information [19]. The post-mortem scheduler is coupled with the memory system simulator and incorporates feedback from the network in issuing trace requests, as described in [50]. The use of this input source is important because it allows the workload set to include large parallel applications written in a variety of styles. Section 4.2.1 describes four of the benchmarks simulated by ASIM.

The simulation overhead for large machines forces a trade-off between application size and simulated system size. Programs with enough parallelism to execute well on a large machine take an inordinate time to simulate. When ASIM is configured with its full statistics-gathering capability, it runs at about 5,000 processor cycles per second on an unloaded SPARCserver 330 and about 20,000 on a SPARCstation 10. At 5,000 processor cycles per second, a 64 processor machine runs approximately 80 cycles per second. Most of the simulations in Chapter 4 ran for roughly one million cycles (a fraction of a second on the A-1000), and took 3.5 hours to complete.

To evaluate the potential benefits of a hybrid hardware/software design approach, ASIM models only an approximation of a software-extended memory system. Section 4.2 describes this simulation technique, which justified the effort required to build the A-1000 and NWO implementations of Alewife.

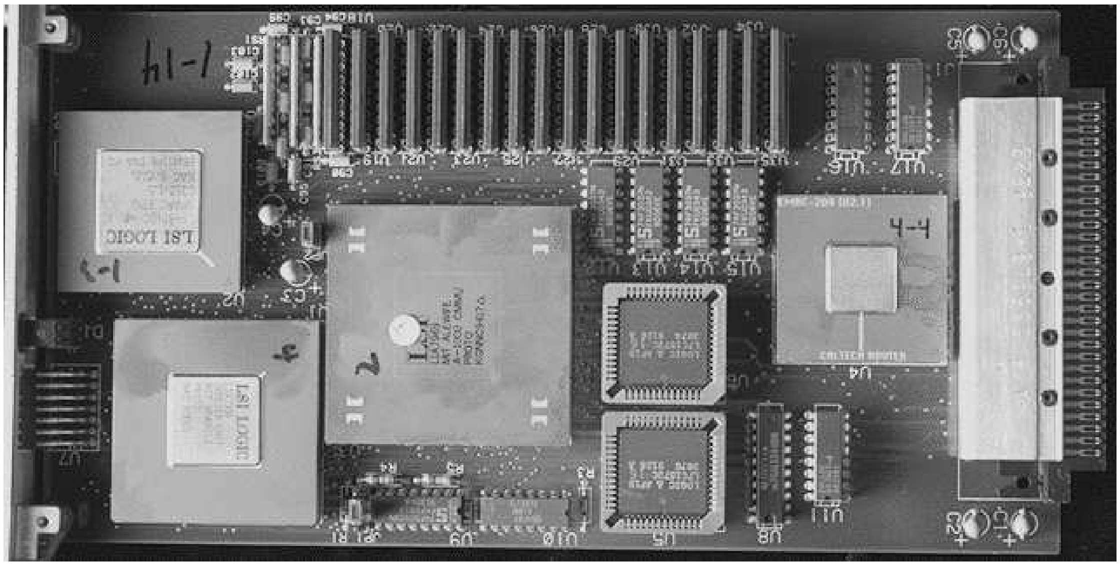


Figure 3-3: A-1000 node board: 12cm $\times$ 22cm

3.1.2 A-1000: the Alewife Machine

A-1000, the hardware implementation of the Alewife architecture, ran its first complete program in May, 1994. As of August, 1994, a sixteen node A-1000 and a few smaller machines have been built. Larger machines will come on-line as additional parts are fabricated.

Figure 3-3 shows an A-1000 node, packaged on a 12cm by 22cm printed circuit board. The large component in the center of the board is the A-1000 CMMU [46]. The chip on the lower left side of the CMMU is the Sparcle processor, and a SPARC-compatible floating-point unit is on the upper left corner of Sparcle. The connector on the right side of the board plugs into a back-plane that wires the nodes together. The Caltech EMRC [28] in the package next to the connector routes messages through a two-dimensional mesh network.

The two square packages on the right side of the CMMU hold SRAM chips that implement 64 Kbytes of direct-map cache, which holds both instructions and data. (Sparcle does not have on-board caches.) The bank of DRAM chips on the top of the board contains 2 Mbytes of private memory, 2 Mbytes of coherence directory memory, and 4 Mbytes of the global shared memory space. In addition to the 512 $\times$ 4 Mbyte configuration, the A-1000 also implements an option for 128 nodes with 16 Mbytes each.

LSI Logic fabricated the CMMU with a hybrid gate-array process. The chip contains 100,000 memory cells (including the cache tags) and a sea of 90,000 gates of logic. Critical paths in the CMMU limit the system speed. The A-1000 runs reliably at 20 MHz and may be tuned to run close to 30 MHz.

The CMMU was designed using the LSI Logic Concurrent Modular Design Environment (CMD-E). One of the outputs of CMD-E is a network file that specifies the chip's gates and connections. The LSI simulator uses switch-level models to simulate the network directly. This switch-level simulator accurately models the operation of the CMMU at about one clock cycle per second on a SPARCstation 10.

In order to provide a platform for shared memory research, the A-1000 supports dynamic reconfiguration of coherence protocols on a block-by-block basis. During both intranode and internode memory accesses, the unit of data transferred between a bank of memory and a cache is 16 bytes. This unit corresponds to the unit of memory coherence, the cache line size, and the shared-memory block size.

The A-1000 supports $Dir_nH_0S_{NB,ACK}$, $Dir_nH_2S_{NB}$ through $Dir_nH_5S_{NB}$, $Dir_5H_5S_B$, and a variety of other protocols. The node diagram in Figure 3-1 illustrates a memory block with two hardware pointers and an associated software-extended directory structure ($Dir_nH_2S_{NB}$). The current default boot sequence configures every block of shared memory with a $Dir_nH_5S_{NB}$ protocol, which uses all of the available hardware pointers.

3.1.3 NWO: the Alewife Simulator

While the machine supports an interesting range of protocols, it does not implement a full spectrum of software-extended schemes. Only a simulation system can provide the range of protocols, the deterministic behavior, and the non-intrusive observation functions that are required for analyzing the spectrum of software-extended protocols.

NWO [15] is a multi-purpose simulator that provides a deterministic debugging and test environment for the Alewife machine[3]. It ran its first complete program in the Spring of 1992. The simulator performs a cycle-by-cycle simulation of all of the components in Alewife. NWO is binary compatible with the A-1000: programs that run on the A-1000 run on the simulator *without recompilation.* In addition, NWO supports an entire range of software-extended protocols, from $Dir_nH_0S_{NB,ACK}$ to $Dir_nH_{NB}S_{-}$.

Furthermore, NWO can simulate an entire Alewife machine with up to 512 processing nodes. This performance — combined with full implementation of the Alewife architecture — allows NWO to be used to develop the software for Alewife, including parallel C and Mul-T compilers, the host interface, the runtime system, and benchmark applications. NWO provides a number of debugging and statistics functions that aid the analysis of programs and their performance on Alewife.

The first implementation of NWO runs on SPARC and MIPS-based workstations. The system simulates 3,000 Alewife clock cycles per second on a SPARCstation 10/30 and 4,700 on a SPARCstation 10/51. Each sequential run of the benchmarks in Section 3.2.2 took several hours to complete. The raw performance must be divided by the number of simulated nodes to get the actual number of simulated cycles per second. Simulating machines with large numbers of nodes (≥ 64) exceeds the physical memory on most workstations, and performance slows to a crawl due to paging.

To permit the simulation of large Alewife machines in a reasonable amount of time, NWO has been ported to Thinking Machine's CM-5 [74]. Determinism is preserved in the CM-5 implementation by executing a barrier after every simulated clock cycle. NWOP, the parallel version of NWO, has proved invaluable, especially for running

[3]NWO stands for *n*ew *w*orld *o*rder, thereby differentiating it from ASIM, the simulator that NWO replaces.

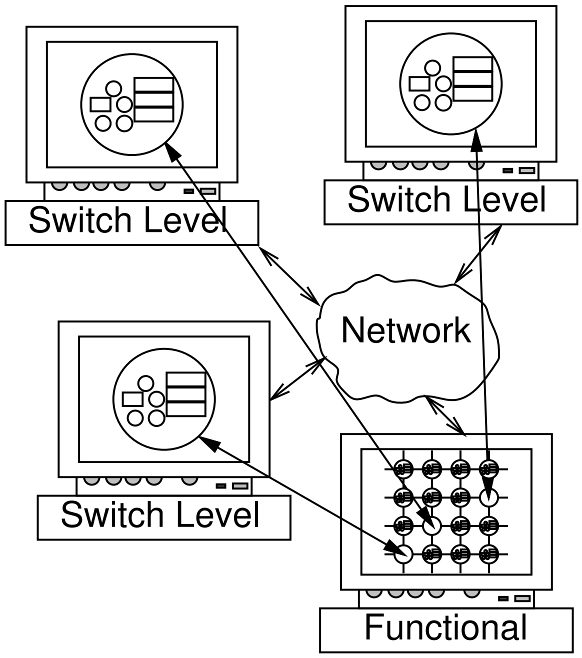


Figure 3-4: The hybrid simulation technique.

simulations of 64 and 256 node Alewife systems. The parallel simulator runs at about 15 Kcycles per second on 32 CM-5 nodes, 27 Kcycles on 64 nodes, and 48 Kcycles on 128 nodes. In addition, the large amount of memory in a CM-5 prevents the need for paging. NWOP generated most of the simulation results in Chapters 6 and 8. Many of the simulations required less than 90 minutes to run on NWOP.

In addition to its role as a valuable research and development tool, NWO performed an essential service during the design and test phases of the A-1000 development. Although NWO's functional model of the CMMU is not nearly as accurate as the LSI switch-level model, the architecture of the NWO CMMU is very similar to the architecture of the A-1000 CMMU. In fact, several finite state machine transition tables that are implemented in LSI's synthesis language are automatically compiled so that NWO can execute them directly. Thus, the internal architecture of NWO is close enough to the hardware so that the functional simulator may be used to drive the tests of the CMMU chip.

To this end, a UNIX socket interface was developed to connect the NWO and LSI simulators. This interface allows the functional simulator to model a complete Alewife machine, except for a small number of nodes, which are modeled by the switch-level simulator. Figure 3-4 illustrates this hybrid simulation technique: one workstation runs the functional simulator of the Alewife machine. The socket interface allows the same workstation or other workstations to simulate some of the Alewife nodes at the switch level. The LSI simulator can be configured to model just the CMMU or a complete Alewife node.

This hybrid simulation system allows a programmer to develop an application at the speed of the functional simulator and then to run the working application on the switch-level simulator or on the A-1000 itself. NWO remains a useful tool, even though it runs five orders of magnitude slower than the A-1000. It is still much faster (and far less expensive) to construct NWO nodes than A-1000 nodes. For this reason, a number of researchers are still using NWO and NWOP to simulate large Alewife systems. In addition, the deterministic property of the simulator and the ease of examining its state facilitate low-level code development, debugging, and research on mechanisms not implemented in the A-1000.

3.1.4 Comparing the Implementations

The NWO implementation of Alewife generated most of the experimental results in the following chapters. There are several reasons for focusing on the simulator, rather than the working hardware. First, NWO ran its first program two years before the A-1000. Running non-trivial applications on the simulator validated the architecture and stabilized the software environment in parallel with the hardware development. Second, NWO implements the whole spectrum of hybrid memory systems while the A-1000 does not. Some of the experiments critical to understanding the software-extension approach could not have been performed on the hardware. Finally, the A-1000 CMMU was primarily the work of John Kubiatowicz. It is his prerogative to experiment with the A-1000 and to report on its performance.

Nevertheless, the existence of the A-1000 validates the NWO simulation strategy. Both systems implement the Alewife architecture, and they share much of the same functional structure. The bulk of the software developed for Alewife on NWO — including the entire software-extended memory system — runs error-free on the A-1000.

There are, however, two significant differences between the Alewife implementations. First, NWO does not model the Sparcle or floating-point unit pipelines, even though it does model many of the pipelined data paths within the CMMU. Second, NWO models communication contention at the CMMU network transmit and receive queues, but does not model contention within the network switches.

Table 3.1 compares the performance of six benchmarks running on a real A-1000 node and on a simulated NWO node. (Section 3.2.2 describes these multiprocessor workloads.) The table shows the time required to run the sequential version of each benchmark, in millions of processor cycles. The NWO simulations were performed several months before the A-1000 ran its first program, so the program object codes are slightly different.

The different NWO and A-1000 floating-point implementations cause all of the difference between the running times of NWO and the A-1000. Two of the benchmarks, TSP and SMGRID, do not use any floating-point operations. Both implementations of Alewife exhibit the same running times for these two programs. Floating-point operations comprise a significant part of the AQ, EVOLVE, MP3D, and Water executions. These benchmarks run slower on the A-1000 than on NWO (in terms of Alewife cycles), because NWO models one simulated cycle per floating-point operation, while the hardware requires between two and forty-five cycles to execute floating-point operations.

The experimental error induced by NWO's single-cycle floating-point operations biases the results towards a higher communication to computation ratio. Such a bias lowers estimates of shared memory performance, because the high communication

Name	NWO Time	A-1000 Time
TSP	37 Mcycles	37 Mcycles
AQ	30 Mcycles	53 Mcycles
SMGRID	100 Mcycles	107 Mcycles
EVOLVE	44 Mcycles	63 Mcycles
MP3D	20 Mcycles	26 Mcycles
Water	16 Mcycles	23 Mcycles

Table 3.1: Sequential time required to run six benchmarks on NWO and the A-1000.

ratio increases the demand on the memory system. If anything, NWO's results for software-extended shared memory are pessimistic. On the other hand, some of the functions in Alewife's mathematics code libraries are extremely inefficient. Since this inefficiency cancels the benefit of fast floating-point operations, NWO may well represent the performance of a production version of the architecture with commercial-grade floating-point code.

Section 6.4 continues this discussion of the differences between NWO and the A-1000 by presenting performance measurements for the A-1000 software-extended memory system. To summarize the results of this study and the experience of other members of the Alewife group, qualitative conclusions derived from NWO agree with those from the A-1000.

3.2 Applications

Two kinds of workloads drive the performance analysis of software-extended memory systems. The first workload is a microbenchmark, called WORKER, that generates a synthetic memory access pattern. The synthetic access pattern allows experimentation with a cross-product of application behavior and memory-system design.

Six other benchmarks comprise the second type of workload, which is intended to test a system's performance on more typical multiprocessor programs. The benchmarks include two graph search algorithms, two solutions to numerical problems, and two physical system simulations. Of course, this small set of applications can only represent a small fraction of parallel applications. The increasing stability, size, and speed of the A-1000 will allow the small existing set of benchmarks to expand to a more representative suite of programs.

3.2.1 WORKER: a Synthetic Workload

The synthetic workload generates an arbitrary number of memory accesses that are typical of a single-program, multiple-data (SPMD) style of parallelism: the processors all run similar sequences of code, and each block of data is read by its worker set and is written by exactly one processor.

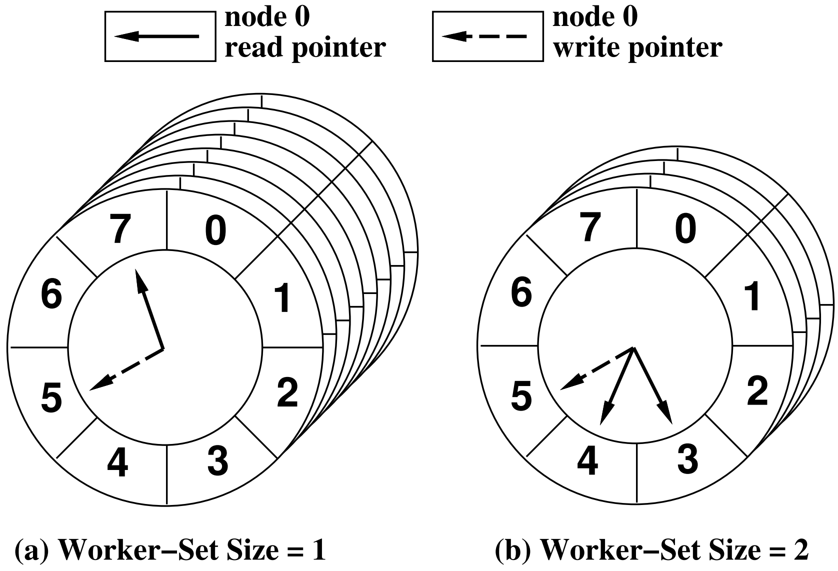


Figure 3-5: The circular worker-set data structure for 8 nodes.

The purpose of the application is to create a deterministic memory access pattern with a configurable number of processors accessing each memory block. WORKER uses a *worker-set data structure* to perform this task. The program consists of an initialization phase that builds the structure and a number of iterations that perform repeated memory accesses to it. The properties of the data structure allow a series of experiments with different worker-set sizes, but a constant number of reads per iteration.

Figure 3-5 illustrates the configuration of this structure when WORKER runs on a system with eight nodes. The basic unit is a circular set of n slots, where n is the number of nodes in the system and each slot is the size of exactly one memory block. WORKER constructs each slot's address from a single template, varying only the 9-bit node identifier within the 32-bit address. In the figure, the slots under each number are physically located in the shared-memory of the node with the corresponding identifier: node identifiers and slot numbers are synonymous. In order to control the total number of reads per iteration, the basic unit is replicated b times, so the total number of memory blocks in the structure is $n \times b$. The depth of the two structures in the figure corresponds to *b*.

The processors begin each iteration by reading the appropriate slots in each basic unit of the worker-set structure. A read offset and a worker-set size (w) determine how each processor accesses the structure. The read offset determines whether a processor reads from its local memory or not, by indicating the slot where each processor begins reading. The worker-set size corresponds to the number of slots that each processor must read. For example, the *read pointers* in Figure 3-5 indicate the slots that processor 0 reads. In Figure 3-5(a), the read offset is 7 and the worker-set size is 1; so, processor 0 reads slot 7; processor 1 reads slot 0; *etc.* In Figure 3-5(b), the read offset is 3 and the worker-set size is 2; so, processor 0 reads slots 3 and 4; processor 1 reads slots 4 and 5; *etc.* The important property of the structure is that w corresponds to the number

Name	Language	Size	Sequential
TSP	Mul-T	10 city tour	1.1 sec
AQ	Semi-C	*see text*	0.9 sec
SMGRID	Mul-T	129×129	3.0 sec
EVOLVE	Mul-T	12 dimensions	1.3 sec
MP3D	C	10,000 particles	0.6 sec
WATER	C	64 molecules	2.6 sec

Table 3.2: Characteristics of applications run on NWO. Sequential time assumes a clock speed of 33 MHz.

of processors that share each memory block. Given this structure, the total number of reads is $n \times b \times w$; thus, the number of reads for both structures (a) and (b) is 64 even though the worker set sizes of the structures are different.

After the reads, the processors execute a barrier and then each performs a single write to one slot in every basic unit of the structure. A write offset indicates the slot that each processor writes, and therefore determines whether or not the write modifies a local or remote memory block. Both of the worker-set structures in Figure 3-5 have a write offset of 5; so, processor 0 always writes to blocks in the memory of processor 5. The total number of writes to the structure during this phase is $n \times b$.

Finally, the processors execute a barrier and continue with the next iteration. Every read request causes a cache miss and every write request causes the directory protocol to send exactly one invalidation message to each reader. This memory access pattern provides a controlled experiment for comparing the performance of different protocols.

3.2.2 NWO/A-1000 Benchmarks

Table 3.2 lists the names and characteristics of the applications in the benchmark suite. They are written in C, Mul-T [44] (a parallel dialect of LISP), and Semi-C [38] (a language akin to C with support for fine-grain parallelism). The table specifies the sequential running time derived from simulation on NWO and scaled to real time using a hypothetical clock speed of 33 MHz[4].

Each application (except MP3D) is studied with a problem size that realizes more than 50% processor utilization on a simulated 64 node machine with a full-map directory. The problem set sizes chosen for the applications permit the NWO simulations to complete in a reasonable amount of time, but represent much smaller data sets than would be used in practice. Since the problem set sizes are relatively small, the following chapters use only the parallel sections to measure speedups. Now that a 16-node A-1000 exists, work is underway to increase the number of benchmarks and to run them with more typical problem sizes.

[4]The running times for Water in Table 3.2 and Table 3.1 differ, due to a change in the compiler during the months between the two experiments.

Traveling Salesman Problem TSP solves the traveling salesman problem using a branch-and-bound graph search. The application is written in Mul-T and uses the `future` construct to specify parallelism. In order to ensure that the amount of work performed by the application is deterministic, the experiments seed the best path value with the optimal path. This program is by no means an optimal implementation of the traveling salesman problem: the study uses it to represent the memory access pattern of simple, parallel, graph search algorithms.

Adaptive Quadrature AQ performs numerical integration of bivariate functions using adaptive quadrature. The core of the algorithm is a function that integrates the range under a curve by recursively calling itself to integrate sub-ranges of that range. This application also uses the `future` construct to specify parallelism, and it exhibits predominantly producer-consumer communication between pairs of nodes. The function used for this study is x^4y^4, which is integrated over the square $((0,0),(2,2))$ with an error tolerance of 0.005.

Static Multigrid SMGRID uses the multigrid method to solve elliptical partial differential equations [32]. The algorithm consists of performing a series of Jacobi-style iterations on multiple grids of varying sizes. Instead of using the default Mul-T task scheduler, this application specifies its own static task-to-processor mapping.

Genome Evolution EVOLVE is a graph traversal algorithm for simulating the evolution of genomes, which is reduced to the problem of traversing a hypercube and finding local and global maxima. The application searches for a path from the initial conditions to a local fitness maximum. The program is rather small, but the graph traversal causes it to have an interesting shared-memory access pattern. In order to eliminate startup effects from the measurements, the algorithm is run twice: speedups are measured only for the second iteration. Experiments show that speedups for the entire run are only slightly lower than just for the second iteration.

MP3D The MP3D application is part of the SPLASH parallel benchmark suite. According to the SPLASH documentation [71],

> MP3D solves a problem in rarefied fluid flow simulation. Rarefied flow problems are of interest to aerospace researchers who study the forces exerted on space vehicles as they pass through the upper atmosphere at hypersonic speeds. Such problems also arise in integrated circuit manufacturing simulation and other situations involving flow at extremely low density.

MP3D is commonly known as a difficult multiprocessor workload. It has a high communication to computation ratio, and achieves only modest speedups on most parallel architectures. The simulations in Chapter 6 use a problem size of 10,000 particles, turn the locking option off, and augment the standard `p4` macros with Alewife's parallel C library [59]. The simulations run five time steps of the application.

Water The Water application, also from the SPLASH application suite, is run for one time step with 64 molecules. The documentation describes Water as follows:

> This N-body molecular dynamics application evaluates forces and potentials in a system of water molecules in the liquid state. The computation is performed over a user-specified number of time-steps. . . Every time-step involves setting up and solving the Newtonian equations of motion for water molecules in a cubical box with periodic boundary conditions. . .

In addition to the `p4` macros, this version of Water uses Alewife's parallel C library with barriers and reductions, rather than the naïve locks used in the standard version.

Chapter 4

Cache Coherence in Alewife

This work on the foundation of the Alewife architecture investigates a range of alternatives for designing cache coherence protocols. The results of this study indicate that no simple, purely-hardware scheme achieves both scalable performance and cost efficiency. Consequently, this chapter proposes the software-extension approach for implementing LimitLESS cache coherence in Alewife. Evaluation of the approach provides the first evidence that supports the concept of a software-extended coherent shared memory. ASIM, the Alewife simulator, generates this evidence by approximating the effects of software-extension without actually implementing the details of the architecture.

4.1 Implementing Directory Protocols

The main thrust of the original memory system experimentation involves the simulation and measurement of several different directory-based protocols, including full-map, limited, and chained varieties. This analysis helps determine the relationship between the implementation of a protocol's directory structure and the performance of a shared-memory system.

The most dramatic differences in performance between protocols is caused by the structure of the directory. For applications that use variables with small worker sets, all of the protocols perform similarly. On the other hand, applications with variables that are shared by many processors exhibit behavior that correlates with the type of directory used by the protocol. Except in anomalous situations, the full-map directory ($Dir_nH_{NB}S_-$) performs better than any other directory-based protocol. This observation should not be surprising, since the full-map protocol is also not scalable in terms of memory overhead. By committing overwhelming resources to cache coherence, it is always possible to achieve good performance.

Simulations show that limited directory protocols can perform as well as full-map directory protocols, subject to optimization of the software running on a system [14]. Although this result testifies to the fact that scalable cache coherence is possible, limited directories are extremely sensitive to the worker sets of a program's variables. Section 4.2 examines a case-study of a multiprocessor application that — when properly modified — runs approximately as fast with a limited directory as with a full-map directory.

However, when one variable in the program is widely shared, limited directory protocols cause more than a 100% increase in time needed to finish executing the application. This sensitivity to worker-set sizes varies with the program running on the system; but in general, the more variables that are shared among many processors, the worse limited directories perform.

4.1.1 LimitLESS Cache Coherence

Alewife's LimitLESS directory protocol uses the software-extension approach to solve the problem of implementing scalable cache coherence. As do limited directory protocols, the LimitLESS directory scheme capitalizes on the observation that only a few shared memory data types are widely shared among processors. Many shared data structures have a small worker set, which is the set of processors that concurrently read a memory location. The worker set of a memory block corresponds to the number of active pointers it would have in a full-map directory entry. When running properly optimized software, a directory entry overflow is an exceptional condition in the memory system. The LimitLESS protocol handles such "protocol exceptions" in software. This is the integrated systems approach — handling common cases in hardware and exceptional cases in software.

The LimitLESS scheme implements a small number of hardware pointers for each directory entry. If these pointers are not sufficient to store the locations of all of the cached copies of a given block of memory, then the state machine that handles the directory interrupts the local processor. The processor then emulates a full-map directory (or possibly a more intelligent protocol) for the block of memory that caused the interrupt. The structure of the Alewife machine supports an efficient implementation of this memory system extension. Since each processing node in Alewife contains both a communications and memory management unit (CMMU) and a processor, it is a reasonable modification of the architecture to couple the responsibilities of these two units. This scheme is called LimitLESS, to indicate that it employs a *Limit*ed directory that is *L*ocally *E*xtended through *S*oftware *S*upport. Figure 3-1, an enlarged view of a node in the Alewife machine, depicts a set of directory pointers that correspond to shared data block *X*, copies of which exist in several caches. In the figure, the software has extended the directory pointer array (which is shaded) into private memory.

Since Alewife's Sparcle processor is designed with a fast trap mechanism, the overhead of the LimitLESS interrupt is not prohibitive. The emulation of a full-map directory in software prevents the LimitLESS protocol from exhibiting the sensitivity to software optimization that is exhibited by limited directory schemes. But given current technology, the delay needed to emulate a full-map directory completely in software is significant. Consequently, the LimitLESS protocol supports small worker sets of processors in its limited directory entries, implemented in hardware.

4.1.2 A Simple Model of the LimitLESS Protocol

Before discussing the details of the software-extended coherence scheme, it is instructive to examine a simple model of the relationship between the performance of a full-map directory and the LimitLESS directory scheme. Let $T_{r,hw}$ be the average remote memory access latency for a $Dir_n H_{NB} S_-$ (hardware) directory protocol. $T_{r,hw}$ encapsulates factors such as the delay in the CMMU, invalidation latencies, and network latency. Given the hardware protocol latency, $T_{r,hw}$, it is possible to estimate the average remote memory access latency for the LimitLESS protocol with the formula: $T_{r,hw} + sT_s$, where T_s (the software latency) is the average delay for the full-map directory emulation interrupt, and s is the fraction of remote memory accesses that overflow the small set of pointers implemented in hardware.

For example, ASIM simulations of a Weather Forecasting program running on 64 node system (see Section 4.2) indicate that $T_{r,hw} \approx 35$ cycles. If $T_s = 100$ cycles, then remote accesses with the LimitLESS scheme will be 10% slower (on average) than with the full-map protocol when $s \approx 3\%$. Since the Weather program is, in fact, optimized such that 97% of accesses to remote data locations "hit" in the limited directory, the full-map emulation will cause a 10% delay in servicing requests for data. Chapter 7 elaborates this model, and uses it to investigate a range of options in designing software-extended systems.

4.1.3 Background: Implementing a Full-Map Directory

Since the LimitLESS coherence scheme is a hybrid of the full-map and limited directory protocols, this new cache coherence scheme may be studied in the context of its predecessors. In the case of a full-map directory, one pointer for every cache in the multiprocessor is stored, along with the state of the associated memory block, in a single directory entry. The directory entry, illustrated in Figure 4-1, is physically located in the same node as the associated data. Since there is a one-to-one mapping between the caches and the pointers, the full-map protocol optimizes the size of the pointer array by storing just one bit per cache. A pointer-bit indicates whether or not the corresponding cache has a copy of the data. The ASIM implementation of the protocol allows a memory block to be in one of four states, which are listed in Table 4.1.

These states are mirrored by the state of the block in the caches, also listed in Table 4.1. It is the responsibility of the protocol to keep the states of the memory and cache blocks coherent. For example, a block in the Read-Only state may be shared by a number of caches (as indicated by the pointer array). Each of these cached copies are marked with the Read-Only cache state to indicate that the local processor may only read the data in the block.

Before any processor modifies a block in an Invalid or Read-Only cache state, it first requests permission from the CMMU that manages the data. At this point, the CMMU sends invalidations to each of the cached copies. The caches then invalidate the copy (change the block's state from Read-Only to Invalid), and send an acknowledgment message back to the memory. The CMMU uses the Write-Transaction state to indicate that a memory location is awaiting acknowledgments, and sets a pointer to designate the cache that initiated the request. A CMMU mechanism for counting the number of invalidations sent and the number of acknowledgments received allows invalidations and acknowledgments to travel through the system's interconnection network in parallel. When the CMMU receives the appropriate number of acknowledgments, it changes the

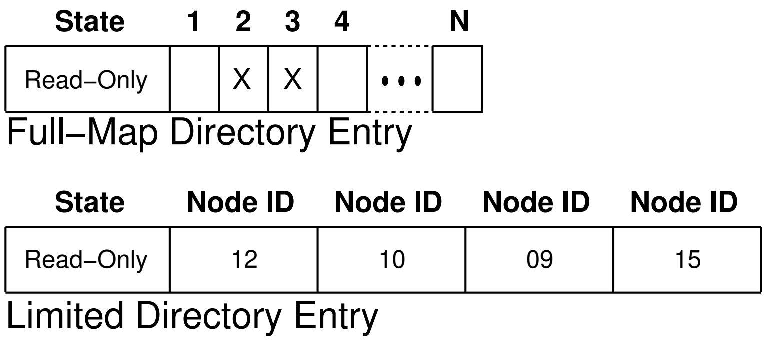


Figure 4-1: Full-map and limited directory entries. The full-map pointer array is optimized as a bit-vector. The limited directory entry has four pointers.

Component	Name	Meaning
Memory	Read-Only	Caches have read-only copies of the data.
	Read-Write	One cache has a read-write copy of the data.
	Read-Transaction	Holding read request, update is in progress.
	Write-Transaction	Holding write request, invalidation is in progress.
Cache	Invalid	Cache block may not be read or written.
	Read-Only	Cache block may be read, but not written.
	Read-Write	Cache block may be read or written.

Table 4.1: Directory states.

state of the block to Read-Write and sends a write permission message to the cache that originated the transaction. In a sense, the cache "owns" the block until another cache requests access to the data.

Changing a block from the Read-Write to the Read-Only state involves an analogous (but slightly simpler) transaction. When a cache requests to read a block that is currently in the Read-Write state, the CMMU sends an update request to the cache that owns the data. The CMMU marks a block that is waiting for data with the Read-Transaction state. As in the Write-Transaction state, a pointer is set to indicate the cache that initiated the read transaction. When a cache receives an update request, it invalidates its copy of the data, and replies to memory with an update message that contains the modified data, so that the original read request can be satisfied.

The protocol might be modified so that a cache changes a block from the Read-Write to the Read-Only (instead of the Invalid) state upon receiving an update request. Such a modification assumes that data that is written by one processor and then read by another will be read by both processors after the write. While this modification optimizes the protocol for frequently-written and widely-shared variables, it increases the latency of access to migratory or producer-consumer data types.

The dilemma of choosing between these two types of data raises an important question: Should a cache coherence protocol optimize for frequently-written and widely-shared data? This type of data requires excessive bandwidth from a multiprocessor's interconnection network, whether or not the system employs caches to reduce the *average* memory access latency. Since the problems of frequently-written and widely-shared data are independent of the coherence scheme, it seems futile to try to optimize accesses to this type of data, when the accesses to other data types can be expedited. This decision forces the onus of eliminating the troublesome data type on the multiprocessor software designer. However, the decision seems reasonable in light of the physical limitations of communication networks. Thus, Alewife always invalidates data when changing a block from Read-Write to Read-Only.

Since this study, two independent groups of researchers have found an answer to the dilemma [63, 25]. Instead of binding either an update or an invalidate policy into hardware, they propose cache coherence schemes that adapt between invalidate and update policies. These schemes require extra bits to implement the directory state, and achieve slightly better performance than either static scheme.

The basic protocol that is described above is somewhat complicated by the associativity of cache lines. In a cache, more than one memory block can map to a single block of storage, called a line. Depending on the memory access pattern of its processor, a cache may need to replace a block of memory with another block of memory that must be stored in the same cache line. While the protocol must account for the effect of replacements to ensure a coherent model of shared memory, in systems with large caches, replacements are rare events except in pathological memory access patterns. Several options for handling the replacement problem in various coherence protocols have been explored.

Simulations show that the differences between these options do not contribute significantly to the bottom-line performance of the coherence schemes, so the final choice of replacement handling for the Alewife machine optimizes for the simplicity of the protocol (in terms of cache states, memory states, and the number of messages): when a cache replaces a block in the Read-Only state, it does not notify the block's home location; when a cache replaces a block in the Read-Write state, it transmits the dirty data back to the home node.

4.1.4 Specification of the LimitLESS Scheme

The model in Section 4.1.2 assumes that the hardware latency ($T_{r,hw}$) is approximately equal for the full-map and the LimitLESS directories, because the LimitLESS protocol has the same state transition diagram as the full-map protocol. The memory side of this protocol is illustrated in Figure 4-2, which contains the memory states listed in Table 4.1. Both the full-map and the LimitLESS protocols enforce coherence by transmitting messages (listed in Table 4.3[1]) between the CMMUs. Every message contains the address of a memory block, to indicate which directory entry should be used when processing the message. Table 4.3 also indicates whether a message contains the data associated with a memory block.

The state transition diagram in Figure 4-2 specifies the states, the composition of the pointer set (P), and the transitions between the states. This diagram specifies a simplified version of the protocol implemented in Alewife. For the purposes of

[1]This table lists (and the rest of this section discusses) the messages in the current Alewife implementation, rather than in ASIM.

describing the implementation of directory protocols, Figure 4-2 includes only the core of the full-map and LimitLESS protocols. Unessential optimizations and other types of coherence schemes have been omitted to emphasize the important features of the coherence schemes. See [46, 49] for more details about Alewife's protocols.

Each transition in the diagram is labeled with a number that refers to its specification in Table 4.2. This table annotates the transitions with the following information: 1. The *input message* from a cache that initiates the transaction and the identifier of the cache that sends it. 2. A *precondition* (if any) for executing the transition. 3. Any *directory entry change* that the transition may require. 4. The *output message* or messages that are sent in response to the input message. Note that certain transitions require the use of an acknowledgment counter (*AckCtr*), which is used to ensure that cached copies are invalidated before allowing a write transaction to be completed. The Alewife CMMU stores this counter in the second pointer of the appropriate directory entry.

For example, Transition 2 from the Read-Only state to the Read-Write state is taken when cache *i* requests write permission (WREQ) and the pointer set is empty or contains just cache *i* ($P = \{\}$ or $P = \{i\}$). In this case, the pointer set is modified to contain *i* (if necessary) and the CMMU issues a message containing the data of the block to be written (WDATA).

