IDENTIFICATION AND OPTIMIZATION OF SHARING PATTERNS FOR SCALABLE SHARED-MEMORY MULTIPROCESSORS

by

STEFANOS KAXIRAS

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

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

1998

Abstract

Distributed shared-memory architectures typically employ a directory-based protocol to maintain cache coherence. Identifying sharing patterns in parallel programs and applying specialized optimizations can increase cache-coherence protocol efficiency and yield performance improvements. In this thesis, I propose and study both optimizations to sharing patterns and techniques to identify sharing patterns.

The main thrust of the thesis is GLOW, a comprehensive optimization for wide sharing—a sharing pattern that is a serious obstacle to scalability to large numbers of processors. I present GLOW in the form of extensions to the SCI ANSI/IEEE standard. GLOW is implemented in special network switches and incorporates characteristics that are not found together in previous proposals: scalable writes *and* scalable reads, network locality (by exploiting the abundance of widely-shared data to satisfy requests locally), simplicity, transparency to the base protocol, and network topology independence. With simulation, I show that for programs with wide sharing, GLOW can be more than twice as fast as SCI in large systems.

Furthermore, I examine techniques to identify sharing patterns. I propose a novel approach based on maintaining the history of load and store instructions in relation to coherence events (e.g., cache-misses) and predicting their future behavior. This *instruction-based* approach, differs from previously proposed adaptive protocols (*address-based techniques*) which maintain data-access history and predict future accesses. Instruction-based prediction can offer accurate detection of sharing patterns using few resources in the form of small predictors per node.

I examine instruction-based prediction for three sharing patterns: wide sharing, migratory sharing, and producer-consumer sharing. For wide sharing, instruction-based prediction compares favorably to static identification and to two novel dynamic address-based identification schemes. For migratory sharing, instruction-based prediction matches or exceeds the performance of previously proposed adaptive protocols (for seven benchmark programs). For producer-consumer sharing, instruction-based prediction is coupled with *speculative pre-send*—a novel optimization based on speculative execution. The low mis-speculation rates of this scheme show promise for performance improvements.

Acknowledgments

I dedicate this thesis to the memory of my father Vasileios, my mother Ekaterini and my beloved partner Angeliki. The love of my mother, my brother Efthimios, my sister Eleni and her family, as well as Angeliki's family kept us going through graduate school.

I would like to thank my advisor and mentor, professor James R. Goodman for supporting me. I am grateful to professor David A. Wood for his insight, professor Gurindar S. Sohi for his support, and professor Mark D. Hill for his help. Their sound advice helped me to complete my Ph.D.

Also, I would like to thank all my friends and colleagues at the University of Wisconsin, especially Scott Breach, Doug Burger, Babak Falsafi, Alain Kägi, Andreas Moshovos, Shubu Mukherjee, Dionisios Pnevmaticatos, Yannis Schoinas, and T.N. Vijaykumar for the many technical conversations, assistance, and advice. My thanks to Todd Bezenek who has proofread an entire version of this thesis. All remaining errors are solely my responsibility.

This work was supported in part by NSF Grants CCR-9207971 and CCR-9509589, funding from the Apple Computer Advanced Technology Group, an unrestricted grant from the Intel Research Council, and equipment donations from Sun Microsystems. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618, with matching funding from the University of Wisconsin Graduate School. This work was also partially supported by the National Center for Supercomputing Applications and utilized the Thinking Machines CM-5 at NCSA, University of Illinois at Urbana-Champaign.

Table of Contents

List of Figures

List of Tables

1 Introduction

The shared-memory multiprocessing paradigm is well established for small-scale parallel machines such as bus-based multiprocessors. The uniform global address space of the shared-memory model, through which all data communication is performed, leads to a clean and elegant programming model that is preferable in many situations over the message-passing programming model.

Larger shared-memory machines built of smaller bus-based symmetric multiprocessors (SMP nodes) are also advancing in the marketplace. Because buses do not scale beyond a small number of processors (usually up to 16), larger shared-memory machines are built by physically distributing the memory among a number of nodes connected with a network. To drive development costs down and shorten time-to-market, distributed shared-memory systems leverage existing commodity parts such as processors, main-boards, and networks designed to support fine-grain communication [41].

Typically, in such architectures, a directory-based coherence protocol is employed to maintain *cache coherence* (*CC*) among the SMP nodes. Contemporary examples of such architectures include the HP/Convex Exemplar [26] and Sequent STiNG [68] that use Scalable Coherent Interface (SCI) networks and SCI cache coherence [41], and the SGI Origin 2000 [62] that uses a directory-based cache coherence protocol originating in Stanford's DASH multiprocessor [65].

The widespread use of hardware shared-memory systems (especially SMP's) presents an opportunity to promote shared-memory parallel programming to a much larger audience of programmers than ever before. However, for widespread use of shared-memory, we need to address performance issues that arise in distributed shared-memory machines. Weber and Gupta's research [100] has revealed the existence of various sharing patterns in shared-memory programs. For example, sharing patterns such as migratory sharing and producer-consumer sharing have been identified and optimizations specific to such patterns have been proposed.

In this thesis I examine optimizations for hardware distributed shared-memory architectures and for various sharing patterns. The main thrust of the thesis is on wide sharing, but I also examine migratory sharing, and producer-consumer sharing. I propose a new optimization for wide sharing called GLOW and a new optimization based on speculative execution for producer-consumer sharing. For migratory sharing I study an optimization inspired by previous work [28,93].

Because these optimizations are specific to a sharing pattern they should not be applied indiscriminately for all accesses. Doing so may result in performance loss. Therefore, we need to identify the data affected by a sharing pattern, or the accesses that belong to a sharing pattern, and selectively apply the corresponding optimization.

Identification of the sharing pattern is orthogonal to the optimization for the sharing pattern. In this thesis I examine the identification of sharing patterns in two dimensions. The first dimension corresponds to the time the identification takes place. Identification can take place at compile-time, or **statically**. In this case the programmer or the compiler identifies the sharing pattern. Alternatively, identification can take place at run-time, or **dynamically**. In this case, run-time mechanisms identify a sharing pattern and invoke the corresponding optimization.

The second dimension corresponds to the method used for the identification. Either the data that are affected by a sharing pattern can be identified, henceforth called **address-based identification**, or the memory instructions that are involved in a sharing pattern can be identified, henceforth called **instruction-based identification**. Dynamic instruction-based identification, using prediction is one of the major contributions of this thesis.

In the rest of this chapter I introduce the optimizations to the three sharing patterns (Section 1.1) and the corresponding dynamic/static and address-based/instruction-based identification techniques (Section 1.2).

1.1 Sharing patterns and optimizations

Several classes of sharing patterns in shared-memory applications have been identified (migratory, read-only, frequently-written sharing, etc. [22,100]). In this thesis I examine wide sharing, and to a lesser degree migratory sharing and producer-consumer sharing. The characteristics of these sharing patterns and the optimizations I propose are introduced below.

1.1.1 Wide sharing

Widely-shared data that are read simultaneously by many—usually all—processors is a distinct sharing pattern that imposes increasingly significant overhead as systems increase in size. When all processors read widely-shared data there is much contention in the home node for servicing the requests as well as in the network around the home node which becomes a *hot spot* [78]. Similarly, when the widely-shared data are written, there is a large number of invalidations (or updates) to be sent all over the system (*i.e.*, non-locally). For systems with no provision for efficient broadcast or multicasts, these invalidations consume much network bandwidth, in a wasteful manner.

In this thesis, I argue that in fact, widely-shared data inherent in some parallel algorithms are a more serious problem than previously recognized, and that furthermore, it is possible to provide support that gives an advantage to widely-shared data. The idea of read-combining [36,95] evolved because of the concern for network contention for widely-shared data. Read-combining is highly dynamic, and reduces traffic in the network by recognizing that simultaneous requests can be merged. The probability of occurrence of simultaneous requests only becomes a factor when serious network contention extends the latency of individual requests, and in general, the best that combining can hope to achieve is a reduction in latency of access to widely-shared data to the latency that would be experienced in an unloaded network.

Alternatively, it is possible to access widely-shared data even faster than non-widely-shared data. The presence of redundant copies of a datum in multiple caches throughout the system offers this possibility: if data are widely shared there has to be network (*geographical*) locality—if some data that are needed by a node are widely shared, it is likely that a cached copy of the data is closer than the original data in the home node. This situation has some resemblance to cache-only memory (COMA) machines [37], where data are quickly accessed if they reside in a cache close to the requester. If an architecture can exploit this fact to improve accessibility of widely-shared data, programmers would find that the best algorithms make extensive use of widely-shared data rather than eschewing it. Thus, the potential for systems that provide high-quality support for widely-shared data may be much larger than would be indicated by a sample of current shared-memory programs, which generally avoid such data wherever possible.

Previous proposals for a wide sharing optimization, such as the STEM Kiloprocessor Extensions to SCI [44] and others [17,69,76] have largely ignored network locality in the network or they are closely tied to a network that is physically hierarchical. In this thesis, I propose a comprehensive solution to optimize wide sharing that borrows from the best attributes of previous proposals. The solution is given in the form of extensions (called GLOW extensions) to other directory-based cache-coherence protocols. The GLOW extensions offer scalable reads and scalable writes to widely-shared data. Scalable reads are achieved by caching directory information in the network, a technique inspired by the request combining proposed for CHoPP [95] and subsequently for the NYU Ultracomputer [36]. Because the directory information is more long-lived, this technique can be effective even when multiple requests are not generated simultaneously. Scalable writes are achieved by exploiting the topology to invalidate or update sharing nodes in logarithmic time. The three main goals that guided the design of GLOW were:

- TO CREATE SHARING TREES THAT MAP WELL ONTO ARBITRARY TOPOLOGIES, thus achieving low-latency protocol messages. To achieve this goal, GLOW builds *logical* sharing trees that follow the request patterns in the network. Multilevel inclusion Appendix 13 is not enforced thus, GLOW is topology independent and effectively avoids deadlocks in arbitrary topologies.
- TO PROVIDE SCALABLE READS by exploiting request combining, independent of the timing of requests.
- TO PROVIDE SCALABLE WRITES by using the tree structure to invalidate or update sharing nodes in parallel.

The GLOW cache-coherence protocol extensions are specifically designed to handle accesses to widely-shared data. GLOW should not be applied to non widely-shared data because in this case the overhead of building a sharing tree may outweigh the benefit.

Through detail execution-driven simulation I study a GLOW implementation on top the SCI IEEE/ANSI standard cache-coherence protocol for six programs that do have widely-shared data and for various system configurations. For the systems and the workload I have studied, I found:

- GLOW provides scalable reads and writes lacking in the base coherence protocol (SCI).

- Widely-shared data start to become detrimental to performance for large systems (for 16 nodes and up).
- In systems with GLOW, the six programs I have studied scale better even with a fixed input size. Increasing the input size increases the performance difference more than linearly among systems with support for wide sharing (GLOW) and systems without.
- Relaxed memory models improve performance by hiding write latencies but cannot replace GLOW which directly reduces both read and write latencies. Thus, GLOW on a strict memory model such as sequential consistency is significantly better than SCI with a relaxed memory model.

I have also studied two sharing-tree replacement algorithms for GLOW and to a limited extent update protocols in GLOW.

1.1.2 Migratory sharing

Migratory sharing refers to data that are read, modified, and written by a single processor at time. Typically, such data are accessed within critical sections. In previous work (by Cox and Fowler [28], and by Stenström, Brorsson, and Sandberg [93]) the optimization is to collapse the coherent read (that first accesses the migratory data) and the coherent write (that updates the migratory data) in a single transaction. This optimization is performed by the home node directory, which is responsible to dynamically detect migratory sharing. For migratory data the directory simply returns a *writable* copy to a read request (after invalidating any older copies). The optimization used in this work is initiated by a node that is accessing migratory data by simply converting the first coherent read of the migratory data to a coherent write. This first appeared in the form of a specialized coherence protocol for migratory sharing in the Munin software Distributed Shared Memory (DSM) system [22].

1.1.3 Producer-consumer sharing

Producer-consumer sharing is one of the more difficult sharing patterns to optimize because conceptually, all sharing can be classified under this pattern. However, optimizations have been proposed for *stable* producer-consumer sharing. The goal of such optimizations is to transfer newly created values from the producer to the consumer as soon as possible. The optimizations are various update protocols and variations of Data Forwarding [59,3,62]. Update protocols have well known problems such as superfluous update messages and implementation constraints with sequential consistency [61,5]. Data Forwarding totally relies on the programmer/compiler for correctness.

In this thesis, I propose a novel optimization for producer-consumer sharing that is both transparent and independent of the memory model. The idea is to *pre-send* data speculatively to potential consumers. The consumers can use the data speculatively, provided that they eventually verify the correctness of the data through the cache-coherence protocol. In this way, processors can race ahead speculatively, while coherence enforcement follows behind.

This optimization trades bandwidth for latency. Because the simulation environment I use in this thesis does not allow speculative execution, I only report mis-speculation statistics (analogous to branch prediction studies). The low mis-speculation rates for four programs indicate that there is potential for performance gains but further research is needed to quantify the performance of this optimization.

1.2 Identification of sharing patterns

Along with the sharing pattern optimizations (primarily the GLOW extensions for wide sharing but also the speculative pre-send for producer-consumer sharing) the major contribution of this thesis is a thorough examination of techniques to identify sharing patterns and selectively apply optimizations. I divide the techniques into dynamic and static and into address-based and instruction-based. Table 1 shows all possible cases.

		TIME	
		Compile-time (Static) (user/compiler)	Run-time (Dynamic) (run-time mechanisms)
TYPE	Address	Static address-based	Dynamic address-based
	Instruction	Static instruction-based	Dynamic instruction-based

Table 1. Techniques to identify a sharing pattern.

***Static vs. Dynamic: To change the interface or not?*—**The choice between static and dynamic depends on: (i) whether it is possible to change the software/hardware interface and (ii) implementation costs. Static methods require an interface to pass information from the program to the hardware. Various reasons may render such an interface infeasible. For example, compatibility reasons or commodity hardware may pose problems. Additionally, the static methods require the involvement of the user (programmer) or, at the very least, the compiler. Static methods may also be undesirable since they subtract from the elegance of the shared-memory programming paradigm. On the other hand, dynamic methods require more expensive mechanisms at run-time to detect sharing patterns. The advantage of dynamic approaches is that they are transparent to the architecture and to the user/compiler. However, they require custom hardware and incur hardware costs.

Address vs. Instruction: Examine what happens to the data or what the program is trying to do?—The choice between address-based or instruction-based approaches depends on many factors. Static address-based and static instruction-based methods are comparable: both require involvement of the user and/or compiler, both have the same implementation problems (software/hardware interface) and their performance depends on the ability to correctly define a sharing pattern using addresses or instructions. Dynamic address-based and dynamic instruction-based techniques differ considerably. In this thesis, I will show that the dynamic instruction-based techniques require few hardware resources and can outperform the more expensive dynamic address-based techniques.

1.2.1 Static techniques

Static address-based techniques have been proposed for all three sharing patterns (Table 2). I discuss previous work and describe static instruction-based techniques to distinguish sharing patterns. However, I evaluate only the dynamic counterparts of these techniques (instruction-based prediction).

1.2.2 Dynamic address-based techniques

Dynamic address-based detection has been previously proposed for migratory sharing and producer-consumer sharing (Table 2). I propose and evaluate two dynamic address-based identification techniques for wide sharing:

- **AGENT DETECTION**: This scheme is inspired by request combining [36] and is based on observing requests in the network switch nodes that implement the GLOW extensions (also called GLOW agents). The switch nodes (agents) observe requests and intercept the ones that appear to repeat frequently. This dynamic scheme tracks closely the performance of the static address-based GLOW while it outperforms ordinary request combining for the programs and the systems I have examined.

- **DIRECTORY DETECTION**: In this scheme, the memory directory discovers widely-shared data by counting the number of reads between writes. Information about widely-shared data is distributed to the nodes which subsequently use the GLOW extensions to access them. In the evaluation I show that this scheme works well when the nature of the widely-shared data is persistent over time.

1.2.3 Dynamic Instruction-based techniques: Instruction-based prediction

Prediction is used for the three dynamic instruction-based techniques to identify sharing patterns. The main idea of *instruction-based prediction* is to examine—at run-time—the behavior of load and store instructions in relation to coherence events. In every node, the past behavior of its load and store instructions is stored in a *small* predictor table. Whenever dynamic instances of load and store instructions generate coherence events (such as misses, or write-faults on read-only cache blocks) we consult the predictors for optimization hints. This means that the optimizations affect the behavior of the processor toward the cache-coherence protocol (*e.g.*, on a load-miss the processor may ask for permission to write) in contrast to address-based prediction optimizations that affect the behavior of the cache-coherence protocol toward the processor (*e.g.*, the cache-coherence protocol—on its own—may decide to return a writable block to a processor that asks for a read-only block).

Instruction-based prediction is not new in the uniprocessor world: it is established research that already appears in commercial processors. Branch prediction is the pioneering instruction-based prediction method studied extensively by many researchers including Smith [92] and Yeh and Patt [105]. Abraham et al. showed that very few loads are responsible for most cache misses [4] and subsequently Tyson et al. proposed instruction-based prediction to selectively bypass the cache for such loads [98]. Gonzalez, Aliagas, and Valero used instruction-based prediction to steer data on caches optimized differently for spatial and temporal locality [33]. Moshovos, Breach, Vijaykumar, and Sohi introduced memory-dependence prediction [72]. They proposed dependence predictors accessed using the address of memory instructions. Subsequently, Moshovos and Sohi proposed memory optimizations based on dependence predictions [73]. Tyson and Austin also proposed similar memory optimizations [97]. This thesis is the first research effort to bring these techniques in the world of parallel shared-memory architectures. Although I believe that such techniques can be generally applicable (from bus-based cache coherence to software-based coherence), in this thesis I present the techniques for hardware directory-based coherence.

The benefits of instruction-based prediction/optimization can be significant:

1. Concise representation of history: Code is much smaller than datasets—static loads and stores can be only so many while the dataset can be arbitrarily large—and keeping track of the history of load and store instructions rather than memory blocks and/or cache blocks consumes far fewer resources.
2. A single technique for many optimizations: The technique I propose can be used to optimize several sharing patterns *using a common, small predictor structure per node*. In con-

trast, each address-based prediction scheme is tailored for a specific sharing pattern. Each may require its own states in the coherence protocol and its own storage (usually on a per block basis) for history information. Although Mukherjee and Hill showed how to generalize address-based prediction [75], the issue of excessive storage for history information remains.

However, there are important issues involved with instruction-based prediction in shared-memory:

1. Implementation issues: Instruction-based prediction calls for a tight integration of the processor core and the coherence mechanisms because information from both places is needed in the predictor.
2. Performance issues: Address-based prediction inherently keeps large amounts of history information and in some situations this might be preferable to the “concise” information we can gather regarding load and store instructions.

The instruction-based prediction techniques I examine for the three sharing patterns are:

- **Wide sharing: Predict whether a load will access widely-shared data.** I propose and evaluate two schemes to predict whether a load instruction will access widely-shared data. These schemes consistently outperform both the AGENT DETECTION and DIRECTORY DETECTION dynamic address-based schemes for the programs studied (Chapter 4).
- **Migratory sharing: Predict whether a load-miss will be followed by a store-write-fault.** This prediction can lead to optimization of migratory sharing patterns. The reasoning is that migratory sharing patterns often generate load-misses closely followed by

store-write-faults. For a set of seven programs studied for this sharing pattern, instruction-based prediction requires far less resources than the (dynamic address-based) adaptive migratory protocols while performing comparably (Chapter 5).

- **Producer-consumer sharing: Predict which node is going to consume a value generated by a store.** This scheme examines store instructions that generate write-faults and keeps track of the potential readers of the newly written cache-blocks. The goal is to predict upon seeing a store-write-fault, whether there is a stable producer-consumer relationship and to predict the identity of the consumer(s). This scheme is inspired by uniprocessor dependece prediction work Appendix 72. There are three degrees of optimization (from conservative to aggressive): (i) Using a simple predictor we can initiate pair-wise sharing with direct cache-to-cache transfers without involving the home directory, (ii) switch to an update protocol—though this is not transparent in the case of a sequential consistent memory system—, and (iii) using enhanced predictors we can *speculatively pre-send* the newly created values to the predicted consumers. In this thesis I examine the first and third cases (Chapter 6).

1.3 Contributions

Here, I summarize the contributions of this thesis. Table 2 illustrates how previous work maps into the optimization space. Table 3 illustrates the contributions of this thesis. The union of these two tables covers all possible cases.

WIDE SHARING	**Static**	**Dynamic**
Address	EC [17] PROXIES [96][15]	STEM [44] Combining [36]
Instruction	—	—

MIGRATORY SHARING	**Static**	**Dynamic**
Address	Munin [22]	Adaptive protocols for migratory data [28][93]
Instruction	—	—

PRODUCER-CONSUMER SHARING	**Static**	**Dynamic**
Address	Munin [22] Update protocols	Competitive Update
Instruction	Data Forwarding [59][3][62]	—

Table 2. Previously proposed identification techniques for three sharing patterns (shaded cells).

In brief, I propose:

- For wide sharing: the GLOW optimization; a static address-based, a static instruction-based, two dynamic address-based, and two dynamic instruction-based methods to selectively apply the optimization.

- For migratory sharing: a dynamic instruction-based method to apply an optimization (which is inspired by previous work). Additionally, I discuss a static instruction-based method.
- For producer-consumer sharing: a dynamic instruction-based method to apply a novel optimization based on speculative execution. I discuss three variations of this method.

WIDE SHARING	**Static**	**Dynamic**
Address	Static GLOW-address	1) Agent Detection 2) Directory Detection
Instruction	Static GLOW-Instruction	Instruction-based prediction: 1) Latency 2) Directory feed-back.

MIGRATORY SHARING	**Static**	**Dynamic**
Address	—	—
Instruction	Critical Section	Instruction-based Prediction

PRODUCER-CONSUMER SHARING	**Static**	**Dynamic**
Address	—	—
Instruction	—	Instruction-based prediction with Speculative pre-send: 1) Last prediction 2) Intersection prediction 3) Two-level adaptive

Table 3. Identification techniques proposed in this thesis.

1.4 Thesis roadmap

Following the Introduction, in Chapter 2 I present the evaluation methodology used throughout this thesis. In Chapter 3, I present the GLOW extensions to optimize wide sharing. GLOW is independent of how wide sharing is defined statically or detected dynamically. Thus, in Chapter 4 I describe a set of techniques (static, dynamic, address-based and instruction-based) to distinguish wide sharing and selectively apply the GLOW optimizations. Chapter 5 describes the dynamic instruction-based prediction for migratory sharing. Chapter 6 presents instruction-based prediction for producer-consumer sharing and the optimization based on speculative execution. Finally, in Chapter 7, I summarize this thesis and conclude.

Nomenclature: In this thesis I use the following naming conventions: *load*, *store* are the actual instructions; *read*, *write* are the cache coherence actions resulting from loads and stores. A cache block that is not *Invalid* can be either *ReadOnly* (*RO*) or *ReadWrite* (*RW*). A load or store can experience a cache *miss* which results in a coherent read or write; furthermore a store can experience a *write fault* on a RO cache block which results in a coherent write.

2 Evaluation Methodology

This section presents the experimental methodology used for all evaluations in this thesis. Simulation is applied to study the effects of optimizations for wide sharing, migratory sharing and producer-consumer sharing. In Section 2.1, I discuss the Wisconsin Wind Tunnel (WWT) [79] simulation environment. To study optimizations for wide sharing, I enhanced WWT with detailed network simulation described in Section 2.2. Finally, in Section 2.3, I describe the benchmark programs used throughout this thesis.