Following the notation in Section 2.1.3, both full-map and LimitLESS are members of the $Dir_n H_X S_{Y,A}$ class of cache coherence protocols. Therefore, from the point of view of the protocol specification, the LimitLESS scheme does not differ substantially from the full-map protocol. In fact, the LimitLESS protocol is also specified by Figure 4-2. The extra notation on the Read-Only ellipse ($S : n > p$) indicates that the state is handled in software when the size of the pointer set (n) exceeds the size of the limited directory entry (p). In this situation, the transitions with the shaded labels (1, 2, and 3) are executed by the interrupt handler on the processor that is local to the overflowing directory. When the protocol changes from a software-handled state to a hardware-handled state, the processor must modify the directory state so that the CMMU can resume responsibility for the protocol transitions.

The hardware mechanisms that are required to implement the software-extended protocols are as follows:

1. A fast interrupt mechanism: A processor must be able to interrupt application code and switch to software-extension rapidly. This ability makes the overhead of emulating a full-map directory (T_s) small, and thus makes the LimitLESS scheme competitive with schemes that are implemented completely in hardware.

2. Processor to network interface: In order to emulate the protocol functions normally performed by the hardware directory, the processor must be able to send and to receive messages from the interconnection network.

3. Extra directory state: Each directory entry must hold the extra state necessary to indicate whether the processor is holding overflow pointers.

In Alewife, none of these mechanisms exist exclusively to support the LimitLESS protocol. The Sparcle processor uses the same mechanism to execute an interrupt

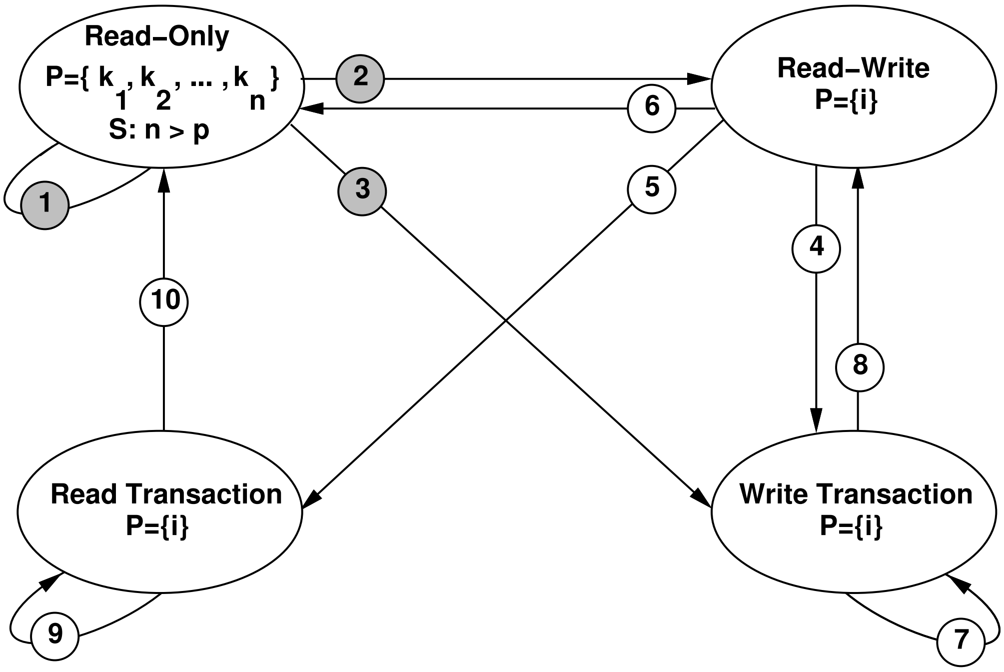


Figure 4-2: Directory state transition diagram for the full-map and LimitLESS coherence schemes.

#	Input Message	Precondition	Directory Entry Change	Output Message(s)
1	$i \rightarrow$ RREQ	—	$P = P \cup \{i\}$	RDATA $\rightarrow i$
2	$i \rightarrow$ WREQ	$P = \{i\}$	—	WDATA $\rightarrow i$
	$i \rightarrow$ WREQ	$P = \{\}$	$P = \{i\}$	WDATA $\rightarrow i$
3	$i \rightarrow$ WREQ	$P = \{k_1, \ldots, k_n\} \wedge i \notin P$	$P = \{i\}$, AckCtr $= n - 1$	$\forall k_j$ INVR $\rightarrow k_j$
	$i \rightarrow$ WREQ	$P = \{k_1, \ldots, k_n\} \wedge i \in P$	$P = \{i\}$, AckCtr $= n - 2$	$\forall k_j \neq i$ INVR $\rightarrow k_j$
4	$j \rightarrow$ WREQ	$P = \{i\}$	$P = \{j\}$	INVW $\rightarrow i$
5	$j \rightarrow$ RREQ	$P = \{i\}$	$P = \{j\}$	INVW $\rightarrow i$
6	$i \rightarrow$ UPDATE	$P = \{i\}$	$P = \{\}$	—
7	$j \rightarrow$ RREQ	—	—	BUSY $\rightarrow j$
	$j \rightarrow$ WREQ	—	—	BUSY $\rightarrow j$
	$j \rightarrow$ ACKC	AckCtr $\neq 0$	AckCtr = AckCtr − 1	—
8	$j \rightarrow$ ACKC	AckCtr = 0, $P = \{i\}$	—	WDATA $\rightarrow i$
	$j \rightarrow$ UPDATE	$P = \{i\}$	—	WDATA $\rightarrow i$
9	$j \rightarrow$ RREQ	—	—	BUSY $\rightarrow j$
	$j \rightarrow$ WREQ	—	—	BUSY $\rightarrow j$
10	$j \rightarrow$ ACKC	$P = \{i\}$	—	RDATA $\rightarrow i$
	$j \rightarrow$ UPDATE	$P = \{i\}$	—	RDATA $\rightarrow i$

Table 4.2: Annotation of the state transition diagram.

Type	Symbol	Name	Data?
Cache to Memory	RREQ	Read Request	
	WREQ	Write Request	
	UPDATE	Update	$\surd$
	ACKC	Acknowledge Invalidate	
Memory to Cache	RDATA	Read Data	$\surd$
	WDATA	Write Data	$\surd$
	INVR	Invalidate Read Data	
	INVW	Invalidate Write Data	
	BUSY	Busy Signal	

Table 4.3: Cache coherence protocol messages.

Type	Symbol	Name	Data?
Cache to Memory	MREQ	Modify Request	
	REPU	Replace Unmodified	
Memory to Cache	MODG	Modify Granted	

Table 4.4: Optional protocol messages.

quickly as it uses to provide a fast context-switch. The processor to network interface is implemented through the interprocessor interrupt (IPI) mechanism, which is designed to increase Alewife's I/O performance and to provide a generic message-passing primitive. A small extension to the extra directory state required for the LimitLESS protocol allows experimentation with a range of memory systems, including the software-only directories ($Dir_n H_0 S_{NB,ACK}$) and the smart memory systems in Chapter 8.

4.1.5 Second-Order Considerations

In addition to the protocol features that have a primary impact on the performance of a cache coherence scheme, there are a number of secondary implementation details that also contribute to the speed of the memory system. Examples of such details include several protocol messages that are not essential for ensuring a memory model. While these features may be interesting from the point of view of protocol design, they have only a small (but not insignificant) effect on the system as a whole.

The messages that are used by the hardware coherence protocols to keep the cache and the memory states consistent are listed in Table 4.3. The *Data?* column indicates the three messages that contain the data of the shared memory block. Table 4.4 lists three optional messages that are not essential to ensure cache coherence. Although the messages have mnemonic names, it is worth explaining their meaning: The RREQ message is sent when a processor requests to read a block of data that is not contained in its cache. The RDATA message is the response to RREQ, and contains the data needed by the processor. The WREQ and WDATA messages are the request/response pair for processor write requests. Since more than one memory word is stored in a cache line, the WDATA message contains a copy of the data in memory.

The MREQ and MODG messages are used to service processor write requests when the cache contains a Read-Only copy of the data to be written. In this case, the processor does not need a copy of the data, so the MODG message does not contain the block of data. MREQ and MODG are really just an optimization of the WREQ and WDATA message combination for the limited directory protocol. This message pair is not essential to a protocol, because it is always *correct* to send a WDATA instead of a MODG message.

It is not obvious that the extra complications needed to implement the MODG message are justified by its performance benefits. The modify request and grant message pair optimizes for data locations that are read by a processor and then immediately written. This optimization is especially important during cold-start periods when an application's working set does not reside in its cache. However, if the protocol needs to send an invalidation message to a cache before completing the write transaction, it is necessary for the directory to store a bit of state that indicates whether the initial request was a WREQ or a MREQ. Due to the complications caused by these messages, they are not included in the transition state diagram, even though they are implemented in Alewife.

The INVR and ACKC message combination is used to purge Read-Only copies of cached data. In the limited directory scheme, when a Read-Only memory block receives a WREQ message, the CMMU sends one INVR message to each cache with a pointer in the directory. When a cache receives the INVR message, it invalidates the appropriate cache line (if the cache tag matches the message's address), and responds with an ACKC. In the full-map and limited protocols, the CMMU may send one INVR message on each cycle, so several INVR messages with the same address may be working their way through the network at the same time.

The CMMU increments the acknowledgment counter when it transmits an INVR message and decrements the counter when it receives an ACKC message. Thus, the counter remembers the total number of ACKC messages that it expects to receive. The counter waits for the acknowledgment counter to reach zero before responding to the initial WREQ message to ensure sequential consistency. To limit the amount of state that must be stored during a write transaction, the CMMU responds with a BUSY signal to any RREQ or WREQ messages to the memory block while invalidations are in progress for a memory block. If a CMMU accepts a RREQ or WREQ message, then the protocol guarantees to satisfy the request eventually. However, if a node receives a BUSY signal, then it must retry the request.

The INVW and UPDATE messages are used to return modified data to memory. If a CMMU receives a RREQ message for a data block in the Read-Write state, the CMMU sends an INVW message to the node that currently has permission to write the data block. When this node receives the INVW message, it responds with an UPDATE message containing the modified data, rather than with an ACKC message, because the cached block is in the Read-Write state. At the same time, the cache invalidates the line that contains the data.

Since multiple addresses map to each cache line, a cache sometimes needs to *replace* one cached block of data with another. If a replaced block is in the Read-Write state, then the UPDATE message is used to send the modified data back to memory. Otherwise, the data is *u*nmodified, and the REPU message is used to notify the directory about the replacement.

The REPU message is optional in the limited and full-map protocols. If a cache replaces a Read-Only copy of data but does not notify the directory, then it may receive a spurious INVR message for the block at some point in the future. However, the address in the INVR will not match the tag in the cache, so the node may acknowledge the spurious invalidation without invalidating the currently cached data. On the other side of the memory system, if a directory receives a RREQ message from a cache that already has a pointer, then it responds with a RDATA message. So, the REPU message may save an INVR message, or it may create unnecessary network traffic. In order to examine the effects of the REPU message, ASIM has been instrumented with an option that selects whether or not the current coherence protocol uses the message.

4.1.6 Evaluation of Secondary Protocol Features

None of the protocol features discussed in this section exhibit more than a ten percent variation in execution time on ASIM. This behavior is expected, because the unessential components of protocols tend to interact with relatively infrequent events, such as cache line replacement or cold-start data accesses. Such low performance returns suggest that issues of complexity and cost can be used to decide whether or not to implement unessential protocol messages. Certain protocol messages may be rejected out-of-hand. For example, the replace unmodified (REPU) message sometimes degrades performance due to an increase in network traffic. Thus, Alewife does not implement REPU, thereby saving the extra complexity and cost of this message.

On the other hand, the modify request/grant (MREQ/MODG) message pair can increase performance by over five percent. While this performance gain does not justify the extra directory state needed to store the modify request during invalidations, it does imply that a simplified version of the feature would be appropriate. Accordingly, the Alewife CMMU responds to a MREQ with a MODG only when no invalidations need to be sent. This simplification eliminates most of the extra cost of the modify grant optimization, while retaining the benefits of reduced latency for simple memory transactions that consist of a read request followed by a write request.

4.2 Preliminary Evaluation

This section documents the preliminary evidence that justified the effort required to build a complete software-extended system. The study demonstrates only the plausibility of the software-extension approach, leaving the implementation and evaluation of an actual system for the following chapters.

For the purpose of estimating the performance of LimitLESS directories, an approximation of the protocol was implemented in ASIM. The technique assumes that the overhead of the LimitLESS full-map emulation interrupt is approximately the same for all memory requests that overflow a directory entry's pointer array. This overhead is the T_s parameter described in Section 4.1.2. During the simulations, ASIM models an ordinary full-map protocol ($Dir_n H_{NB} S_-$). When the simulator encounters a pointer array overflow, it stalls both the CMMU and the processor that would handle the LimitLESS interrupt for T_s cycles. While this evaluation technique only approximates the actual behavior of a fully-operational LimitLESS scheme, it is a reasonable method for determining whether to expend the greater effort needed to implement the complete protocol.

For applications that perform as well with a limited directory as with a full-map directory, the LimitLESS directory causes little degradation in performance. When limited directories perform significantly worse than a full-map directory, the LimitLESS scheme tends to perform about as well as full-map, depending on the number of widely-shared variables. If a program has just one or two widely-shared variables, a LimitLESS protocol avoids hot-spot contention that tends to destroy the performance of limited directories. On the other hand, the performance of the LimitLESS protocol degrades when a program utilizes variables that are both widely-shared and frequently written. But as discussed in previous sections, these types of variables tend to exhaust the bandwidth of the interconnection network, no matter what coherence scheme is used by the memory system.

In general, the ASIM results indicate that the LimitLESS scheme approaches the performance of a full-mapped directory protocol with the memory efficiency of a limited directory protocol. The success of this new coherence protocol emphasizes two key principles: first, the software-extension approach can successfully be applied to the design of a shared-memory system. Second, the implementation of a protocol's directory structure correlates closely with the performance of the memory system as a whole.

4.2.1 Results

Table 4.5 shows the simulated performance of four applications, using a four-pointer limited protocol ($Dir_4H_{NB}S_-$), a full-map protocol ($Dir_nH_{NB}S_-$), and a LimitLESS ($Dir_nH_4S_{NB}$) scheme with $T_s = 50$. The table presents the performance of each application/protocol combination in terms of the time needed to run the program, in millions of processor cycles. All of the runs simulate a 64-node Alewife machine.

Multigrid is an early version of the program described in Section 3.2.2, Weather forecasts the state of the atmosphere given an initial state, SIMPLE simulates the hydrodynamic and thermal behavior of fluids, and Matexpr performs several multiplications and additions of various sized matrices. The computations in Matexpr are partitioned and scheduled by a compiler. The Weather and SIMPLE applications are measured using dynamic post-mortem scheduling of traces, while Multigrid and Matexpr are run on complete-machine simulations.

Since the LimitLESS scheme implements a full-fledged limited directory in hardware, applications that perform well using a limited scheme also perform well using LimitLESS. Multigrid is such an application. All of the protocols require approximately the same time to complete the computation phase. This evidence confirms the assumption that for applications with small worker sets, such as Multigrid, the limited (and therefore the LimitLESS) directory protocols perform almost as well as the full-map protocol. [14] has more evidence of the general success of limited directory protocols.

The SIMPLE application indicates the performance of LimitLESS under extreme conditions: this version of the applications uses a barrier synchronization implemented

Application	$Dir_4H_{NB}S_-$	$Dir_nH_4S_{NB}$	$Dir_nH_{NB}S_-$
Multigrid	0.729	0.704	0.665
SIMPLE	3.579	2.902	2.553
Matexpr	1.296	0.317	0.171
Weather	1.356	0.654	0.621

Table 4.5: Performance for three coherence schemes running on 64 ASIM nodes, in terms of millions of cycles.

using a single lock (rather than a software combining tree). Although the worker sets in SIMPLE are small for the most part, the globally shared barrier structure causes the performance of the limited directory protocol to suffer. In contrast, the LimitLESS scheme performs almost as well as the full-map directory protocol, because LimitLESS is able to distribute the barrier structure to as many processors as necessary.

The Matexpr application uses several variables that have worker sets of up to 16 processors. Due to these large worker sets, the processing time with the LimitLESS scheme is almost double that with the full-map protocol. The limited protocol, however, exhibits a much higher sensitivity to the large worker sets.

Weather provides a case-study of an application that has not been completely optimized for limited directory protocols. Although the simulated application uses software combining trees to distribute its barrier synchronization variables, Weather has one variable initialized by one processor and then read by all of the other processors. Additional ASIM simulations show that if this variable is flagged as read-only data, then a limited directory performs just as well for Weather as a full-map directory.

However, it is easy for a programmer to forget to perform such optimizations, and there are some situations where it is very difficult to avoid this type of sharing. Figure 4-3 gives the execution times for Weather when this variable is not optimized. The vertical axis on the graph displays several coherence schemes, and the horizontal axis shows the program's total execution time (in millions of cycles). The results show that when the worker set of a single location in memory is much larger than the size of a limited directory, the whole system suffers from hot-spot access to this location. So, limited directory protocols are extremely sensitive to the size of a heavily-shared data block's worker set.

The effect of the unoptimized variable in Weather was not evident in the initial evaluations of directory-based cache coherence for Alewife [14], because the network model did not account for hot-spot behavior. Since the program can be optimized to eliminate the hot-spot, the new results do not contradict the conclusion of [14] that system-level enhancements make large-scale cache-coherent multiprocessors viable. Nevertheless, the experience with the Weather application reinforces the belief that complete-machine simulations are necessary to evaluate the implementation of cache coherence.

As shown in Figure 4-4, the LimitLESS protocol avoids the sensitivity displayed by limited directories. This figure compares the performance of a full-map directory, a four-pointer limited directory ($Dir_4H_{NB}S_-$), and the four-pointer LimitLESS ($Dir_nH_4S_{NB}$) protocol with several values for the additional latency required by the LimitLESS proto-

Figure 4-3: Limited and full-map directories, 64 ASIM nodes.

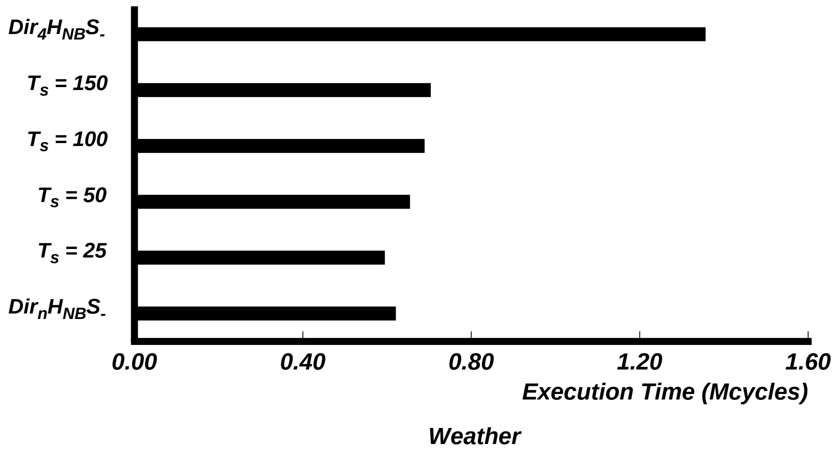


Figure 4-4: LimitLESS $Dir_nH_4S_{NB}$, 25 to 150 cycle emulation latencies.

col's software ($T_s = 25, 50, 100,$ and 150). The execution times show that the LimitLESS protocol performs about as well as the full-map directory protocol, even in a situation where a limited directory protocol does not perform well. Furthermore, while the LimitLESS protocol's software should be as efficient as possible, the performance of the LimitLESS protocol is not strongly dependent on the latency of the full-map directory emulation.

Section 6.1 shows that a software read request handler implemented with the flexible coherence interface requires 400 cycles of processing. Amortized over 4 directory pointers, this processing time (T_s) is 100 cycles per request. A hand-tuned, assembly language implementation of the software requires about 150 cycles of processing, or less than 40 cycles per request.

It is interesting to note that the LimitLESS protocol, with a 25 cycle emulation latency, actually performs better than the full-map directory. This anomalous result is caused by the participation of the processor in the coherence scheme. By interrupting and slowing down certain processors, the LimitLESS protocol produces a slight back-off

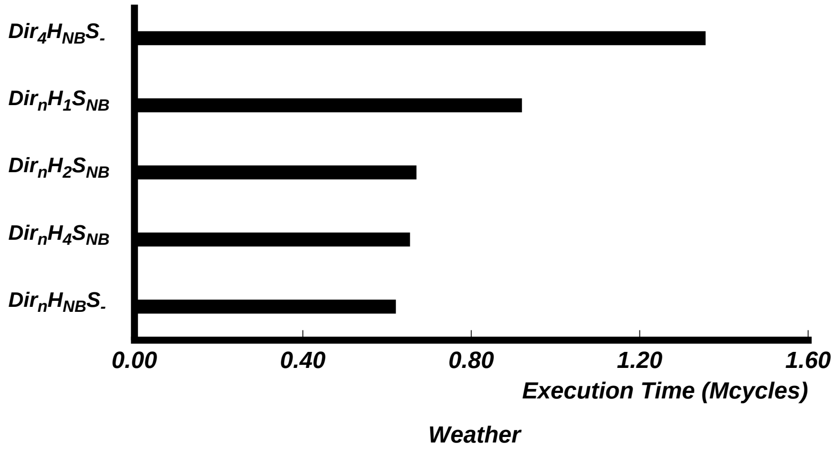


Figure 4-5: LimitLESS with 1, 2, and 4 hardware pointers.

effect that reduces contention.

The number of pointers that a LimitLESS protocol implements in hardware interacts with the worker-set size of data structures. Figure 4-5 compares the performance of Weather with a full-map directory, a limited directory, and LimitLESS directories with 50 cycle emulation latency and one ($Dir_nH_1S_{NB}$), two ($Dir_nH_2S_{NB}$), and four ($Dir_nH_4S_{NB}$) hardware pointers. The performance of the LimitLESS protocol degrades gracefully as the number of hardware pointers is reduced. The one-pointer LimitLESS protocol is especially bad, because some of Weather's variables have a worker set that consists of two processors.

This behavior indicates that multiprocessor software running on a system with a LimitLESS protocol will require some of the optimizations that would be needed on a system with a limited directory protocol. However, the LimitLESS protocol is much less sensitive to programs that are not perfectly optimized. Moreover, the software optimizations used with a LimitLESS protocol should not be viewed as extra overhead caused by the protocol itself. Rather, these optimizations might be employed, regardless of the cache coherence mechanism, since they tend to reduce hot-spot contention and to increase communication locality.

4.2.2 Conclusion

The preliminary investigation of the Alewife memory system indicates the viability of the software-extension approach. Although the experiments evaluate only the hardware side of software-extended shared memory, they demonstrate enough evidence to justify the time and expense of building an actual machine. The next two chapters complete the design and evaluation of the Alewife memory system: Chapter 5 describes the design of the software part of cache coherence protocols, and Chapter 6 evaluates the performance of the system as a whole.

Chapter 5

Flexible Coherence Interface

There are two goals that define flexibility and drive abstraction in a software-extended memory system. The first goal — the one that motivated the development of the flexible coherence interface — is the ability to implement a variety of different memory systems that all work on one hardware base. This ability facilitated the experiments that helped evaluate the trade-offs involved in designing software-extended shared memory. The interface also enables the development of smart memory systems, which require dynamic changes in the mapping between memory blocks and coherence protocols.

The second goal is the ability to write a single set of modules that implement a memory system for a number of different hardware platforms. This goal motivated the `pmap` abstraction in the Mach operating system [65]. `pmaps` allow Mach's virtual memory system to run efficiently on a wide range of different architectures.

Although a good abstraction between the hardware and software parts of a memory system should achieve both goals, the system designer usually has one or the other in mind when building a system. The initial version of the flexible coherence interface ignored the problem of multiple hardware bases, because — at the time that it was developed — Alewife was the only architecture with a software-extended memory system. The rest of this section concentrates on the goal of implementing many memory systems on a single hardware base, leaving speculation about the other side of the interface for Section 8.5.

5.1 Challenges and Solutions

Alewife's flexible memory system grew out of several previous software-extended systems that use the raw CMMU interface directly. The software protocol handlers of the older systems are written in Sparcle assembly code and are carefully tuned to take full advantage of the features of the Alewife architecture. Such implementations take a long time to construct: for example, John Piscitello (a graduate student with the Alewife project) spent most of a semester writing a software-only directory protocol [64].

These systems take a long time to write, because programming the memory hardware directly is difficult for several reasons:

1. Similar events have different possible representations, depending on whether they are initiated by local or remote nodes.

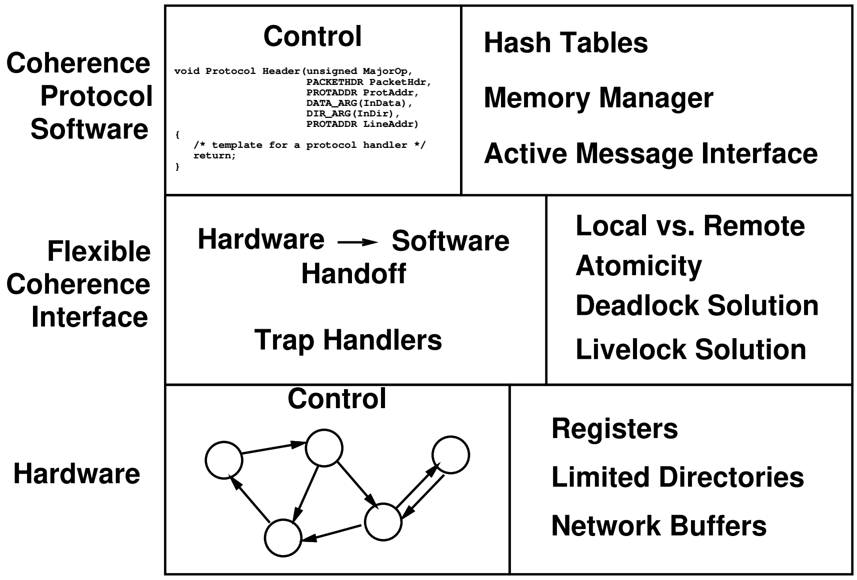


Figure 5-1: Hardware, interface, and software layers of a memory system.

2. The parallelism between hardware and software makes atomic protocol transitions problematic.

3. The memory system can starve the processor and livelock the system.

4. Limited network buffering can cause the system to deadlock.

The flexible coherence interface relieves the operating system programmer of the burden of solving these problems and provides a convenient abstraction for writing software-extended systems. Memory events that are handled in software generate calls to procedures written in C and linked into the operating system kernel. These event-handling procedures have message-passing semantics and do not use the shared memory model. Since the procedures run in supervisor mode, they must be written carefully; but as long as a handler follows the conventions of the interface, it executes atomically and does not cause the system to deadlock or livelock.

The interface provides an abstraction for directory entries, protocol messages, and the chunks of data stored in shared memory. The abstraction attempts to hide Alewife's encoding of these structures in order to make the software easier to write and to understand. Despite this indirection, the interface's implementation is efficient because it represents the structures in processor registers with the same encoding used by the A-1000 CMMU [46].

There are three ways to specify the mapping between an memory event and its software handler. First, the kernel contains a default set of event handlers and a dispatch table. When Alewife is booted normally, these handlers process events for all blocks in shared memory. The default procedures implement $Dir_nH_5S_{NB}$. Second, the Alewife host interface has an option that allows the user to choose an alternate default handler. The alternate handler processes events for all blocks in shared memory instead of the default handlers[1], and must accept all memory events, performing its own dispatch if necessary. This option allows the user to configure the system with a special protocol such as $Dir_n H_0 S_{NB,ACK}$. Finally, while the system is running, the interface allows the software to set the handler for any individual memory block. (See Section 5.1.2 for the syntax and implementation of this *call-back* functionality.) This feature enables efficient implementations of protocol state machines and allows each memory block to have its own semantics.

Figure 5-1 illustrates the hierarchy of the system. The lowest level of the architecture is the memory-system hardware. Finite-state machines in the hardware control the operation of this level, and the coherence state is stored in registers, limited directories, and network buffers. The flexible coherence interface is implemented as a layer of software on top of the hardware. It consists mostly of assembly-language trap handlers that field hardware-generated exceptions. The interface trap handlers call the procedures of the software-extended memory system that implement the cache coherence protocol. These procedures can then call the interface functions described in the next few sections. In addition, a library of useful routines that facilitate code development accompanies the interface. The library implements hash-table manipulation, a free-listing memory manager, and an active message [76] interface.

5.1.1 Representations

In the Alewife architecture, internode memory accesses invoke software via an asynchronous interrupt, while intranode accesses cause a synchronous trap — an exception that the processor must handle before a pending load or store can complete. In the former case, a network packet carries information about the transaction. In the latter case, information about the transaction is stored in CMMU registers. In order to mask this complexity, the interface transforms all memory-system events into procedure calls, with a message-passing abstraction that hides the details of the mechanisms involved in transmitting and receiving packets.

Figure 5-2 shows an example of a procedure that rides on top of the interface. The coherence protocol software in the figure handles an event caused by a read request that exceeds the number hardware directory pointers. All of the information required to process the associated memory system event — roughly the same information as in a network protocol packet [46] — is passed as arguments to this procedure. By the time that the interface calls the procedure, the interface has already disposed of the packet and freed the space in the appropriate CMMU network buffer. That is, if such a packet ever existed! Instead, the interface may have gathered the information from various CMMU registers, thereby simplifying the task of the handler.

The arguments to the procedure have the following meaning: `MajorOp` corresponds to the specific type of coherence protocol message. Some software handlers use this value to dispatch to subroutines. `PacketHdr` contains additional information about the memory transaction, including the identifiers of the requesting and responding nodes.

[1]It would be easy to change this option to configure different handlers for arbitrary subdivisions of shared memory.

```
void ProtocolHandler(unsigned MajorOp,
                     PACKETHDR PacketHdr, PKTADDR ProtAddr,
                     DATA_ARG(InData), DIR_ARG(InDir),
                     PROTADDR LineAddr)
{
   /* Upon Entry, hardware directory is locked and
      network packet disposed (if it ever existed!) */

  unsigned NumDirPtrs = DIR_PTRS_IN_USE(InDir);
  PDIRSTRUCT pDirStruct;

  /* set the directory back to hardware-handled state */
  DIR_CLR_PTRS_IN_USE(InDir), DIR_WRITEUNLOCK(LineAddr, InDir);

  /* put source and directory pointers into the dirstruct */
  pDirStruct = (PDIRSTRUCT)HashLookup(LineAddr);
  EnterDirPtr(pDirStruct, PKTHDR_SRC(PacketHdr));
  EnterDirPtrs(pDirStruct, DIR_VALUE(InDir), NumDirPtrs);

  /* normal C function return */
}
```

Figure 5-2: Sample protocol message handler.

Operation	Action
`HashLookup(PROTADDR)`	look up an address in a hash table
`HashInsert(PROTADDR,POINTER)`	insert an address into a hash table
`HashReplace(PROTADDR,POINTER)`	replace an address in a hash table
`HashDelete(PROTADDR)`	remove an address from a hash table
`local_malloc(SIZE)`	free-listing memory allocation
`local_free(POINTER)`	free-listing memory release
`kernel_do_on(NODE,FUNCTION,...)`	active message
`kernel_do_on_dma(NODE,FUNCTION,...)`	" with direct memory access

Table 5.1: Functionality related to FCI.

`ProtAddr` contains the shared memory address plus synchronization information, and is accessed through the flexible coherence interface. Depending on `MajorOp`, `InData` may contain data associated with the memory event. `InDir` contains the current state of the hardware directory. `LineAddr` is a copy of the shared memory address that can be used as a key for hash table functions and for accessing hardware state.

Some of the symbols in Figure 5-2 are specific to the protocol, rather than part of the interface: the `EnterDirPtr` and `EnterDirPtrs` subroutines store pointers into a directory structure of type `PDIRSTRUCT`. When the handler finishes processing the memory event, it simply returns, and the interface takes care of returning the processor back to the application.

The `HashLookup` function is part of the library that is not formally part of the interface, but is provided with it. Table 5.1 lists this function and some of the others in the library: the first four implement hash tables that key off of protocol addresses; the next two access a free-listing memory manager; and the last two provide active message functionality.

Table 5.2 lists the operations that provide an abstraction for data. Note that the coherence protocol software never needs to know the actual size of the unit of data transfer, nor does the size have to be fixed. The first three macros provide the declarations required to implement the data abstraction on top of the C compiler's type system. `WriteBackData` and `RetrieveData` provide access to the DRAM storage for the data. The last two operations manipulate the state of the data in the local processor's cache.

The set of macros in Table 5.3 complete the message-passing abstraction. The first two sets of operations read and write the `PKTHDR` and `PKTADDR` components of a message. The last three macros in the table allow the software to transmit messages in response to the original memory event.

5.1.2 Atomicity

The atomicity problem arises from parallelism in the system: since both hardware and software can process transactions for a single memory block, the software must

Operation	Action
`DATA_ARG(LABEL)`	specifies data as function argument
`DATA_TYPE(LABEL)`	definition for data as variable
`DATA_VALUE(LABEL)`	specifies data as function call value
`WriteBackData(PROTADDR,LABEL)`	write data to memory
`RetrieveData(PROTADDR,LABEL)`	read data from memory
`ProtocolInvalidate(PROTADDR)`	transmit local invalidation
`FlushExclusive(PROTADDR,LABEL)`	force cached data to memory

Table 5.2: FCI operations for manipulating local data state.

Operation	Action
`PKTHDR_SRC(PKTHDR)`	get source field of packet header
`PKTHDR_DST(PKTHDR)`	get destination field of header
`PKTHDR_FORMAT(MSG,TO,FROM)`	format a packet header
`ADD_PKTHDR_DST(PKTHDR,AMOUNT)`	add amount to destination field
`PKTADDR_PROTADDR(PKTADDR)`	get line address field of address
`PKTADDR_CLR_FE(PKTADDR)`	clear full/empty field of address
`PKTADDR_SET_FE(PKTADDR,NEW_FE)`	set full/empty field of address
`PKTADDR_FE(PKTADDR)`	get full/empty field of address
`TRANSMITREQUEST(MAJOP,PKTADDR,TO,FROM)`	transmit protocol request packet
`TRANSMITDATA(MAJOP,PKTADDR,TO,FROM)`	transmit protocol data packet
`TRANSMITDOUBLE(PKTHDR,PKTADDR)`	transmit two-word packet

Table 5.3: FCI operations for transmitting and receiving packets.

orchestrate state transitions carefully. Accesses to hardware directory entries provide an example of this problem. Since both the CMMU hardware and the Sparcle software may attempt to modify directory entries, the interface needs to provide a mechanism that permits atomic transitions in directory state.

Table 5.4 lists the relevant macros implemented in the interface. The first three macros are analogous to the `DATA` macros in Table 5.2. `DIR_READLOCK` and `DIR_WRITEUNLOCK` provide an interface to the CMMU mutual exclusion lock for directory entries, thereby allowing a software protocol handler to modify directory state atomically. The rest of the operations in Table 5.4 access and modify fields in the abstract directory entry.

Flexible, directory-based architectures should provide some level of hardware support for specifying protocols on a block-by-block basis. The two `DIR_HANDLER` macros in Table 5.4 provide this call-back functionality. In the current version of the flexible coherence interface, these macros store an entire program counter in a directory entry. This encoding is convenient, but inefficient. It would be easy to change the software to use a table lookup, rather than a direct jump to each handler. Such a scheme would use far fewer bits in each directory entry, enabling low-cost hardware support.

Since typical protocol handlers modify directory state, the interface automatically locks the appropriate directory entry before calling any handler. Figure 5-2 shows an atomic state transition: the handler retrieves a field of the directory entry with `DIR_PTRS_IN_USE`, clears the same field with `DIR_CLR_PTRS_IN_USE`, and then uses `DIR_WRITEUNLOCK` to commit the modification and release the mutual exclusion lock.

Directory modification is only a simple instance of an atomicity problem: the interface's solution consists of little more than convenient macros for using features of the CMMU. Other atomicity problems pose more serious difficulties. For example, the CMMU may transmit data in response to a remote (asynchronous) read request, but relay the message to the processor for additional software handling. Before the memory-system software fields the associated memory event, a local (synchronous) write to the memory block might intervene. In order to ensure the atomicity of the read request, the interface must guarantee the proper order of the actions: the higher-level software must process the partial asynchronous event before handling the synchronous event. Again, the coherence protocol sees normal message-passing passing semantics, while the flexible interface takes care of such details.

5.1.3 Livelock and Deadlock

There are also issues of livelock and deadlock involved in writing the coherence protocol software. Livelock situations can occur when software-extension interrupts occur so frequently that user code cannot make forward progress [64]. The framework solves this problem by using a timer interrupt to implement a watchdog that detects possible livelock, temporarily shuts off asynchronous events, and allows the user code to run unmolested. In practice, such conditions happen only for $Dir_nH_0S_{NB,ACK}$ and $Dir_nH_1S_{NB,ACK}$, when they handle acknowledgments in software. The `timer_block_network` function in Table 5.5 provides an interface that allows these protocols to invoke the watchdog directly.

The possibility of deadlock arises from limited buffer space in the interconnection

Operation	Action
DIR_ARG(LABEL)	specifies entry as function argument
DIR_TYPE(LABEL)	definition for entry as variable
DIR_VALUE(LABEL)	specifies entry as function call value
DIR_READLOCK(PROTADDR,LABEL)	atomically read and lock directory
DIR_WRITEUNLOCK(PROTADDR,LABEL)	atomically write and unlock directory
DIR_HANDLER(LABEL)	get protocol handler
DIR_SET_HANDLER(LABEL,HANDLER)	set protocol handler
DIR_SET_EMPTY(LABEL)	clear directory state
DIR_COPY(FROM,TO)	copy one directory to another
DIR_PTRN(LABEL)	get pointer N
DIR_SET_PTRN(LABEL,NEW_PTR)	set pointer N
DIR_CLR_PTRN(LABEL)	clear pointer N
DIR_SET_LOCAL_BIT(LABEL)	set special local pointer
DIR_CLR_LOCAL_BIT(LABEL)	clear special local pointer
DIR_PTRS_IN_USE(LABEL)	number of valid pointers
DIR_CLR_PTRS_IN_USE(LABEL)	set number of valid pointers to zero
DIR_SET_PTRS_IN_USE(LABEL,NEW_PIU)	set number of valid pointers
DIR_SET_PTRS_AVAIL(LABEL,NEW_PA)	set number of pointers available
DIR_STATE(LABEL)	get directory state field
DIR_CLR_STATE(LABEL)	clear directory state field
DIR_SET_STATE(LABEL,NEW_STATE)	set directory state field
DIR_UDB(LABEL)	get directory user defined bits
DIR_SET_UDB(LABEL,NEW_UDB)	set directory user defined bits
DIR_FE(LABEL)	get directory full/empty state
DIR_CLR_FE(LABEL)	clear directory full/empty state
DIR_SET_FE(LABEL,NEW_FE)	set directory full/empty state

Table 5.4: FCI operations for manipulating hardware directory entries.

Operation	Action
timer_block_network()	temporarily block the network
info = launch_from_trap_prologue()	set up state for transmit
launch_from_trap_epilogue(info)	restore state after transmit

Table 5.5: FCI operations for livelock and deadlock avoidance.

network: if two different processors simultaneously attempt to transmit large numbers of packets, there is a chance that the network buffers between the nodes will fill to capacity. If both of the processors continue to try to transmit, the buffers will remain clogged forever. Such scenarios rarely happen during normal Alewife operation, but they have the potential to lock up the entire system when they occur. The CMMU provides a mechanism that detects such scenarios and interrupts the local processor when they occur. Upon receiving the interrupt, the processor empties the packets in the CMMU's network buffers and retransmits them when the deadlock subsides.

For performance reasons, the flexible coherence interface normally disables all interrupts, including the one that prevents deadlocks. This policy allows protocol handlers that do not use the network to run efficiently. The last two operations in Table 5.5 allow the memory system to invoke the deadlock solution: as long as the programmer places calls to `launch_from_trap_prologue` and `launch_from_trap_epilogue` around portions of code that transmit messages, the interface makes sure that deadlock does not occur. These semantics allow the interface to save the state of the user code's transmissions only when necessary.

5.2 Experience with the Interface

During the course of the research on software-extended memory systems, the flexible coherence interface proved to be an indispensable tool for rapidly prototyping a complete spectrum of protocols. A single set of routines use the interface to implement all of the protocols from $Dir_nH_1S_{NB}$ to $Dir_nH_{NB}S_{-}$. Other modules linked into the same kernel support $Dir_nH_0S_{NB,ACK}$, $Dir_nH_1S_{NB,LACK}$, and $Dir_nH_1S_{NB,ACK}$. The smart memory system software described in Chapter 8 also uses the interface.

Anecdotal evidence supports the claim that the interface accelerates protocol software development. With the interface, the software-only directory ($Dir_nH_0S_{NB,ACK}$) required only about one week to implement, compared to the months of work in the previous iteration. The $Dir_nH_1S_{NB,ACK}$ protocol reused much of the code written for other protocols and required less than one day to write and to test on NWO.

Interestingly enough, the CMU paper on virtual memory reports that a staff programmer took approximately three weeks to implement a `pmap` model. Similarly, Donald Yeung (a graduate student with the Alewife project) recently took about three weeks to implement a coherence protocol on top of the flexible coherence interface. The two programming tasks are not directly comparable, because the CMU staff member implemented hardware-dependent code and Don wrote high-level code; however, the analogy is compelling.

5.3 The Price of Flexibility

Unfortunately, flexibility comes at the price of lower performance. All of the interface's mechanisms that protect the memory-system designer from the details of the Alewife architecture increase the time that it takes to handle protocol requests in software.

This observation does not imply that the implementation of the interface is inefficient: it passes values to protocol handlers in registers, leaves interrupts disabled as often as possible, usually exports macros instead of functions, and uses assembly code for critical prologue and epilogue code. Yet, any layer of abstraction that enables flexibility lowers performance, because handling general-case behavior almost always requires more computation than handling the behavior of a specific application.

The next chapter investigates the performance of software-extended shared memory and begins with an assessment of the trade-off between flexibility and performance. Chapter 7 uses an analytical model to investigate the relationship between the speed of the software part of a memory system and the end-to-end performance of a multiprocessor.

Chapter 6

Cost, Flexibility, and Performance

The implementations of the Alewife architecture provide a proof-of-concept for the software-extension approach. The following empirical study uses the tools described in the previous chapter to show that this approach leads to memory systems that achieve high performance without prohibitive cost. NWO allows the investigation of a wide range of architectural parameters on large numbers of nodes [13]; the A-1000 demonstrates a real, working system. While the study does provide detailed measurements of software-extended systems, it focuses on their most important characteristics by using two metrics: the size of a system's hardware directory and the speedup that a parallel system achieves over a sequential one. The former metric indicates cost and the latter measures performance.

Rather than advocating a specific machine configuration, this chapter seeks to examine the performance versus cost trade-offs inherent in implementing software-extended shared memory. It begins by measuring the performance of two different implementations of Alewife's memory system software, in order to evaluate the impact of flexibility on performance. The study then uses a synthetic workload to investigate the relationship between application behavior and the performance of a software-extended system. The chapter concludes by presenting six case studies that examine how application performance varies over the spectrum of software-extended protocols.

6.1 Flexibility and Performance

Flexible and inflexible versions of $Dir_nH_5S_{NB}$ have been written for Alewife. Comparing the performance of these two implementations demonstrates the price of flexibility in a software-extended system.

One version of the software is written in the C programming language and uses the flexible coherence interface. It implements the entire range of protocols from $Dir_nH_0S_{NB,ACK}$ to $Dir_nH_{NB}S_{-}$ and the smart memory systems in Chapter 8. The other version of the software is written in Sparcle assembly language and uses the CMMU interface directly. The code for this optimized version is hand-tuned to keep instruction counts to a minimum. To reduce memory management time, it uses a special free-list of extended directory structures that are initialized when the kernel boots the machine.

Readers Per Block	C Read Request	Assembly Read Request	C Write Request	Assembly Write Request
8	436	162	726	375
12	397	141	714	393
16	386	138	797	420

Table 6.1: Average software-extension latencies for C and for assembly language, in simulated execution cycles.

Activity	C Read Request	Assembly Read Request	C Write Request	Assembly Write Request
trap dispatch	11	11	9	11
system message dispatch	14	15	14	15
protocol-specific dispatch	10	N/A	10	N/A
decode and modify hardware directory	22	17	52	40
save state for function calls	24	N/A	17	N/A
memory management	60	65	28	11
hash table administration	80	N/A	74	N/A
store pointers into extended directory	235	74	99	45
invalidation lookup and transmit	N/A	N/A	419	251
support for non-Alewife protocols	10	N/A	6	N/A
trap return	14	11	9	11
total (median latency)	480	193	737	384

Table 6.2: Breakdown of simulated execution cycles measured from median-latency read and write requests. Each memory block has 8 readers and 1 writer. N/A stands for not applicable.

The assembly-language version also takes advantage of a feature of Alewife's directory that eliminates the need for a hash table lookup. Since this approach requires a large programming effort, this version only implements $Dir_nH_5S_{NB}$.

The measurable difference between the performance of the two implementations of the protocol extension software is the amount of time that it takes to process a protocol request. Table 6.1 gives the average number of cycles required to process $Dir_nH_5S_{NB}$ read and write requests for both of the implementations. These software handling latencies were measured by running the WORKER benchmark (described in Section 3.2.1) on a simulated 16-node system. The latencies are relatively independent of the number of nodes that read each memory block. In most cases, the hand-tuned version of the software reduces the latency of protocol request handlers by about a factor of two or three.

These latencies may be understood better by analyzing the number of cycles spent on each activity required to extend a protocol in software. Table 6.2 accounts for all of the cycles spent in read and write requests for both versions of the protocol software. These counts come from cycle-by-cycle traces of read and write requests with eight readers and one writer per memory block. The table uses the median request of each type to investigate the behavior of a representative individual. (Table 6.1 uses the average to summarize aggregate behavior.)

The dispatch and trap return activities are standard sequences of code that invoke hardware exception and interrupt handlers and allow them to return to user code, respectively. (The dispatch activity does not include the three cycles that Sparcle takes to flush its pipeline and to load the first trap instruction.) In the assembly-language version, these sequences are streamlined to invoke the protocol software as quickly as possible. The C implementation of the software requires an extra protocol-specific dispatch in order to set up the C environment and hide the details of the Alewife hardware. For the types of protocol requests that occur when running the WORKER benchmark, this extra overhead does not significantly affect performance. The extra code in the C version that branches to the protocols supported only by NWO also impacted performance minimally.

The assembly code uses the CMMU mechanisms to decode and modify the hardware directory, while the C code uses the flexible coherence interface operations in Table 5.4. The C code was generated by Alewife's normal compiler, so it saves state on the stack before making function calls; the hand-tuned assembly code uses perfect interprocedural analysis, thereby avoiding the need to save any state.

The primary difference between the performance of the C and assembly-language protocol handlers lies in the flexibility of the C interface. The assembly-language version avoided most of the expense of memory management and hash table administration by implementing a special-purpose solution to the directory structure allocation and lookup problem. This solution relies heavily on the format of Alewife's coherence directory and is not robust in the context of a system that runs a large number of different applications over a long period of time. However, it does place a minimum bound on the time required to perform these tasks. The flexible code uses the more general functions in Table 5.1, which integrate the memory-system software with the rest of the kernel.

Both the flexible and inflexible versions of the read request code store exactly five pointers from the hardware directory and one pointer that corresponds to the source of the request. Thus, read request software processing is amortized over six remote requests. The assembly code takes only about 12 cycles to store each pointer into the extended directory, which is close to the minimum possible number on Sparcle. The C code uses a pointer-storing function. While this function is less efficient by more than a factor of three, it is a better abstraction: the same function is used by all of the $Dir_n H_X S_{Y,A}$ protocols.

The write request code stores two additional pointers (raising the total to eight, the worker-set size) before transmitting invalidations. Again, the assembly code performs these functions much more efficiently than the C code, but the C code uses a better abstraction for building a number of different protocols. The appropriate balance of flexibility and performance lies somewhere between the two extremes described above. As the Alewife system evolves, critical pieces of the protocol extension software will be hand-tuned to realize the best of both worlds.

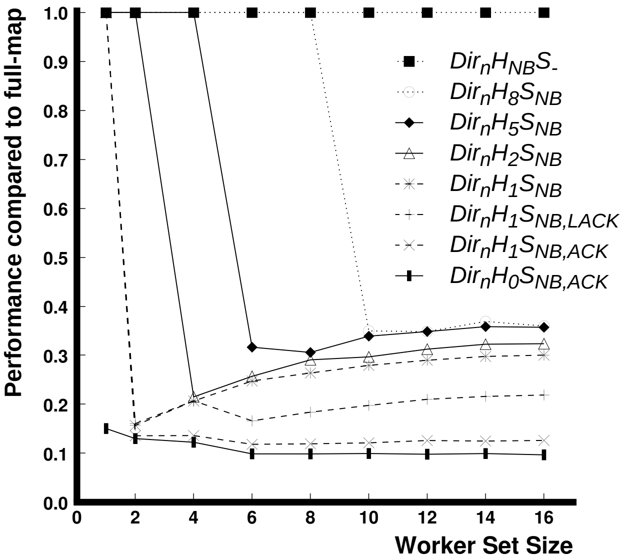


Figure 6-1: Synthetic workload shows the relationship between protocol performance and worker-set size on a simulated 16-node machine.

Section 7.4.2 uses an analytical model to translate the information about the relative performance of protocol handlers into predictions about overall system behavior. The model indicates that the factors of 2 or 3 in handler running times (as in Table 6.1) generally do not cause significant differences in aggregate system performance.

6.2 Worker Sets and Performance

Section 2.3 defines a worker set as the set of nodes that access a unit of data between subsequent modifications of the data block. The software-extension approach is predicated on the observation that, for a large class of applications, most worker sets are relatively small. Small worker sets are handled in hardware by a limited directory structure. Memory blocks with large worker sets must be handled in software, at the expense of longer memory access latency and processor cycles that are spent on protocol handlers rather than on user code.

This section uses the WORKER synthetic benchmark to investigate the relationship between an application's worker sets and the performance of software-extended coherence protocols. Simulations of WORKER running on a range of protocols show the relationship between worker-set sizes and the performance of software-extended shared memory. The simulations are restricted to a relatively small system because the benchmark is both regular and completely distributed, so the results would not be qualitatively different for a larger number of nodes.

Figure 6-1 presents the results of a series of 16-node simulations. The horizontal axis gives the size of the worker sets generated by the benchmark. The vertical axis measures the ratio of the execution time of a full-map protocol ($Dir_nH_{NB}S_-$) to the execution time of each protocol running the same benchmark configuration.

The solid curves in Figure 6-1 indicate the performance of some of the protocols that are implemented in the A-1000. As expected, the more hardware pointers, the better the performance of the software-extended system. The performance of $Dir_nH_5S_{NB}$ is particularly easy to interpret: its performance is exactly the same as the full-map protocol up to a worker-set size of 4, because the worker sets fit entirely within the hardware directory. For small worker-set sizes, software is never invoked. The performance of $Dir_nH_5S_{NB}$ drops for larger worker sets, due to the expense of handling memory requests in software.

At the other end of the performance scale, the $Dir_nH_0S_{NB,ACK}$ protocol performs significantly worse than the other protocols, for all worker-set sizes. Since WORKER is a shared memory stress test and exaggerates the differences between the protocols, Figure 6-1 shows the worst possible performance of the software-only directory. The measurements in the next section, which experiment with more realistic applications, yield a more optimistic outlook for the zero and one-pointer protocols.

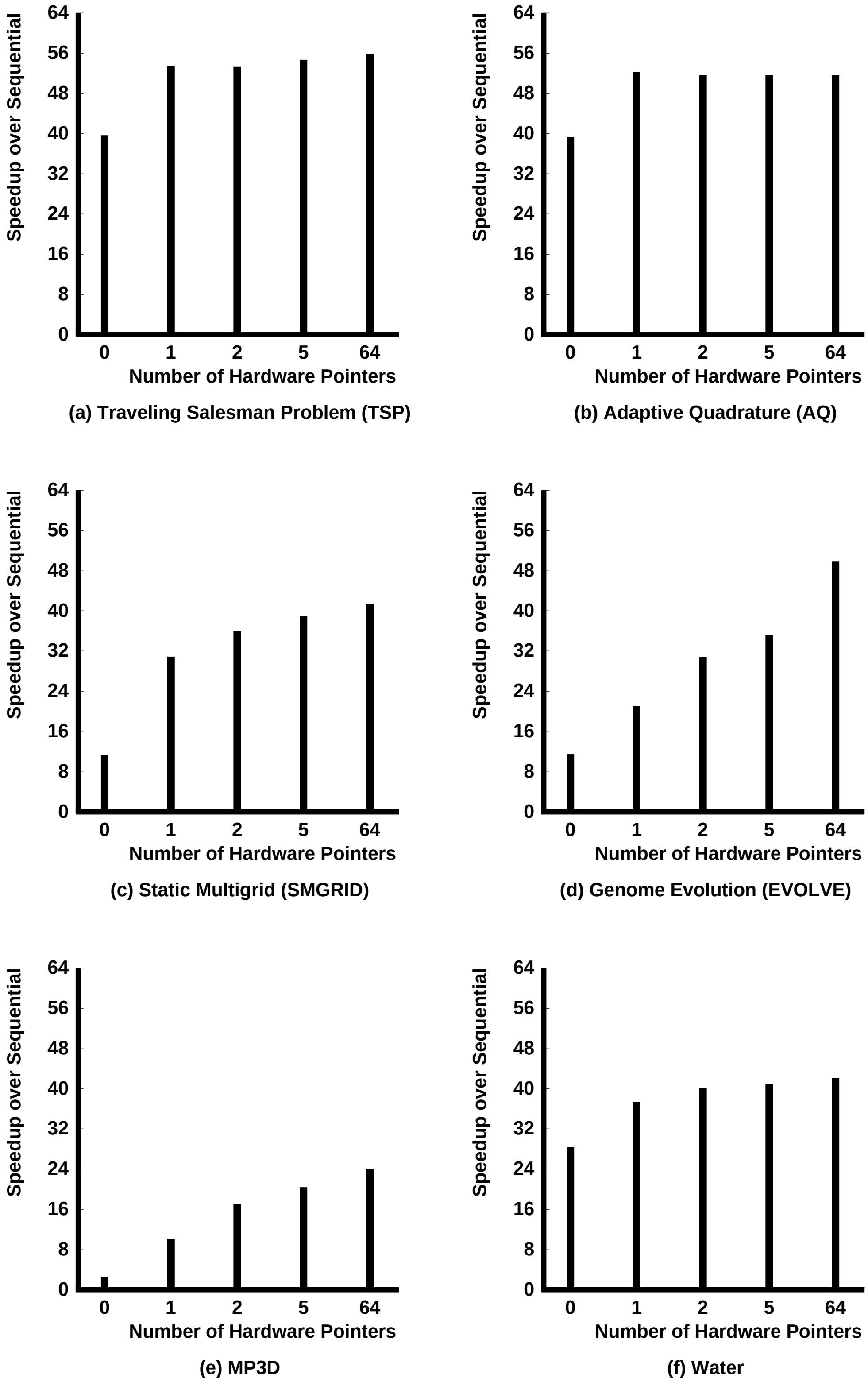
The dashed curves correspond to one-pointer protocols that run only on NWO. These three protocols differ only in the way that they handle acknowledgment messages (see Section 2.1.3). For all non-trivial worker-set sizes, the protocol that traps on every acknowledgment message ($Dir_nH_1S_{NB,ACK}$) performs significantly worse than the protocols that can count acknowledgments in hardware. $Dir_nH_1S_{NB}$, which never traps on acknowledgment messages, has very similar performance to the $Dir_nH_2S_{NB}$ protocol, except when running with size 2 worker sets. Since this version of $Dir_nH_1S_{NB}$ requires the same amount of directory storage as $Dir_nH_2S_{NB}$, the similarity in performance is not surprising.

Of the three different one-pointer protocols, the protocol that traps only on the last acknowledgment message in a sequence ($Dir_nH_1S_{NB,LACK}$) makes the most cost-efficient use of the hardware pointers. This efficiency comes at a slight performance cost. For the WORKER benchmark, this protocol performs between 0% and 50% worse than $Dir_nH_1S_{NB}$. When the worker-set size is 4 nodes, $Dir_nH_1S_{NB,LACK}$ performs slightly better than $Dir_nH_1S_{NB}$. This anomaly is due to a memory-usage optimization that attempts to reduce the size of the software-extended directory when handling small worker sets. The optimization, implemented in the $Dir_nH_1S_{NB,LACK}$, $Dir_nH_1S_{NB,ACK}$ and $Dir_nH_0S_{NB,ACK}$ protocols, improves the run-time performance of all three protocols for worker-set sizes of 4 or less.

6.3 Application Case Studies

This section presents more practical case-studies of several programs and investigates how the performance of applications depends on memory access patterns, the coherence protocol, and other machine parameters.

Figure 6-2 presents the basic performance data for the six benchmarks described in Section 3.2.2, running on 64 simulated nodes. The horizontal axis shows the number of directory pointers implemented in hardware, thereby measuring the cost of the system. The vertical axis shows the speedup of the multiprocessor execution over a sequential run without multiprocessor overhead. The software-only directory is always on the left and



(a) Traveling Salesman Problem (TSP)

(b) Adaptive Quadrature (AQ)

(c) Static Multigrid (SMGRID)

(d) Genome Evolution (EVOLVE)

(e) MP3D

(f) Water

Figure 6-2: Application speedups over sequential, running on a simulated 64 node Alewife machine.

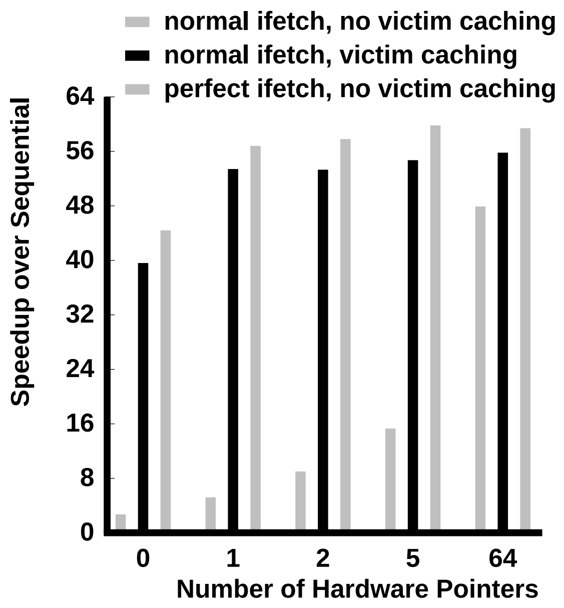


Figure 6-3: TSP: detailed performance analysis on 64 NWO nodes.

the full-map directory on the right. All of the figures in this section show $Dir_nH_1S_{NB,ACK}$ performance for the one-pointer protocol.

The most important observation is that the performance of $Dir_nH_5S_{NB}$ is always between 71% and 100% of the performance of $Dir_nH_{NB}S_-$. Thus, the graphs in Figure 6-2 provide strong evidence that the software-extension approach is a viable alternative for implementing a shared memory system. The rest of this section seeks to provide a more detailed understanding of the performance of software-extended systems.

Traveling Salesman Problem Given the characteristics of the application's memory access pattern, one would expect TSP to perform well with a software-extended protocol: the application has very few large worker sets. In fact, most – but not all – of the worker sets are small sets of nodes that concurrently access partial tours.

Figure 6-3 presents detailed performance data for TSP running on a simulated 64 node machine. Contrary to initial expectations, TSP suffers severe performance degradation when running with the software-extended protocols. The gray bars in the figure show that the five-pointer protocol performs more than three times worse than the full-map protocol. This performance decrease is due to instruction/data thrashing in Alewife's combined, direct-map caches: profiles of the address reference pattern of the application show that two memory blocks that were shared by *every* node in the system were constantly replaced in the cache by commonly run instructions.

A simulator option confirms this observation by allowing one-cycle access to every instruction, without using the cache. This option, called *perfect ifetch*, eliminates the effects of instructions on the memory system. The hashed bars in Figure 6-3 confirm that instruction/data thrashing was a serious problem in the initial runs. Absent the effects of instructions, all of the protocols except the software-only directory realize performance

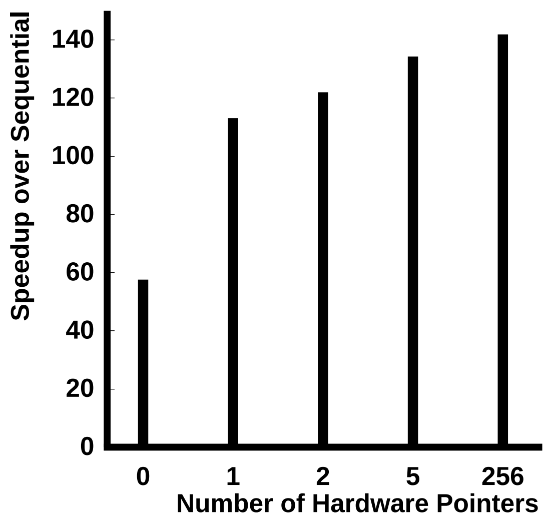


Figure 6-4: TSP running on 256 NWO nodes.

equivalent (within experimental error) to a full-map protocol.

While perfect instruction fetching is not possible in real systems, there are various methods for relieving instruction/data thrashing by increasing the associativity of the cache system. Alewife's approach to the problem is to implement a version of victim caching [39], which uses the transaction store [48] to provide a small number of buffers for storing blocks that are evicted from the cache. The black bars in Figures 6-2(a) and 6-3 show the performance for TSP on a system with victim caching enabled. The few extra buffers improve the performance of the full-map protocol by 16%, and allow all of the protocols with hardware pointers to perform about as well as full-map. For this reason, the studies of all of the other applications in this section enable victim-caching by default.

It is interesting to note that $Dir_nH_0S_{NB,ACK}$ with victim caching achieves almost 70% of the performance of $Dir_nH_{NB}S_-$. This low-cost alternative seems viable for applications with limited amounts of sharing.

Thus far, the simulations have shown the performance of the protocols under an environment where the full-map protocol achieves close to maximum speedup. On an application that requires only 1 second to run, the system with victim caching achieves a speedup of about 55 for the 5 pointer protocol. Running the same problem size on a simulated 256 node machine indicates the effects of running an application with lower speedups. Figure 6-4 shows the results of this experiment, which indicate a speedup of 142 for full-map and 134 for five-pointers. These speedups are quite remarkable for this problem size. In addition, the software-extended system performs only 6% worse than full-map in this configuration. The difference in performance is due primarily to the increased contribution of the transient effects over distributing data to 256 nodes at the beginning of the run.

Adaptive Quadrature Since all of the communication in AQ is producer-consumer, the application should perform equally well for all protocols that implement at least one directory pointer in hardware. Figure 6-2(b) confirms this expectation by showing the performance of the application running on 64 simulated nodes. Again, $Dir_nH_0S_{NB,ACK}$ performs respectably due to the favorable memory access patterns in the application.

Static Multigrid SMGRID's speedup over sequential is limited by the fact that only a subset of nodes work during the relaxation on the upper levels of the pyramid of grids. Furthermore, data is more widely shared in this application than in either TSP or AQ. The consequences of these two factors appear in Figure 6-2(c): the absolute speedups are lower than either of the previous applications, even though the sequential time is three times longer.

The larger worker-set sizes of multigrid cause the performance of the different protocols to separate. $Dir_nH_0S_{NB,ACK}$ performs more than three times worse than the full-map protocol. The others range from 25% worse in the case of $Dir_nH_1S_{NB,ACK}$ to 6% worse in the case of $Dir_nH_5S_{NB}$.

Genome Evolution Of all of the applications in Figure 6-2, EVOLVE causes the five-pointer protocol ($Dir_nH_5S_{NB}$) to exhibit the worst performance degradation compared to $Dir_nH_{NB}S_{-}$: the worker sets of EVOLVE seriously challenge a software-extended system. Figure 6-5 shows the number of worker sets of each size at the end of a 64 node run. Note that the vertical axis is logarithmically scaled: there are almost 10,000 one-node worker sets, while there are 25 worker sets of size 64.

The significant number of nontrivial worker sets implies that there should be a sharp difference between protocols with different numbers of pointers. A more detailed analysis in Chapter 7 shows that a large number of writes to data with a worker-set size of six reduces the performance of all protocols with fewer hardware pointers. The large worker sets impact the 0 and 1 pointer protocols most severely. Thus, EVOLVE provides a good example of a program that can benefit from a system's hardware directory pointers.

In addition to the performance degradation caused by large worker sets, EVOLVE suffers from severe data/data thrashing in the nodes' direct-mapped caches. To help diagnose the cause of the thrashing, the simulator generates a trace of all of the directory modification events that occur during a run with the full-map protocol. Analysis of the thrashing addresses in the traces shows that frequently accessed, shared data structures allocated on different nodes conflict in a number of lines of every node's cache.

Once diagnosed, the data/data thrashing problem is easy to solve. The pathological situation is caused by the regularity of the application, and the fact that the memory allocation heap in each node starts at a position that is aligned to every other node's heap. Due to the regularity of this thrashing, a simple change to the operating system can eliminate it: by skewing the nodes' heaps before running the program, the conflicting memory blocks can be shifted into different cache lines.

Figure 6-6 shows the results of the thrashing effect on a 64 node run of the application. The gray bars show the performance of EVOLVE without skewed heaps; the black bars

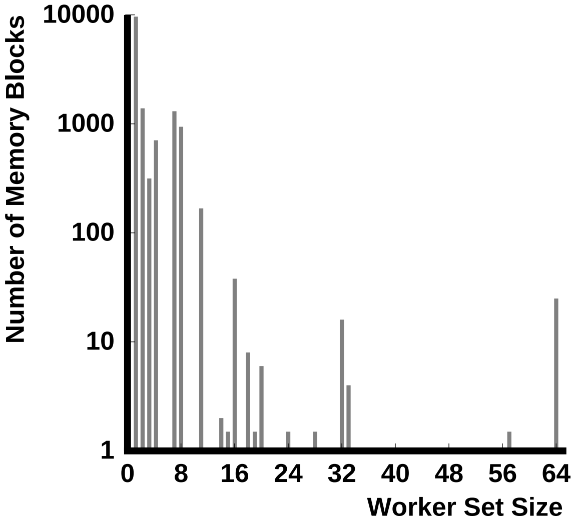


Figure 6-5: Histogram of worker-set sizes for EVOLVE, running on 64 NWO nodes.

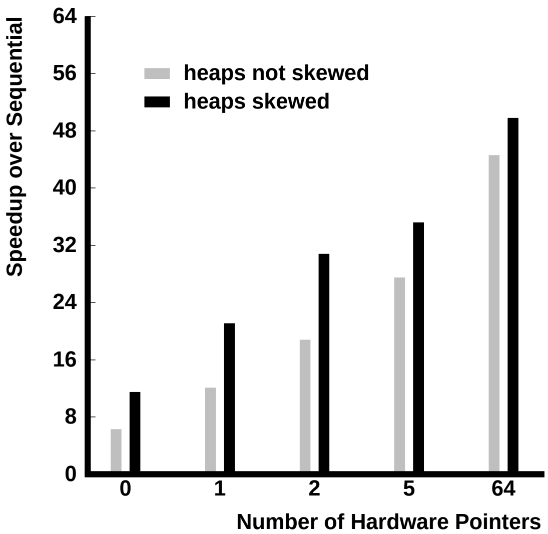


Figure 6-6: EVOLVE performance with and without heap skew.

show the performance gained by skewing each node's heap. (The black bars correspond to the ones in Figure 6-2(d).) When thrashing occurs, the $Dir_nH_{NB}S_{-}$ speedup is almost 45 and $Dir_nH_5S_{NB}$ reaches 27, about 40% slower. In the absence of significant thrashing, the full-map protocol achieves a speedup of 50, while the five-pointer protocol lags full-map by about 30%.

MP3D Since MP3D is notorious for exhibiting low speedups [56], the results in Figure 6-2(e) are encouraging: $Dir_nH_{NB}S_{-}$ achieves a speedup of 24 and $Dir_nH_5S_{NB}$ realizes a speedup of 20. These speedups are for a relatively small problem size, and absolute speedups should increase with problem size.

The software-only directory exhibits the worst performance (only 11% of the speedup of full-map) on MP3D. Thus, MP3D provides another example of an application that can benefit from at least a small number of hardware directory pointers.

Water The Water application, also from the SPLASH application suite, is run with 64 molecules. This application typically runs well on multiprocessors, and Alewife is no exception. Figure 6-2(f) shows that all of the software-extended protocols provide good speedups for this tiny problem size. Once again, the software-only directory offers almost 70% of the performance of the full-map directory.

6.4 A-1000 Performance

In a sense, the results in Figure 6-7 are much more dramatic than those presented in the rest of this chapter: the graphs show the performance of the six benchmarks on the largest available A-1000, an Alewife machine with 16 nodes and a 20 MHz processor clock speed. From the point of view of a system architect, this evidence is exciting because it confirms that ideas can be transformed into practice. The fact that the same benchmark and memory-system object code runs on both the A-1000 and the NWO simulator validates both the software-extended design technique and the simulation technology.

The figure shows the entire range of directory sizes that work on the A-1000 hardware, from $Dir_nH_0S_{NB,ACK}$ to $Dir_nH_5S_{NB}$, skipping the one-pointer protocols. It is hard to compare the data in Figure 6-7 directly to Figure 6-2 due to the different number of processors, a new version of the compiler, and different floating-point speeds. Nevertheless, the qualitative results are the same: as predicted by NWO, $Dir_nH_2S_{NB}$ through $Dir_nH_5S_{NB}$ all offer similar performance, while $Dir_nH_0S_{NB,ACK}$ achieves acceptable performance only for some of the benchmarks. The usual exception to this rule is EVOLVE, which benefits from $Dir_nH_5S_{NB}$'s hardware directory pointers.

As rewarding as it is to build a real system, thus far the A-1000 offers no more insight into the trade-offs involved in software-extended design than does NWO. The hardware certainly supports a smaller range of protocols and design options. On the other hand, the A-1000 allows users to develop real applications and to run data sets that could never be attempted under simulation. As the number of benchmarks and the machine size grow, the software-extended memory system must prove itself as the foundation for a system with a stable user environment.

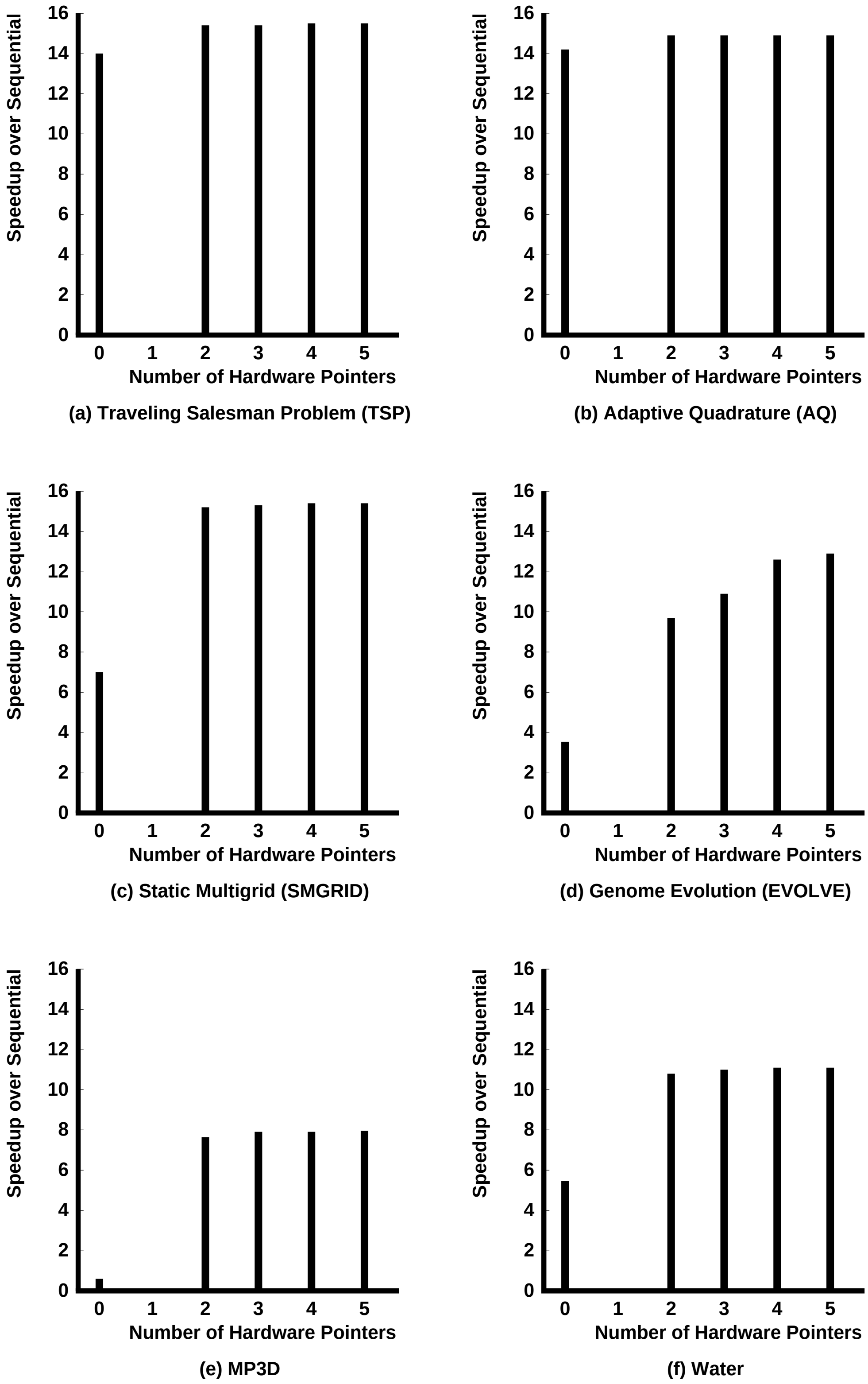


Figure 6-7: Application speedups over sequential, running on a 16 node A-1000.