2.1 Wisconsin Wind Tunnel

A detailed study of the optimizations proposed in this thesis requires execution driven simulation because of the complex interactions among the processors, their caches, the memory modules, and the network. The Wisconsin Wind Tunnel [79] is a well-established tool for evaluating large-scale parallel systems through the use of massive, detailed simulation. WWT simulates shared-memory systems on a message-passing host system (a Thinking Machines Corp. CM-5 [40]). It executes target parallel programs at hardware speeds (without intervention) for the common case when there is a hit in the simulated coherent cache. In the case of a miss, the simulator takes control and executes the appropriate actions defined by the simulated protocol. WWT keeps track of virtual time in processor cycles. The direct execution nature of WWT poses certain limitations: only instructions that generate coherence events are observable; the coherent caches are blocking; the cache-block size must be a power-of-two multiple of the hardware cache-block size (32 bytes) of the host system (CM-5); and finally speculative execution is not supported.

Kägi, Aboulenein, Burger, and Goodman have simulated SCI extensively under WWT [47] and all the optimizations I study (for wide sharing, for migratory sharing, and for producer-consumer sharing) have been applied to this simulation environment. I simulated systems that resemble SCI systems made of readily available components such as SCI rings and workstation nodes.

For the evaluations in Chapters 3 and 4—which require detailed network simulation—I have simulated k-ary n-cube systems (such as the Wisconsin Multicube proposed by Goodman and

Woest [34]) from 16 to 128 nodes in two and three dimensions. For practical reasons, in the evaluations of Chapters 5 and of 6—where accurate network simulation is not critical—I simulated a constant latency (100 processor cycles) network.

The nodes comprise a processor, an SCI cache, memory, a memory directory, a GLOW agent that implements the GLOW extensions, and a number of ring interfaces. The processors run at 500MHz and execute one instruction per cycle in the case of a hit in their cache. Each processor is serviced by a 64K 4-way set-associative cache with a cache-block size of either 32 or 64 bytes (see the discussion of cache parameters in Section 2.3). Processor, memory, and network interface (including GLOW agents) communicate through a 166 MHz 64-bit *split-transaction* bus (1.266 GB/s peak bandwidth). The bus transfers 64 bits per bus cycle (every 3 processor cycles). Memory and directory accesses require 4 bus cycles (12 processor cycles) before the start of the data transfer while cache access requires 1 bus cycle (3 processor cycles).

The GLOW agents are enhanced switches that connect the ring interfaces and pass messages from one to the other. To implement the GLOW optimizations for widely-shared data each GLOW agent is equipped with a 1024-entry directory cache and 64K of data storage. To minimize conflicts, the agent's directory is organized as a 4-way set-associative cache. Although I assume uniprocessor nodes, GLOW applies equally well to symmetrical multiprocessor (SMP) nodes. In this case, the GLOW agent resides in the network interface of the SMP node and is responsible for servicing the set of processors inside the node.

2.2 Network

For the evaluation of the wide-sharing optimizations it is imperative to accurately simulate the network of a parallel system. This is because the effect of widely-shared data on the network performance is critical. Although the GLOW extensions described in the next chapter can be applied to many different topologies, I use the k-ary n-cube networks of SCI rings in 2 and 3 dimensions. Because of its scalability, this topology has great appeal in real world systems such as the CRAY T3D [21] and T3E [87], SGI Origin 2000 [62], Convex Exemplar 2000 [1].

The rings use a 500 MHz clock; 16 bits of data can be transferred every clock cycle through every link, giving a total of 1GB/sec bandwidth. This is equivalent to the actual IEEE 1596 standard which describes a 250 MHz network that sends 16 bits of data at both clock edges.

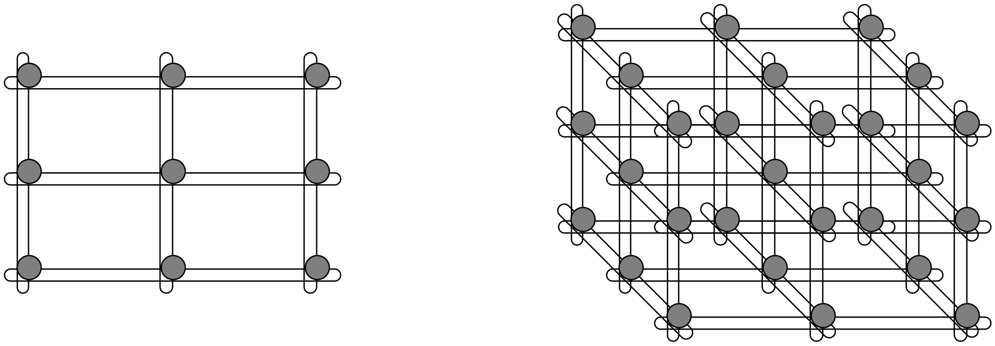


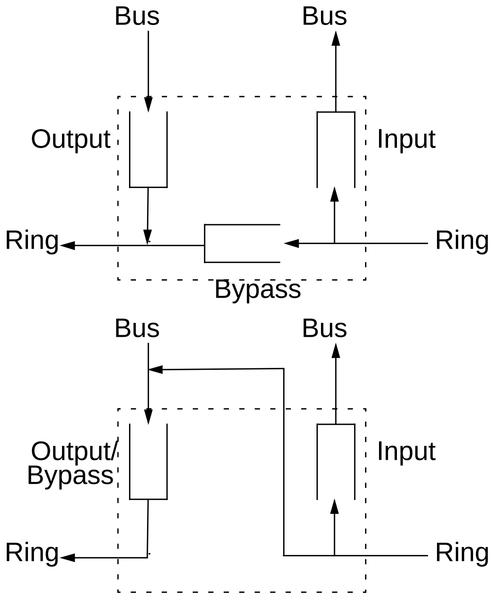
Figure 2.1. 2-dimensional and 3-dimensional networks made of SCI rings.

I simulate contention throughout the network, but messages are never dropped since I assume infinite queues. The model I use deviates from the SCI specification in that I do not simulate SCI's bandwidth allocation algorithms (based on the so-called *"Go-bits"* [41]), nor SCI's scheduling between ring traffic and outgoing traffic from the nodes. As can be seen in Figure

2.2, the SCI interface specifies three classes of queues: the Input queues, the Output queues and the Bypass buffer. The Input and Output queues are actually pairs of queues, each pair comprises a queue for requests and a queue for responses. The Bypass buffer captures the ring traffic while the output link is busy with outgoing traffic from the Output queue. SCI specifies that while the Bypass buffer is not empty, the Output queue cannot be drained. SCI guarantees fairness in bandwidth allocation with Go-bits that throttle the outgoing traffic of nodes. Eventually, the Bypass buffer will be empty and the Output queue can transmit. I do not model the arbitration between the Output queue and the Bypass buffer. Any messages in these queues are interleaved with a total FIFO order. I do not model the Go-bits algorithm for bandwidth allocation. However, since both the Output queue and the Bypass buffer have equal access to the output link, there is no starvation of the Output queue.

Queues are simulated using WWT's servers. A WWT server keeps autonomous virtual time and can generate and accept messages. A server's virtual time is updated on message arrivals and also internally, by advancing its virtual clock by some cycles. The WWT has three default servers per node: the CPU server, the cache server, and the directory server. WWT orders messages in virtual time and delivers them to the appropriate server. In the network model, each queue is a different server with its own virtual time. The queueing in a node is shown in Figure 2.3. The ring interfaces are connected to the processor, cache, memory, and directory through the system bus. However, the GLOW agent and the ring switch are directly connected to the ring interfaces. I simulate all queues with a fixed latency of 10 processor cycles and a transfer latency of 1 processor cycle per 16 bits.

I performed comparisons with a different model of the SCI network developed by Burger



In the standard SCI ring Interface, messages in the Output buffer are not transmitted while the Bypass buffer is not empty. Eventually the Bypass buffer will empty because of the bandwidth allocation algorithms of the SCI ring.

In the simplified model the arbitration between the Bypass and Output buffer is missing, hence a single queue for both.

Figure 2.2. SCI ring interface assumptions.

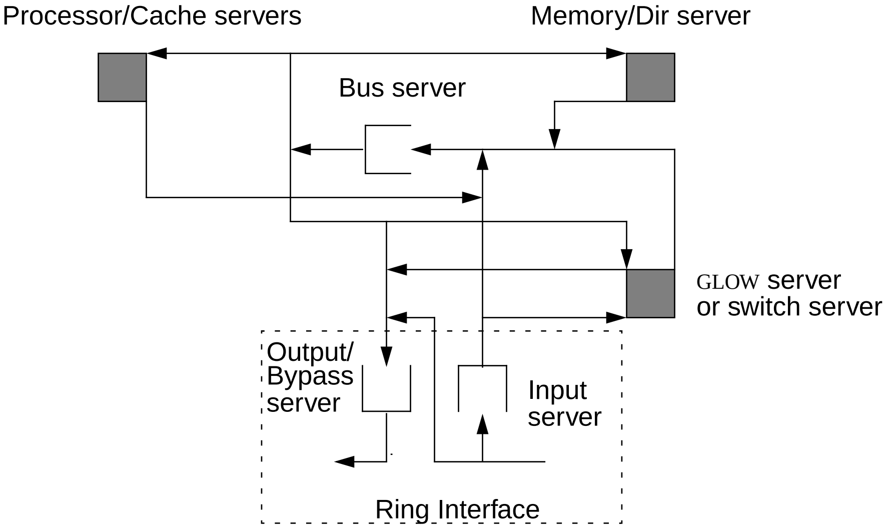


Figure 2.3. Queueing in a node (only one ring interface shown for clarity). The GLOW server is close to the ring interface and does not have to go through the bus to communicate with other nodes. This is because the GLOW server represents the switch that connects multiple ring interfaces together and transfers messages from one to the other.

[19,20]. This model uses finite queues and retransmissions. It takes into account the arbitration between the Output queues and the Bypass buffers, but it does not implement the Go-bit algorithm for bandwidth allocation on the rings (hence it does not guarantee forward progress for the Output queues). The difference of the execution time of two benchmarks (GAUSS and SPARSE benchmarks described in Section 2.3) for 32- and 64-node 2-dimensional systems using the two network models—setting all other parameters to equivalent values—is less than 5% (Burger's simulation model is consistently faster). Since the two network models produce similar read and write latencies, the discrepancy in execution time is due to other simulator differences, such as the modeling of contention in the local bus. Burger's network simulation is centralized in one node: messages are routed to a single node for the purpose of network simulation and then to their final destination. Unfortunately, this is unsuitable for studying wide-sharing optimizations (discussed in the next chapter) which require the availability of the messages in many places along their network path.

2.3 Benchmarks

Here, I describe the ten benchmark programs used in this thesis to study optimizations for three types of sharing: wide sharing, migratory sharing and producer-consumer sharing. The benchmarks were chosen because they exhibit one or more of the sharing patterns. Benchmarks that do not exhibit a sharing pattern are used as *control* benchmarks to study potential negative effects on performance. The ten programs are:

- GAUSS, SPARSE, APSP, TC, CG, BARNES (wide sharing)
- CHOLESKY, MP3D, PTHOR (various degrees of migratory sharing)
- OCEAN (producer-consumer sharing)

Benchmark	**Input Size**	**Cache size/ Block size**	**Wide Sharing**	**Migratory Sharing**	**Prod.- Cons. Sharing**	**References**
CHOLESKY	bsstk14	64K/32		Yes		[91][28][93]
MP3D	10K/10 iter	64K/32		Yes		[91][28][93]
PTHOR	risc	64K/32		Little		[91][28][93]
GAUSS	512x512	64K/64	Yes (dyn.)		Yes	[17][25][50]
SPARSE	512x512	64K/64	Yes (static)			[50]
APSP	256x256	64K/64	Yes (dyn.)			[17][50]
TC	256x256	64K/64	Yes (dyn.)			[50]
BARNES	4K part.	64K/64	Yes (static)			[91][50]
CG	128x128	256K/64	Yes			[14]
OCEAN	130x130	64K/32			Yes	[91][47]

Table 4. Benchmarks used in this work. For each benchmark, the input size, the cache and block size, and prominent sharing patterns are described. The references point to papers where the benchmarks are described or used in the same way as in this work.