6.5 Conclusions

The NWO and A-1000 implementations of Alewife prove that the software-extension approach yields high-performance shared memory. The hardware components of a software-extended system must be tuned carefully to achieve high performance. Since the software-extended approach increases the penalty of cache misses, thrashing situations cause particular concern. Adding extra associativity to the processor side of the memory system, by implementing victim caches or by building set-associative caches, can dramatically decrease the effects of thrashing on the system as a whole.

Alewife's flexible coherence interface enabled the study of the spectrum of software-extended memory systems by facilitating the rapid development of a number of protocols. However, flexibility comes at the price of extra software processing time. The next chapter uses an analytical model to show that the extra processing time does not dramatically reduce end-to-end performance. The following chapter proves the usefulness of flexibility by demonstrating memory systems that use intelligence to improve performance.

Chapter 7

The Worker-Set Model

A working system provides copious information about a narrow range of design choices. In contrast, an analytical model provides a way to explore a design space without the high overhead of building new systems. This chapter combines these two methods for evaluating an architecture: first, the Alewife system validates a mathematical model; then, the model examines a range of architectural mechanisms used to implement software-extended shared memory.

Rather than attempting to reproduce the details of operation, the model predicts the gross behavior of systems by aggregating its input parameters over the complete execution of programs. The inputs characterize the features of an architecture and the memory access pattern of an application, including a description of the application's worker sets. The model uses these parameters to calculate the frequency and latency of various memory system events. From these derived quantities, the model estimates the system's processor utilization, a metric that encompasses both the frequency of memory access and the efficiency of a memory system.

This chapter is organized as follows: the next section defines the model inputs and outputs precisely. Following the definitions, the model is described at both the intuitive and mathematical levels. Before exploring the design space, comparisons between the predictions and experimental results validate the model. Once validated, the model is used to investigate a number of options in memory system design by predicting the six benchmarks' performance on a variety of architectures. Then, synthetic workloads help explore the relationship between application memory access patterns and performance. The final section enumerates a number of conclusions about the memory system design space and discusses the problem of modeling cost.

7.1 Model Inputs and Outputs

Figure 7-1 illustrates the function of the worker-set model. The two types of model inputs come from several sources: a high-level description of the memory system architecture; measurements from microbenchmarks that determine the latency of important transactions in a software-extended system; and experimental measurements of each application's memory access patterns. For this study, the latter two input sources cor-

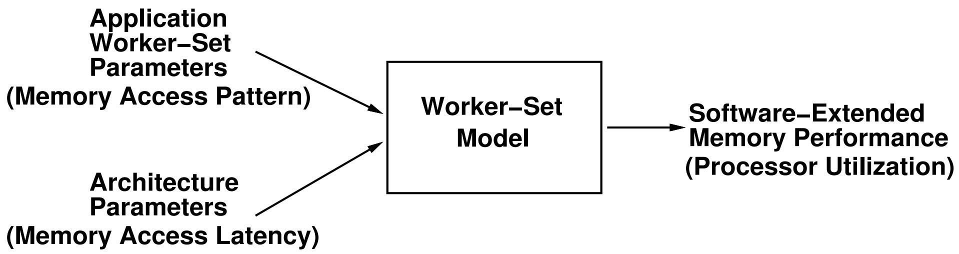


Figure 7-1: The worker-set model.

respond to simulations that require up to several hours to generate each data point. In contrast, given a set of inputs, the model can predict the performance for a whole range of architectures within a few minutes, if not seconds.

From a distance, the model outputs in the following sections look very much like the performance graphs in the previous chapter. The horizontal axis measures an architectural mechanism (such as the number of hardware pointers in a directory entry) or a measure of an application's memory access pattern (such as worker-set size). The vertical axis gives processor utilization, a measure of the performance of the system as a whole. Although the performance metrics are different, the model outputs give the same qualitative information as the studies that use detailed implementations of the Alewife architecture.

7.1.1 Notation

Table 7.1 lists some basic notation used to describe the model. P stands for the number of processing nodes in a system and i denotes the number of pointers in each directory entry of a software-extended system.

The model uses two types of time intervals — or latencies — as parameters. T_x represents the time required to process x, some sort of memory transaction, in a software-extended system. $T_{x,hw}$ is the latency of the same transaction when it can be handled completely in hardware. In some memory systems, an event x may need to be handled partially in hardware and partially in software. t_x denotes the amount of time delay induced by the software component of the system. All time intervals are measured in units of processor cycles.

The model's metric of system performance is always processor utilization, represented by $\mathcal{U}$. This symbol is typically subscripted to indicate the model from which it is derived.

7.1.2 Application Parameters

The worker-set model parameterizes the behavior of a workload as a set of hit ratios at each level of the memory hierarchy, the latencies of access at each level, and its worker-set behavior. These inputs to the model may be derived from different sources, depending on the type of workload. For the benchmark applications, all of the model inputs come from statistics gathered by simulating each application with a $Dir_nH_{NB}S_-$

Symbol	Meaning
P	number of processors in a system
i	number of hardware directory pointers
T_x	the latency of x, an event in a memory system
$T_{x,hw}$	latency of x when using a hardware-only directory
t_x	the additional overhead of handling x due to software extension
$\mathcal{U}_M$	processor utilization predicted by model M

Table 7.1: Notation for the analytical model.

memory system. In a sense, the model uses a characterization of an application running on a hardware-only system to predict the performance on the gamut of software-extended systems.

WORKER, the synthetic workload, requires $Dir_nH_{NB}S_-$ simulations to determine its hit ratios and latencies; its worker-set behavior is completely determined by its structure and can be specified by examination. For this reason, WORKER was used to develop the model and to perform the initial model validation.

Other synthetic workloads may be created by specifying a complete set of model inputs without any data from simulations. Such workloads can not represent the performance of the system on real applications, but they are useful for examining the sensitivity of software-extended systems to an application's memory access behavior.

Table 7.2 lists all of the model inputs derived from applications or synthetic workloads. The first set of variables contains the hit ratios and access latencies. These parameters are the familiar quantities used to characterize a memory system with caches. The average memory access time for a hardware-only system ($T_{a,hw}$) may be calculated directly from them:

$$T_{a,hw} = hT_h + lT_l + rT_{r,hw} \tag{7.1}$$

The access latencies (T_x) are application-dependent, as opposed to constant parameters of the architecture, because they take contention into account. For example, T_h — the cache hit latency — incorporates cache access delays due to processor versus CMMU contention for the address/data bus. Similarly, T_l contains contention between local and remote requests for a node's DRAM and $T_{r,hw}$ incorporates contention within the interconnection network. Appendix A lists the values of model parameters for the six benchmarks.

Definition *A* worker set *is the set of processors that access a block of memory between subsequent modifications of the block's data.*

The worker set for a *read-only* block of memory, which is initialized but never modified, consists of all of the processors that ever read it. An *instantaneous worker set* is the set of processors that access a block of memory between an instant in time and the previous modification. At the time of a write, the instantaneous worker set is equal to the worker set.

Symbol	Meaning
N_i	number of instruction fetches
N_a	number of memory accesses
h	cache hit ratio
l	local memory access ratio
r	remote memory access ratio
T_h	cache hit latency
T_l	local memory access latency
$T_{r,hw}$	hardware-only remote access latency
R_k	number of reads with k processors in instantaneous worker set
W_k	number of writes with k processors in worker set

Table 7.2: Inputs to the analytical model from applications.

In order to minimize the amount of record-keeping, the model assumes that the memory system has a load/store architecture: all memory accesses are either reads or writes. Consequently, two sequences of parameters specify an application's worker-set behavior: the first sequence $\{R_k \mid 0 \leq k \leq P\}$ indicates the size of the instantaneous worker set at the time of a each read; the second sequence $\{W_k \mid 0 \leq k \leq P\}$ indicates the size of the worker set at the time of writes and dirty reads. A *dirty* read causes the Alewife protocols to invalidate a read-write copy of the requested data. This type of read transaction requires approximately the same amount of time as a write transaction that invalidates a single read-only copy of data.

The two sequences may be plotted as histograms, with worker-set size on the ordinate and the number of corresponding accesses on the abscissa. Figures 7-2 and 7-3 show the histograms for the six benchmarks. These histograms are produced by collecting traces of directory modification events during simulated runs of the benchmarks with $Dir_nH_{NB}S_-$. After the end of each simulation, a trace interpreter scans the event traces and constructs the histograms. The interpreter uses the following operational rules to construct the worker-set behavior histograms:

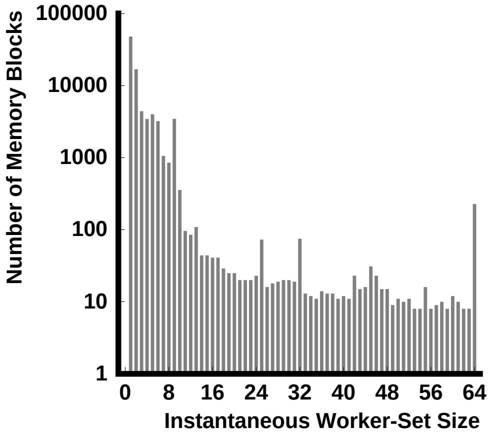
1. When a processor reads a data object, the bin with the corresponding instantaneous worker-set size is incremented once.

2. When a processor writes a data object, the bin with the corresponding worker-set size is incremented once.

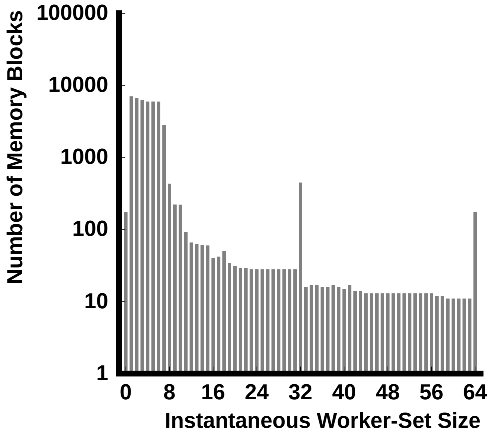
TSP exhibits the type of worker-set behavior that takes advantage of a software-extended system: the application primarily reads and writes data objects that are shared by only a few processors. There are also two memory blocks that are widely-shared and are distributed to all of the nodes in the system. The spike at the right end of Figure 7-2(a) indicates some residual thrashing of the widely-shared objects, even with the victim caches enabled. The thrashing causes extra accesses after the instantaneous worker sets reach the size of the corresponding worker set.

(a) Traveling Salesman Problem (TSP)

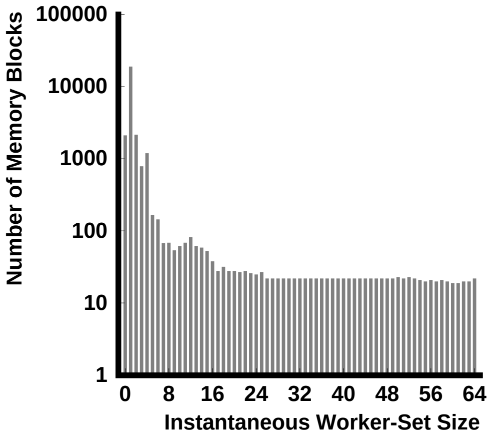
(b) Adaptive Quadrature (AQ)



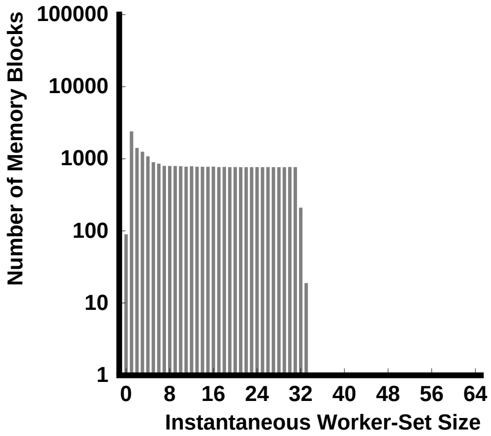
(c) Static Multigrid (SMGRID)



(d) Genome Evolution (EVOLVE)



(e) MP3D



(f) Water

Figure 7-2: Read access instantaneous worker-set histograms.

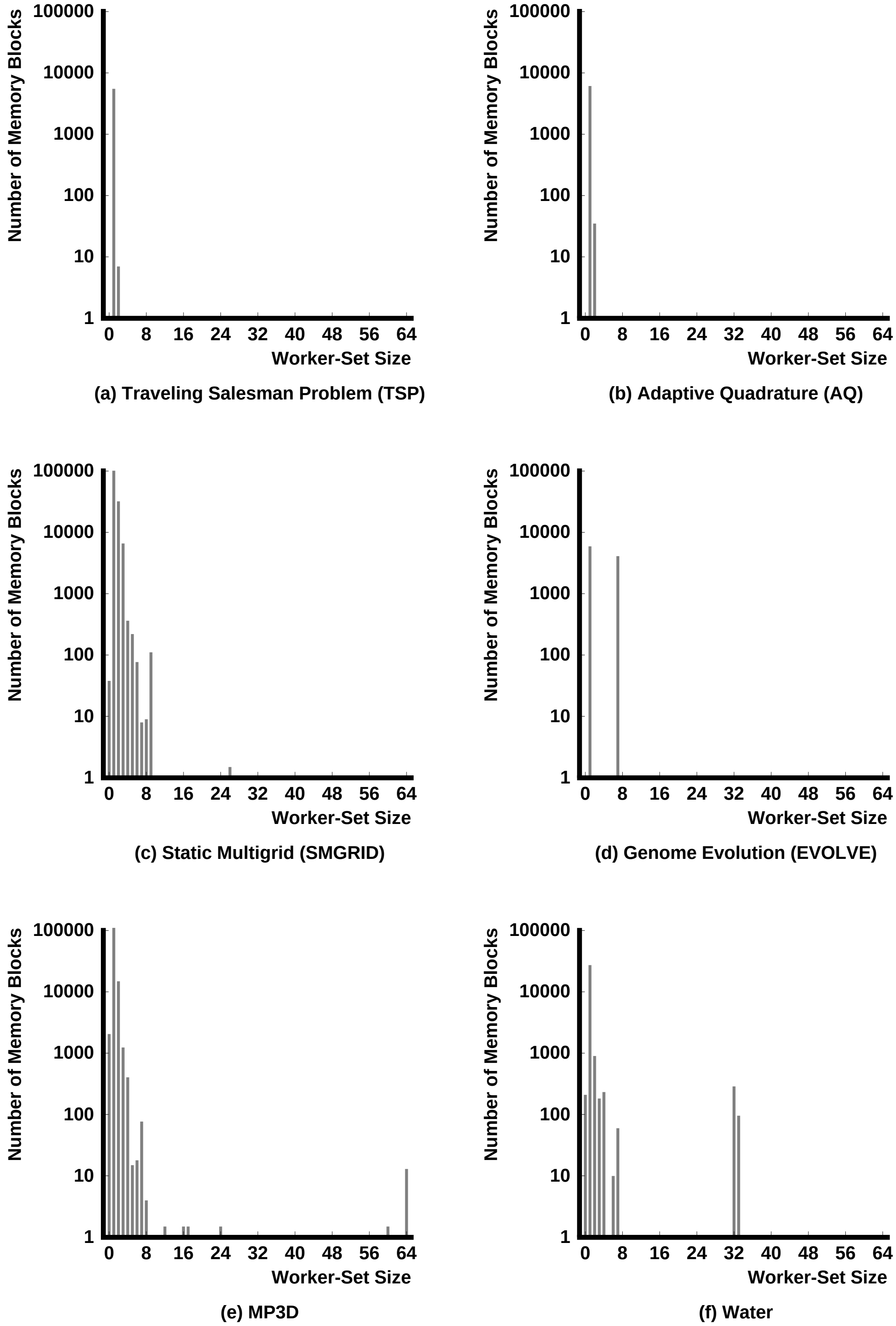


Figure 7-3: Write access worker-set histograms.

The worker-set histograms show that AQ performs more writes than reads. This behavior makes sense in light of the fact that synchronizing reads are treated as writes with respect to coherence. Since the application uses producer-consumer communication — orchestrated by the `future` construct, almost of the data is communicated through synchronizing reads and writes.

The other applications show similar worker-set profiles with some amount of widely-shared, read-only data and read-write data with predominately small worker sets. The histograms are useful for understanding the way that the benchmarks use shared memory and for interpreting their performance on shared-memory systems [78].

Software-only directories are particularly sensitive to the values of R_0, W_0, and W_1, which can be affected by purely intranode accesses. For the sake of completeness, the graphs in Figures 7-2 and 7-3 contain all intranode and internode accesses. However, the software-only directory implemented for Alewife ($Dir_nH_0S_{NB,ACK}$) includes a one-bit optimization for intranode data (see Section 3.1). For this reason, the model uses the values of R_k and W_k that do not include purely intranode accesses, unless otherwise specified.

A similar Alewife feature uses a special one-bit directory pointer reserved for the local node. Unless otherwise specified, the event trace interpreter assumes the existence of this pointer, and does not include the local node in the worker set when calculating R_k and W_k. Section 7.4.5 analyzes the effect of both one-bit optimizations.

7.1.3 Architecture Parameters

Since the analytical model is used primarily to investigate the performance of software-extended systems, its architectural parameters describe the time required to handle memory accesses in software. Table 7.3 lists the software-handling latency parameters and their measured values. These values are comparable to the measurements in Tables 6.1 and 6.2. The numbers are similar because they are also derived from the statistics gathered from WORKER simulations; they are different because they measure the average — not the median time — to execute each activity. Furthermore, many of the values given are the result of simple linear regressions that specify execution time as a function of the number of directory pointers or of worker-set size.

$t_{r,base}$ and $t_{r,extra}$ are an example of a pair of parameters derived from a linear regression of WORKER measurements. Together, they specify the time for memory-system software to process a read request (t_r) in an architecture with i hardware directory pointers:

$$t_r = t_{r,base} + i t_{r,extra}$$

All of the (*base,extra*) pairs in Table 7.3 are regression parameters. The other variables in the table are simple average latencies for software-extension processing. Section 7.2.2 explains how each of these parameters contributes to memory system latencies.

Some of the higher-level architectural parameters are incorporated into the model by modifying these parameters. For instance, Section 7.4.1 investigates the effect of trap latency (the time to clear the processor pipeline, save state, and switch to supervisor mode) on the performance of software-extended systems. The model generates extra

Symbol	Type of Latency	Value	Trap
$t_{r,prologue}$	read software handler prologue	130	$\surd$
$t_{r,base}$	base read software handling	205	$\surd$
$t_{r,extra}$	extra read software handling, per pointer	47	
$t_{w,base}$	base write software handling	605	$\surd$
$t_{w,extra}$	extra write software handling, per copy	12	
$t_{w,1,ack}$	1 pointer, handle acknowledgment	188	$\surd$
$t_{w,1,lack}$	1 pointer, handle last acknowledgment	452	$\surd$
$t_{r,0,sm,base}$	0 pointers (small wss), base read	322	$\surd$
$t_{r,0,sm,extra}$	0 pointers (small wss), extra read, per copy	11	
$t_{r,0,lg}$	0 pointers (large wss), read latency	433	$\surd$
$t_{w,0,sm,base}$	0 pointers, base write latency	388	$\surd$
$t_{w,0,sm,extra}$	0 pointers, extra write latency per copy	41	
$t_{w,0,lg,base}$	0 pointers, base write latency	1138	$\surd$
$t_{w,0,lg,extra}$	0 pointers, extra write latency per pointer	13	
$t_{w,0,ack}$	0 pointers, cost of handling acknowledgment	182	$\surd$
$t_{w,0,lack}$	0 pointers, cost of handling last acknowledgment	283	$\surd$

Table 7.3: Inputs to the analytical model from protocol measurements, values are in processor cycles.

trap latency by increasing the value of the parameters in Table 7.3 with a check-mark in the “Trap” column. Similarly, the study in Section 7.4.2 modifies code efficiency by dividing all of the t_x parameters by a constant factor.

Other architectural parameters require changes to the model's equations. These parameters include the number of directory pointers, special processors dedicated to memory-system software, and other details of protocol implementation.

7.1.4 Performance Metric

The goal of the model is to produce the same qualitative predictions as the results in the previous chapter, and then to extend the empirical results by exploring the design space. For reasons described in [17], the model can not be expected to calculate actual execution times or speedup. To summarize the discussion in [17], the model aggregates the architectural and application parameters over the entire duration of execution. This technique ignores actual forward progress in the application and neglects common multiprocessing phenomena such as network hot-spots and computation bottlenecks.

Nevertheless, the model can predict processor utilization, a measure of the amount of time that the program spends executing the program, as opposed to waiting for the memory system. More formally, processor utilization is defined as

$$\mathcal{U} = \frac{1}{1 + aT_a}, \tag{7.2}$$

Symbol	Meaning
$\mathcal{U}_{hw}$	processor utilization, hardware-only directory
$\mathcal{U}_{nfb}$	processor utilization, no feedback
$\mathcal{U}_{fb}$	processor utilization, with feedback
$\mathcal{U}_{phases}$	processor utilization, phases model
N_r	number of software-handled read requests
N_w	number of software-handled write requests
N_s	number of software-handled requests
N_d	number of requests delayed by software handling
a	memory access ratio
T_a	access latency
s	software handling ratio
T_s	software handling access latency
T_r	remote access latency
T_i	expected residual idle time during memory access
r_t	remote access ratio, time-average
L_{T_r}	expected residual life of remote accesses
t_s	software handling latency
t_r	read software handling latency
t_w	write software handling latency

Table 7.4: Derived parameters of the analytical model.

where $a = N_a/N_i$ is the memory access ratio and T_a is the average latency of access to memory.

The aT_a term gives the average fraction of time that each processor spends waiting for the memory system. Thus, the maximum processor utilization is $\lim_{aT_a \to 0} \mathcal{U} = 1$ and the minimum processor utilization is $\lim_{aT_a \to \infty} \mathcal{U} = 0$. Roughly speaking, $\mathcal{U} = 1$ corresponds to a system that achieves perfect speedup over sequential, and $\mathcal{U} = 0$ means that a system makes no forward progress whatsoever.

Preliminary studies that examined a wide range of options for the Alewife architecture [14] used processor utilization to compare different system configurations. Subsequent phases of the Alewife research determined that qualitative conclusions derived from this metric were generally correct.

7.2 Model Calculations

Most of the intuition about software-extended memory systems that drives the analytical model is incorporated into the equations that calculate processor utilization. The rest of the model requires little more than good accounting in order to count the number of occurrences and the latency of each type of event in the memory system. Table 7.4 lists all of the quantities derived as part of the analysis.

7.2.1 Calculating Utilization

The model uses equation 7.2 directly to calculate the processor utilization of a $Dir_nH_{NB}S_{-}$ system:

$$\mathcal{U}_{hw} = \frac{1}{1 + aT_{a,hw}}$$

The formula for utilization in software-extended systems must take into account both the additional memory access latency due to software and the cycles stolen from user code by the memory system. Substituting T_a (the predicted memory latency) for $T_{a,hw}$ accounts for the longer access latency. In order to deduct the stolen cycles from the system's performance, the equation requires an additional term:

$$\mathcal{U}_{nfb} = \frac{1}{1 + \underbrace{aT_a}_{request} + \underbrace{arst_s}_{response}}, \tag{7.3}$$

where r is the remote access ratio, s is the fraction of remote requests that require software handling, and t_s is the average time that the memory system software requires to process a request. Thus, each memory request takes, on average, T_a cycles of latency and steals rst_s cycles from some processor on the response side. Section 7.2.2 describes the model's calculations of T_a, s, and t_s.

This model of processor utilization assumes that every cycle the memory system steals from a processor could otherwise have been spent performing useful work for the application. While this assumption might be true for applications that achieve $\mathcal{U} \approx 1$, the penalty for stealing cycles in systems with lower utilization is not as extreme. It is possible that when the memory system needs to steal cycles from a processor, the processor is waiting for the memory system. Given this situation, there should be no penalty for stealing cycles, because the processor has no useful work to do.

This scenario leads to an interesting observation: the lower the processor utilization, the less likely that there will be a penalty for stealing cycles. This property of software-extended systems is actually a positive feedback loop. When such a system performs poorly, the extra processing that the system requires does not reduce the performance as it does when the system performs well.

Equation 7.3 does not take this feedback into account, so it calculates $\mathcal{U}_{nfb}$, the processor utilization without feedback. (*nfb* stands for no feedback.) In order to model positive feedback, the stolen-cycles term changes slightly:

$$\mathcal{U}_{fb} = \frac{1}{1 + aT_a + ars(t_s - (1 - \mathcal{U}_{fb})T_i)} \tag{7.4}$$

This term subtracts a number of idle cycles from t_s, the average number of stolen cycles: $(1 - \mathcal{U}_{fb})$ gives the time-average probability that a processor is idle, and T_i is the expected number of idle cycles remaining at the time that the memory system interrupts a processor. Solving this equation for $\mathcal{U}_{fb}$ yields a closed form expression for $\mathcal{U}_{fb}$:

$$(arsT_i)\mathcal{U}_{fb}^2 + (1 + aT_a + arst_s - arsT_i)\mathcal{U}_{fb} - 1 = 0$$

The problem of modeling T_i remains. The expected amount of idle time may be viewed as a stochastic process with time intervals drawn from the distribution of remote memory access latencies. For the sake of argument, assume that these time intervals are independent of each other and that the memory system interrupts are independent of the stochastic process. Then, the stochastic process is a renewal process and

$$T_i = \min(r_t L_{T_r}, t_s),$$

where r_t is the time-average probability that a processor is making a remote access, given that it is making any memory access; and L_{T_r} is equal to the time-average residual life of the renewal process, or

$$L_{T_r} = \frac{E[T_r^2]}{2E[T_r]}. \tag{7.5}$$

It is necessary to take the minimum of $r_t L_{T_r}$ and t_s so that the number of idle cycles in equation 7.4 never exceeds the number of cycles required for software-extension.

While this model of T_i yields some valuable intuition, it does not provide a good approximation of the behavior of either the synthetic workload or the benchmarks. Calculations show that the model constantly overestimates the utilization, sometimes by more than a factor of two. The failure of this model suggests that the remote latency process is not a renewal process.

Since the renewal process model does not yield accurate results, and since the calculation of L_{T_r} is relatively tedious, the details of the residual life calculation will be omitted. To summarize the L_{T_r} derivation, both the expectation and the variance of each component of the remote latency (T_r) must be calculated. These values are propagated through to the residual life equation 7.5.

Some thought about the nature of shared memory accesses in multiprocessor programs sheds some light on the remote latency process: in many parallel algorithms, when one processor makes a shared memory access, all of the other processes are also making similar accesses. Thus, the accesses of requesting and responding nodes should be highly correlated. Taking this observation to the logical extreme, a model could assume that:

Phases Hypothesis *If one node's remote memory request coincides with a remote access on the responding node, then the responding node's access is exactly the same type of access.*

The name *phases* hypothesis indicates the assumption that computation proceeds in phases, during each of which all processors execute similar functions. This intuition is especially true for single-program multiple-data applications like WORKER, which synthesizes the same memory access pattern on every processor. The hypothesis leads to a new value of T_i: $(r_t t_s/2)$ replaces $(r_t L_{T_r})$. r_t corresponds to the predicate "if one node's remote memory request coincides with a remote access on the responding node;" $t_s/2$ corresponds to the consequent "then the responding node's access is exactly the same type of access." This term is $t_s/2$ rather than t_s, because the expected residual life when entering at a random time into an interval of *deterministic* length Z is $Z/2$. Thus,

$$\mathcal{U}_{phases} = \mathcal{U}_{fb}[T_i = (r_t t_s/2)] \tag{7.6}$$

Section 7.3 compares the validity of the model without feedback and the phases model. The rest of this section describes the derivation of all of the parameters in the utilization equations.

7.2.2 Counting Events and Cycles

The parameters in equations 7.3, 7.4, and 7.6 that the model must calculate are T_a, s, T_s, r_t, and t_s. Average memory access latency may be stated in terms of the other parameters:

$$\begin{aligned} T_a &= hT_h + lT_l + rT_r \\ &= hT_h + lT_l + r(T_{r,hw} + sT_s) \\ &= T_{a,hw} + rsT_s \end{aligned} \tag{7.7}$$

This equation uses the standard formulation for a memory hierarchy, with T_r representing the average remote access latency in a software-extended system. Note that the formula for T_r is exactly the same as the simple model used in Section 4.1.2. Given T_a and T_r, the fraction of memory access time spent processing remote requests may be calculated as $r_t = rT_r/T_a$.

Since the model assumes a load/store memory architecture, the fraction of remote requests that require software handling (s) and the associated overhead (t_s) can be broken down into their read and write components. Let N_s be the total number of software-handled requests:

$$N_s = N_r + N_w,$$

where N_r and N_w are the number of software-handled read and write requests, respectively. Thus,

$$\begin{aligned} s &= \frac{N_s}{rN_a} \\ t_s &= \frac{N_r t_r + N_w t_w}{N_s}, \end{aligned}$$

where t_r and t_w are the average latencies for using software to handle read and write requests.

The memory access latency seen on the processor side (T_s) also has read and write components, but the breakdown depends on the implementation of the memory system: the Alewife architecture includes a feature that allows the hardware to transmit read-only copies of data in parallel with the associated software handling. With this *hardware read transmit* capability, read transactions may require software handling, but the software is not in the critical path from the point of view of the requesting processor. Thus, the number of requests delayed by software handling (N_d) depends on this architectural feature:

$$N_d = \begin{cases} N_w & \text{with hardware read transmit} \\ N_r + N_w & \text{without hardware read transmit} \end{cases}$$

T_s also depends on the same feature:

$$T_s = \begin{cases} t_w & \text{with hardware read transmit} \\ \frac{N_r t_{r,prologue} + N_w t_w}{N_d} & \text{without hardware read transmit} \end{cases},$$

where $t_{r,prologue}$ is the amount of time required before the software can transmit a block of data. The components of the above quantities (N_r, N_w, t_r, and t_w) all depend on the specific software-extended protocol and on each application's worker-set behavior. For example, they are all zero for $Dir_nH_{NB}S_{-}$. The next few paragraphs detail the derivation of the parameters for the other protocols.

$\underline{Dir_nH_1S_{NB} \leftrightarrow Dir_nH_{n-1}S_{NB}}$ For the directory protocols with two or more hardware pointers per entry[1], calculating the number of software-extended writes (N_w) and their average latency (t_w) is straightforward:

$$\begin{aligned} N_w &= \sum_{k=i+1}^{P} W_k \\ t_w &= t_{w,base} + \frac{\sum_{k=i+1}^{P} (k t_{w,extra}) W_k}{N_w} \end{aligned}$$

Calculating the number of software-extended reads (N_r) and average latency (t_r) is slightly more complicated, due to an option in the implementation of the extension software. After the first directory overflow interrupt, the software can choose to handle all subsequent read requests. The software can also choose to reset the hardware directory pointers, thereby allowing the hardware to continue processing read requests. Not only does the *pointer reset option* allow parallelism between the hardware and software, it allows every software-handled request to be amortized over the number of hardware directory pointers. The following equations count the number of events and cycles with and without this option:

$$\begin{aligned} N_r &= \begin{cases} R_i + \frac{\sum_{k=i+1}^{P} R_k}{i+1} & \text{with pointer reset option} \\ \sum_{k=i}^{P} R_k & \text{without pointer reset option} \end{cases} \\ t_r &= \begin{cases} t_{r,base} + i t_{r,extra} & \text{with pointer reset option} \\ \frac{N_r t_{r,base} + i R_i t_{r,extra}}{N_r} & \text{without pointer reset option} \end{cases} \end{aligned}$$

$\underline{Dir_nH_1S_{NB,ACK} \text{ and } Dir_nH_1S_{NB,LACK}}$ All of the event and cycle counts for the single-pointer protocols are the same as for the multi-pointer protocols, with the exception of the average software-extended write latency. t_w is longer because the one-pointer protocols must handle one or more acknowledgment messages in order to process write transactions. For $Dir_nH_1S_{NB,ACK}$,

$$t_w = \frac{\sum_{k=2}^{P} (t_{w,base} + k t_{w,extra} + (k-1) t_{w,1,ack} + t_{w,1,lack}) W_k}{N_w},$$

[1] $Dir_nH_1S_{NB}$ implements enough storage for two pointers.

and for $Dir_n H_1 S_{NB,LACK}$,

$$t_w = \frac{\sum_{k=2}^{P} (t_{w,base} + k t_{w,extra} + t_{w,1,lack}) W_k}{N_w}$$

The amount of time to handle an acknowledgment message in the middle of a transaction is $t_{w,1,ack}$, and $t_{w,1,lack}$ cycles are required to handle the last acknowledgment before completing the transaction.

$Dir_n H_0 S_{NB,ACK}$ The software-only directory architecture is considerably more complicated to model, because it has a much wider range of software-handled transactions. In addition, the Alewife version of $Dir_n H_0 S_{NB,ACK}$ attempts to optimize the time and memory space required to handle small worker sets. Consequently, the model differentiates between transactions involving small (*sm*) and larger (*lg*) worker-set sizes. The equations used to count events and cycles are otherwise similar to the calculations above:

$$\begin{aligned}
N_r &= \sum_{k=0}^{P} R_k \\
N_w &= \sum_{k=0}^{P} W_k \\
t_r &= \frac{\sum_{k=0}^{3} (t_{r,0,sm,base} + k t_{r,0,sm,extra}) R_k + \sum_{k=4}^{P} t_{r,0,lg} R_k}{N_r} \\
t_w &= \frac{\sum_{k=0}^{4} (t_{w,0,sm,base} + k t_{w,0,sm,extra} + (k-1) t_{w,0,ack} + t_{w,0,lack}) W_k}{N_w} + \\
&\quad \frac{\sum_{k=5}^{P} (t_{w,0,lg,base} + k t_{w,0,lg,extra} + (k-1) t_{w,0,ack} + t_{w,0,lack}) W_k}{N_w}
\end{aligned}$$

7.2.3 Restrictions and Inaccuracies

As specified in the preceding section, the model calculations are valid only for the software-extended protocols used in Alewife. However the model is purposely built with an abstraction for the type of software-extension scheme: four values (N_r, N_w, t_r, and t_w) encapsulate the protocol. Given the specification of a protocol not described above, it would be relatively easy to recalculate values for these four values and to run the model with these values.

Even though intuition behind multiprocessor algorithms and software-extended systems drives the model, it is not obvious that the model can produce accurate predictions of system performance. Certain assumptions of independence are particularly troublesome. For example, all of the input parameters (such as access ratios and latencies) are assumed to be independent of the amount and latency of memory accesses handled by software. This assumption should prove to be valid given large processor caches and a high-bandwidth interconnection network, but certainly can not be verified *a priori.* The next section proves that despite the simplicity of the model, its results match the performance of the experimental systems described in Chapter 6.

7.3 Validating the Model

Before using the analytical model to make any predictions, it is important to validate it by comparing its predictions against known experimental values. This section uses both qualitative and quantitative methods to compare the model's predictions with the empirical results from the previous chapter. To summarize the results of this study:

- The model always matches the qualitative results of the experimental data.
- The *phases* model predicts the behavior of the synthetic WORKER application better than the model without feedback.
- The *no feedback* model exhibits slightly less systematic error than the *phases* model when trying to predict the performance of the six benchmarks.

7.3.1 The WORKER Synthetic Workload

Using the WORKER synthetic workload together with the NWO simulation system allows experiments with the entire cross-product of worker-set sizes and software-extended systems. This software provides a good apparatus for developing and training the analytical model. Figure 7-4 illustrates the results of this process by using the *phases* model to compare the performance of various software-extended systems to $Dir_nH_{NB}S_{-}$ performance.

The graph in the figure reproduces the one in Figure 6-1 on page 68, which uses simulation data to make the same comparison. Note that the performance metrics are not exactly the same in the two figures! Figure 7-4 uses the ratio of processor utilizations, while Figure 6-1 uses the (inverted) ratio of execution times. Were it not for the small anomalies in the empirical data, it would be difficult to tell the difference between the two sets of curves. Qualitatively, they are the same.

The results for $Dir_nH_0S_{NB,ACK}$ require some elaboration. Recall that the worker-set behavior of WORKER is completely determined by its structure. A synthetic histogram generator creates the worker-set behavior (R_k and W_k) values required to produce the results in Figure 7-4. When using the basic outputs of this histogram generator, the model does not predict the rise in performance at the left end of the $Dir_nH_0S_{NB,ACK}$ curve (corresponding to small worker-set sizes).

Examination of the detailed statistics generated by NWO reveals a thrashing situation in each processor's cache. This thrashing causes one node to send read-write copies of memory blocks back to memory just before other nodes read the same blocks. This scenario improves performance by converting dirty reads (similar in latency to writes) into clean reads that do not require a protocol to send invalidations. Eliminating the thrashing in the simulated system proves to be a difficult task, but adjusting the synthetic histogram generator is easy. The $Dir_nH_0S_{NB,ACK}$ curve in Figure 7-4 corresponds to the data with the corrected worker-set histograms. Although the model underestimates the software-only directory performance slightly, it does show the same performance increase at low worker-set sizes.

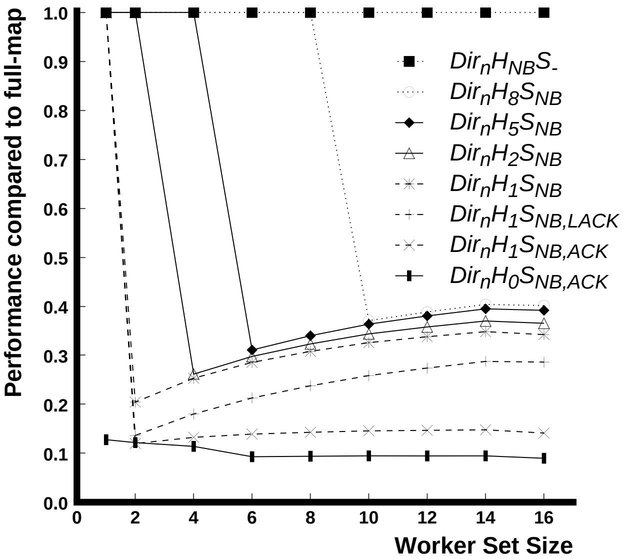


Figure 7-4: Predicted performance of the synthetic workload, used to validate the model. Compare to Figure 6-1.

Figure 7-5 presents a quantitative comparison of the model and the experimental data. The scatter plots in the figure show the percent error of the model versus the number of hardware directory pointers. The error in the model is defined as

$$\begin{aligned} \text{error} &= \text{empirical result} - \text{model's prediction} \\ &= \frac{\text{execution cycles for } Dir_n H_{NB} S_-}{\text{execution cycles for } Dir_n H_X S_{Y,A}} - \frac{\mathcal{U}_{model}}{\mathcal{U}_{hw}} \end{aligned}$$

The percent error is the error divided by the empirical result, multiplied by 100. The figure shows percent error for both $\mathcal{U}_{nfb}$ (a) and $\mathcal{U}_{phases}$ (b).

The dashed line on the plots indicates no error. Following from the definition of error, positive error (above the line) indicates that the model has underpredicted the system performance; negative error (below the line) indicates an overprediction. Since both models ignore contention, they should both overpredict performance: however, the model without feedback systematically underpredicts the system performance, while the phases model overpredicts performance. Thus, the feedback modeled in $\mathcal{U}_{phases}$ is present in the synthetic benchmark and has a second-order effect on performance.

Table 7.5 summarizes the characteristics of the error distribution. It lists the distributions expectation (E[error]) and standard deviation (σ[error]), as well as the 90% confidence interval as defined in [35]. The low absolute expected error indicates a good match between the phases model and the synthetic benchmark.

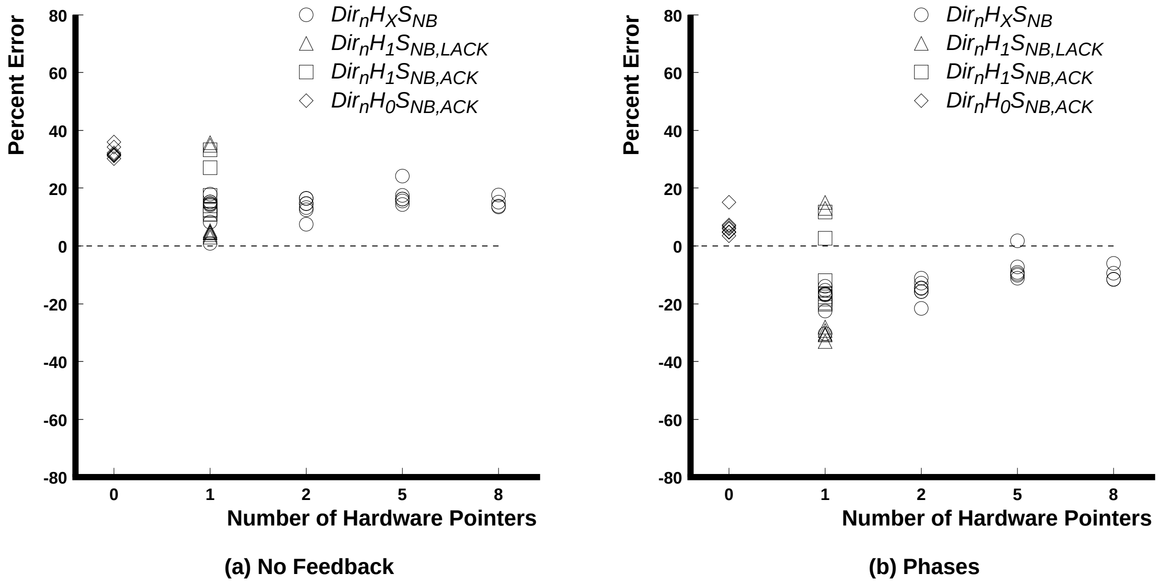


Figure 7-5: Model error in predictions of synthetic workload performance.

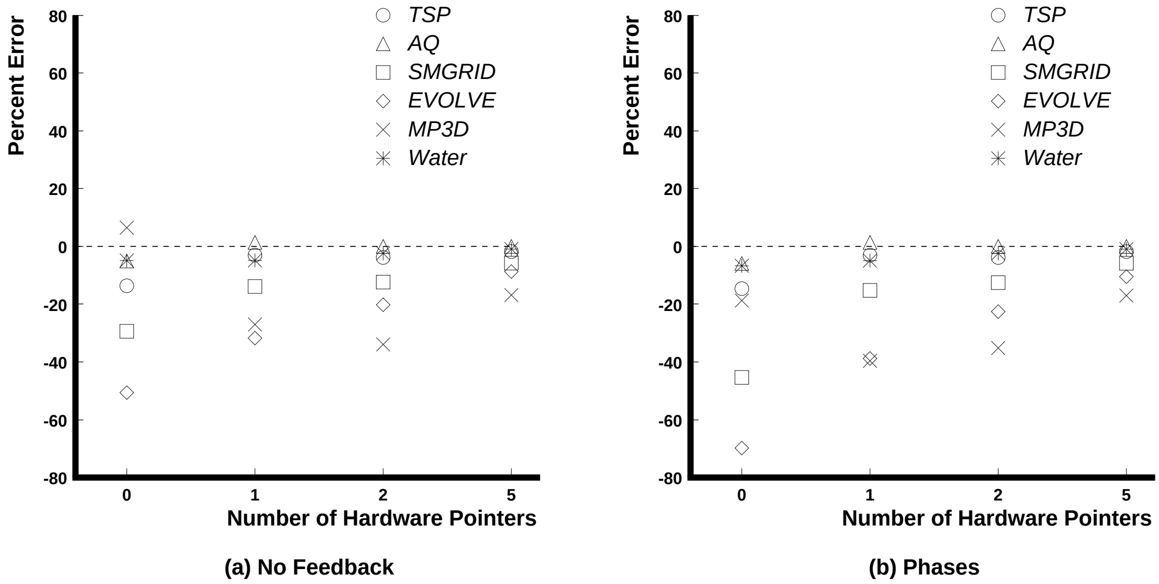


Figure 7-6: Model error in predictions of benchmark performance.

Workload	Model	E[error]	σ[error]	90% Confidence Interval
Synthetic	No Feedback	0.0361	0.0186	(0.0317,0.0404)
Synthetic	Phases	-0.0262	0.0259	(-0.0322,0.0202)
Benchmarks	No Feedback	-0.0662	0.0600	(-0.0876,-0.0447)
Benchmarks	Phases	-0.0777	0.0672	(-0.102,-0.0537)

Table 7.5: Summary of error in analytical model.

7.3.2 The Benchmarks

The *nfb* model's predictions for the benchmarks, which are graphed in Figure 7-7, also match the qualitative results of the empirical study. All of the major conclusions derived from Figure 6-2 on page 70 could be derived equally well from the analytical model.

However, predicting the performance of applications is by nature more complicated than modeling a synthetic workload. In fact, the quantitative comparison in Figure 7-6 shows some systematic error in the model. The error in this figure is calculated as

$$\begin{aligned} \text{error} &= \text{empirical result} - \text{model's prediction} \\ &= \frac{\text{speedup for } Dir_n H_X S_{Y,A}}{\text{speedup for } Dir_n H_{NB} S_{-}} - \frac{\mathcal{U}_{model}}{\mathcal{U}_{hw}} \end{aligned}$$

Figure 7-6 shows that the model without feedback tends to overestimate the system's performance, and the phases model makes slightly worse predictions. Table 7.5 quantifies the systematic error. The error is probably caused by the fact that the model ignores network hot-spots and computation bottlenecks, which are serious factors in real multiprocessor applications. The error is particularly high for the software-only directory ($Dir_n H_0 S_{NB,ACK}$), which achieves such low performance that small mispredictions look relatively large on a percent error graph.

The reader should bear in mind that small differences in processor utilization are not significant; yet, the larger percent error for $Dir_n H_0 S_{NB,ACK}$ usually does not obscure the qualitative conclusions from the model. Since $Dir_n H_0 S_{NB,ACK}$ requires more software than any of the other software-extended memory systems, the software-only directory tends to exhibit much more sensitivity to architectural features.

In fact, the model never misorders the performance of different software-extended schemes. Thus, the model may be used to choose between alternative implementations, if not to predict their relative performance exactly.

7.4 Model Predictions

The real utility of the analytical model is its ability to extend the exploration of software-extended memory systems beyond the limits of the Alewife implementations. This section uses the model to investigate the effect of a number architectural mechanisms and features, including trap latency, code efficiency, dedicated memory processors,

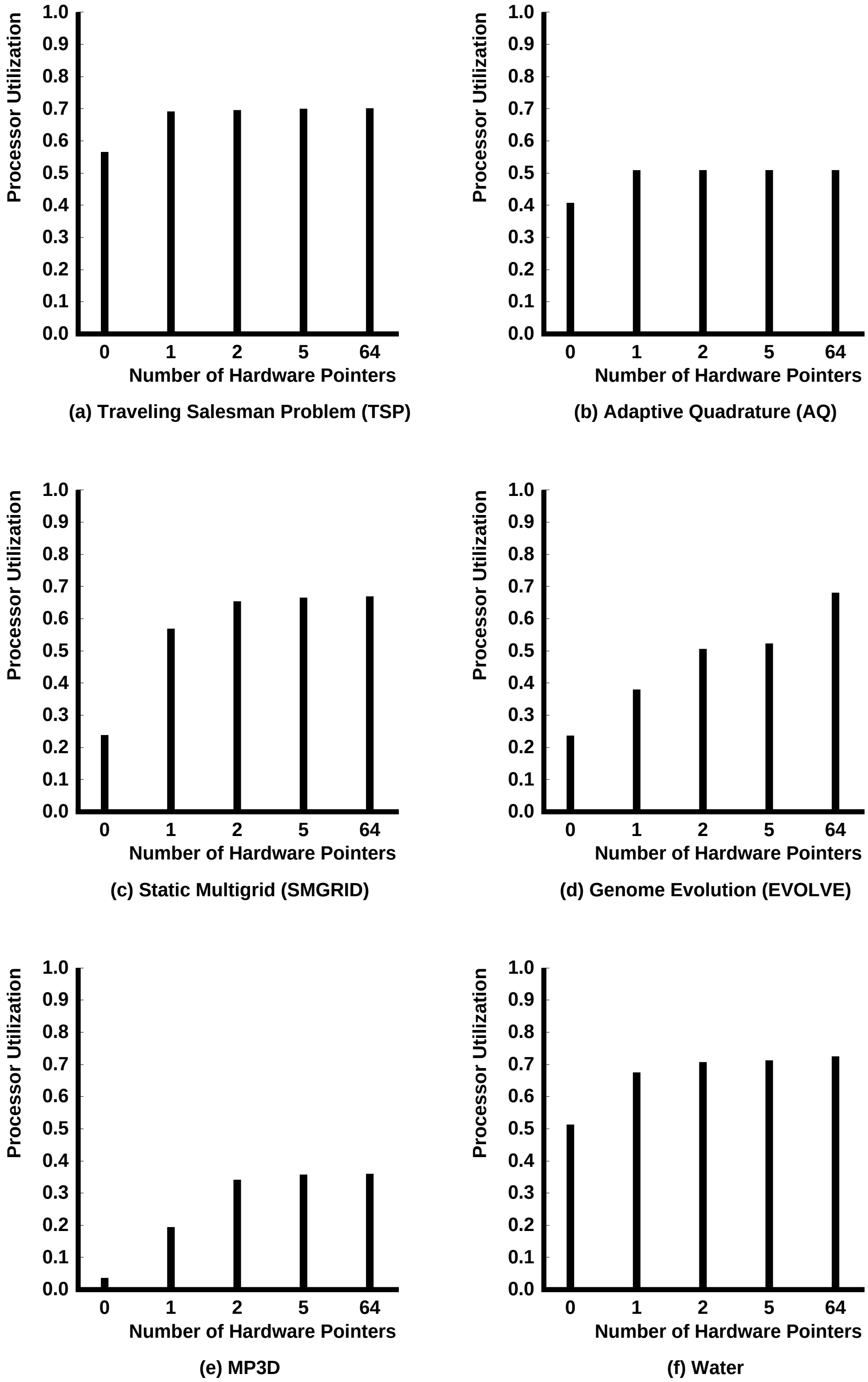


Figure 7-7: Predicted performance of benchmark applications, used to validate the model. Compare to Figure 6-2.

network speed, and protocol implementation details. The time required to perform this kind of study by fabricating actual hardware or by building and running a simulation system would have been prohibitive.

The figures presented below differ slightly from the graphs in previous sections. The performance metric on the vertical axis is always raw processor utilization, while the horizontal axis shows the number of hardware directory pointers. All of the figures plot the model's predictions for a 64 processor system. In order to provide a reference, every graph contains a curve marked with unfilled triangles. This curve gives the performance of the stock Alewife architecture, enhanced only to implement the full range of hardware pointers.

The units of the x-axis require some explanation: since many of the architectural features discussed below primarily affect the protocols with small hardware directories, the graphs emphasize the left end of the scale by using a linear scale. The associated curve segments are plotted with solid lines. The right end of the scale is approximately logarithmic, plotting 8, 12, 16, 32, and 64 hardware pointers. Dashed curve segments indicate this portion of the graph.

7.4.1 Trap Latency

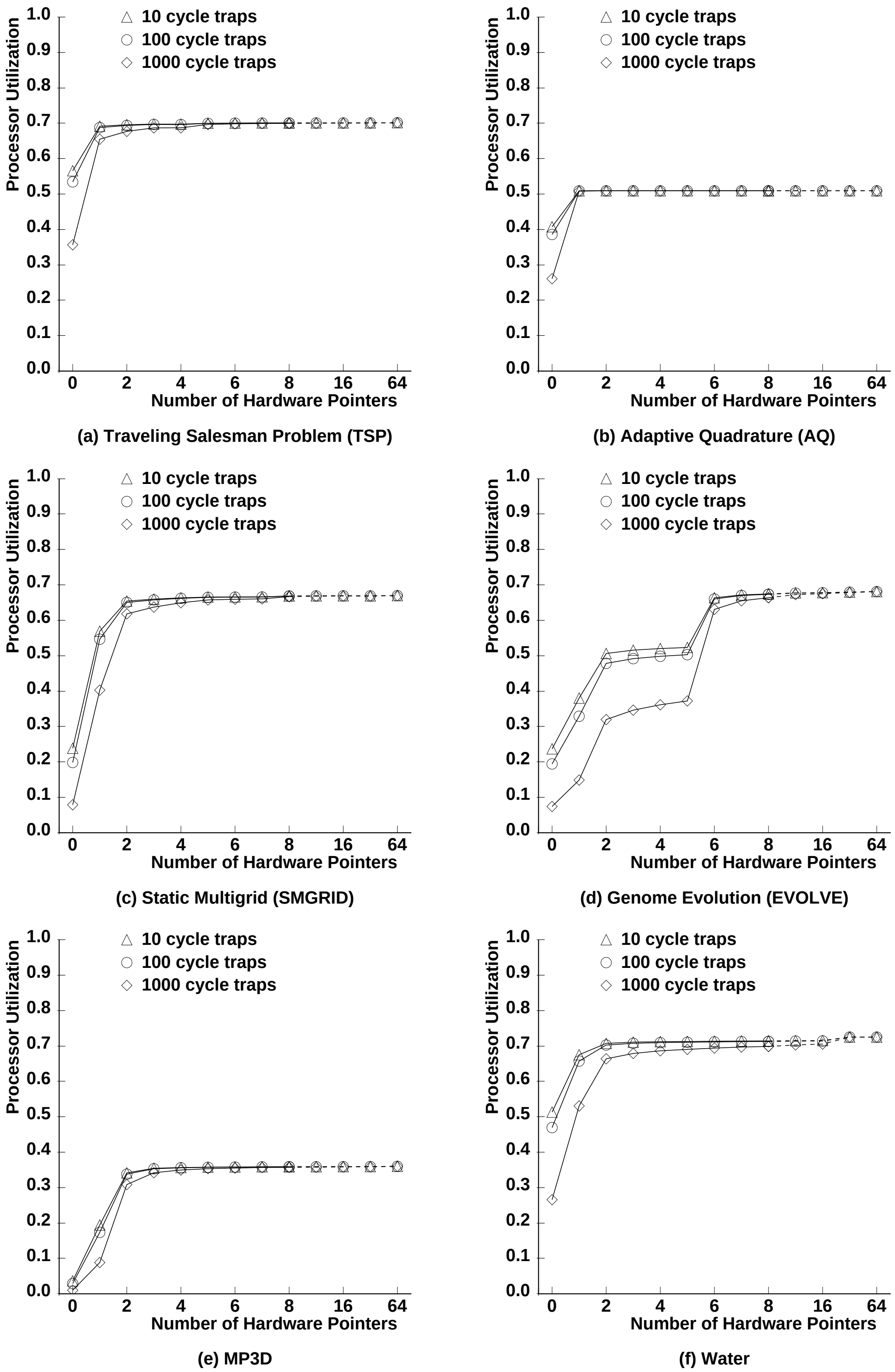
One of the most common questions about the Alewife memory system involves the relationship between software-extended performance and the trap latency of the system. Trap latency is defined as the time between the instant that the processor receives an asynchronous interrupt (or synchronous exception) and the time that it processes the first instruction of the interrupt handling code. (For the purposes of this discussion, the prologue instructions that save machine state are not considered part of the handling code.)

Due to Alewife's mechanisms for performing a fast context switch, its trap latency is about 10 cycles (see Table 6.2), plus 3 cycles to flush the processor's pipeline. This trap latency is extremely low, especially compared to heavy-weight process switch times, which can take thousands of cycles in popular operating systems. What happens to the performance of a software-extended system when trap latency increases?

Figure 7-8 answers this question by showing the model's predictions for systems with 10, 100, and 1000-cycle trap latency. The 10 cycle latency indicates the aggressive Alewife implementation; 100 cycles corresponds to an architecture that does not incorporate special features for context-switching; 1000 cycles indicates a reasonable lower bound on performance.

The figure shows that, within an order of magnitude, trap latency does not significantly affect the performance of software-extended systems. Given appropriate hardware support — in the form of enough directory pointers to contain major application worker sets — even two orders of magnitude does not cause a dramatic change in performance. In other words, an architecture similar to Alewife in all respects except trap latency would not perform significantly different on any of the benchmarks except EVOLVE. This benchmark would suffer by a factor of about 30% with the long trap latency.

At the other end of the scale, the software-only scheme does show sensitivity to large changes in trap latency. 1000-cycle trap latency causes $Dir_nH_0S_{NB,ACK}$ performance



(a) Traveling Salesman Problem (TSP)

(b) Adaptive Quadrature (AQ)

(c) Static Multigrid (SMGRID)

(d) Genome Evolution (EVOLVE)

(e) MP3D

(f) Water

Figure 7-8: Modeled performance of software-extended schemes with trap latency increased to 100 cycles and 1000 cycles.

to suffer by factors ranging from 36% to 72%. Section 7.4.3 explores a different implementation of software-only directories that avoids traps entirely.

7.4.2 Code Efficiency

Another factor that affects the performance of software-extended protocols is the quality of the memory-system software itself. Section 6.1 analyzes the difference between protocol handlers written with the flexible coherence interface, versus hand-tuned versions of the same handlers. The investigation shows that hand-coding can increase the efficiency of the memory system by a factor of 2 or 3.

Figure 7-9 continues this investigation by predicting the increased performance of the benchmark suite, assuming that protocol code efficiency can be increased by a factor of 2 or 4. The speed of the handlers significantly affects the performance of the software-only directory, but has no effect on directories that are large enough to contain each benchmark's worker sets.

This result suggests a trade-off between cost and flexibility: a flexible interface generally requires layers of abstraction, and therefore lower code efficiency than a less flexible system may achieve. Increasing the cost of a system by adding directory pointers (DRAM chips) essentially buys flexibility, or the ability to add abstraction without incurring a performance penalty. Unfortunately, experience shows that flexibility is almost indispensable when writing complex protocols such as $Dir_nH_0S_{NB,ACK}$. This point will surface again in the following section.

In any case, gratuitously poor code quality never benefits a system. Over time, both the Alewife compiler and appropriate portions of the flexible coherence interface will improve. Faster protocol handlers should have an impact on benchmark performance, especially if they accelerate the system during bottleneck computations, which are neglected by the analytical model. For example, Figure 6-4 shows the performance of TSP running on 256 nodes. In this configuration, the time required to distribute many copies of data impacts the performance of the software-extended systems. Faster protocol handlers would help reduce the impact of this type of transient effect.

7.4.3 Dedicated Memory Processors

The sensitivity of $Dir_nH_0S_{NB,ACK}$ to trap latency and code efficiency suggests an alternative design approach for multiprocessor memory systems: build two-processor nodes as in *T [60], and dedicate one of the processors exclusively to handling software for a memory system. Kuskin [51] and Reinhardt [66] claim that such a methodology results in flexible, high-performance systems.

Figure 7-10 uses the model to examine these claims. In order to predict the performance of dual processor systems, the model uses the processor utilization equation 7.2 (without the software-extension term), but uses equation 7.7 to predict the average memory access latency. The figure shows the performance of the base software-extended system (triangles) with one processor and a communications and memory management unit (CMMU), a two-processor system using the same memory-system software (circles), and a two-processor system than manages to increase code-efficiency by a heroic

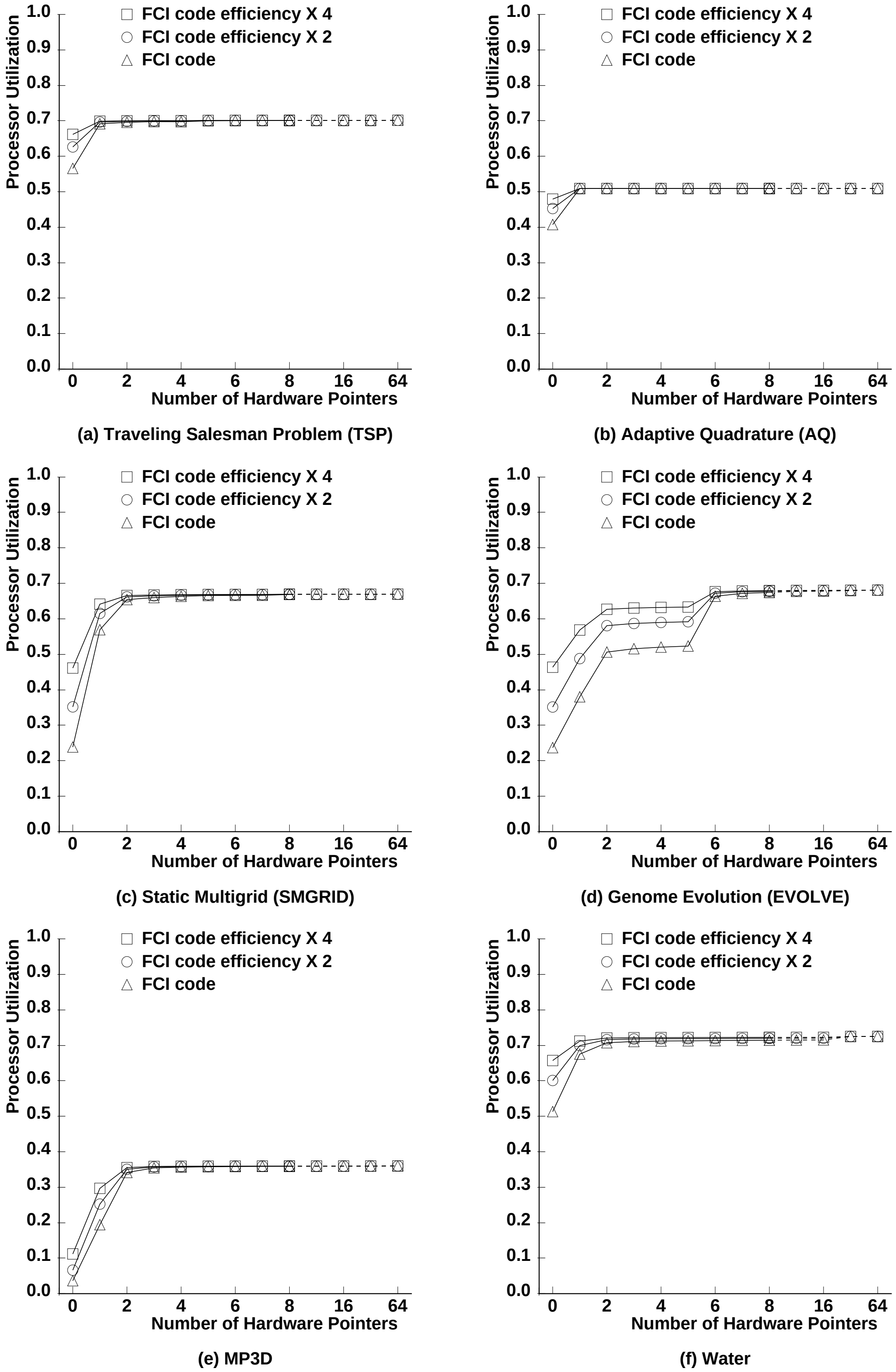


Figure 7-9: Modeled performance of software-extended schemes with code efficiency improved by a factor of 2 and a factor of 4.

factor of 10 (squares).

The model predicts that a dedicated processor and limited directories are mutually exclusive hardware acceleration techniques: the increase in performance achieved by combining these two features never justifies their combined cost. In fact, EVOLVE shows the only significant increase in performance generated by a second processor. Chapter 8 discusses costless changes to the LimitLESS system that produce similar gains in performance.

It is important to note that simply dedicating a processor to $Dir_nH_0S_{NB,ACK}$ (represented by the unfilled circle at $i = 0$) does not produce a system that achieves the same performance as the Alewife $Dir_nH_5S_{NB}$ (represented by the unfilled triangle at $i = 5$). For all of the benchmarks, the code efficiency on the two processor systems must reach a factor of about 10 in order to achieve comparable performance.

In practice, it is difficult to implement such fast memory-system software without seriously sacrificing the quality of abstraction, and therefore flexibility. Dual processor systems require a combination of special processor mechanisms implemented specifically for memory-system software and either hand-tuned code or a completely new type of compilation technology.

Up to this point, the analysis of dual processor systems has assumed that all of the $Dir_nH_0S_{NB,ACK}$ implementations take advantage of Alewife's optimization for purely intranode accesses. (See Section 3.1.) The filled triangle, circle, and square on the graphs in Figure 7-10 show the effect of removing this optimization from each of the proposed systems. The model predicts that three out of the six benchmarks would perform significantly worse without this optimization. Thus, dual processor systems are extremely sensitive to the efficiency of the memory system software, especially when the software impacts purely intranode accesses.

7.4.4 Network Speed

What range of operating conditions require the acceleration afforded by hardware directories or by a dual processor architecture? One of the key parameters of a multiprocessor architecture is the network speed, measured by the latency of internode communication. In the Alewife system, the network latency is about the same order of magnitude as the hardware memory access time. Figure 7-11 presents graphs that show the relationship between network speed and software-extended systems.

The model adjusts network latency by multiplying the average remote access latency ($T_{r,hw}$) by factors of 5, 10, 20, and 100. The top curve in each graph shows the predictions for the base system, and the key indicates the base value of $T_{r,hw}$. High-performance multiprocessors should have network latencies close to or faster than Alewife's; communication in networks of workstations is between 20 and 100 times slower than in Alewife. The new adjustment constitutes a major — and unvalidated — change in the model! Thus, the graphs certainly do not predict performance accurately, but they do show the relative importance of different parameters of the model .

For all of the benchmarks, the relative benefit of the hardware directory pointers decreases as network latency increases. This trend indicates that hardware acceleration is not useful for systems with slow networks. In fact, the curves in Figure 7-11 indicate

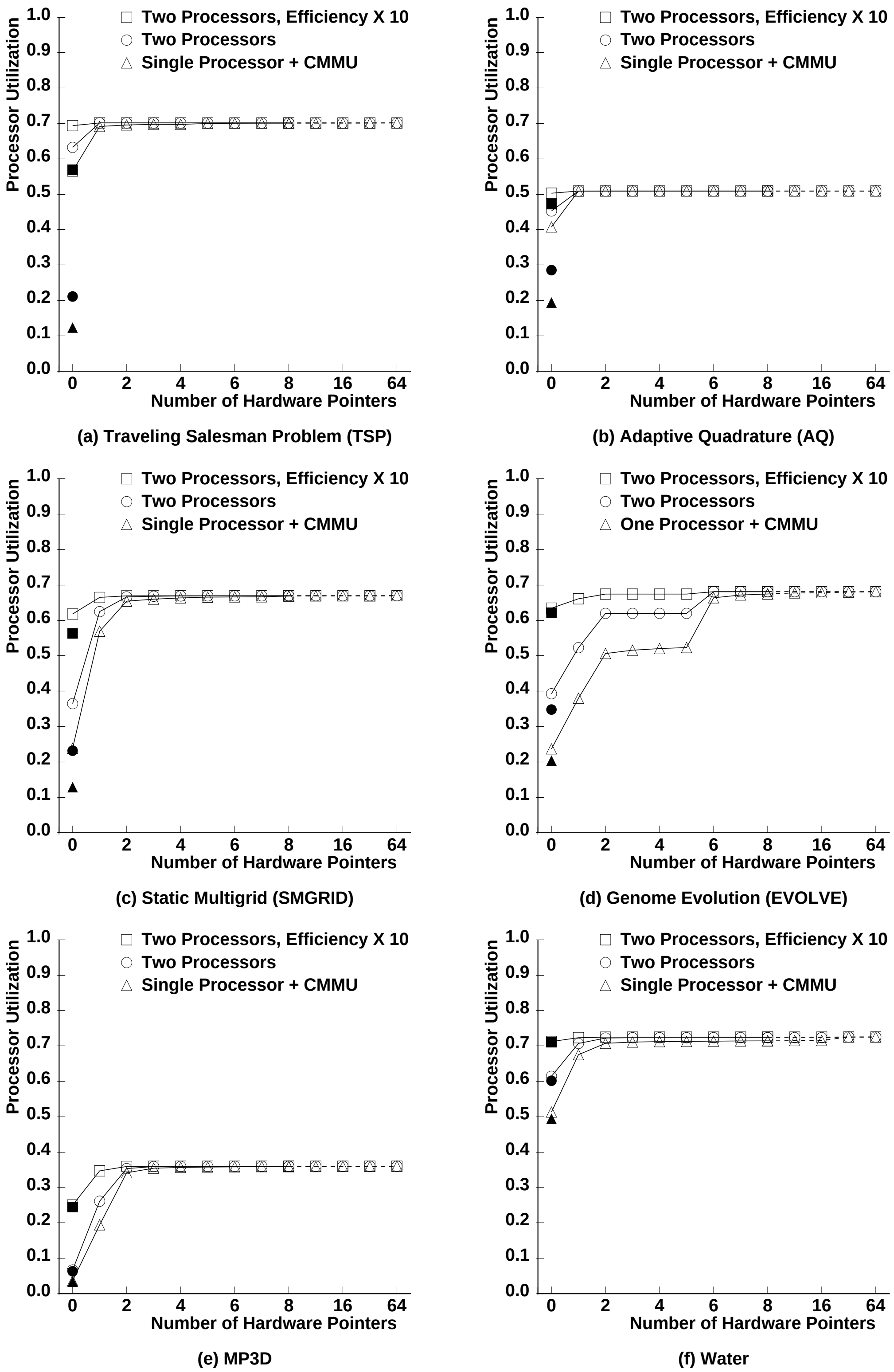


Figure 7-10: Modeled performance of software-extended schemes with three different architectures. Solid symbols show the performance of the zero-pointer scheme without hardware support for purely intranode accesses.

that a software-only directory does not seriously impact performance when remote access latencies are high.

Of course, the graphs also show that as the network latency increases, performance decreases to an unacceptable level. However, the model's predictions about absolute performance are predicated on a very specific memory hierarchy: namely, one that performs 16-byte transfers to fetch data into 64 Kbyte caches. Actual systems with long network latencies must transfer much larger blocks of data in order to amortize the network latency. If such systems can tolerate long communication delays, they should also be able to tolerate the extra processing required by software-only directories.

For systems with slow networks, software-only directories (or even completely software distributed shared memory) should provide reasonable alternatives for implementing shared memory. This conclusion leads to an interesting scheme for using the flexible coherence interface to implement a range of memory systems. Section 8.5 describes this scheme.

7.4.5 Protocol Implementation Details

This section examines the effects of a number of details of coherence protocol implementation. These details tend to have only a minor impact on the cost and the performance of software-extended systems. Although the conclusions are not terribly surprising, this investigation shows that the model is useful for understanding both important architectural mechanisms and the inner workings of a system.