Table 4 summarizes the benchmarks. The input size describes the input data set of the programs. Because I use detailed network simulation, multiple WWT servers per node (eight or more compared to WWT's default of three), and because I simulate multiple target nodes per-host node (I simulate target systems of up to 128 nodes on 32 host nodes) it is feasible to simulate only small datasets. Large datasets either exceed the memory capacity of the host system or take an unreasonably long time to simulate. Nevertheless, I simulated a limited number of cases with large datasets to examine scalability. I used 64K caches for all benchmarks. This size is somewhat large for most of the small datasets of the benchmarks. Although capacity misses are not eliminated, the emphasis is on coherence misses. I used 64-byte cache blocks for studying wide-sharing optimizations. SCI defines 64-byte cache blocks and this size generally results in good performance for the base SCI case. I used 32-byte cache blocks for the migratory and producer-consumer optimizations where finer-grain sharing works better.

2.3.1 GAUSS

The GAUSS program solves a linear system of equations using the well-known method of Gaussian elimination. Details of the shared-memory program are discussed by Chandra et al. [25]. A coefficient matrix $N \times N$ is filled with random numbers and then the linear system is solved using a known vector. The work is divided among the processors by distributing the rows block-wise.

GAUSS consists of two phases: In the first phase, for each column, a pivot row is chosen between all processors. Subsequently, this pivot row is read by all processors, multiplied by a factor and subtracted from the rest of the rows. In the second phase (the backward substitution phase), processors compute the vector of the unknown variables, starting from the last row. As each variable is computed, every processor reads it and the appropriate factor is subtracted from every row.

The program has three structures that are widely shared:

1. In every iteration of the first phase a pivot row is read by all processors; in subsequent iterations, elements of previous pivot rows are updated. Potentially, every row of the coefficient matrix can be widely shared since the pivot row can change in every iteration. The pivot row of an iteration is only known at run time.
2. An unknown variable vector is widely shared in the second phase as every variable that is computed is read by most processors.
3. Some variables used for reduction operations are widely shared.

2.3.2 SPARSE

This program solves ***AX=B,*** where ***A*** and ***B*** are matrices (***A*** being a SPARSE matrix) and ***X*** is a vector. The main data structures in the SPARSE program are ***A,*** the *N*×*N* SPARSE matrix, and ***X***, the vector that is widely shared. This vector has *N* elements (the number of columns in the SPARSE matrix) and is not updated. Three more global variables in the program (used in reduction operations) are also widely shared and are frequently updated. In SPARSE, the nature of the data does not change during the execution of the program.

2.3.3 APSP, TC

The All-Pairs Shortest Path program (APSP) solves the problem of finding the shortest paths between all pairs of vertices in a graph. For this program I used a dynamic-programming formulation that is a special case of the Floyd-Warshall [27] algorithm.

In this program, an *N*-vertex graph is represented by an *N*×*N* adjacency matrix. The *(i,j)* element of this matrix represents the weight, or distance, between the *i* and *j* vertices. The Floyd-

Warshall algorithm computes a series of *N* new matrices, each based on the previous matrix. I have parallelized the algorithm using row decomposition and optimized it by grouping reads, computations, and writes to reduce the number of barriers between iterations. The resulting program is reasonably optimized for SCI. The input graphs used for the simulations are 256- and 512-vertex dense graphs (most of the vertices are connected). The adjacency array contains widely-shared data. However, wide sharing depends on the input graph.

Transitive Closure (TC), is another application of the Floyd-Warshall algorithm. In this program an *N*×*N* matrix represents the connectivity of the graph with ones and zeroes. Again the algorithm iterates *N* times and in each step it updates the matrix based on what was computed in the previous step. The inputs are 256- and 512-vertex graphs with a 50% chance of two vertices being connected. APSP and TC differ considerably in their write behavior. Similarly to APSP, TC has also been reasonably optimized for SCI. The main array in TC contains widely-shared data, but again wide sharing depends on the input graph.

2.3.4 CG

In the CG program, the conjugate gradient method is applied to solve a Poisson equation. The code is structured similarly to the NAS CG benchmark [14]. CG is computationally intensive and simulation is limited to small inputs of 64×64 or 128×128. Abandah studied the behavior of CG and found that it exhibits significant wide sharing [2].

2.3.5 BARNES

The BARNES benchmark from the SPLASH suite [91] simulates the interaction of a system of bodies (such as celestial objects) in a three-dimensional space. The main data structure in

BARNES is an *octree* [91] that divides the three-dimensional space into cubes of various volumes. Since eight small cubes fit in a larger cube—of twice the edge length—the octree has a degree of 8. BARNES is an example of a program with little widely-shared data. The top of the octree is widely shared. However, it is difficult to determine statically how many levels of the octree are widely shared.

2.3.6 CHOLESKY

CHOLESKY is a kernel that factors a SPARSE matrix into the product of a lower triagonal matrix and its transpose [91]. A task queue is used to distribute work to the nodes. The handling of the task queue involves migratory sharing. I use CHOLESKY to study optimizations for migratory sharing (Chapter 5) and also as a control benchmark for wide sharing optimizations (Chapter 4).

2.3.7 MP3D

MP3D employs a Monte Carlo method to perform fluid flow simulation (such as simulation of an obstacle placed in a wind-tunnel) [91]. In MP3D, a number of particles move though a wind tunnel colliding with each other and with obstacles in the wind tunnel. A particle array is distributed statically to the nodes. Many processors can update cells of the particle array and cells of a space array which describes the position of the particles. Regardless of whether accesses to the cells of the particle array and the cells of the space array are protected by locks, these two arrays constitute migratory data. Thus, I use MP3D to study migratory-sharing optimizations (Chapter 5).

2.3.8 PTHOR

PTHOR is a parallel, event-driven simulator of logic circuits. For the evaluations, a simple RISC processor executing instructions is simulated. PTHOR has a task queue which involves migratory sharing (Chapter 5).

2.3.9 OCEAN

OCEAN, from the SPLASH 2 benchmark suite [101], is an improved version of the original SPLASH OCEAN [91]. This program simulates currents and water movement in oceans. The SPLASH 2 OCEAN uses a multigrid equation solver. I use OCEAN to study producer-consumer optimizations (Chapter 6) and as a control program for wide sharing (Chapter 4, Section 4.4) and migratory sharing (Chapter 5).

3 Wide Sharing

In this chapter I present the GLOW extensions designed to optimize accesses to widely-shared data. I first motivate the need for an efficient method to handle widely-shared data (Section 3.1) and subsequently I present the main idea of the GLOW framework, which is to exploit the redundancy of widely-shared data to provide scalable reads and scalable writes (Section 3.2). GLOW incorporates various features found scattered in previous research efforts. In Section 3.3, I review related work and briefly discuss features and shortcomings of the various schemes previously proposed. I then proceed with an overview of GLOW (Section 3.4), where I discuss how GLOW integrates and builds upon ideas from previous work to provide comprehensive support for wide sharing. The main part of this chapter (Section 3.5) presents a detailed GLOW implementation on top of SCI, an IEEE standard cache coherence protocol. Other generic implementations are outlined at the end of this chapter in Section 3.7.

3.1 Widely-shared data

Widely shared are the data that are read by many or all the nodes in a system between successive writes. In contrast, migratory data, although read by many or all the nodes in a system, they are only read by a single node between successive writes. Typically, widely-shared data represent a small percentage of the dataset of a program. Weber and Gupta have studied shared-memory parallel programs and observed that the average degree of sharing (the number of nodes that simultaneously share the same data) is low [100]. Their observation, however, does not indicate the serious performance degradation resulting from accessing such data. Even if widely-shared data represent a negligible percentage of the dataset, they can be detrimental to performance because the number of reads (or invalidates) corresponding to such data can be excessively large: widely-shared data imply that a great many processors read them simultaneously.

To make this point clear I use the concept of *read-runs* as a tool to investigate sharing behavior. In Section 4.5 of Chapter 4 I am making extensive use of read-run analysis to explain the performance of various GLOW schemes. Analogous to a *write-run* defined by Eggers and Katz [29], I define a read-run for a data block as a sequence of reads (from any processor) between two writes (from any writer). The size of a read-run is thus related to the number of simultaneous cache copies in the system (if we ignore for a moment multiple reads because of replacements). Figure 3.1 shows the sharing behavior of the GAUSS program (discussed in the previous chapter) running on 128 nodes. The left graph in Figure 3.1 is the read-run histogram for GAUSS. The horizontal axis is the read-run size and the vertical axis is the number of times a read-run appears in the execution of the program. Despite the fact that the number of read-runs of size 128 (corresponding to widely-shared data) is negligible, and despite the modest degree of sharing of 2.75, about one half of all the reads (or alternatively invalidates) in the program correspond to widely-shared data. The abundance of reads (invalidates) corresponding to widely-shared data is evident in the right graph of Figure 3.1, which shows the number of accesses (reads or invalidates) that correspond to read-runs of various sizes (horizontal axis). The explanation for this is that each read-run of size *W* contains *W* reads (or *W* invalidations) and even the very few—but large—read-runs encompass as many reads (or invalidations) as a great number of small read-runs.

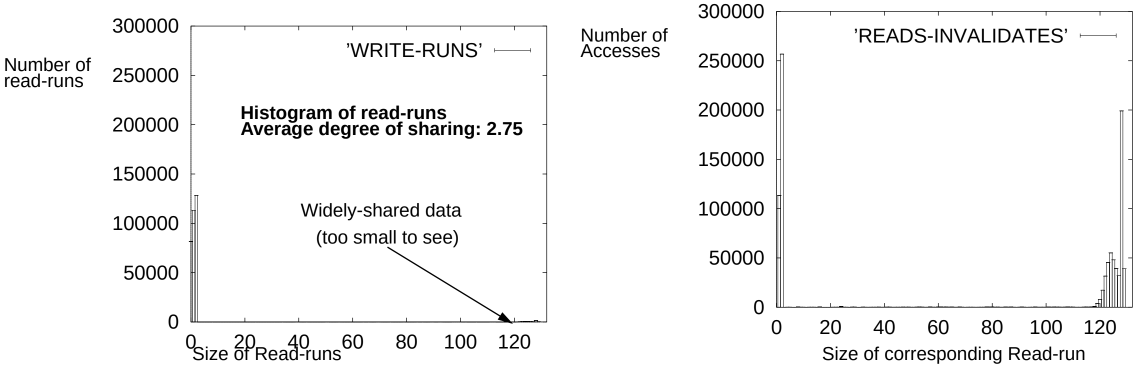


Figure 3.1. Read-run histogram and corresponding Reads/Invalidations for GAUSS **running in 128 nodes.**

Not only are the accesses to widely-shared data numerous but they are also the most expensive in terms of latency, because they create congestion in the network and even worse congestion in the home node directories of the data. When many nodes simultaneously access a single data block, each experiences much greater latency than if all nodes accessed different data blocks in different home node directories.

Providing support for such accesses is essential for scalability of programs. A simple application of Amdahl's law shows why: as the system size grows, widely-shared data accesses eventually will dominate the execution time of programs (since their number is a function of system size and they become increasingly expensive because of congestion). Thus, the speed-up will be limited by widely-shared data, regardless of how much the execution time of the rest of the program is reduced.

3.2 The idea behind GLOW: exploit redundancy of widely-shared data

The GLOW extensions are intended to provide support for widely-shared data. Such data are a serious threat to the performance of *large* systems. Typically, a large distributed shared-memory multiprocessor comprises a number of nodes (16 or more) —each containing one or more processors, caches and a part of the global memory. The nodes are connected with a high-performance network. In real systems (CRAY T3D [21] and T3E [87], SGI Origin 2000 [62], Convex Exemplar 2000 [1]), large diameter networks such as multi-dimensional torri are preferred over large centralized switches (*e.g.*, crossbar-switches), since the latter can be significantly more expensive with current technology. GLOW is intended for systems with non-trivial network topologies (such as those mentioned above), rather than for systems that are based on large crossbar switches where all nodes are equally distant.

For many such network topologies, the bisection bandwidth does not grow linearly with the bandwidth needed for random communication. As the diameter of the network increases with system size, we must optimize communication patterns to make efficient use of the network bandwidth. Increasing communication locality (by making communicating nodes neighbors in the network) is a promising course of action. Not only does locality reduce read and write latencies (since node communication is localized), but it also yields better utilization of the network by allowing multiple communication patterns to coexist in the network without conflict. In general, increased communication locality can be achieved by statically or dynamically mapping the data in the nodes [62] or by dynamically mapping the computation to the nodes [55,56].

Widely-shared data offer a better opportunity for exploiting locality than other data. By their nature widely-shared data are usually present in many nodes in the system. Thus, instead of satisfying read requests for widely-shared data by going to the home node directory (which may be at the far end of the system), we can simply access the closest copy of the data. The GLOW framework provides the mechanisms to do exactly this. Similarly, instead of invalidating or updating all sharing nodes using multiple point-to-point messages that travel all over the network and consume bandwidth, we can follow a hierarchical approach where invalidation/update is initiated locally and expands through the network recursively. The communication pattern of such a hierarchical invalidation/update technique bears close resemblance to a *spanning tree* of the network topology (whose root is the initiator of the invalidation/update). Again the GLOW framework specifies such a recursive invalidation/update technique.

GLOW specifies, in the network domain, similar functionality to hierarchical directories (discussed in the next section). GLOW's main function is to intercept requests for widely-shared data and locate the closest copy to satisfy them. GLOW is not constrained by a physical hierarchy (as for example the hierarchical directories of cache-only memory architectures [37,57]) and it can be implemented in any network topology—even completely random topologies. GLOW implements a hierarchical directory structure that is based on a *logical* hierarchy resulting from the convergence of requests to the home node of the requested data. The home node directory is the root of any GLOW hierarchy. A significant difference between GLOW and previous work with hierarchical directories is that GLOW does not impose *multilevel inclusion* Appendix 13 among its logical hierarchical levels. This is a significant advantage (as it will be discussed in Section 3.5.3) because it allows tremendous flexibility. GLOW is intended *only* for widely-shared data since only in this case the probability of finding a copy nearby significant to justify the effort.

Central to the GLOW framework is the concept of the agent in the network. A GLOW agent handles requests from a small set of nodes. It is responsible for servicing requests for widely-shared data by locating the nearest copy. It is also responsible for invalidating or updating nodes that it has serviced. GLOW agents are discussed in detail in Section 3.4.

3.3 Related work

A major concern for distributed shared-memory systems is their scalability. Scalability in the context of this work refers to speeding-up parallel programs by increasing the number of processors in the system. Ideally, this means making the performance of primitive operations (such as loads, stores, locks and unlocks) independent of the number of processors in the system. In reality, engineering and cost considerations largely prohibit this. For example, one could envision a system with a “free” broadcast capability (*e.g.*, with a fully connected network where each node has a dedicated link to every other node). In such a system, the cost of writing shared data (and consequently invalidating or updating all other copies of the data in the system) is independent of the number of nodes sharing the data and therefore independent of the total number of nodes in the system. However, such a system would be impractical. The next best thing is to make the cost of operations (at worst case) a logarithmic function of the total number of nodes in the system. This was the route followed by many researchers who proposed *sharing tree structures* and *hierarchical directories* for scalable writes and *request combining* for scalable reads. In the following sections, I describe these three approaches and specific examples of each.

3.3.1 Sharing trees

A sharing tree is a directory structure that keeps track of the location of copies of data blocks throughout the system. The appeal of a sharing tree stems from its potential to offer scalable writes: all the nodes of a sharing tree can be invalidated or updated in time proportional to the logarithm of their number. In previous work, sharing tree protocols have been proposed, such as the Scalable Tree Protocol (STP) [76], the Tree Directory (TD) [69], and —by far the most serious design— the STEM extensions to SCI [44].

Theoretically, all these protocols (STP, TD, and STEM) offer logarithmic invalidation *under the assumption of unit latency messages*. In reality, message latency *does* depend on the network topology, the size of the system, and the congestion characteristics of the network. Scott discusses the scalability of latency in his thesis and argues that asymptotically latency must increase as at least $O(N^{1/2})$, where N is the number of nodes in the system [86]. Since the aforementioned protocols do not take into account network locality—their tree structure maps arbitrarily on top of the network topology—the latency of their writes is worse than logarithmic. Furthermore, these protocols actually increase traffic in the network over non-tree-based protocols [76,69,44].

3.3.1.1 STEM kiloprocessor extensions to SCI

The STEM extensions to SCI, originally developed by Johnson [44], provide a logarithmic-time algorithm to build, maintain and invalidate a binary sharing tree (as opposed to GLOW's *k*-ary trees) without regard to the topology of the interconnection network. STEM is a complete upgrade of the SCI protocol, although STEM and SCI nodes interoperate at the lower level of performance. STEM does not distinguish between widely-shared data and the rest of the data, since it can be used for all accesses. In the case of non-widely-shared data, STEM has the same performance as the SCI protocol with similar transactions

STEM employs combining in the interconnect to provide scalable reads. Combining operates on two requests that happen to be in a queue at the same time, generating a single new request along with a response to one of the original requests. Combining depends heavily on traffic patterns and in fact will rarely occur, except in the presence of substantial congestion [78]. Combining only returns list-pointer information (and not data, which are returned later). This is much simpler than other approaches (such as the typical NYU Ultracomputer combining [36]), which leave residual state within the interconnect for modifying the responses when they return.

STEM defines one additional pointer for the SCI caches. The representation (triangle) of a STEM cache (devised by James [43]) is shown in Figure 3.2. The *forward* and *backward* pointers are the same as the ones in SCI caches. The *down* pointer is used to build the binary sharing trees. Notice that in Figure 3.2 the different pointers correspond to the different vertices of the triangle.

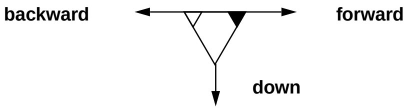


Figure 3.2. STEM cache pointers.

***STEM sharing tree creation*—**The STEM sharing tree creation starts with an ordinary SCI prepend list (Figure 3.3 A). Nodes prepend to this list and eventually they will get the data. A merge process converts this linear list into a list of subtrees. In this structure, the subtrees are strictly ordered according to their height. The first subtree (connected directly to the memory directory) is of height one. Larger subtrees are placed further away from the memory directory. Nodes attaching in front of a stable list of subtrees will trigger further merges: adjacent equal-height subtrees merge to form a single higher subtree, until a stable subtree structure is

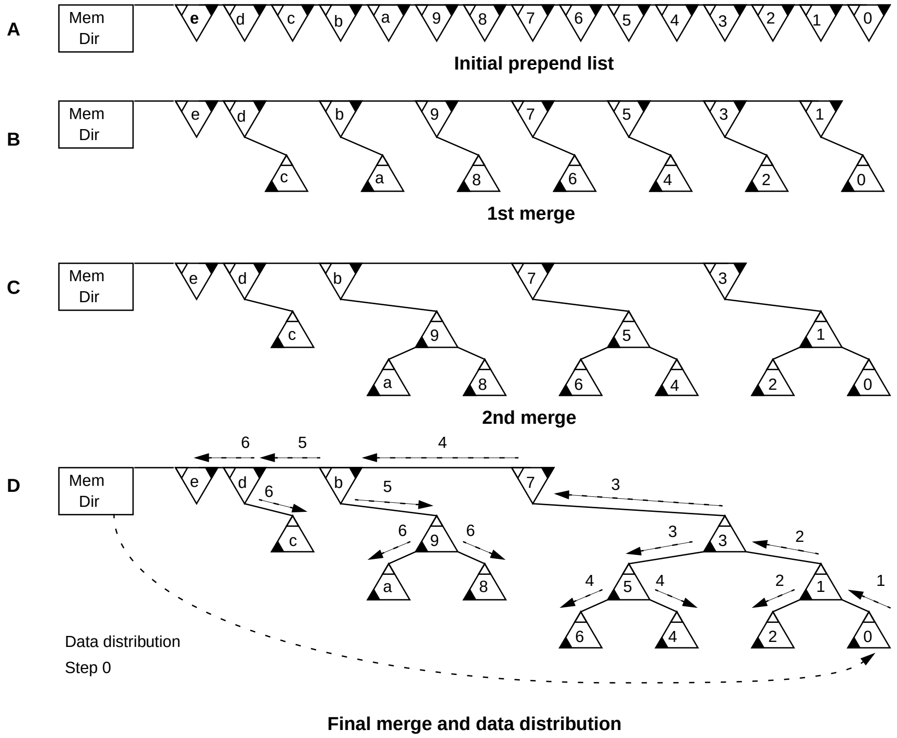


Figure 3.3. STEM sharing tree construction.

formed. Up to three transactions are needed to merge subtree pairs.

The merge process is distributed and performed concurrently by the sharing caches. Data distribution is partially performed during the merging process. Multiple steps are involved (some overlapped in time), one for each level of the binary tree that is created. Under ideal conditions, the first step of the merge process generates a list of subtrees of height one (Figure 3.3 B), the second generates a list of subtrees of height two (Figure 3.3 C), etc. The process continues until the sharing list has reached a stable state, as illustrated in Figure 3.3 D.

During the subtree-merge process, a distributed algorithm selects which pairs of subtrees are merged. To avoid conflicts, nodes two and one cannot be merged if one and zero are being merged. To generate balanced trees, nodes three and two should be merged if nodes one and zero are being merged. Techniques for selecting the nodes to be merged are discussed by James [43] and by Johnson [44]. Following the merge, data are distributed to the nodes. Data are propagated by cache-to-cache writes, as illustrated in Figure 3.3 D. Some of these transactions can take place concurrently with the sharing-tree creation.

***STEM* rollout—**Rollout in STEM occurs for the same reasons as in SCI caches. Rollouts are more complex for trees than for lists when the rolling out node has two children. In this case, the rolling out node *walks* down the tree to a leaf node, removes the leaf from the tree, and finally replaces itself with the leaf node.

If a node has zero or just one child, the basic SCI rollout protocol is used. If a node has two children, it must find a replacement for itself before it rolls out in order to keep the tree structure intact. The leaf node that is used to replace the rollout node is found by first using the *down* pointer and then the *forward* pointer. This search procedure creates a one-to-one correspondence among nodes with two children and leaf nodes and there is no overlap between paths that connect pairs of nodes. After a leaf node has been located with the walkdown, the leaf is swapped with the rollout node. The walkdown and node replacement protocols can introduce up to eight transactions per deleted node. However, half of the nodes in any binary tree are leaf nodes, so the walkdown is only performed half of the time. Furthermore, the average number of walkdown steps required for nodes in a balanced binary tree is one and this average becomes smaller as the tree becomes unbalanced.

***STEM* sharing tree invalidation—**Sharing tree invalidation is initiated by the writer that is the *root* of the tree (connected directly to the memory directory). In the first phase, invalidations are distributed to other nodes in the sharing tree, as illustrated in Figure 3.4 A. The forwarding of invalidations stops with the leaf nodes. In the second phase, leaf nodes invalidate themselves and notify their parents. This trims the leaf nodes from the sharing tree all the way up to the root node, as illustrated in Figures 3.4 B and C.

The sharing tree can also be updated with new values rather than invalidated. As in the first phase of invalidation, updates rather than invalidates are distributed all the way to the leaves (Figure 3.4 A). In the second phase, acknowledgments are returned to the *root* node (writer) without affecting the sharing tree (Figure 3.4 D).