Pointer reset Alewife's pointer reset feature allows the software to choose to reset the hardware directory pointers. This mechanism allows the system to avoid $i/(i+1)$ interrupts caused by directory overflow events on a memory block, after the first such event. Figure 7-12 shows the effects of this feature. The curves marked with unfilled symbols show that when trap latency is low, the pointer reset feature improves performance only slightly: the 6% improvement for $Dir_nH_2S_{NB}$ running EVOLVE is the largest in any of the graphs. The curves with filled symbols show that resetting the directory is more important when trap latency is high. When software handling takes a long time, it is important to avoid as many interrupts as possible.

Hardware transmit Figure 7-13 shows the performance benefit of Alewife's feature that allows the hardware to transmit read-only copies of data in parallel with the associated software handling. ($Dir_nH_0S_{NB,ACK}$ can not use this feature.) Since this feature is a prerequisite for pointer reset, the curves marked with circles show the model's prediction for systems without either the hardware transmit or the pointer reset mechanism.

Software-extended architectures with longer trap latencies benefit from transmitting data in hardware, because the feature eliminates the need for many interrupts. The curves marked with filled symbols show the performance improvement when the trap latency is 1000 cycles. With the exception of AQ, all of the benchmarks realize improved performance, sometimes even when the number of hardware pointers is large.

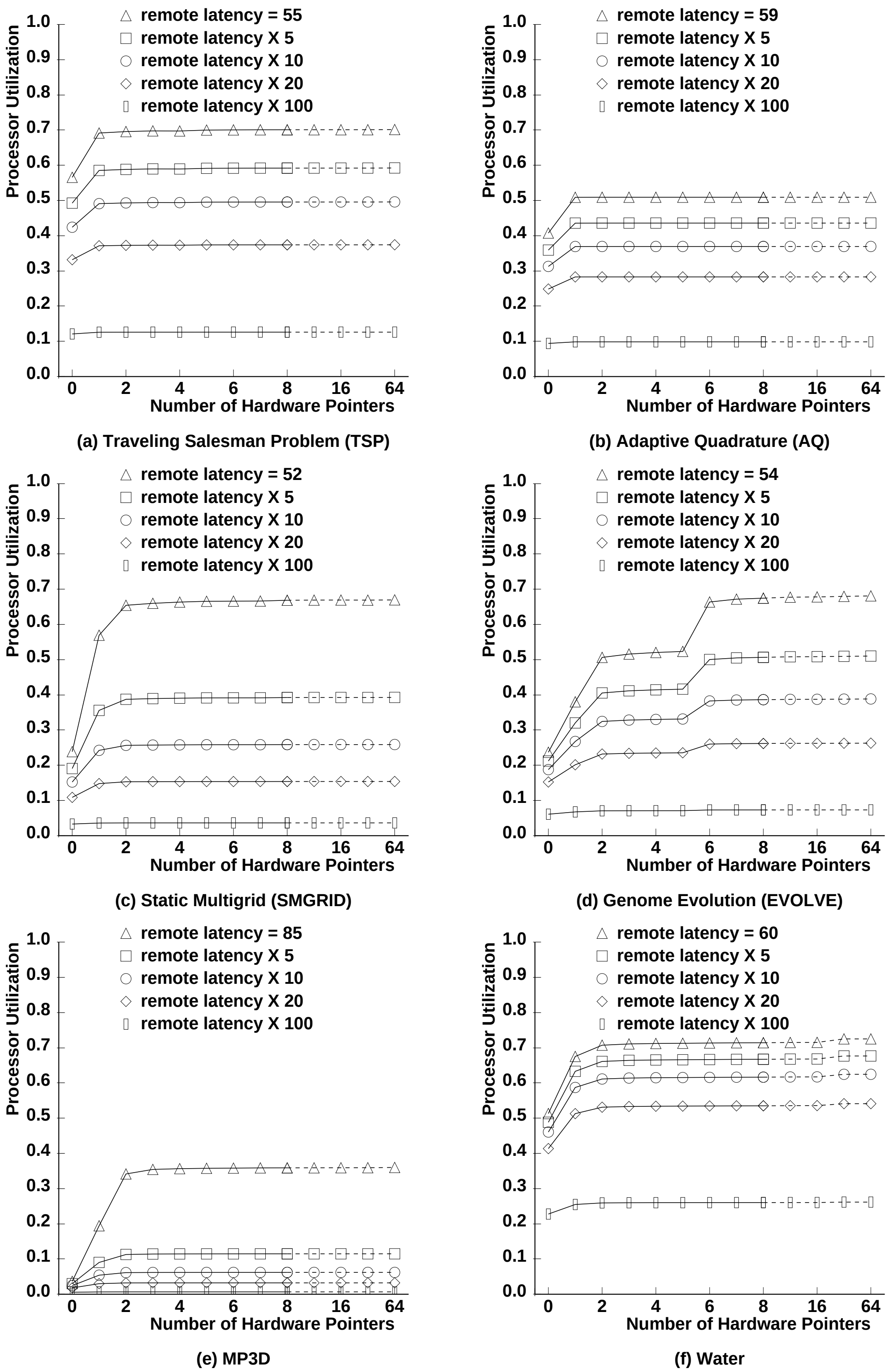


Figure 7-11: Modeled performance of software-extended schemes with increasing network latency. The base remote latencies are experimental values from $Dir_nH_{NB}S_{-}$ simulations.

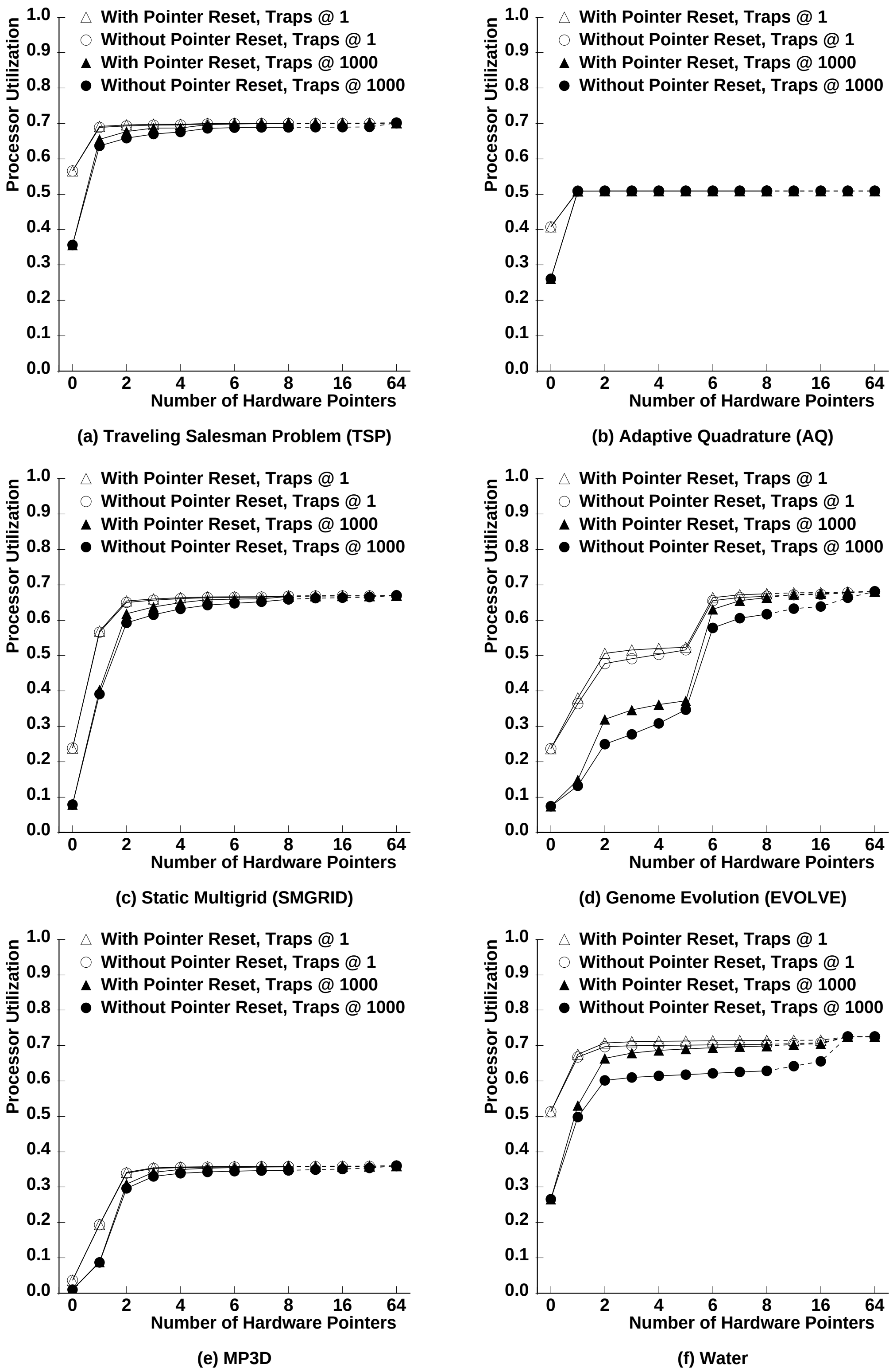


Figure 7-12: Modeled performance of software-extended schemes with and without the software optimization that resets the hardware directory pointers after storing them in local memory.

The contribution of pointer reset is a little less than half of the total performance improvement derived from the two mechanisms combined. This observation is true because the hardware reset feature removes all of the directory-overflow interrupts from the average memory access time (T_a), while the pointer reset option eliminates the need for only $i/(i+1)$ of the interrupts on responding nodes.

Local bit The local bit feature in Alewife provides an extra one-bit pointer in every directory entry that is used exclusively for intranode accesses. This feature is useful primarily because it ensures that directory overflows will never occur on intranode memory accesses, thereby simplifying the design of the CMMU and the memory-system software. In addition, this bit corresponds to the amount of mechanism required for $Dir_nH_0S_{NB,ACK}$ to optimize purely intranode memory accesses.

Figure 7-14 shows that the feature is very important for the software-only directory. (The $Dir_nH_0S_{NB,ACK}$ circles in the figure correspond to the filled triangles in Figure 7-10.) The local bit also has a slight performance benefit for the other protocols, because it provides one additional pointer when local nodes are members of important worker sets. Thus, the local-bit shifts the SMGRID and EVOLVE performance curves to the left by one hardware pointer.

7.5 Node Architecture

The previous section used the worker-set model to predict the performance of software-extended systems as a function of features of the memory system. This section extends the scope of the investigation by exploring the node architecture in parallel systems. First, the model predicts the change in performance when using superscalar processors, as opposed to Alewife's Sparcle (single-issue RISC) processors. Then, the model shows the impact of building a machine with several processors per node, rather than a single processor per node.

7.5.1 Superscalar Processors

Throughout the preceding analysis, all time intervals were measured in units of processor clock cycles. Since Alewife has a single-issue RISC processor, a clock cycle is equal to the time required to retire each non-memory, integer instruction. Superscalar processors with more functional units and higher execution bandwidths suffer from intrinsic hardware latency in a memory system more than Alewife's conventional processors.

The worker-set model can estimate the interaction between software-extended shared memory and superscalar processors by changing time units. Let the processor bandwidth (b) be equal to the peak number of instructions that can be retired per cycle. Multiplying the model's time interval by the inverse of this bandwidth ($1/b$) approximates processors that can retire b instructions per cycle. This transformation is equivalent to multiplying the hardware memory access latencies (T_l and $T_{r,hw}$) by the processor bandwidth (b). No other latencies need to be modified: assuming a well-engineered memory hierarchy, the effective cache hit latency (T_h) would be equal to the new time interval; and assuming

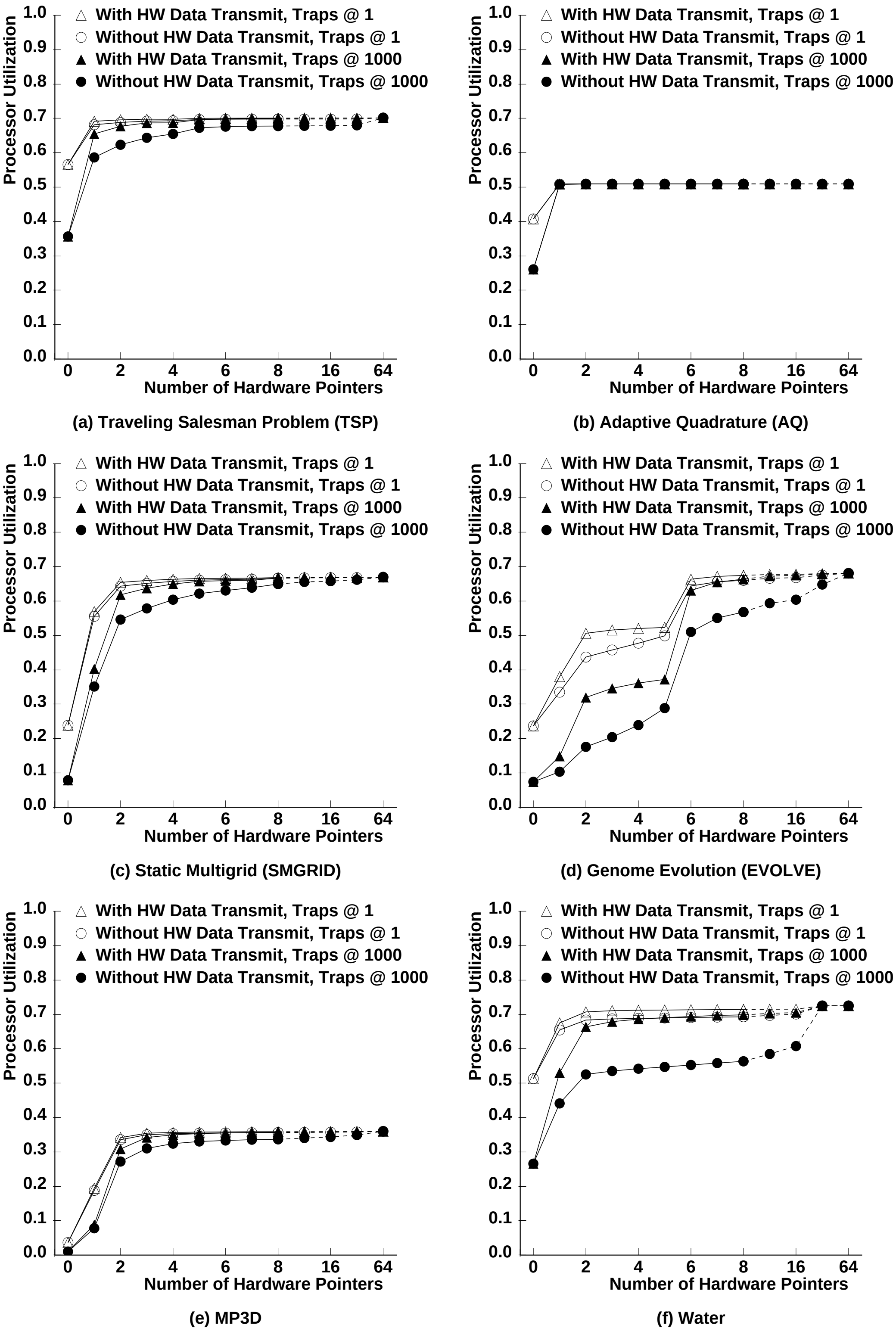


Figure 7-13: Modeled performance of software-extended schemes with and without the optimization that allows hardware to transmit read-only copies of data before invoking software handlers.

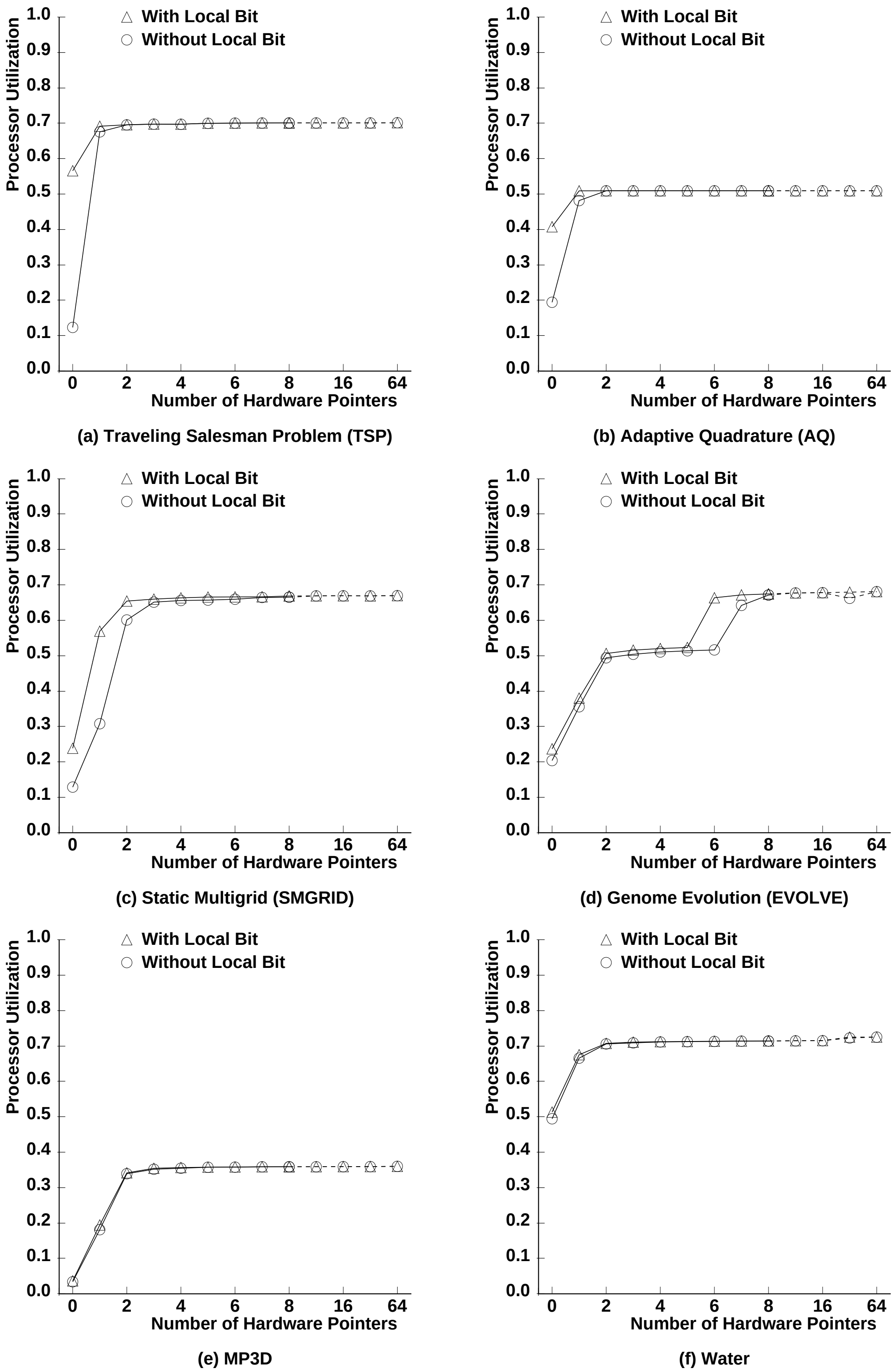


Figure 7-14: Modeled performance of software-extended schemes with and without the hardware optimization that employs a special one-bit directory pointer for the local node.

appropriate compilation technology, protocol software handling (represented by the t_x parameters) would speed up with the processors.

Figure 7-15 shows the predicted behavior of the system with superscalar processors running the six benchmark applications. As in the previous sections, the horizontal axis of the graph shows the number of hardware pointers in the software-extended coherence scheme. The vertical axis shows the instructions per cycle achieved for each node architecture, calculated as utilization ($\mathcal{U}_{nfb}$) multiplied by the processor bandwidth (b). This calculation makes the optimistic assumption that the benchmarks exhibit enough instruction-level parallelism to fill the functional units of any superscalar processor, up to 8-way instruction issue. Conversely, the model makes the pessimistic assumption that the memory system is the performance bottleneck. Each line on the graph shows the performance for a different processor bandwidth value.

The results show that all of the applications (except MP3D) could benefit from superscalar processors. However, just as single-issue processors will not achieve 100% utilization, superscalar processors will not achieve their peak instruction issue rate. Due to the impact of memory latency and bandwidth limitations, doubling the processor bandwidth never doubles actual performance. Certainly, increasing intranode memory bandwidth along with processor bandwidth would help ameliorate some of this effect.

In addition, hardware support for shared memory (in the form of limited directories) proves to be extremely important when a system has superscalar processors. Even though multiple-issue processors might be able to execute software-extension code swiftly, the utilization penalty of stealing processor cycles increases with processor bandwidth. While the relative performance difference between $Dir_nH_0S_{NB,ACK}$ and $Dir_nH_{NB}S_-$ decreases as processor bandwidth increases, the absolute difference in performance increases with processor bandwidth.

7.5.2 Multiple Processors per Node

Another parameter in multiprocessor node design involves the number of processors per node. The nodes of a number of existing parallel architectures consist of several processors and memory modules connected by a bus [55, 79, 23]. These clustered nodes communicate over a higher-level interconnection network. From the point of view of the memory system, implementing multiple processors per node has two beneficial effects and one detrimental effect. First, the data locality changes: since the amount of memory in each node increases with the number of processors, a larger percentage of an application's data resides in each node, increasing the ratio of local to remote accesses. Second, the worker-set profiles collapse: each "worker" corresponds to several processors, so the worker-set sizes become smaller (assuming proper task and data placement). This effect reduces the demands for coherence directory pointers. The locality and worker-set effects decrease the demands on the memory system and interconnection network, thereby improving performance.

Third, the local and remote memory access times increase. No matter how well a bus is engineered, physical limitations make a multidrop bus slower than a one-to-one link between a processor and a local memory module. In addition, contention for shared memory resources decreases the memory bandwidth available to each processor. The

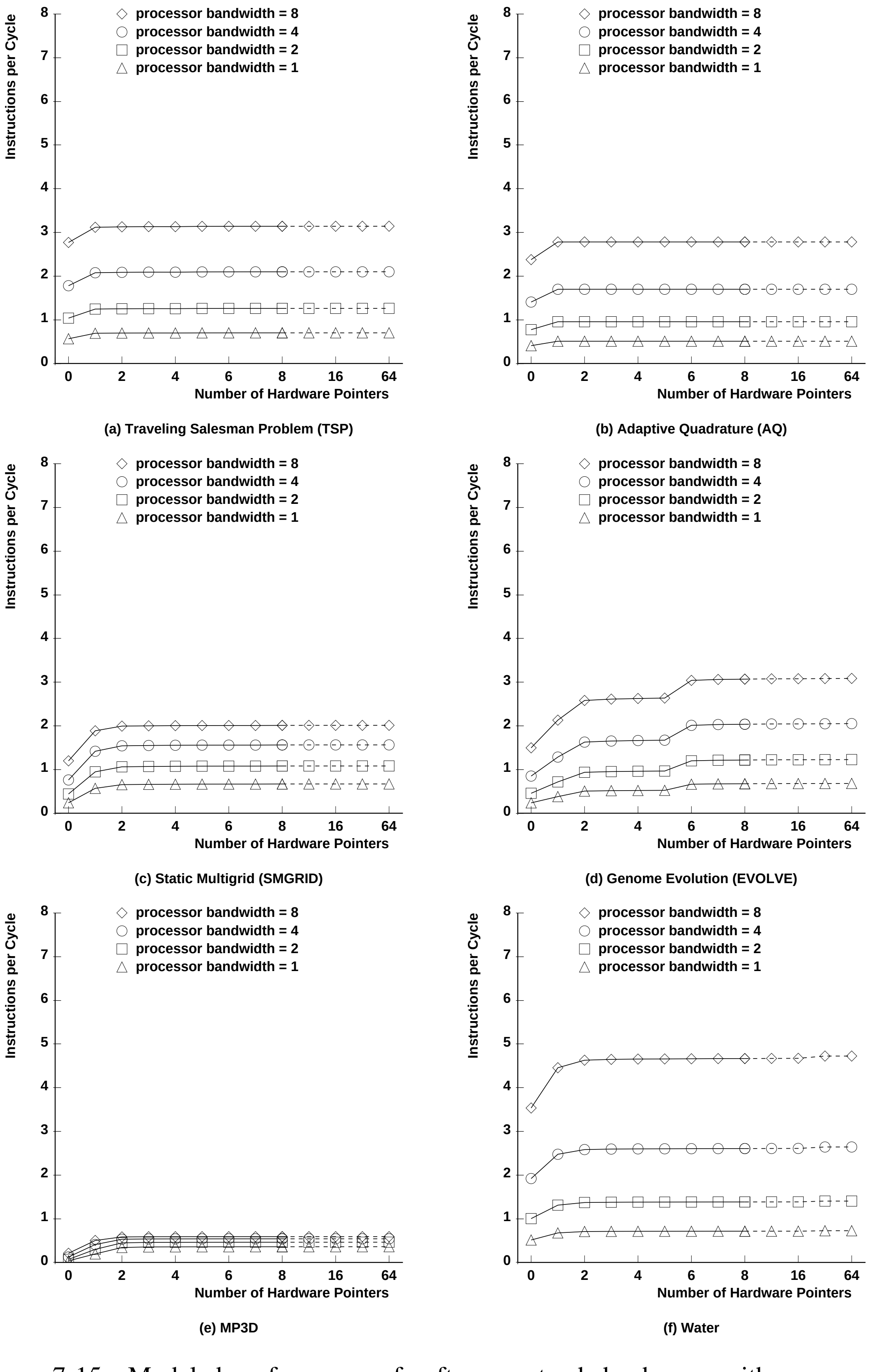


Figure 7-15: Modeled performance of software-extended schemes with superscalar processors. Processor bandwidth indicates the peak number of instructions per cycle.

results below show that this effect can cancel any improvements in data locality.

In order to separate the effects of data locality and access time, first consider a hypothetical system that clusters processors without any additional intranode latency or contention. This type of architecture can be modeled by accounting for the changes in locality (modifying l and r) and in worker-set profiles (scaling the R_k and W_k sequences).

Figure 7-16 shows the predicted performance for each of the six benchmarks, running on this hypothetical system. The horizontal axis shows the number of processors per node, and the vertical axis shows processor utilization. The general trend in each graph is for the performance of all of the schemes (except for $Dir_nH_0S_{NB,ACK}$) to approach the performance of $Dir_nH_{NB}S_{-}$ as the number of processors per node increases. This trend is caused by the collapsing worker-sets. The (very) slight increase in $Dir_nH_{NB}S_{-}$ performance is due to the increase in the local to remote access ratio.

With at least two hardware pointers ($Dir_nH_2S_{NB}$), the performance benefit of multiple processors is small, except when running the EVOLVE application. Collapsing the large worker-sets in EVOLVE dramatically improves the performance of all of the software-extended schemes. With only two processors per node, $Dir_nH_5S_{NB}$ performs as well as $Dir_nH_{NB}S_{-}$. On the other hand, $Dir_nH_0S_{NB,ACK}$ improves only slightly: software-only directories hamper performance even with multiple processors per node.

While these results make clustered nodes look moderately promising, they do not take into account the effect of increased memory access time. The worker-set model can help understand this effect by quantifying how much latency can increase before it cancels the benefits of multiple processors per node. A good metric for examining the trade-off is the *latency equivalent*, the increase in latency that causes a clustered-node architecture to achieve exactly the same performance as an architecture with single-processor nodes.

For example, Figure 7-16(e) shows that MP3D, running on a system with $Dir_nH_5S_{NB}$, will exhibit a processor utilization of 0.36 with one processor per node and 0.38 with eight processors per node. However, solving the worker set model for the latency equivalent shows that if the local memory access latency increases by 61% (latency equivalent = 1.61), the system with eight processors per node achieves a processor utilization of 0.36, the same as the architecture with single-processor nodes. Building a node with eight processors rather than with one processor would surely increase local memory latency by more than 61%, so a system with single-processor nodes would run MP3D faster than a system with clustered nodes.

Figure 7-17 shows the latency equivalents for the benchmarks running on the gamut of node architectures. The vertical axis of these graphs plots the latency equivalent for each configuration on a logarithmic scale. Figure 7-17(e) shows the latency equivalent described above: the triangle at $(8, 1.61)$ indicates that for eight processors per node, a factor of 1.61 increase in latency negates the benefits of multiple processors per node.

For $Dir_nH_2S_{NB}$, $Dir_nH_5S_{NB}$, and $Dir_nH_{NB}S_{-}$, the latency equivalent remains below a factor of 2.5 for all of the benchmarks except EVOLVE. This evidence indicates that clustered nodes would have to be engineered extremely well to beat the performance of single-processor nodes. Since single-processor nodes are easier to build and permit more modular packaging than clustered nodes, parallel architectures with $Dir_nH_2S_{NB}$ (or a higher-performance protocol) should use single-processor nodes.

This conclusion must be weakened when considering applications like EVOLVE.

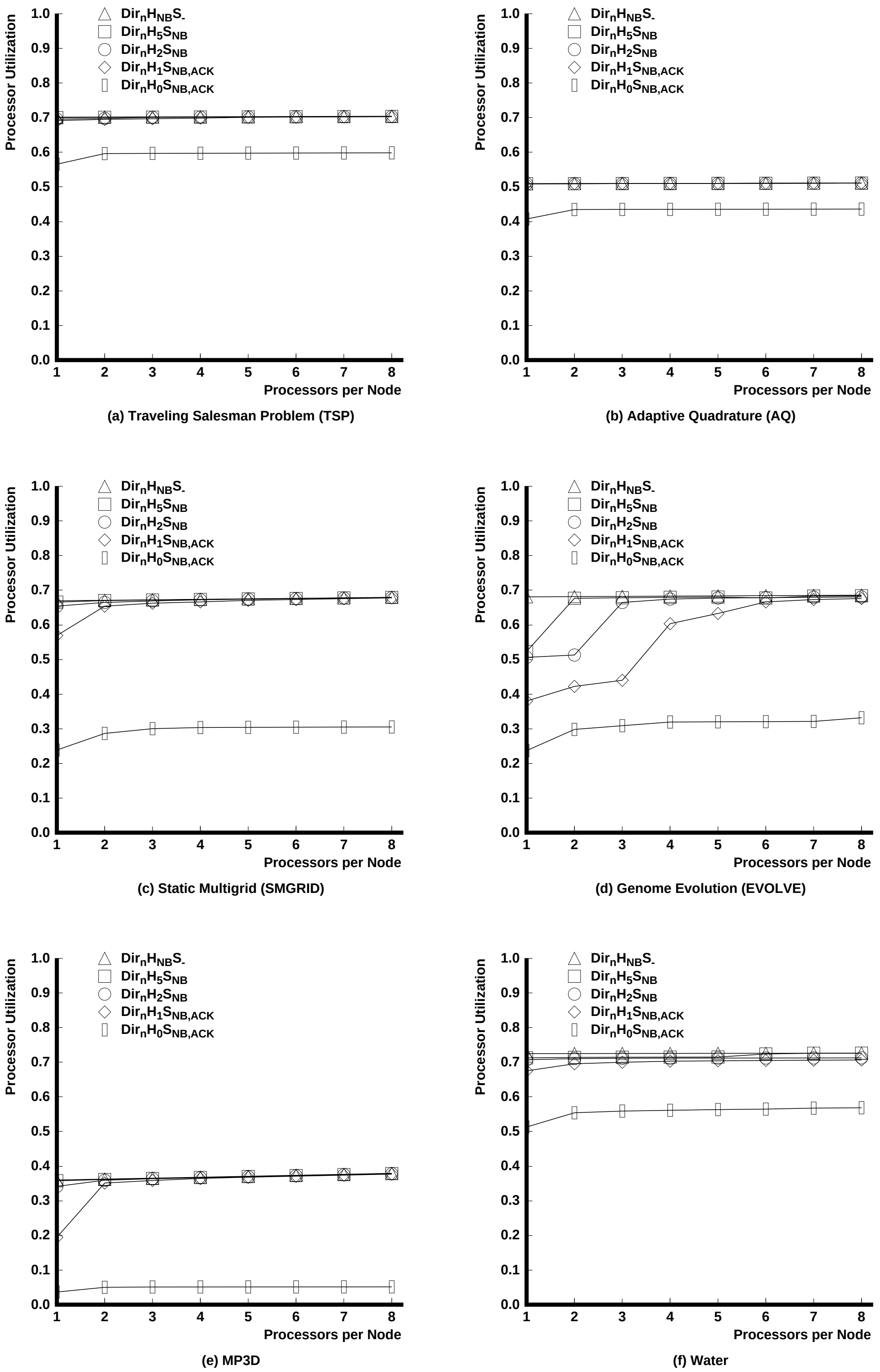


Figure 7-16: Modeled performance of software-extended schemes with multiple processors per node, assuming constant intranode latency and contention.

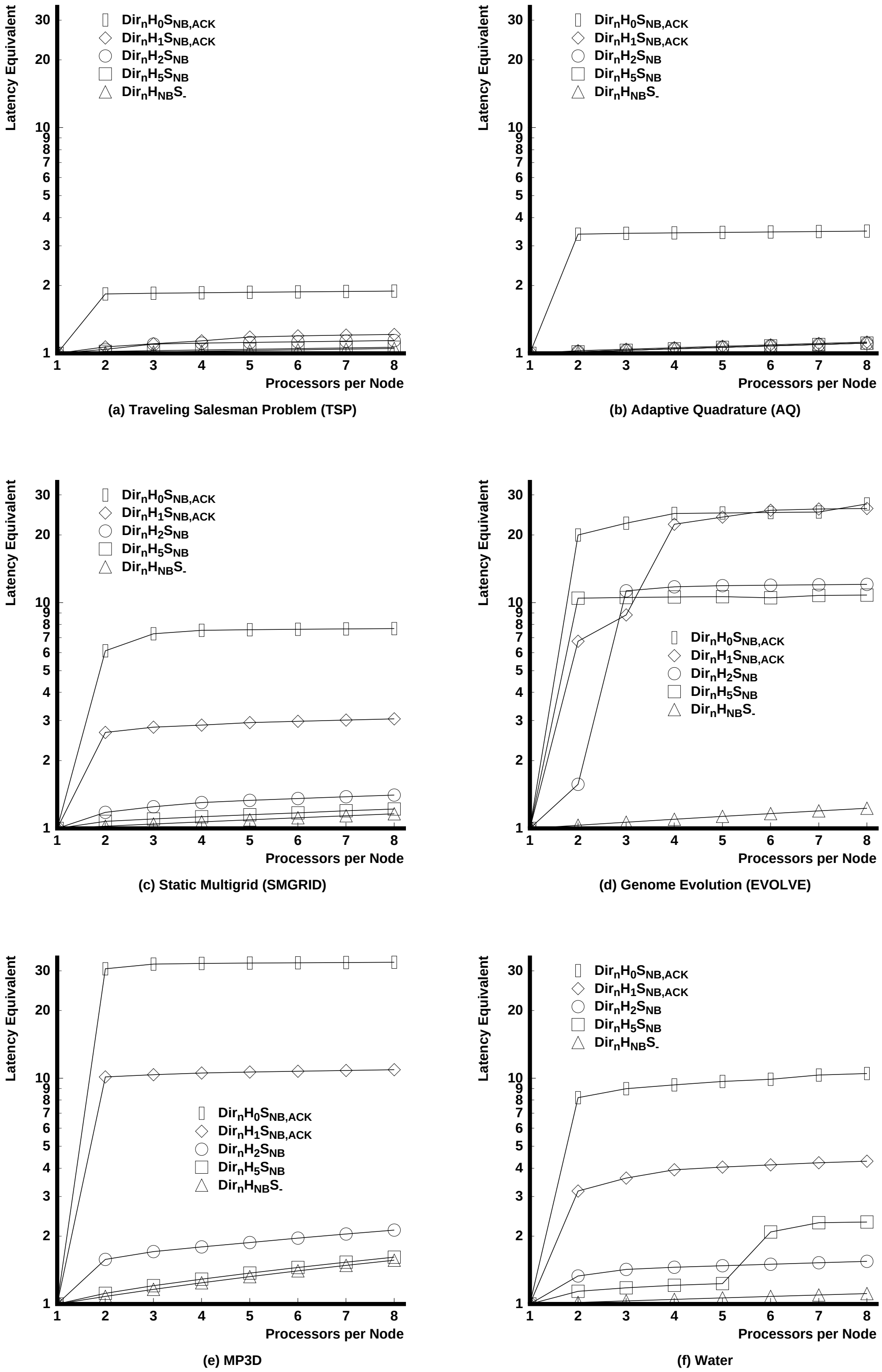


Figure 7-17: Latency equivalent, the increase in latency that causes a system with multiple processors per node to perform the same as a system with single-processor nodes.