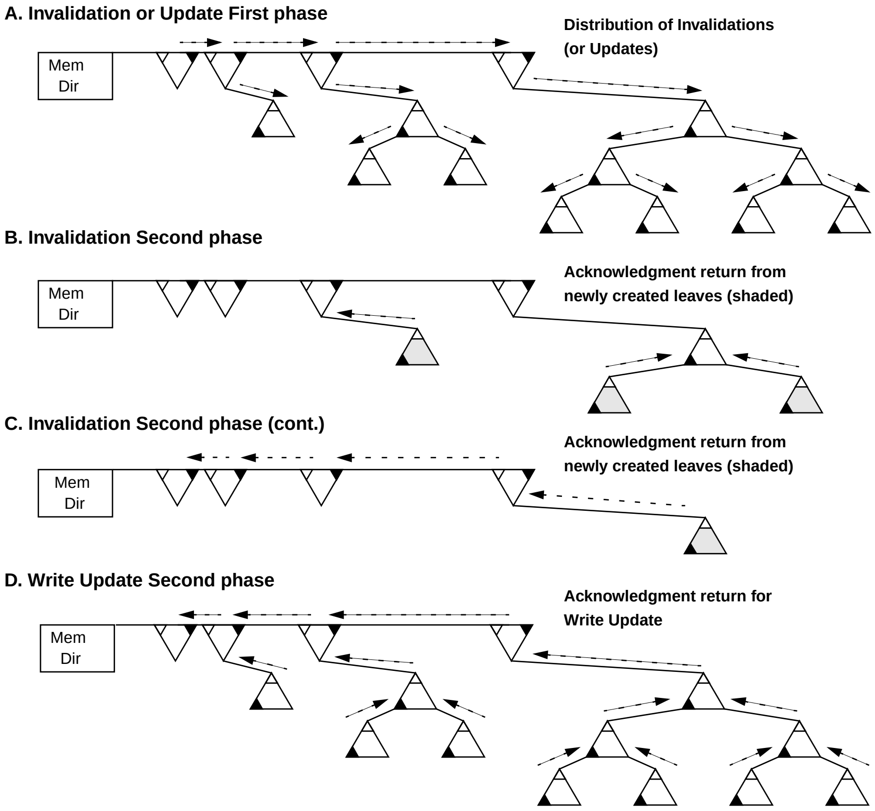

A. Invalidation or Update First phase
Distribution of Invalidations (or Updates)
Mem Dir
B. Invalidation Second phase
Acknowledgment return from newly created leaves (shaded)
Mem Dir
C. Invalidation Second phase (cont.)
Acknowledgment return from newly created leaves (shaded)
Mem Dir
D. Write Update Second phase
Acknowledgment return for Write Update
Mem Dir

Figure 3.4. STEM sharing tree invalidation.

The STEM extensions have not yet been implemented in full detail because of their complexity. However, Johnson simulated the behavior of STEM for regular access patterns and in his thesis [44] he shows that indeed its performance for both reads and writes is *O(logN),* where *N* is the number of nodes sharing. The simulations were performed with the assumption of unit-latency messages.

3.3.1.2 STP and TD

The Scalable Tree Protocol (STP) proposed by Nilsson and Stenström [76] defines a sharing-tree protocol that speeds up the writing of shared data by invalidating the tree in logarithmic time. The notable feature of the protocol is that additions to and deletions from the tree leave the tree balanced. The tree does not map onto the underlying topology (thus the protocol does not benefit from physical locality). This is because its structure depends on the timing of the requests, and deletions from the tree (replacements) change its structure without respect to the underlying topology. Furthermore, no support has been proposed to speed up reads (*e.g.*, combining). Nilsson and Stenström's STP work did not address correctness issues such as deadlocks, starvation etc. Simulation [76] shows a 15% speedup over SCI for the GAUSS program in a 16 node system but also a 35% *increase* in network traffic. This may be a serious drawback for larger systems.

The Tree Directory (TD) protocol, proposed by Maa, Pradhan, and Thiebaut[69], is based on a *K*-ary tree structure that is maintained in the sharing caches. This scheme was proposed to improve the storage overhead problem of limited directories. It is a limited pointer directory expanded in a tree structure. TD does not take into account physical locality—the sharing tree does not map onto the network topology—and it does not provide for scalable reads. TD strictly enforces multilevel inclusion, so nodes have a unique position in the tree. The authors report improvements in performance compared to a basic limited directory but equivalent performance compared to a chained directory scheme such as SCI.

The two sharing tree protocols described above were applied indiscriminately on all data. This diminishes the potential benefit since the overhead of the more complex protocols is incurred for all accesses. Building a sharing tree does not come for free and doing so for data that is not widely shared may result in degradation for the most common access patterns. Only when the number of nodes that participate in the sharing tree is large is the overhead sufficiently leveraged. Bianchini and LeBlanc distinguished widely-shared data (“hot” data) from other data in their work [17]. This is feasible as long as we can distinguish requests and responses for widely-shared data and for non-widely-shared data. In this thesis, I decouple the optimization to a sharing pattern from the identification of the sharing pattern and show that there are many identification techniques with different tradeoffs.

3.3.2 Request combining

Sharing-tree coherence protocols provide support for writes. However, a truly effective solution should also provide support for reads. Traditionally, request combining hardware in the network (explored by Gottlieb et al. in the NYU Ultracomputer [36]) provides scalable reads and alleviates hot spots [78] in the system. Many types of combining are discussed in detail by Lebeck and Sohi [63]—who also introduced a classification of combining schemes. The success of Interconnect-Interconnect-Combining—using the terminology of Lebeck and Sohi [63]—such as the NYU Ultracomputer combining, depends on whether requests happen to be in the same network queue at the same time. This in turn depends on many factors such as the timing of the requests, the latencies in the system, the size of the network queues, and unrelated network activity taking place concurrently. Although combining is most often successful when it is most needed (when there is congestion in the network), and where it is most needed (at the hot spot), clearly its success depends on network characteristics. Of the sharing tree protocols described previously, only STEM employs combining in the interconnect to provide scalable reads.

3.3.3 Caching in the network

Caching in network switches has been proposed by Mizrahi, Baer, Lazowska, and Zahorjan [71]. In the context of this proposal, network caching is used to “migrate” the functionality of the home node close to where the data are actively used. This proposal was not intended for wide sharing and it does not handle widely-shared data. GLOW is performing a similar function: it transfers the functionality of the home node close to where the data are accessed, but it does so in a much larger scale with many GLOW agents impersonating the home node.

3.3.4 Hierarchical directories

The GLOW concept is similar to that of the Hierarchical Full Map Directory (HFMD) [69], which is based on full-map directories embedded in the network topology. HFMD is strictly based on the inclusion property. In contrast, GLOW is far more general (since it does not impose the multilevel inclusion property) and can be easily applied to many topologies. Relevant ideas to the GLOW framework have also been proposed for Cache-Only Memory Architectures (COMA) such as the Data Diffusion Machine (DDM) proposed by Hagersten et al. [37] and the Kendall Square Research KSR-1 [57].

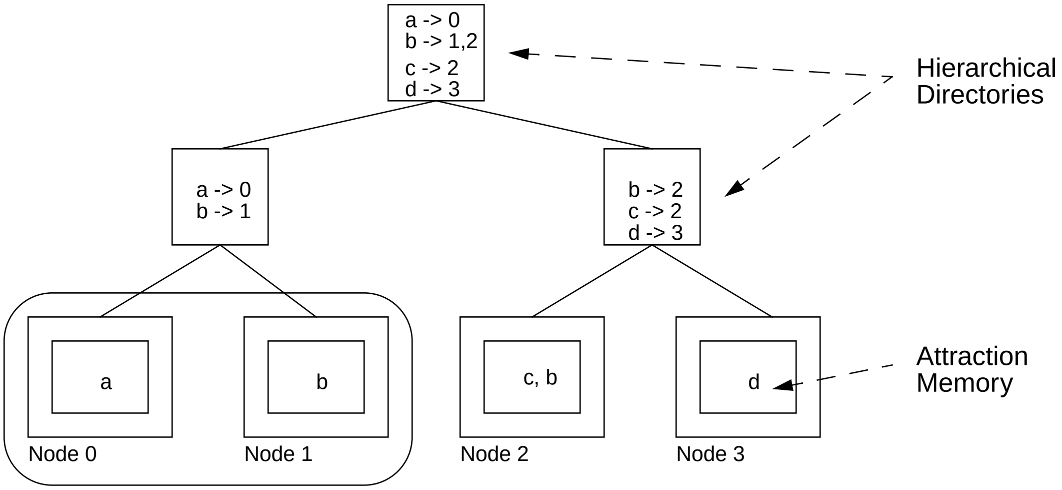


Figure 3.5. COMA architectures. A request for non-local data traverses the hierarchical directories until it locates a directory entry that specifies the location of the data. For example, when node 0 requests data block "b" it will find it in its immediate vicinity; when it requests data block "d" it has to ascend to the root of the hierarchical directory structure to find the data block in the remote node 3.

In COMA architectures, a data block does not have a fixed home node location as in the typical directory-based cache-coherent DSM architectures. Instead, data blocks reside in the cache-like memory or Attraction Memory (AM) [37] of the nodes that access them (Figure 3.5). Thus, in a COMA system, a read request for a non-local data block does not have an *a priori* fixed destination. Instead, a copy of the data block is first located and the read-request is directed toward that copy. As in the case of widely-shared data, satisfying requests from the *closest* copy leads to increased communication locality. In COMA machines such as the KSR-1, this is achieved by providing an explicitly hierarchical directory structure. Although the first DDM implementation was built using a common bus to connect the nodes, its abstract design also called for a hierarchical directory structure. In such architectures, requests are first routed to the lowest level of the hierarchical directory that keeps track of the data blocks in the immediate vicinity of the requesting node; if a copy is not located in this level the request is routed to the next level of the directory that keeps track of data blocks available in a larger part of the system; eventually the request will locate a copy in some directory level or it will reach the top level of the directory (root level) that keeps track of the location of all data blocks in the system. However, this hierarchical directory has proven to be a serious disadvantage of the COMA architecture and it has been discarded in subsequent proposals such as the Flat-COMA [94] or Simple-COMA (S-COMA) [82], which specify home nodes for the data blocks. A serious shortcoming of the hierarchical directory approach is its mandatory use for all requests whether or not they are likely to locate a copy in lower levels of the hierarchical directories.

3.4 GLOW

The goal of GLOW is to preserve the simplicity and elegance of the shared memory paradigm and to avoid modifying parallel algorithms that depend on widely-shared data, by providing—transparently—the necessary mechanisms for efficient access of such data. To avoid limiting the scope of this work, instead of proposing GLOW as specific cache coherence protocol to handle widely-shared data, I describe it as a set of extensions for existing cache coherence protocols. The extensions specify how to enhance networks using special switches —the GLOW agents— that know how to build sharing trees by recursively applying the underlying cache coherence protocol. Only the GLOW agents are aware of the sharing trees; the rest of the nodes in the system need only use the base cache coherence protocol.

The GLOW extensions are intended to provide support exclusively for widely-shared data. However, they are independent of how the widely-shared data are identified and exposed to the hardware. For purposes of discussing the extensions I assume the existence of special requests that are used to access widely-shared data. In subsequent sections I describe how to generate such special requests either statically or dynamically.

3.4.1 GLOW features

The GLOW extensions improve on previous efforts (TD [69], STP [76], STEM [44]) by embodying the following four characteristics:

- TRANSPARENCY: GLOW is not a protocol itself but rather extensions to other protocols to improve their handling of widely-shared data. The functionality of the GLOW extensions is implemented in selected network switch nodes called GLOW agents. A requirement for all

GLOW implementations is that the GLOW agents be uniquely identified. When GLOW is applied to linked-list protocols such as SCI, GLOW agents can have their own unique identifiers (IDs) and can be implemented as stand-alone nodes anywhere in the topology (Section 3.5). However, in full-map protocols (*e.g.*, $\mathrm{Dir_nNB}$ [8]) the node addressing is more restrictive. In situations where we cannot assign unique IDs to GLOW agents, these can be unified with ordinary nodes and share the nodes' ID (Section 3.7.1). GLOW agents intercept special requests for widely-shared data, but other requests (for non-widely-shared data) remain unaffected and proceed through the switch nodes at full speed. For the SCI protocol—and to a large extent for full-map directory protocols—the node caches and the node memory directories are unaware of the existence of the GLOW agents. Minimal changes are needed for the base cache-coherence protocol. In contrast, other proposals (*e.g.*, STEM, STP, TD) call for a complete upgrade of the cache-coherence protocol in all nodes.

- SIMPLICITY: The GLOW agents behave both as memory directories and cache nodes using the underlying cache coherence protocol recursively: GLOW agents impersonate remote memory nodes toward a local cluster of nodes they service; toward the home node directory, agents behave as if they were ordinary caches. Implementing a GLOW agent is therefore a matter of copying the protocol implemented in the node caches, copying the protocol implemented in the memory directories, and providing the necessary glue logic to make them behave as a single entity.
- NETWORK (GEOGRAPHICAL) LOCALITY. A sharing tree is constructed out of the GLOW agents and other caches in the system to match the tree that fans-in from all the sharing

nodes to the actual home node of the widely-shared data. GLOW captures *network (geographical) locality* so that neighboring nodes in the sharing tree are in physical proximity.

- NO MULTILEVEL INCLUSION. In contrast to hierarchical directory schemes for cache coherence, GLOW does not impose *multilevel inclusion* Appendix 13 in its sharing trees. In hierarchical directory schemes, each level of the hierarchy includes all the information of the lower level. In GLOW, each hierarchical level of the sharing tree is independent of the one below. The sharing nodes do not have a fixed level in the tree, but they can occupy any level. Thus, the involvement of the GLOW agents is optional. This characteristic gives GLOW the necessary flexibility to be applicable to many different network topologies without fear of deadlock (see Section 3.5.3 for a detailed discussion). Additionally, it allows for algorithms that are impossible with multilevel inclusion, such as the *LINEARIZING ROLL-OUT* algorithm that handles replacements in the sharing trees (see Section 3.5.6 for a detailed discussion).
- SCALABLE READS. Since the GLOW agents intercept multiple requests for a data block and generate only a single new request toward the home node, an effect similar to read-combining is achieved, eliminating hot spots [78].
- SCALABLE WRITES. On a write, GLOW invokes in parallel the underlying protocol's invalidation or update mechanisms. On receipt of an invalidation (update) message, an agent recursively starts the invalidation (update) process on the other agents or nodes it services. This parallel invalidation (update), coupled with the geographical locality of the tree, permits fast, scalable writes that require low bandwidth.

3.4.2 An overview of the GLOW extensions

Every cache coherence protocol has two facets: the memory and the cache facet. In GLOW, selected switch nodes in the topology (the GLOW agents) behave both as memory and cache nodes. A GLOW agent behaves as the home node memory directory (thus giving the illusion of memory being nearby, locally) toward the nodes it services. Toward the actual home node memory directory the GLOW agents act as ordinary caches.

The GLOW agents are directory caches in the network (with optional data storage) whose function is to cache the structure of sharing trees in the network. A sharing tree consisting of these agents and other caches in the system is constructed to match the tree that fans-in from all the sharing nodes to the actual home node of the widely-shared data. Nodes become children of a GLOW agent when their requests for widely-shared data are serviced by that agent. The GLOW agents themselves join the sharing tree when they send their own requests toward the remote memory in order to service their *children* (other nodes or other agents). The sharing tree is thus created recursively, bottom-up. In general, the sharing nodes will be the leaves of the sharing tree and the GLOW agents will comprise the internal nodes. The root of the tree is the home node directory of the widely-shared data. However, because the involvement of the GLOW agents is optional, any mix of nodes and agents in any level of the tree is allowed.

An example of a general K-ary sharing tree that can be formed using GLOW agents is shown in Figure 3.6. In this figure the home node directory is represented by the square in the top-left corner, nodes that have a copy of the data block by white circles and GLOW agents by black circles. As far as the home node directory is concerned, it points to a number of sharing nodes, whereas in reality it points to the first level of agents that hold the rest of the sharing tree. Sim-

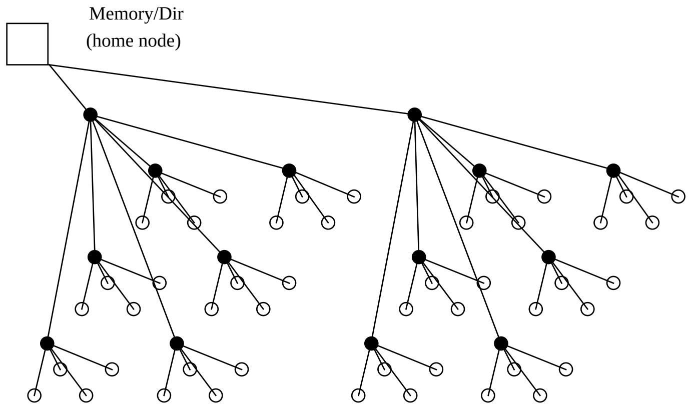


Figure 3.6. Logical K-ary sharing tree. Black nodes are GLOW agents, white nodes are sharing nodes.

ilarly, as far as the nodes that actually share the data (the leaves of the sharing tree) are concerned, they are serviced directly by the home node directory, where in fact they are serviced by the agents impersonating the memory.

GLOW extensions can be implemented on top of a wide range of topologies, including hypercubes, meshes, trees, butterfly topologies [45] and many others. For linked-list protocols (SCI) the GLOW agents can be stand-alone nodes anywhere in the network topology (Section 3.5) but with other protocols such as full-map directory protocols GLOW agents may be restricted to coincide with other nodes in the topology (Section 3.7.1). GLOW can also be used in irregular topologies (*e.g.*, an irregular network of workstations). As I described in the previous chapter, I study GLOW on k-ary n-cube topologies that have proven to be popular with real implementations (CRAY T3D [21], T3E [87], SGI Origin 2000 [62], Convex Exemplar 2000 [1]).

3.5 GLOW on SCI

In this section, I describe a detailed implementation of GLOW on top the IEEE/ANSI standard SCI cache coherence protocol. This implementation demonstrates that GLOW is feasible on top of a very complex and feature-rich protocol. GLOW requires virtually no changes to the basic SCI protocol. A few special considerations to accommodate GLOW are pointed out in the description. Since GLOW makes heavy use of the underlying cache-coherence protocol, a full description of the SCI protocol appears in Appendix 1. Here, I describe in detail the GLOW agents in an SCI system and how they can create, invalidate (update), and manipulate sharing trees in the network.

3.5.1 SCI GLOW Agents

Central to any GLOW implementation are the special network switches, the GLOW agents, that build the sharing trees. In an SCI network a GLOW agent is implemented on a switch node that transfers messages from one ring to another. In Figure 3.7 I show a small SCI system built of two SCI rings (ovals) and six nodes (squares). A single switch (triangle) connects the two rings.

To perform transactions safely with other memory directories and other caches, the agent has its own unique node identifier. Lets assume that widely-shared data reside in the top node (labeled “M”) and all nodes access them. Instead of letting the requests of the three lower nodes pass through to the upper ring, the GLOW agent intercepts them and requests its own copy of the data, pretending to be a simple cache node in the upper ring. The agent then pretends to be the remote memory node “M” locally on the lower ring. The three lower nodes get

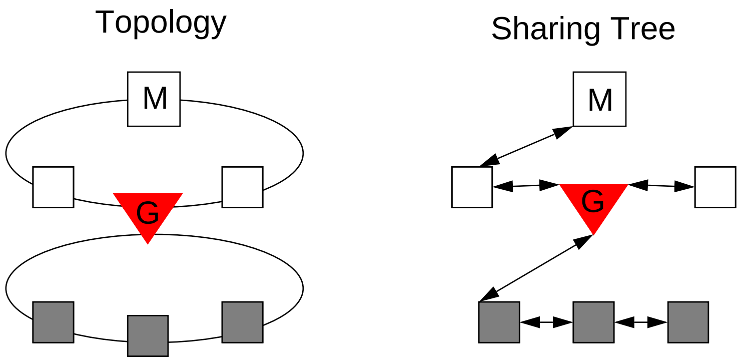


Figure 3.7. A GLOW agent in a 2-ring SCI system.

their data from the agent, as if the actual home node serviced them.[1] They form an SCI list that is confined to the lower ring. This list is called a *child list* of the GLOW agent. The resulting sharing tree is shown in the right of Figure 3.7. It consists of two levels, each being a standard SCI sharing list. The key observation is that the GLOW agent looks like a cache node in the top level of the sharing tree and as a memory node for the lower level of the sharing tree.

An important characteristic of any GLOW agent is that *its behavior does not depend on any topological parameters other than the direction of the request traffic*. The GLOW agents are unaware of the network topology and of their position in the network. The only thing they know is which direction is the way to the home node for any request. The way to the home node is the way to a higher level of the GLOW hierarchy; everything else is the lower level. Thus, based on the direction of the request traffic, the single agent in Figure 3.8 can assume the role of memory for the lower ring for one sharing tree (left sharing tree in Figure 3.8) and

[1] That a node with a different node identifier than the home node of the data serviced the requests makes no difference in the SCI protocol. However, the backward pointer of the head of the small SCI list must point to the agent. This pointer is undefined in SCI because it is implicitly assumed that it points to the home node of the data.

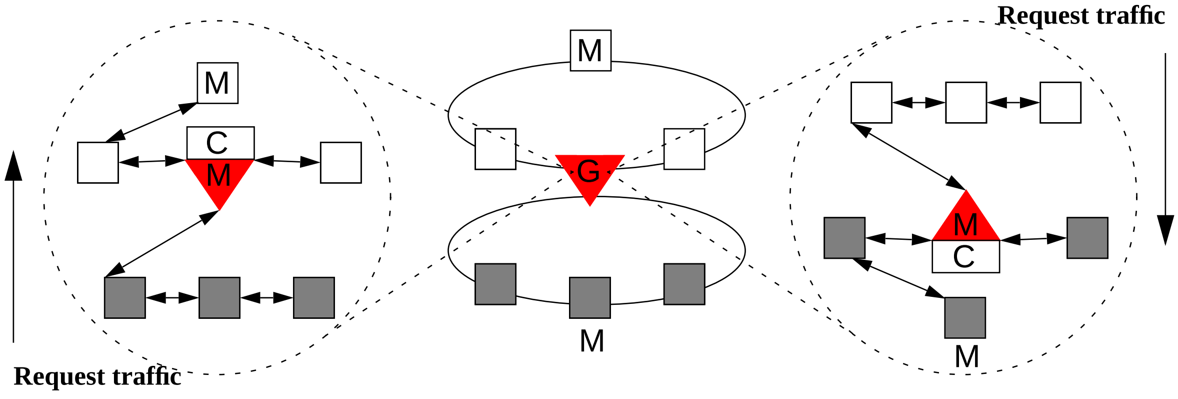


Figure 3.8. The behavior of a GLOW agent is independent of topological information; it depends only on the direction of the request traffic.

at the same time assume the role of memory on the upper ring for another sharing tree (right sharing tree in Figure 3.8).

The behavior of the GLOW agents is slightly more complex than the simplified version described above:

- First, a GLOW agent can appear as multiple distinct memory directories for a single data block. This capability is necessary for the agent to support different child lists for the different rings it handles. The number of distinct child lists an agent can support is called the *degree* of the agent. The maximum useful degree of an agent is one less than the total number of rings connected by the agent, since one of the rings is the way to home node and cannot have a child list. The minimum useful degree is two, since a degree of one leads to one-dimensional trees. The degree of an agent is an implementation parameter, reflecting both topology and cost considerations.
- Second, in contrast to the memory directories which only point to the head of a list, the GLOW agents hold child lists from both ends. This is achieved by the agent presenting

itself as a *virtual tail* to the first node that joins each child list. The reason is that the agents need a way to pass the data fetched from memory to their waiting child lists. Since waiting SCI nodes always get the data from their downstream neighbor (except tail nodes that get them from memory), appearing as a virtual tail is a transparent method for the agent to distribute data. This also facilitates rollout algorithms such as the LINEARIZING ROLLOUT described in Section 3.5.6.