Figure 7-17(d) shows that the performance improvements for EVOLVE promised by Figure 7-16(d) can be realized with ordinary engineering techniques. The high latency equivalents indicate that it would be easy to build a clustered-node system that outperforms a single-node system. If EVOLVE is a typical application, then architectures could improve performance by implementing multiple processors per node, thereby collapsing worker sets.

$Dir_nH_0S_{NB,ACK}$ and $Dir_nH_1S_{NB,ACK}$ could also benefit from clustered nodes. However, an architect would need strong motivation to implement one of these inexpensive protocols, yet increase the system cost by implementing multiple processors per node. One such design might use $Dir_nH_0S_{NB,ACK}$ to connect preexisting bus-based multiprocessors into a larger system.

7.6 The Workload Space

This section drives the model with completely synthetic inputs in order to investigate the relationship between an application's worker-set behavior and the performance of software-extended systems. This study is intended to give some insight into the behavior of shared memory, rather than making any specific predictions.

The synthetic inputs are primarily a composite of the six benchmark applications. The hit ratios and access latencies in the input set are held constant at values that are similar to those found in the benchmarks. The worker-set inputs simulate a bimodal distribution of data objects: one type of data object has a small worker-set size, and the other type of data object is shared by every node in the system. Since the workload is completely synthetic, it is easy to model 256 processors, rather than 64.

Three variables in the worker-set inputs are changed. First, the size of the smaller worker sets ranges from one node to P. The default worker-set size is 5. Second, the percentage of data objects with the small worker-set size varies from 0% to 100%. The default percentage of small worker sets is 50%. Third, the percentage of data objects that are modified (as opposed to read-only) varies from 0% to 100%. The default percentage of written objects is 50%.

The data access model requires a bit of explanation. If a data object is written and its worker-set size is S, then the worker-set histograms contain exactly S reads and one write for that object. Consider two classes of data objects, one with worker-set size S_l and the other with worker-set size S_b, where $S_l < S_b$. If both of these classes have the same percentage of modified objects, then the class with worker-set size S_l will have a higher *write:read* ratio than the class with worker-set size S_b.

This effect is visible in Figure 7-18. The figure projects a three-dimensional plot with the number of hardware pointers and the size of the smaller worker set on the horizontal axes, and processor utilization on the vertical axis. The axis with the number of pointers has a similar scale to the two-dimensional plots in the previous sections. Low numbers of hardware pointers are plotted on a linear scale; high numbers are plotted on a logarithmic scale. The line on the upper right-hand side of this graph (and of the other two graphs in this section) shows the $Dir_nH_{NB}S_-$ performance for the synthetic workload. All of the points on this line correspond to a processor utilization of 0.58.

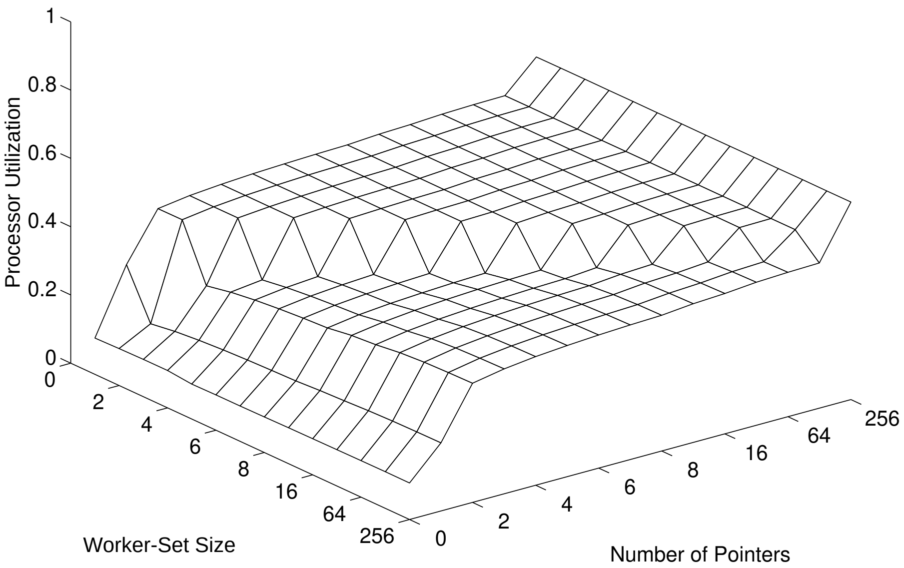


Figure 7-18: Dependence of performance on worker-set size and hardware directory size.

There are two plateaus in the graph: the lower plateau in the foreground of the plot corresponds to the performance of software-extended systems that implement at least two directory pointers in hardware. The cliff in front of this plateau indicates the lower performance of $Dir_nH_0S_{NB,ACK}$ and $Dir_nH_1S_{NB,ACK}$. The second plateau corresponds to the protocols that implement enough hardware pointers to contain the small worker sets. A more complicated worker-set profile would create one plateau for every different worker-set size.

The cliff at the far end of the higher plateau indicates the jump in performance between the software-extended systems and $Dir_nH_{NB}S_{-}$. This cliff is caused by the data objects that are shared by every processor in the system. The cliff between the two plateaus indicates the increase in performance achieved by processing all of the small worker sets in hardware. The height of this cliff depends on the *write:read* ratio. Thus, the cliff is higher on the left side of the graph where worker sets are small than on the right side of the graph.

Figure 7-19 shows the effect of the *write:read* ratio more effectively, because it holds all worker-set variables constant, except the percentage of modified data objects (measured by the left axis). The cliff between the two plateaus is now parallel to the y-axis, because the worker set is fixed at 5. For the same reason, the 50% curve in this figure is exactly the same as the curve in Figure 7-18 with a worker-set size equal to 5.

Both $Dir_nH_0S_{NB,ACK}$ and $Dir_nH_1S_{NB,ACK}$ improve with lower percentages of modified data, because these protocols handle the invalidation process entirely in software. The $Dir_nH_1S_{NB,ACK}$ curve is particularly dramatic: at 0% data objects modified, the protocol performs almost as well as $Dir_nH_2S_{NB}$. On this side of the plot, the only difference

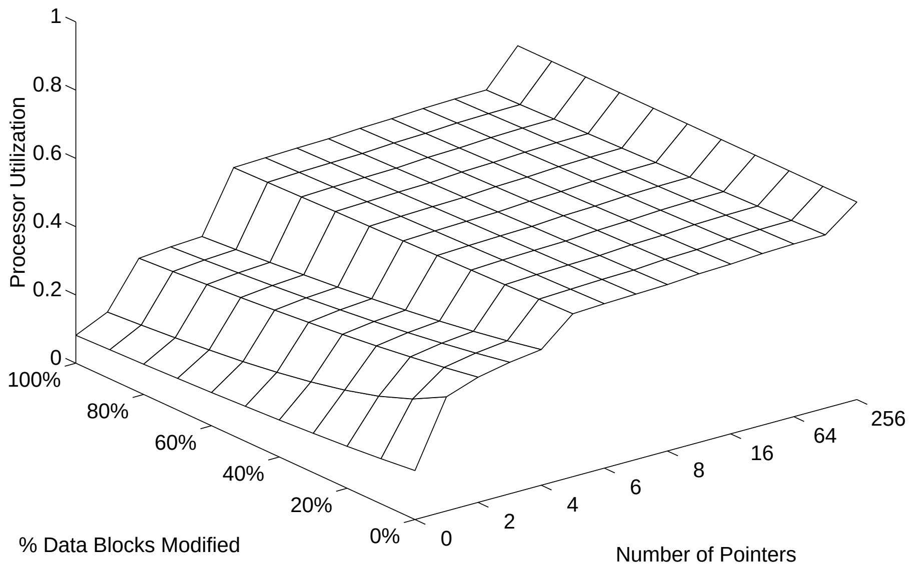


Figure 7-19: Dependence of performance on percentage of modified data and hardware directory size.

between the two schemes is the extra hardware directory pointer. At 100% data objects modified, $Dir_nH_1S_{NB,ACK}$ performs approximately the same as $Dir_nH_0S_{NB,ACK}$, because processing invalidations becomes the protocols' dominant activity.

Figure 7-20 shows the effect of the balance of the two worker-set sizes. The left axis now shows the percent of the data objects that are shared by all of the nodes in the system, as opposed to the data objects shared by only 5 nodes. Again, the 50% curve in this graph is the same as the curve in Figure 7-18 with worker-set size equal to 5, and the same as the 50% curve in Figure 7-19.

The figure shows that when few data objects are widely-shared, there is a steep cliff between the software-extended systems with too few pointers and the ones with enough pointers to hold the smaller worker set. The cliff is steep, because the second plateau reaches the performance of $Dir_nH_{NB}S_{-}$.

The height of the cliff seems to belie the claim that software-extended systems offer a graceful degradation in performance. This claim remains true for most real workloads, which have a more complicated range of worker-set sizes compared to the simple bimodal distribution used to generate this plot. Even with a bimodal distribution, the performance of the $Dir_nH_XS_{NB}$ protocols should still be higher than the corresponding limited directories ($Dir_iH_{NB}S_{-}$), because the software-extended directories do not suffer from pointer thrashing.

When most data objects are widely-shared, the cliff disappears and the two plateaus merge together at a relatively low processor utilization. Given this type of worker-set behavior, none of the software-extended protocols perform well compared to $Dir_nH_{NB}S_{-}$. The next chapter describes three techniques that use the flexibility of the software-

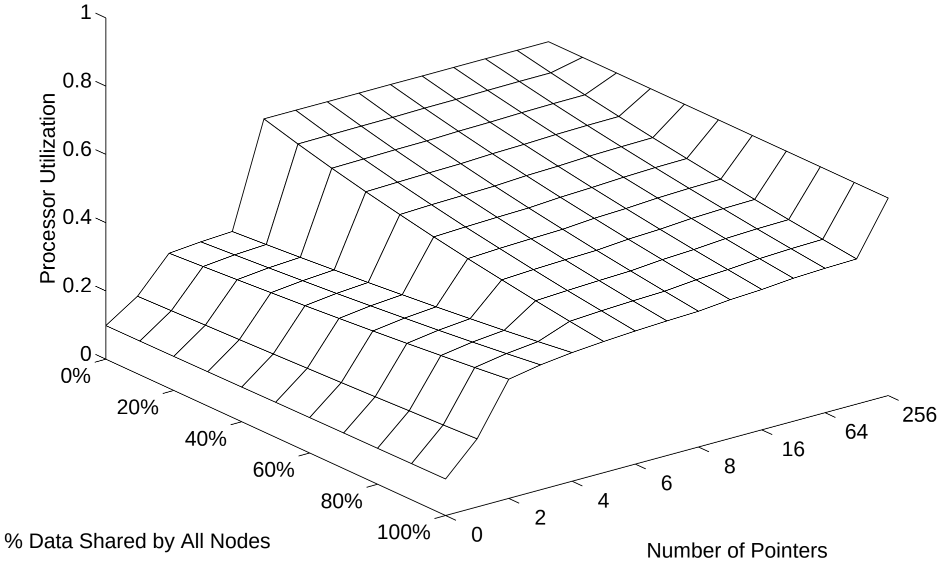


Figure 7-20: Dependence of performance on mix of worker-set sizes and hardware directory size.

extended design to help eliminate the problems caused by widely-shared data.

7.7 Conclusions and an Open Question

The model answers a number of questions about the sensitivity of software-extended systems to architectural mechanisms and to application workloads. If desired, this analytical tool may be expanded to incorporate a wider range of software-extended systems. The conclusions may be divided roughly into those that are valid for software-extended systems that incorporate a limited hardware directory, and those that are valid for the software-only directory architecture.

Given an appropriate number of hardware pointers, software-extended systems are not sensitive to an order of magnitude difference in trap latency, but do require a better implementation of interrupts than is provided by most popular operating systems. Adding hardware directories to a system buys insensitivity to memory-system code efficiency, thereby encouraging flexibility. Adding an inexpensive mechanism that streamlines intranode accesses is useful, but not essential. Hardware directories and dedicated memory-system processors are mutually exclusive: having one obviates the need for the other. Finally, hardware directories are useless in systems with extremely slow networks.

Implementing software-only directories is more problematic, even when the system dedicates processors for the exclusive use of the memory system. The performance of such systems depends strongly on the efficiency of the memory-system software and on mechanisms that streamline intranode accesses. This sensitivity stems from the fact that these systems expend hardware on all memory accesses, instead of accelerating common case accesses. Contrary to the conventional wisdom, the sensitivity of software-only schemes may result in fewer abstractions, and therefore less flexibility in actual implementations. Without the development of a new compilation strategy, protocols for software-only directories will have to be painstakingly hand-coded.

One important metric neglected by the analytical model is the cost of software-extended systems, in terms of both component price and design time: is the cost of $Dir_n H_X S_{NB}$ close to a standard processor/MMU combination, or do software-extended systems cost about the same as a dual-processor system? There are arguments for both comparisons. Certainly, a typical virtual memory system requires a unit with associative memory, extra memory for page tables, and a processor interface. On the other hand, the first iteration of the A-1000 CMMU required more design time and gates than did Sparcle, the A-1000 processor. An evaluation of the costs involved in building each type of shared-memory system will play a critical role in determining the implementation (and even the existence) of large-scale, shared memory systems.

Chapter 8

Smart Memory Systems

The philosophy of shared memory is to make the application programmer's job as easy as possible. Given this goal, the memory systems described in the previous chapters achieve only partial success. Although the basic $Dir_nH_XS_{Y,A}$ protocols do provide a convenient programming model at reasonable cost, they do nothing to help the programmer make the most efficient use of a multiprocessor.

Implementing part of a memory system in software presents an opportunity for the system designer to do much more for application programmers. Incorporating intelligence into the software can allow the system to improve application performance with and without the help of the programmer. This chapter presents three methods for building this kind of smart memory system: adaptive broadcast protocols; LimitLESS profiling; and a profile-detect-optimize scheme. The following research is not intended to be a definitive work on the topic of software-enhanced memory systems; rather, it attempts to indicate directions and to provide motivation for future investigation.

8.1 Challenges

Two interwoven challenges emerge when trying to build a smart memory system: collection of performance data and information feedback. Data collection refers to the ability of a system to measure its own performance. If a measurement technique is to become part of the permanent, everyday operation of a system, it must have low overhead in terms of both computation and memory usage. That is, it must be simple. Expensive and complicated measurements also have a place in the system, but they tend to be used during the development phase of applications rather than as part of production versions. In any case, the more overhead required by a measurement technique, the less it will be used in practice.

Information feedback involves transforming measurements into performance improvement. By far, the most convenient time for providing feedback (for both the system designer and the programmer) is on-line, during the actual operation of the system. In the best scenario, a system can adapt to various memory access patterns while a program is running. The better the performance improvement, the more complicated the measurement, feedback, and optimization can be. However in most cases, memory systems with latencies on the order of a microsecond require this kind of dynamic feedback mechanism to be extraordinarily simple. Section 8.2 describes one technique for allowing a software-extended memory system to adapt dynamically to program behavior.

Due to the short lengths of time involved, more complicated feedback must take place during the development of an application. The general methodology requires the programmer to put the system into a special state before running a program. While the system runs the program, it *profiles* the execution, measuring and recording performance data. At some point, the programmer requests the system to produce a report that summarizes the gathered information. Typically, the production versions of applications do not make use of a system's profiling capability.

Unfortunately, this static feedback technique requires mechanisms to convert measurements of the dynamic operation of the system into information about the program itself. This conversion process requires cooperation between the compiler and the smart memory system. Such cooperation is made difficult by dynamic data allocation and by the aliasing problem.

Most programs use some form of dynamic data allocation, which prevents a compiler (even in combination with a linker and loader) from knowing the actual memory addresses of data objects. Thus, while a memory system can easily collect information that relates memory addresses to data object behavior, it must feed the compiler information that maps instructions (or program counters) to behavior. Section 8.3 describes a scheme for Alewife that produces such a mapping. The compiler can then correlate instructions with positions in a program's source code, thereby completing the feedback loop.

When presented in an appropriate human-readable format, this type of information is often helpful to the programmer. Information about the how a program uses shared memory often indicates ways for a programmer to optimize the performance. Even so, it is much more convenient for the programmer when the compiler and memory system can combine to optimize performance automatically.

When a solution to the challenge of dynamic data allocation exists, the aliasing problem still makes automatic feedback difficult: given a variable, it is usually not possible for a compiler to track all of the accesses to the variable throughout a program. Conversely, given an instruction that accesses memory, it is often impossible to determine all of the variables that correspond to that access. Any completely automatic optimization scheme must function in the absence of complete information about the mapping between variables and program counters. Section 8.4 describes a system that addresses this challenge by asking the programmer for just enough information to solve the aliasing problem. After profiling and feedback steps, the system automatically performs *safe* optimizations on the program. Such optimizations generally improve performance, and never cause a program to produce the wrong results.

8.2 Adaptive Broadcast Protocols

The premise underlying the LimitLESS protocols states that most, but not all, data objects are accessed by small worker sets. Consequently, the family of software-extended memory systems provides graceful degradation in performance as the amount of widely-shared data increases.

But how graceful? When transforming a memory block from Read-Only to Read-Write state, both the $Dir_nH_{NB}S_-$ and its naïve $Dir_nH_XS_{NB}$ extensions transmit one invalidation and receive one acknowledgment message for every member of the worker set. Thus, the time and communication bandwidth required for this transition grows linearly with worker-set size. Although the constant factors are relatively low, especially for the unscalable $Dir_nH_{NB}S_-$ protocol, the invalidation process becomes a bottleneck in systems with a large number of processors.

Software-extended systems provide an efficient alternative to this inherently sequential process. If the size of a memory block's worker set exceeds a predetermined threshold, then the software can choose to broadcast invalidation messages to every node in the system (or to some appropriate subset of nodes). Such a policy caps the maximum latency and bandwidth for the invalidation process. Furthermore, this adaptation to widely-shared data is simple enough that it adds very little to memory access latency.

This strategy is very similar to the $Dir_iH_BS_-$ protocols, except that the software broadcast has far more available options. For example, the extension software can choose a broadcast threshold that is significantly higher than the number of pointers implemented in hardware. The software can also choose a broadcast algorithm that is optimized for the exact size and topology of the a given system. A flexible choice of threshold and algorithm should prove especially useful in multiprocessors that may be logically partitioned into smaller virtual machines.

A broadcast protocol has an additional benefit to the $Dir_nH_XS_{NB}$ protocols. If the software decides to use a broadcast invalidation strategy for a memory block, then it can conserve significant time and memory: subsequent interrupts caused by overflowing the block's directory entry may safely be ignored. No additional pointers need to be recorded, because the next write will cause the data to be removed from every cache in the system. In fact, if the directory hardware allows the software to disable the directory overflow interrupt for a particular memory block (but not the associated trap-on-write), the extension software can eliminate all of the overhead from subsequent directory overflows.

Figure 8-1 shows the difference in performance between various protocol options discussed above. The graph shows the performance of EVOLVE, with the broadcast threshold on the ordinate and speedup on the abscissa. Since the $Dir_nH_{NB}S_-$ protocol has no broadcast threshold, its performance is shown as a horizontal line at the top of the plot. Similarly, the line for the $Dir_nH_5S_{NB}$ protocol appears at the bottom. Of the six benchmark applications, EVOLVE provides the best case study due to the difference in performance between $Dir_nH_{NB}S_-$ and $Dir_nH_5S_{NB}$ [1].

Due to its low constant factor in transmitting messages, $Dir_5H_BS_-$ (shown as a horizontal, dotted line) achieves higher performance than any of the software-extended schemes. This protocol performs worse than $Dir_nH_{NB}S_-$, because it uses a sequential broadcast and transmits many more invalidations than necessary. $Dir_5H_5S_B$, which corresponds to the horizontal line just above $Dir_nH_5S_{NB}$, uses the same type of naïve

[1]The values are slightly different than in Figure 6-2 due to several months of changes in the Alewife kernel and compiler.

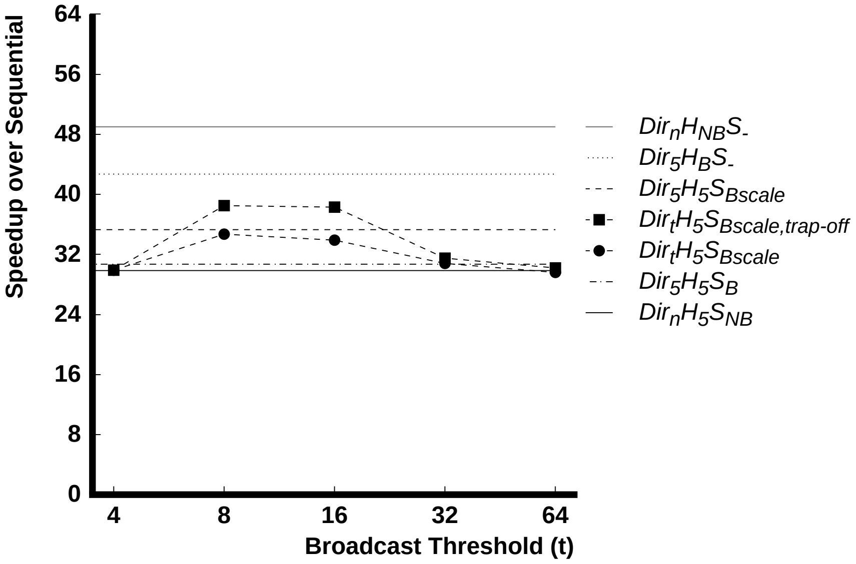


Figure 8-1: Performance of EVOLVE, running on 64 simulated nodes with various broadcast protocols.

broadcast, but implements it in software.

The dashed lines in the figure show the performance of the software-extended system with a scalable broadcast scheme. This scheme uses active messages and a hierarchical algorithm to distribute invalidations to every cache in the system. The hardware acknowledgment counter mechanism then tallies the corresponding acknowledgments sequentially. When running on 64 NWO nodes, this strategy provides better performance than several other software broadcast schemes that have been implemented. The horizontal, dashed line shows the performance for a protocol called $Dir_5H_5S_{Bscale}$, a protocol that performs a broadcast for all memory blocks that overflow their hardware directory entry. This protocol is a software-extended version of $Dir_5H_BS_-$, and allows EVOLVE to perform better than with $Dir_nH_5S_{NB}$. In order to investigate a realistic implementation of $Dir_5H_5S_{Bscale}$, the simulator is configured so that the processor never receives a directory overflow trap. This option is not available on the A-1000.

The $Dir_tH_5S_{Bscale}$ protocol is similar to $Dir_5H_5S_{Bscale}$, except that $Dir_tH_5S_{Bscale}$ uses a broadcast to transmit invalidations only for memory blocks with worker-set sizes greater than a certain threshold (t). Each circle on the plot indicates the performance for one experimental threshold, with all other experimental factors constant. $Dir_tH_5S_{Bscale}$ achieves its best performance with a threshold set to 8, which is slightly larger than one of the dominant worker-set sizes in EVOLVE shown in Figure 7-3(d). At this threshold, the protocol — which can run on the A-1000 without modification — achieves performance close to the unsupported $Dir_5H_5S_{Bscale}$ protocol.

$Dir_tH_5S_{Bscale,trap\text{-}off}$ is exactly the same as $Dir_tH_5S_{Bscale}$, except that this protocol takes advantage of a hardware directory feature that allows the software to disable the directory overflow interrupt. Like $Dir_5H_5S_{Bscale}$, $Dir_tH_5S_{Bscale,trap\text{-}off}$ can avoid excessive interrupts; like $Dir_tH_5S_{Bscale}$, $Dir_tH_5S_{Bscale,trap\text{-}off}$ only broadcasts to worker sets over a threshold size. The performance of this hybrid protocol also peaks at a threshold of eight, and exceeds the performance of both of the other scalable broadcasts. Again, the A-1000 does not implement the feature that allows the software to disable directory overflow interrupts; however, changing exactly one line of source code in the CMMU specification would enable this feature. Given the potential gain in performance, this change seems worthwhile.

It is important to interpret Figure 8-1 in light of the 64-node system size and the EVOLVE benchmark. For example, while $Dir_5H_BS_-$ performs admirably well on a 64 node machine, it outperforms $Dir_tH_5S_{Bscale,trap\text{-}off}$ by only a small factor. On slightly larger machines and on bandwidth-limited applications, the scalable broadcast protocols should be even more beneficial. Given the asymptotic bandwidth and latency requirements of the sequential and scalable broadcasts, it is certainly possible that $Dir_tH_5S_{Bscale,trap\text{-}off}$ and $Dir_tH_5S_{Bscale}$ would be able to outperform $Dir_nH_{NB}S_-$ for machines within the 512 node A-1000 limit.

While EVOLVE certainly provides the best available workload for studying broadcasts, its simple structure and corresponding worker-set behavior do not have a great dynamic range. Figures 7-2(d) and 7-3(d) show the application's behavior for the second iteration of the genome evolution algorithm. EVOLVE exhibits slightly more interesting behavior — in the form of larger worker sets — over the entire length of a run. The large worker sets do not produce a significant effect in the model, because the model does not consider the impact of hot-spots and bottlenecks; however, they do affect the speedups in Figure 8-1. A bandwidth-hungry application with more widely-shared data than EVOLVE would show more significant differences between the various protocols.

The preceding case-study suggests a question that needs to be answered before incorporating $Dir_tH_5S_{Bscale}$ into a production system: given an application and a system configuration, what is the appropriate setting for the broadcast threshold? The answer is determined by the relative performance of the sequential and the broadcast schemes, as well as the requirements of the application.

For sufficiently large systems, there must be a worker-set size such that the latency of broadcasting invalidations is equal to the latency of a sequential scheme. Similarly, there must be a worker-set size such that the bandwidth of a broadcast is equal to the bandwidth of sequential invalidations. If the latency-equivalence point is equal to the bandwidth-equivalence point, the broadcast threshold should always be set to the worker-set size corresponding to this mutual equivalence point. Then, the adaptive protocol would always select the optimal strategy.

Due to the small constant factor of the sequential invalidation scheme, the bandwidth-equivalence point should be lower than the latency-equivalence point. In this case, the appropriate choice of threshold would depend on the requirements of each application. For bandwidth-limited applications, the threshold should be set to the bandwidth-equivalence point; the opposite is true for latency-limited applications. Most real programs would probably fall somewhere in between. In practice, it might be difficult to calculate the equivalence points *a priori*. Instead, calibration techniques such as the one used with Strata [10] would be appropriate for determining the protocol balance thresholds.

Despite the remaining questions involved in the implementation and configuration of adaptive broadcast protocols, the case study shows that they can be implemented and promise to improve the performance of large systems. This class of protocols also serves as a good example of a (necessarily simple) scheme that uses on-line feedback to improve the performance of the system as a whole.

8.3 The LimitLESS Profiler

While on-line adaptive techniques might be able to ameliorate the effects of certain classes of data, it is the programmer who has the knowledge required to make real improvements in the performance of an application. The LimitLESS profiler is a measurement and feedback mechanism that uses Alewife's software-extended system to give programmers information about the way that applications use shared memory. This section describes the implementation and the interface of this optimization tool.

8.3.1 The Profiler Interface

The LimitLESS profiler is a working part of the Alewife programming tool set. In order to use it, a programmer sets a single variable in the runtime environment. Subsequently, when the programmer runs a program, the memory system gathers information whenever it invokes software to handle a memory request.

The overhead of the profiling mechanism depends on the operating characteristics of the application. For applications that run for many seconds and cause little communication between processors, the overhead is small. For example, when running the Water benchmark (216 molecules, 10 iterations) on a 16 node A-1000, the time required to execute the application increases by only 1% with LimitLESS profiling enabled. Applications that run for only a fraction of a second and make constant use of shared memory incur a higher overhead. MP3D (10000 particles, 5 iterations) requires 45% more time to run with the profiler than without it.

After the application finishes running or at any time during its execution, the programmer can request the system to dump the information gathered by the profiler. This information may be stored on disk for subsequent, off-line examination. Having dumped the information, the programmer can request a report of memory usage. Although many possible formats for presenting the data are possible, only one has been released to programmers thus far.

Figure 8-2 shows a sample LimitLESS profiler report, which displays information about the MP3D application. The figure shows a window containing an `Emacs` (text editor) display. The lower half of the screen displays records in the report; the first record gives information for hexadecimal program counter `101618`, which corresponds to the function `boundary`, line `49`, in the MP3D source file `adv.c`. Specifically, the program counter indicates the first remote memory access to the structure slot `BCond->bc_A`.

During the course of execution, the instruction at this program counter caused the first remote access to 13 different memory blocks (termed `cache lines` in the report). After the first remote reference to these 13 memory blocks, their worker sets grew to a maximum size of between 6 and 15, with a median size of 8. Furthermore, MP3D accesses `Bcond->bc_A` as read-only data. That is, none of the memory blocks required

```
static void boundary(bn, Part)
int bn;                  /* boundary number */
struct particle *Part;   /* molecule */
{
  int i,
      local_next;
  float d2,
        temp;
  struct bcond *BCond = &Bconds[bn];
  struct ares *RPart;

  ++local_BC;
  if (bn < COND_OFFSET)   /* Solid wall case */ {
█   d2 = 2*(BCond->bc_A*Part->x
          + BCond->bc_B*Part->y
          + BCond->bc_C*Part->z
          + BCond->bc_D);
    if (d2 < 0.0) {
      Part->x -= d2 * BCond->bc_A;
      Part->y -= d2 * BCond->bc_B;
--%%-Emacs: adv.c                  (C)---- 6%------------------
#x101618: adv.c:49: in boundary
    d2 = 2*(BCond->bc_A*Part->x
            ^

   13 LimitLESS cache lines:
   worker set size between 6 and 15, median is 8
   read-only data when in LimitLESS mode
--------------------------------------------------------
#x101634: adv.c:51: in boundary
          + BCond->bc_C*Part->z
            ^

   8 LimitLESS cache lines:
   worker set size between 6 and 15, median is 8
   read-only data when in LimitLESS mode
--------------------------------------------------------
#x101A00: adv.c:61: in boundary
      temp = BCond->bc_temp*(1.0 - BCond->bc_coef);
                                   ^

   1 LimitLESS cache line
   worker set size is 11, read-only data when in LimitLESS mode
--------------------------------------------------------
-----Emacs: *compilation*          (Fundamental)---- 6%---------
M-x next-error
```

Figure 8-2: Sample report with LimitLESS profile information for MP3D.

the LimitLESS software to process a write request by sending invalidation messages. Section 8.4 describes the importance of the read-only designation.

The programmer can use the `next-error` command, shown at the bottom of the display, to scroll through each record in the report. This command causes `Emacs` to move the subsequent record to the first line of the bottom window, load the appropriate source file, and position the cursor on the relevant line of code. In Figure 8-2, the top half of the screen displays `adv.c`, with the cursor on line 49.

8.3.2 Example of Use

The LimitLESS profiler proved its usefulness almost as soon as it became available. When displaying information about a program named `gauss`, the profiler displayed this record:

```
-----------------------------------------------------------
#x1004A0: gauss.c:134: in threadproc
      for ( i = myid; i < N; i += Globl->nprocs )
                                 ^
   1 LimitLESS cache line
   worker set size is 15, 29 writes in LimitLESS mode
-----------------------------------------------------------
```

The 29 writes to the `nprocs` slot of `Globl` were completely unexpected. Upon examining the program, Ricardo Bianchini (a graduate student visiting the Alewife group from the University of Rochester) found the following lines of code:

```
typedef struct
{
    element  a[N][N];
    int nprocs;
    p4_lock_t pivot_done[N];
}  GlobalData;

GlobalData *Globl;
```

This structure definition caused `nprocs`, a read-only slot with a large worker set, to be stored in the same memory block as `pivot_done`, a synchronization object accessed by a small worker set with read-write semantics. Ricardo realized that by adding some padding into the structure, the accesses to the two conflicting data types could be decoupled, thereby improving performance.

8.3.3 Implementation

There are four components of the LimitLESS profiler: configuration, on-line profiling, information dump, and report generation. Configuration allows the programmer to enable the profiler; on-line profiling gathers information about program counters and associated memory addresses while running the program; information dump transfers the profile data from the Alewife machine to the programmer's workstation, and report generation matches program counters with memory blocks and formats the reports. The last two components correspond to the feedback portion of the smart memory system.

From the point of view of this study, the on-line profiling component is the most interesting, because it is built on top of the flexible coherence interface. The memory protocol used to gather information about program counters and memory blocks is a hybrid of $Dir_nH_0S_{NB,ACK}$ and $Dir_nH_XS_{NB}$, where $X \geq 2$.

Each memory block starts out in the initial state of $Dir_nH_0S_{NB,ACK}$, which allows all intranode accesses to complete without software intervention, but generates an interrupt upon the first remote access to a memory block. When the software detects the first remote access, it performs two actions: first, it changes the state of the memory block to the appropriate state in $Dir_nH_XS_{NB}$. Then, the protocol uses an active message, rather than a normal protocol message, to transmit the memory block's data back to the requesting node. Upon receiving this active message, the requesting node records the current program counter and memory block address in a special buffer. This information is used to *map program counters and memory blocks,* thereby solving the dynamic data allocation problem in the feedback step. Since this extra processing happens only upon the first remote access to data, programs that run for long periods of time without allocating new memory blocks tend to have lower overhead than programs that run for short amounts of time or allocate many blocks.

The rest of the information about shared memory usage comes from the normal $Dir_nH_XS_{NB}$ operation. In order to gather data, the profiler splices into the hash table functions listed in Table 5.1. During execution, the profiler spoofs the operation of the hash table and saves information about worker-set sizes and the number of each type of access. This component of the profiler only gathers information for memory blocks that require LimitLESS software handling. Thus, MP3D incurs more overhead than Water because it requires more LimitLESS software processing.

By selecting a protocol with fewer hardware pointers than 5 (the default), the programmer can observe the behavior of smaller worker sets at the expense of more profiling overhead. In general, higher profiler overhead indicates more useful information for the programmer. Since the profiler uses a non-trivial amount of storage for every shared memory block, the programmer usually can not profile an application running on its maximum data set size. However, experience shows that small data sets capture much of the information about the behavior of many applications.

8.4 Profile, Detect, and Optimize

While it is useful for a system to be able to provide information directly to the programmer, a truly smart memory system should require little, if any, input from the programmer to improve performance. This section describes a method that allows the compiler and the memory system to cooperate and automatically optimize the performance of widely-shared, read-only data. This method has three stages. First, the runtime system profiles an application's execution. Second, an analysis routine uses the profile information to detect read-only memory blocks. Third, the compiler generates code that causes the runtime system to optimize accesses to the detected blocks. Hence, this method is called

PRODO for *pro*file, *d*etect, and *o*ptimize.

A PRODO system has been implemented for Alewife programs written in Mul-T. This section describes the system and some initial experience, including the user interface, the implementation, and a case study using the EVOLVE benchmark. The description concludes by examining some relevant architectural features in the Alewife machine.

8.4.1 The PRODO Interface

Although the PRODO system detects and optimizes memory accesses automatically, it requires some direction on the part of the programmer. Most of the programmer's tasks involve specifying interesting data objects and providing information that helps the system work around dynamic memory allocation and aliasing problems. The programmer must annotate an application, compile it in a special environment, run the LimitLESS profiler, and then instruct the compiler to perform its optimizations.

Table 8.1 shows the three program annotations, which are all easy to use. When `prodo` is used to call a primitive allocation function (*e.g.* `make-vector`), it causes the compiler to apply the PRODO method to the data object returned by the function. The `define-alloc` and `call-alloc` macros are used to define and to call allocation functions that either call other allocation functions or contain `prodo` declarations. For example, the following line of code creates a high-level function that always makes a vector with five elements:

```
(define-alloc (make-5-vector) (prodo make-vector 5))
```

A corresponding line of code might allocate a top-level instance of this type of vector:

```
(define *my-5-vector* (call-alloc make-5-vector))
```

Given this code, `*my-5-vector*` would be a global variable that points to a 5-element vector. The PRODO system would automatically profile `*my-5-vector*` and optimize it, if possible.

After annotating a program, running the PRODO system is almost trivial. At present, the actions required to run PRODO include the normal compilation commands, the LimitLESS profiler commands, and three other commands. For a production version of the system, the entire process could easily be automated completely: the programmer would only need to provide a list of the source files and arguments to be used when running the program with the profiler.

8.4.2 Implementation

The PRODO system consists of a heuristic for detecting read-only data and an optimization that uses a broadcast protocol to ensure safety. The heuristic attempts to match allocated data objects with the behavior of memory blocks. It assumes that there is a one-to-one mapping between each unique type of data object and the procedure call tree when it is allocated. The macros listed in Table 8.1 help the PRODO system keep track

Macro	Purpose
`(prodo prim-alloc . args)`	call `prim-alloc` with `args` and PRODO the result
`(define-alloc (alloc-fun . args)` `. body)`	define an allocation function define an allocation function
`(call-alloc alloc-fun . args)`	call `alloc-fun` with `args`

Table 8.1: PRODO macros.

of important call sequences. In a sense, the programmer uses the macros to specify the branches of the call tree that are important to track.

During the compilation phase that takes place before profiling, each instance of the `prodo` and `call-alloc` macros generates a unique identifier. The `define-alloc` macro declares an additional argument for the corresponding function. This argument is used to pass through the identifier generated by `call-alloc`. During the profiling phase, each `call-alloc` within a function labeled with `define-alloc` creates a two-word cell that is used to chain the procedure call information into an actual call-tree data structure. As the program runs, the macros generate enough information to trace the call tree down to the allocation of every object labeled with the `prodo` macro. In addition, the PRODO system stores a record of the address and size of every `prodo` object.

After the profiling phase, the PRODO system combines the information about call trees and data object sizes with the information gathered by the LimitLESS profiler. The system first matches all data objects with their call trees. Then, the analyzer does a brute-force search that attempts to correlate the location of read-only data within each object. More formally, it attempts to find (*base,stride*) pairs in each object such that

$$\forall x[((y = \textit{base} + x \times \textit{stride}) \wedge (y \in \texttt{object})) \Rightarrow y \in \texttt{read-only}],$$

where *base* is a constant integer offset from the address of each object, *stride* is a multiple of the memory block size, `object` is the set of all memory blocks contained within a unique type of object, and `read-only` is the set of all read-only memory blocks. The search algorithm checks all possible (*base,stride*) pairs and runs in time $O(n^2)$, where n is the total number of memory blocks in the profiled data objects.

Having identified all of the (*base,stride*) pairs that correspond to read-only data, the PRODO system is ready to optimize the program in a final compilation phase. During this phase, the system simplifies the call-tree identification code and inserts a safe optimization for each (*base,stride*) pair. The code simplification replaces instructions that dynamically construct the call-tree data structure with instructions that pass just enough information between functions to indicate when an optimization should take place. This information is much more concise than the call-tree data structure, because only a few leaves of the call-tree correspond to read-only data. While the profiled application allocates cells dynamically and chains them together, the optimized program uses shift-and-add sequences and passes integers down through the call tree.

The safe optimization for each (*base,stride*) pair is a system call that sets the appropriate directory entries into a special read-only state. When the program runs, allocating

EVOLVE version	Memory System	Speedup
Base	$Dir_nH_5S_{NB}$	29.8
PRODO	$Dir_nH_5S_{NB}$	36.4
PRODO	$Dir_nH_5S_{NB,trap\text{-}off}$	37.2
Base	$Dir_nH_{NB}S_{-}$	49.0

Table 8.2: Performance of different versions of EVOLVE.

read-only structures takes slightly longer due to this extra work. During the operation of the program, the memory system treats this read-only data as if it had exceeded the threshold of a broadcast protocol. The memory system software never extends the directory entry, thereby avoiding most of the overhead associated with read-only data. In a memory system that allows the software to disable directory overflow traps, the optimization would avoid all of the overhead.

This optimization is *safe,* because the protocol can use a broadcast to invalidate cached copies should the program ever attempt to write the data. At the end of the broadcast, the memory block reverts back to the normal $Dir_nH_5S_{NB}$ protocol, thereby cancelling the incorrect optimization.

8.4.3 A Case Study

In order to determine the efficacy of this type of smart memory system, the EVOLVE benchmark was annotated with the appropriate macros and optimized with the PRODO system. The annotations required modifying only 3 out of the 117 lines of code in the benchmark, and 40 out of 394 lines in a code library. The profiling phase took place on a 16 node A-1000 with a ten dimensional problem set, and the entire PRODO process took less than a minute.

Table 8.2 compares the performance of the base (unoptimized) version and the PRODO (optimized) versions of EVOLVE. The measurements use both a larger system size (64 simulated nodes) and a larger problem size (12 dimensions). The first row of the table shows the base performance of the application with the $Dir_nH_5S_{NB}$ protocol, and the last row shows the performance with $Dir_nH_{NB}S_{-}$. The middle two lines show the increase in performance due to the PRODO optimizations. Although the table specifies the $Dir_nH_5S_{NB}$ protocol, the protocol would have transmitted broadcasts if it detected any incorrect optimizations. Instrumentation in NWO shows that no such events occurred during the simulations of the optimized version of EVOLVE.

The second line of the table indicates that the PRODO optimizations resulted in a 22% improvement in performance with $Dir_nH_5S_{NB}$. $Dir_nH_5S_{NB,trap\text{-}off}$, which allows the software to disable directory overflow traps, permits the optimized program to achieve a 25% performance improvement. The *trap-off* feature would be enabled by the same one-line change to the CMMU source code mentioned in Section 8.2.

The improvement shown by the PRODO system should encourage more work on this type of method for performance enhancement. From the point of view of the programmer, the memory system is truly "smart," because it can increase an application's performance with very little external help.

8.4.4 Architectural Mechanisms

Having implemented a smart memory system that automatically takes advantage of the semantics of read-only data, it is worth reexamining the original proposal for such a system. Even the early work on the Alewife protocols recognized the importance of optimizing the memory accesses for this important type of data [14]. As a result, the Alewife instruction set architecture includes a *software-coherent load,* which causes the memory system to fetch a read-only copy of data without allocating a directory pointer. When using this instruction, the application *software* (written by the programmer or generated by the compiler) assumes the responsibility for guaranteeing coherence.

Such an instruction may be used by the compiler to improve the performance of accesses to read-only data. In fact, the Alewife compiler has used the instruction for a few data objects, including instructions and procedure closures. However in the present system, instructions are stored in each node's private memory, so the software-coherent load is rarely used.

It would also be possible to follow the lead of other systems (*e.g.* [11]) that allow the programmer to designate data as read-only. The compiler would merely translate read-only data accesses into software-coherent loads. While it is useful to provide this feature in shared-memory systems, all of the responsibility for the correctness and the success of this approach lies with the programmer: if a variable is mislabeled, the program returns the wrong answer; if the programmer forgets to add a label, performance suffers; if a generic function is used to manipulate many types of data, its memory accesses can not be optimized using the software-coherent load.

Nevertheless, the original proposal for the PRODO scheme assumed that the memory system could give the compiler enough information to use this instruction successfully. Automatic optimization could relieve the programmer of the burden of labeling data throughout the program! There are two reasons why this hypothesis is wrong: first, the aliasing problem makes it impossible to map all instructions to data objects or vice versa. In general, information from the memory system about one `(instruction,data-object)` pair indicates nothing about other data objects accessed by the `instruction` or about other instructions that access the `data-object`. Without a general method for propagating information from instruction to instruction or from object to object, the requisite analysis becomes very difficult.

Second, the software-coherent load is an unsafe optimization. When it is used, the system assumes that application software will ensure the coherence of the data. Since the programmer ostensibly has total knowledge about the application, it is reasonable to provide an opportunity for improved performance at the risk of incorrect behavior. Neither the same knowledge nor the same options are appropriate for an automatic optimization system. Since the information provided by the profiler is at best statistically valid, any optimization has a chance of being incorrect. Given the choice between correct behavior and improved performance, almost every (sane) programmer would choose the former. Thus, the safe memory-side optimization implemented for Alewife serves as a

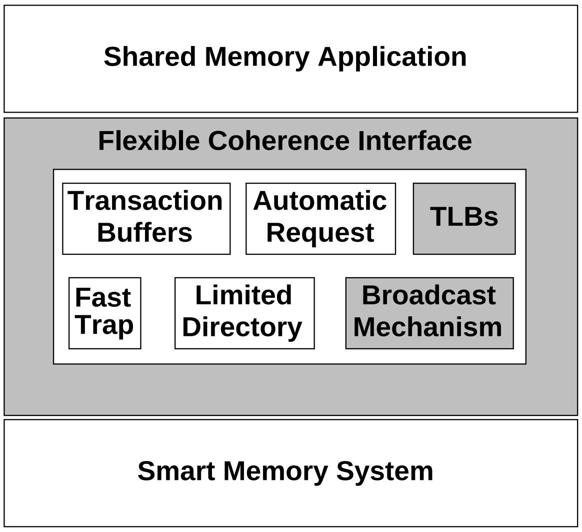


Figure 8-3: A flexible coherence interface serves as a basis for hardware-accelerated coherent shared memory.

much more stable foundation for the PRODO system.

8.5 Hardware-Accelerated Coherent Shared Memory

By balancing the time required to collect data with the benefits that the information provides, a software-extended memory system can use intelligence to augment the shared-memory model and to improve multiprocessor performance. The adaptive broadcast protocols, the LimitLESS profiler, and the PRODO system achieve this balance in different ways. Yet, they are all implemented on top of a single hardware base with the flexible coherence interface.

Reversing the software-extended approach leads to a top-down methodology for building a distributed shared memory architecture. Instead of extending limited hardware with software, the goal is to use hardware to accelerate slow software. Figure 8-3 illustrates an architecture that follows this hardware-acceleration approach. The foundation of the architecture is a flexible coherence interface that provides an abstraction between shared memory software and hardware. In addition to expediting software development, the interface allows multiprocessor applications and smart memory systems to run on a variety of different hardware platforms with varying support for shared memory.

The support takes the form of hardware mechanisms that accelerate functions in the memory-system software. Functions that hardware does not accelerate must be emulated completely in software. For example, Alewife's hardware mechanisms include limited directories, fast traps, automatic remote data requests, and transaction buffers. Alewife must emulate broadcasts and page tables in software, because the architecture lacks a broadcast network and translation-lookaside buffers.

In fact, several independent conclusions derived from the work on software-extended memory systems support this approach to shared-memory design. First, the budding success of smart and flexible memory systems indicates an increasing number of techniques for using intelligence to improve multiprocessor efficiency. For these techniques to become generally useful, they need to be portable. Only a software library with a flexible and standard interface will achieve this goal.

Second, the analytical model indicates that different operating regimes demand different hardware acceleration techniques. While limited directories benefit a system with Alewife's low-latency network, they would not be appropriate for a system with much higher latencies. Since computer markets demand a range of cost versus performance trade-offs, any portable system must be able to accommodate a range of hardware options.

Finally, Alewife's flexible coherence interface already supports partial hardware independence: the interface hides the binary format of directories, the details of message transmission, and the size of the unit of data coherence. Using this abstraction, it would not be hard to port Alewife's adaptive broadcast protocols to a software-only directory architecture. The port would require an emulation of the underlying LimitLESS directory scheme, a task that is certainly easier than writing an entire $Dir_nH_0S_{NB,ACK}$ protocol from scratch. For performance reasons, a software-only directory architecture with high network latency would need to use a larger unit of coherence than Alewife. Since the flexible interface hides the memory block size, not a single line of code in the higher-level adaptive protocol would need to change. The same sort of methodology could be used to abstract all of the mechanisms illustrated in Figure 8-3.

Software extension and its dual, hardware acceleration, provide an architecture for building a range of cost-efficient implementations of shared memory. Low-cost, moderate performance systems can use software-only directories; more expensive, high performance systems can implement the hardware structures required to sustain low latency memory access. With a flexible coherence interface, this range of implementations would support a common set of applications, running on top of a single smart memory system. Packaged as a line of products, this architecture would solve the ultimate goal of cost-effectivity: customers could choose price/performance combinations according to their own needs for computational power.

Chapter 9

Conclusions

The software extension approach offers a cost-efficient method for building scalable, coherent, high-performance shared memory. Two implementations of the Alewife architecture testify to this conclusion: the A-1000 proves that the approach works in practice, and the NWO experiments demonstrate a range of performance versus cost alternatives.

An analytical model — validated by the empirical study — surveys the design space. Architectural features such as trap latency and code efficiency do not seriously impact the performance of software-extended systems, as long as they implement at least one hardware directory pointer. Specific hardware optimizations for features of cache coherence protocols can enhance performance slightly, but they are not essential.

Alewife's flexible coherence interface facilitates the rapid development of new software-extended protocols. This interface hides hardware implementation details and provides a convenient message-based programming abstraction. Anecdotal evidence suggests that the abstraction is well worth a small decrease in performance; however, production software-extended systems will blend hand-tuned optimizations and higher-level code.

The software-extended approach, combined with a flexible coherence interface, enables a class of smart memory systems. These systems use dynamic and static feedback techniques to boost multiprocessor performance, with and without the help of application programmers. Experience with three such techniques on Alewife indicates that incorporating software into a memory system has far greater benefits than mere cost-management.

9.1 Recommendations for Distributed Shared Memory

In order to achieve high performance with a large number of processors, a cost-efficient shared memory architecture requires a LimitLESS hardware directory and a flexible coherence interface. The minimal architecture, called *1dir* (pronounced "wonder"), includes:

- Associative processor caches. The associativity may take the form of victim caches.

- One pointer per directory entry ($Dir_nH_1S_{NB,LACK}$) that stores an arbitrary node identifier and serves as an acknowledgment counter. The directory entry should also implement a one-bit pointer for the local node. This configuration optimizes for producer-consumer communication.
- One processor and memory module per node, with streamlined intranode accesses.
- A fast interface between processors and the interconnection network. Transmitting or receiving a message should take on the order of ten instructions.
- A processor interrupt mechanism. Processors should execute no more than 100 instructions between an asynchronous event and the beginning of its handler.
- Intelligent software that rides on top of a flexible coherence interface. Smart memory systems maximize performance by generating synergy between programmers, compilers, and runtime environments.

This recommendation agrees with the findings of Hill, Wood, *et. al.*: suitably tuned one-pointer protocols yield performance comparable to more expensive architectures.

The software-extension approach leads to the *1dir* architecture and enables a cost versus performance trade-off in shared memory design. Designers can add additional directory pointers to the architecture, spending DRAM for increased performance and flexibility. Extra pointers improve performance by capturing more complicated communication patterns than producer-consumer in hardware. The corresponding reduction in software-extension events decreases a system's sensitivity to software processing time, thereby increasing flexibility.

Distributed shared memory designers might also choose to eliminate the hardware directory entirely. Such software-only directory architectures lie at the inexpensive end of the spectrum of software-extended systems and are a deceptively low-cost implementation of shared memory. While such systems do avoid the cost of the control and state required to implement a hardware directory, they are extremely sensitive to the quality of memory-system software. The designers of these systems will be faced with the choice of reducing network speed and amortizing communication delay over large blocks of data, or implementing hardware mechanisms — such as dedicated processors — to speed up the software.

The performance of *1dir* depends, in part, on the quality of its memory system software. The architecture will benefit from a library of smart memory techniques that optimize the way that programs use shared memory. The future of research on multiprocessors lies not in new methods for implementing distributed shared memory, but in finding ways to integrate the hardware and software components of multiprocessors into a coherent whole.

9.2 Future Work

Unfortunately, it is hard to extrapolate from a single data point. Since Alewife is the only working multiprocessor with a software-extended memory system, conclusions about the entire design space are tentative at best. Fortunately, the research area is an active one and new machines will add additional data points to the design space.

In the near term, the Alewife project will continue progress towards building a production version of software-extended shared memory. The addition of more applications and microbenchmarks to the machine's repertoire will drive a phase of performance and reliability tuning. Performance tuning will result in improvements in many parts of the software system, including the flexible coherence interface. Reliability tuning will involve adding features that allow the system to run for long periods of time.

The long term will see new systems that investigate the design and cost of various hardware mechanisms for accelerating multiprocessor performance. In an effort to provide a convenient and efficient abstraction to application programmers, these systems will attempt to consolidate shared memory, message passing, and a virtual machine model. Ultimately, the market will base decisions about parallel architectures on cost-effectiveness, convenience, and politics. I hope that elegance survives the frenzy.

Appendix A

Experimental Data

A.1 Simulation Measurements.

Coherence Scheme	Protocol Options	Execution Time
$Dir_n H_{NB} S_-$	None.	620874
$Dir_4 H_{NB} S_-$	None.	1356447
$Dir_2 H_{NB} S_-$	None.	1527552
$Dir_1 H_{NB} S_-$	None.	1575031
$Dir_n H_4 S_{NB}$	25 cycle latency	594716
	50 cycle latency	654444
	100 cycle latency	689113
	150 cycle latency	703801
$Dir_n H_3 S_{NB}$	50 cycle latency	614585
$Dir_n H_2 S_{NB}$	50 cycle latency	669784
$Dir_n H_1 S_{NB}$	50 cycle latency	920283

Table A.1: ASIM: Execution times for Weather. (Figures 4-3, 4-4, and 4-5.)

Dir_n	Worker-Set Size								
Protocol	1	2	4	6	8	10	12	14	16
$H_8 S_{NB}$	1	1	1	1	1	0.350	0.348	0.369	0.360
$H_5 S_{NB}$	1	1	1	0.316	0.306	0.339	0.348	0.358	0.357
$H_2 S_{NB}$	1	1	0.215	0.257	0.291	0.297	0.312	0.323	0.324
$H_1 S_{NB}$	1	0.157	0.207	0.248	0.264	0.279	0.290	0.298	0.300
$H_1 S_{NB,LACK}$	1	0.161	0.207	0.166	0.184	0.198	0.210	0.216	0.219
$H_1 S_{NB,ACK}$	1	0.136	0.136	0.118	0.119	0.121	0.126	0.125	0.126
$H_0 S_{NB,ACK}$	0.150	0.130	0.122	0.0987	0.0986	0.0994	0.0979	0.0993	0.0967

Table A.2: NWO: WORKER performance compared to full-map. (Figure 6-1.)

Pointers	TSP	AQ	SMGRID	EVOLVE	MP3D	Water
0	39.6	39.3	11.4	11.5	2.6	28.4
1	53.4	52.3	30.9	21.1	10.2	37.4
2	53.3	51.6	36.0	30.8	17.0	40.1
5	54.7	51.6	38.9	35.2	20.4	41.0
64	55.8	51.6	41.4	49.8	24.0	42.1

Table A.3: NWO: Speedup over sequential, 64 nodes. (Figure 6-2.)

Pointers	normal ifetch no victim	normal ifetch victim	perfect ifetch no victim	256 nodes
0	2.7	39.6	44.4	57.6
1	5.2	53.4	56.8	113.1
2	9.0	53.3	57.8	122.0
5	15.3	54.7	59.8	134.3
64	47.9	55.8	59.4	141.9

Table A.4: NWO: Detailed TSP measurements. (Figures 6-3 and 6-4.)

WSS	Number
1	9646
2	1391
3	316
4	708
7	1308
8	942
11	168
14	2
15	1
16	38
18	8
19	1
20	6
24	1
28	1
32	16
33	4
57	1
64	25

Table A.5: NWO: EVOLVE worker-set sizes. (Figure 6-5.)

Pointers	not skewed	skewed
0	6.3	11.5
1	12.1	21.1
2	18.8	30.8
5	27.5	35.2
64	44.6	49.8

Table A.6: NWO: Detailed EVOLVE measurements. (Figure 6-6.)

Pointers	TSP	AQ	SMGRID	EVOLVE	MP3D	Water
0	14.0	14.2	7.00	3.54	0.60	5.46
1	15.4	14.9	15.2	9.69	7.64	10.8
2	15.4	14.9	15.3	10.9	7.91	11.0
5	15.5	14.9	15.4	12.6	7.91	11.1
64	15.5	14.9	15.4	12.9	7.96	11.1

Table A.7: A-1000: Speedup over sequential, 16 nodes. (Figure 6-7.)

Protocol	Threshold	Speedup
$Dir_nH_{NB}S_-$		49.0
$Dir_5H_BS_-$		42.7
$Dir_5H_5S_{Bscale}$		35.3
$Dir_tH_5S_{Bscale,trap\text{-}off}$	4	29.9
	8	38.5
	16	38.3
	32	31.5
	64	30.2
$Dir_tH_5S_{Bscale}$	4	29.9
	8	34.7
	16	33.9
	32	30.8
	64	29.6
$Dir_5H_5S_B$		30.7
$Dir_nH_5S_{NB}$		29.9

Table A.8: NWO: EVOLVE with broadcasts. (Figure 8-1.) The values are slightly different than in Table A.6 due to several months of changes in the Alewife kernel and compiler.

A.2 Model Parameters

Table A.9 lists extra model parameters that are needed to model the Alewife memory hierarchy. These parameters, which were omitted from the discussion in Chapter 7 to eliminate gratuitous detail, are used as follows:

$$\begin{aligned}
h &= h_{cache} + h_{txnbuf} \\
T_h &= \frac{h_{cache} T_{h,cache} + h_{txnbuf} T_{h,txnbuf}}{h} \\
l &= l_{nonet} + l_{net} \\
T_l &= \frac{l_{nonet} T_{l,nonet} + l_{net} T_{l,net}}{l}
\end{aligned}$$

The values of all of these parameters are listed in Table A.10, along with the other model inputs.

Symbol	Meaning
h_{cache}	cache hit ratio
h_{txnbuf}	transaction buffer hit ratio
$T_{h,cache}$	cache hit latency
$T_{h,txnbuf}$	transaction buffer hit latency
l_{nonet}	ratio of local accesses that require no network messages
l_{net}	ratio of local accesses that require network messages
$T_{l,nonet}$	latency of local accesses that require no network messages
$T_{l,net}$	latency of local accesses that require network messages

Table A.9: Additional model parameters for the Alewife memory hierarchy.

Symbol	TSP	AQ	SMGRID	EVOLVE	MP3D	Water
N_i	28186444	17933534	98117235	19056262	19669795	110747983
N_a	5892391	7447602	11409727	4317830	3860909	20077454
h	0.96	0.988	0.91	0.984	0.894	0.99
l	0.0339	0.00861	0.0472	0.0058	0.0176	0.00806
r	0.00568	0.00334	0.0432	0.00997	0.0885	0.00227
T_h	1.32	2.03	1.31	1.38	1.35	1.86
T_l	13.4	13.4	16.7	28.7	16.5	13.9
$T_{r,hw}$	55.2	59.5	52.5	54.4	85.4	60.0
h_{cache}	0.95	0.988	0.899	0.983	0.892	0.989
h_{txnbuf}	0.0101	0.000291	0.011	0.00167	0.00159	0.000420
$T_{h,cache}$	1.27	2.03	1.24	1.37	1.34	1.86
$T_{h,txnbuf}$	6.02	7.01	6.76	7.56	8.18	10.1
l_{nonet}	0.0338	0.00856	0.0417	0.00455	0.0163	0.00776
l_{net}	0.0000996	0.0000467	0.00543	0.00124	0.00132	0.000299
$T_{l,nonet}$	13.2	13.1	13.0	12.8	12.1	10.7
$T_{l,net}$	68.0	69.6	44.6	86.6	70.9	96.0

Table A.10: Model input parameters for six benchmarks.

Nodes	TSP	AQ	SMGRID	EVOLVE	MP3D	Water
0	0	0	0	175	2120	90
1	1362	62	47997	7042	19054	2409
2	645	0	16856	6675	2169	1420
3	362	0	4414	6250	790	1259
4	478	0	3444	5961	1199	1087
5	95	0	4007	5962	167	900
6	42	0	3214	5953	145	859
7	11	0	1058	2824	68	800
8	5	0	854	432	69	799
9	4	0	3458	222	54	798
10	4	0	356	221	62	792
11	2	0	96	92	69	781
12	2	0	85	66	82	791
13	2	0	109	63	62	779
14	2	0	44	61	59	779
15	2	0	44	60	53	777
16	2	0	41	40	38	782
17	2	0	41	42	28	770
18	2	0	29	50	32	775
19	2	0	25	34	28	769
20	2	0	25	31	28	771
21	2	0	20	29	27	768
22	2	0	20	29	28	768
23	2	0	20	28	26	768
24	2	0	23	28	25	770
25	2	0	73	28	27	768
26	2	0	16	28	22	770
27	2	0	18	28	22	769
28	2	0	19	28	22	768
29	2	0	20	28	22	768
30	2	0	20	28	22	773
31	2	0	19	28	22	769

Table A.11: Read access worker-set histograms: 0 – 31 nodes. (Figure 7-2.)

WSS	TSP	AQ	SMGRID	EVOLVE	MP3D	Water
32	2	0	75	448	22	211
33	2	0	13	16	22	19
34	2	0	12	17	22	0
35	2	0	11	17	22	0
36	2	0	14	16	22	0
37	2	0	13	16	22	0
38	2	0	13	17	22	0
39	2	0	11	16	22	0
40	2	0	12	15	22	0
41	2	0	11	17	22	0
42	2	0	23	14	22	0
43	2	0	15	14	22	0
44	2	0	16	13	22	0
45	2	0	31	13	22	0
46	2	0	23	13	22	0
47	2	0	15	13	22	0
48	2	0	15	13	22	0
49	2	0	9	13	22	0
50	2	0	11	13	23	0
51	2	0	10	13	22	0
52	2	0	11	13	23	0
53	2	0	8	13	22	0
54	2	0	8	13	21	0
55	2	0	16	13	20	0
56	2	0	8	13	21	0
57	2	0	9	12	20	0
58	2	0	10	12	21	0
59	2	0	8	11	20	0
60	2	0	12	11	19	0
61	2	0	10	11	19	0
62	2	0	8	11	20	0
63	2	0	8	11	20	0
64	491	0	227	174	22	0

Table A.12: Read access worker-set histograms: 32 – 64 nodes. (Figure 7-2.)

WSS	TSP	AQ	SMGRID	EVOLVE	MP3D	Water
0	0	0	38	0	2054	210
1	5491	6102	101071	5916	306494	27298
2	7	35	32009	0	14872	902
3	0	0	6591	0	1241	183
4	0	0	363	0	405	233
5	0	0	220	0	15	0
6	0	0	77	0	18	10
7	0	0	8	4095	77	60
8	0	0	9	0	4	0
9	0	0	111	0	0	0
12	0	0	0	0	1	0
16	0	0	0	0	1	0
17	0	0	0	0	1	0
24	0	0	0	0	1	0
26	0	0	1	0	0	0
32	0	0	0	0	0	288
33	0	0	0	0	0	96
60	0	0	0	0	1	0
64	0	0	0	0	13	0

Table A.13: Write access worker-set histograms. (Figure 7-3.)

Symbol	Value
N_i	100000000
N_a	20000000
h	0.95
l	0.00
r	0.05
T_h	1.2
T_l	—
$T_{r,hw}$	50

Table A.14: Model synthetic input parameters. (Figures 7-18, 7-19, and 7-20.)

Bibliography

[1] Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14. IEEE, June 1990.

[2] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of the Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. Available as MIT/LCS/TM-454.

[3] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An Evolutionary Processor Design for Multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[4] Anant Agarwal, Beng-Hong Lim, David A. Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114. IEEE, June 1990.

[5] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289. IEEE, June 1988.

[6] Andrew W. Appel and Kai Li. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107. ACM, April 1991.

[7] *ANSI/IEEE Std 1596-1992 Scalable Coherent Interface*, 1992.

[8] Henri E. Bal and M. Frans Kaashoek. Object Distribution in Orca using Compile-Time and Run-Time Techniques. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, September 1993.

[9] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the 38th IEEE Computer Society International Conference*, pages 528–537. IEEE, February 1993.

[10] Eric A. Brewer. *Portable High-Performance Supercomputing: High-Level Platform-Dependent Optimization*. PhD thesis, MIT, Department of Electrical Engineering and Computer Science, September 1994.

[11] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of MUNIN. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[12] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[13] David Chaiken and Anant Agarwal. Software-Extended Coherent Shared Memory: Performance and Cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 314–324. IEEE, April 1994.

[14] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache-Coherence in Large-Scale Multiprocessors. *IEEE Computer*, 23(6):41–58, June 1990.

[15] David Chaiken and Kirk Johnson. NWO User's Manual. ALEWIFE Memo No. 36, Laboratory for Computer Science, Massachusetts Institute of Technology, June 1993.

[16] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234. ACM, April 1991.

[17] David Lars Chaiken. Cache Coherence Protocols for Large-Scale Multiprocessors. Technical Report MIT/LCS/TR-489, Massachusetts Institute of Technology, September 1990.

[18] Rohit Chandra, Kourosh Gharachorloo, Vijayaraghavan Soundararajan, and Anoop Gupta. Performance evaluation of hybrid hardware and software distributed shared memory protocols. In *Proceedings of the 1994 International Conference on Supercomputing*, pages 274–288, July 1994.

[19] Mathews Cherian. A study of backoff barrier synchronization in shared-memory multiprocessors. Technical Report MIT/LCS/TR-452, Massachusetts Institute of Technology, May 1989.

[20] David R. Cheriton, Gert A. Slavenberg, and Patrick D. Boyle. Software-Controlled Caches in the VMP Multiprocessor. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 367–374. IEEE, June 1986.

[21] Trishul M. Chilimbi and James R. Larus. Cachier: A Tool for Automatically Inserting CICO Annotations. In *Proceedings of the 23nd International Conference on Parallel Processing*. IEEE, August 1994.

[22] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. *IEEE Transactions on Computers*, C-37(8), August 1988.

[23] Alan Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, and Willy Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 106–117. IEEE, April 1994.

[24] Alan Cox and Robert Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 32–44, December 1989.

[25] Alan L. Cox and Robert J. Fowler. Adaptive Cache Coherence for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual Symposium on Computer Architecture*, pages 98–108. IEEE, May 1993.

[26] CS9239 M/PAX Multi-Processor Platform Technical Overview, 1990. CHIPS and Technologies, Inc.

[27] Peter J. Denning. Working Sets Past and Present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[28] Charles M. Flaig. VLSI Mesh Routing Systems. Technical Report 5241:TR:87, California Institute of Technology, May 1987.

[29] Matthew J. Frank and Mary K. Vernon. A Hybrid Shared Memory/Message Passing Parallel Machine. In *Proceedings of the 22nd International Conference on Parallel Processing*. IEEE, August 1993.

[30] Aaron J. Goldberg and John L. Hennessy. Mtool: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):28–40, January 1993.

[31] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th International Conference on Computer Architecture*, pages 254–263. IEEE, May 1991.

[32] W. Hackbusch, editor. *Multigrid Methods and Applications*. Springer-Verlag, Berlin, 1985.

[33] Erik Hagersten, Anders Landin, and Seif Haridi. DDM – A Cache-Only Memory Architecture. *IEEE Computer*, 25(9):44–54, September 1992.

[34] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 262–273. ACM, October 1992.

[35] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., New York, New York, 1991.

[36] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi. Distributed-Directory Scheme: Scalable Coherent Interface. *IEEE Computer*, 23(6):74–77, June 1990.

[37] Truman Joe and John L. Hennessy. Evaluating the Memory Overhead Required for COMA Architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 82–93. IEEE, April 1994.

[38] Kirk Johnson. Semi-C Reference Manual. ALEWIFE Memo No. 20, Laboratory for Computer Science, Massachusetts Institute of Technology, August 1991.

[39] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373. IEEE, June 1990.

[40] Herbert I. Kavet. *Computers*. Ivory Tower Publishing Company, Inc., Watertown, Massachusetts, 1992.

[41] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

[42] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory; Early Experience. In *Proceedings of the Practice and Principles of Parallel Programming*, pages 54–63. ACM, May 1993.

[43] David A. Kranz. ORBIT: An Optimizing Compiler for Scheme. Technical Report YALEU/DCS/RR-632, Yale University, February 1988.

[44] David A. Kranz, Robert Halstead, and Eric Mohr. Mul-T: A High-Performance Parallel Lisp. In *Proceedings of the Symposium on Programming Languages Design and Implementation*. ACM SIGPLAN, June 1989.

[45] KSR-1 Technical Summary, 1992. Kendall Square Research, Waltham, Massachusetts.

[46] John Kubiatowicz. User's Manual for the A-1000 Communications and Memory Management Unit. ALEWIFE Memo No. 19, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1991.

[47] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the International Supercomputing Conference*. IEEE, July 1993.

[48] John Kubiatowicz, David Chaiken, and Anant Agarwal. Closing the Window of Vulnerability in Multiphase Memory Transactions. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274–284. ACM, October 1992.

[49] John D. Kubiatowicz. Closing the Window of Vulnerability in Multiphase Memory Transactions: The Alewife Transaction Store. Technical Report MIT/LCS/TR-594, Massachusetts Institute of Technology, November 1994.

[50] Kiyoshi Kurihara, David Chaiken, and Anant Agarwal. Latency Tolerance through Multithreading in Large-Scale Multiprocessors. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*. IPS Press, April 1991.

[51] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313. IEEE, April 1994.

[52] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9), September 1979.

[53] Richard P. LaRowe Jr., Carla Schlatter Ellis, and Laurence S. Kaplan. The Robustness of NUMA Memory Management. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 137–151, October 1991.

[54] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159. IEEE, June 1990.

[55] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[56] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–60, January 1993.

[57] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[58] David J. Lilja and Pen-Chung Yew. Improving Memory Utilization in Cache Coherence Directories. *IEEE Transactions on Parallel and Distributed Systems*, 4(10):1130–1146, October 1993.

[59] Beng-Hong Lim. Parallel C Functions for the Alewife System. ALEWIFE Memo No. 37, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1993.

[60] Rishiyur S. Nikhil, Gregory M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 156–167. IEEE, May 1992.

[61] Brian W. O'Krafka and A. Richard Newton. An Empirical Evaluation of Two Memory-Efficient Directory Methods. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 138–147. IEEE, June 1990.

[62] Per Stenström and Truman Joe and Anoop Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 80–91. IEEE, May 1992.

[63] Per Stenström, Mats Brorsson, and Lars Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual Symposium on Computer Architecture*. IEEE, May 1993.

[64] John D. Piscitello. A Software Cache Coherence Protocol for Alewife. Master's thesis, MIT, Department of Electrical Engineering and Computer Science, May 1993.

[65] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39. ACM, October 1987.

[66] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336. IEEE, April 1994.

[67] Charles L. Seitz. Concurrent VLSI Architectures. *IEEE Transactions on Computers*, C-33(12):1247–1265, December 1984.

[68] Richard Simoni. Cache Coherence Directories for Scalable Multiprocessors. Technical Report CSL-TR-92-550, Stanford University, October 1992.

[69] Richard Simoni and Mark Horowitz. Dynamic Pointer Allocation for Scalable Cache Coherence Directories. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*. IPS Press, April 1991.

[70] Richard Simoni and Mark Horowitz. Modeling the Performance of Limited Pointers Directories for Cache Coherence. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 309–318. IEEE, May 1991.

[71] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-92-526, Stanford University, June 1992.

[72] SPARC Architecture Manual, 1988. SUN Microsystems, Mountain View, California.

[73] C. K. Tang. Cache Design in the Tightly Coupled Multiprocessor System. In *AFIPS Conference Proceedings, National Computer Conference*, pages 749–753, June 1976.

[74] The Connection Machine CM-5 Technical Summary, 1992. Thinking Machines Corporation, Cambridge, Massachusetts.

[75] Jory Tsai and Anant Agarwal. Analyzing Multiprocessor Cache Behavior Through Data Reference Modeling. In *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, pages 236–247. ACM SIGMETRICS, May 1993.

[76] Thorsten von Eicken, David Culler, Seth Goldstein, and Klaus Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*. IEEE, May 1992.

[77] Wolf-Dietrich Weber. Scalable Directories for Cache-Coherent Shared-Memory Multiprocessors. Technical report, Stanford University, January 1993. CSL-TR-93-557.

[78] Wolf-Dietrich Weber and Anoop Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256. ACM, April 1989.

[79] Andrew W. Wilson Jr. and Richard P. LaRowe Jr. Hiding shared memory reference latency on the galatica net distributed shared memory architecture. *Journal of Parallel and Distributed Computing*, 15(4):351–367, 1992.

[80] David A. Wood, October 1993. Private Communication.

[81] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for Cooperative Shared Memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–167. IEEE, May 1993.