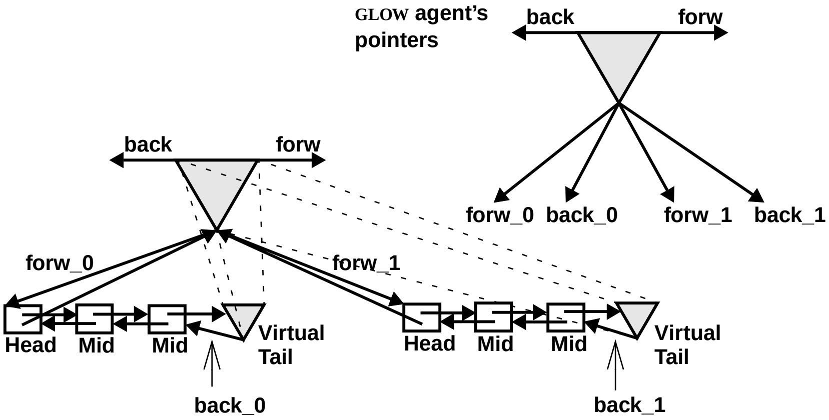


Figure 3.9. Schematic representation of a GLOW agent holding 2 child lists.

In Figure 3.9 the agent is represented by a triangle and the SCI caches by rectangles. For every child list, the GLOW agent has a *memory forward* pointer and a *virtual-tail backward* pointer (see Figure 3.9 and Figure 3.10). The agents also have a *forward* and a *backward* pointer, permitting them to join SCI lists and act like ordinary SCI caches.

The internals of a GLOW agent can be designed in many different ways. Figure 3.10 depicts a sample design that is used in all the subsequent sections. The GLOW agent is a switch that is connected to a number of rings and to the bus of a corresponding node. This arrangement is used in the k-ary n-cube topologies, where there is an agent for every node in the system.

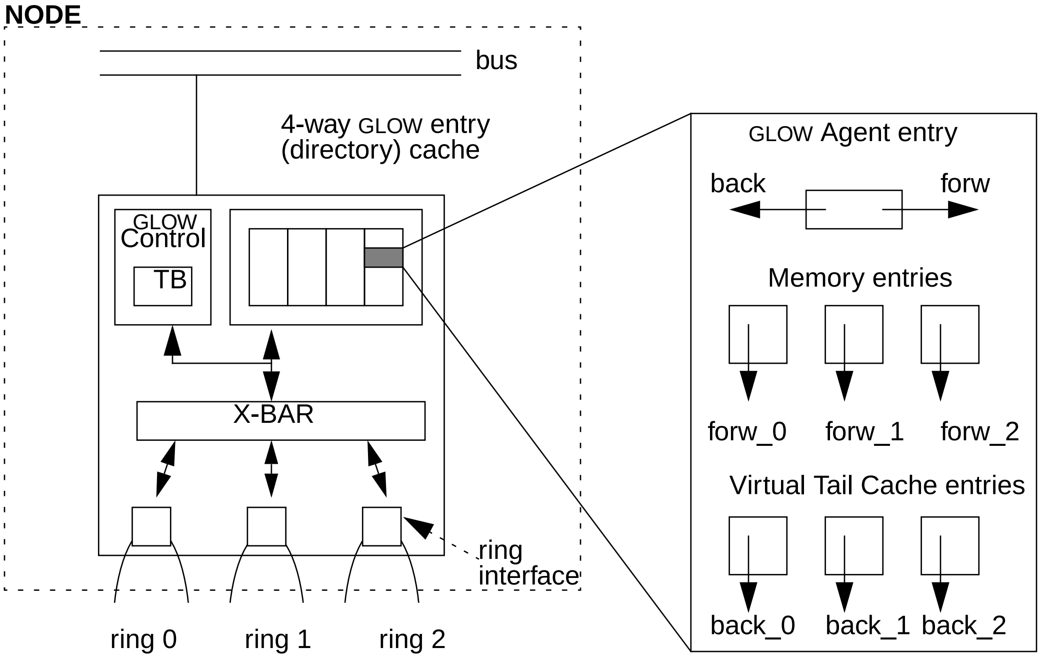


Figure 3.10. Internals of a GLOW agent. The right scheme is a representation of an uncompressed GLOW entry in the directory cache.

However, in the general case, GLOW agents can be a stand-alone switch nodes.

The agent can pass a message from any ring to any other ring. If a request is a special GLOW request, it is taken out of the rings and delivered to the GLOW control logic. This logic accesses the GLOW directory cache that in the case of Figure 3.10 is a 4-way set-associative cache. This cache can be expanded with data storage. However, as I will show in Section 4.2.1, data storage does not provide enough performance improvement to be cost-effective. The control logic also contains a Task Buffer (TB) which accommodates up to 64 tasks, as defined by the SCI standard [41]. The Task Buffer can temporarily hold the equivalent of a few GLOW entries that are in the process of being replaced or invalidated, each associated with multiple outstanding tasks. The task buffer is not a victim cache; all its entries are temporary and soon-to-be deleted.

Each GLOW entry in the GLOW directory cache can store the necessary pointers and state for each personality of the GLOW agent. Figure 3.10 shows a *fully-expanded* GLOW entry which has: (i) a single cache entry (two pointers and state) to impersonate a cache in the high-level ring, (ii) a per-ring memory directory entry (a single forward pointer and state) to hold a child list, (iii) a per-ring cache entry (a single backward pointer and state) to impersonate a virtual tail. At any time, the memory directory entry and the virtual tail entry for the high-level ring are *unused* and can be discarded completely, at the expense of having to dynamically rename the rest of the entries to correspond to the correct rings. The GLOW entry depicted in Figure 3.10 is a fully-expanded entry, meaning that it has full support for all the rings. However, as mentioned above, the degree of the agent is an independent parameter and GLOW implementations are not required to support all the rings.

3.5.2 Creation of sharing trees

The GLOW sharing trees are constructed with the involvement of the GLOW agents that impersonate the memory directory in various places in the topology. A node uses a special request to access a block of widely-shared data. A normal request would pass through the GLOW agents and would reach the actual home-node memory directory. In this case, the node would be instructed to attach to an SCI list and get the data from the head (if the list was stable) or the last node that joined the list if the data were still in transit (explained in Appendix 1 and depicted in Figure 8.3). However, the special request is intercepted at the first agent (where the request would change rings) *at the agent’s discretion*. The intercept causes a lookup in the agent’s directory storage to find information (a GLOW entry) about the requested block. The lookup will result in a *hit* if there is an allocated tag or a *miss* otherwise:

- If the lookup results in a miss, the agent sends its own special request for the data block toward the home node (Figure 3.11). The agent instructs the requesting node to attach itself to a child list comprised only of the virtual tail (the agent itself) and wait for the data. As soon as the agent gets a copy of the data, it will pass them to the child lists via the virtual tails. By using virtual tails, the SCI nodes cannot tell the difference between a GLOW agent and the actual memory directory.
- If the lookup results in a hit, the requesting node is instructed to attach itself to the appropriate child list. It will get the data from either the agent (if it caches data) or the previous head of the child list (Figure 3.12).

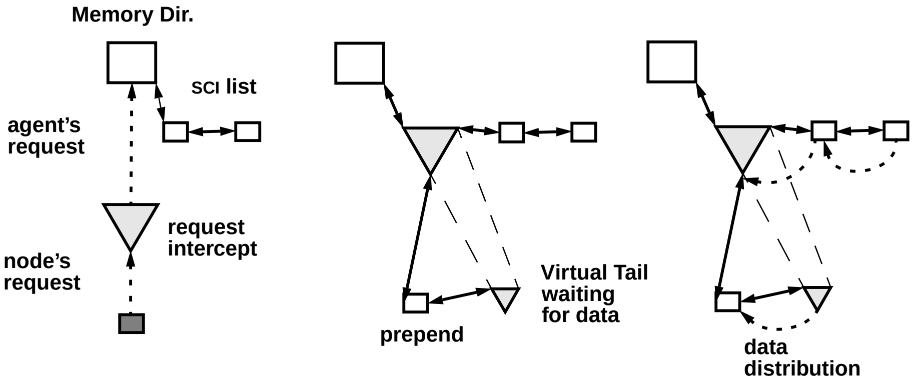


Figure 3.11. GLOW agent intercepts node's request and generates its own request (miss in the agent's directory).

The requesting node might be the first node in a new child list, even when a GLOW agent already has multiple child lists. In this case, when the requesting node attaches directly to the virtual tail in a new child list, the agent has to *fetch* the data from one of its other child lists (Figure 3.13). The agent cannot repeat its request to get the data, since this would result in

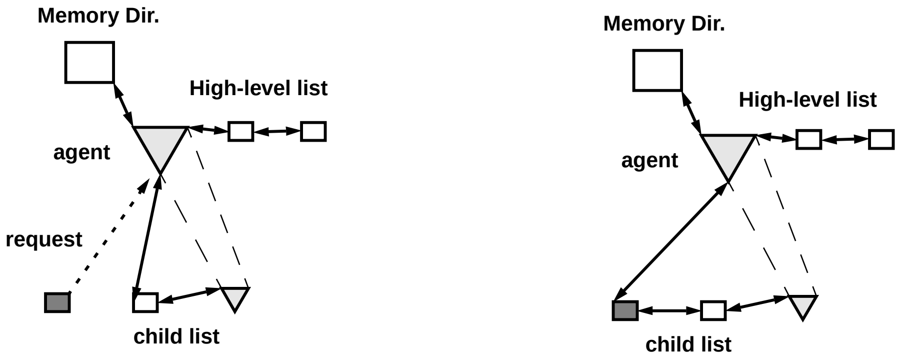


Figure 3.12. Hit in agent; node joins a pre-existing child list.

joining the sharing tree twice. The fetching of data is achieved by the agent attaching as a virtual head in front of one of its populated child lists and reading the data. For the duration of the fetch operation, an additional temporary pointer is allocated to the GLOW entry for the virtual head. The fetch operation can be applied recursively (descending the sharing tree) until a node that has the data is found.

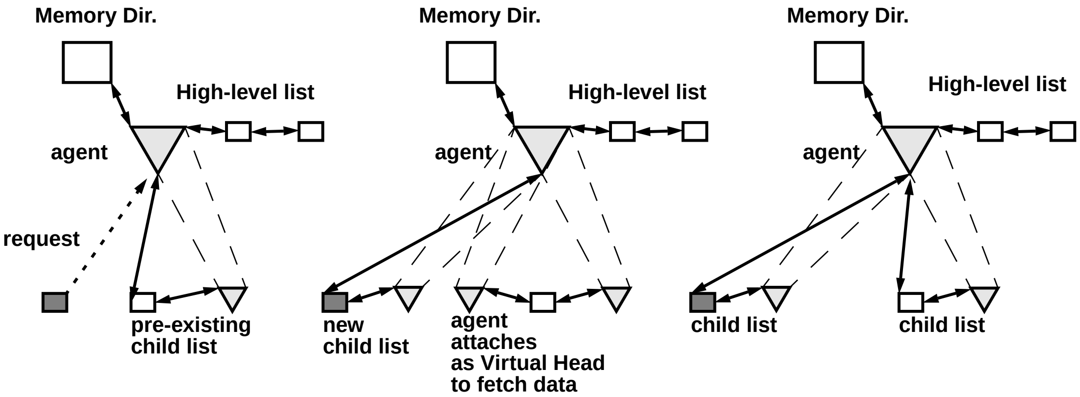


Figure 3.13. Hit in agent, node joins a new child list.

In the case of a miss in the agent, the agent sends its own request which will be treated the same way at the next agent. Eventually, the last agent before the home node will get the data (either from memory or from the node connected directly to the memory directory) and pass them to its child lists. When all nodes simultaneously read a cache block (for example, after a global barrier) all the child lists can be constructed in parallel. Copies of the cache block are distributed down the tree concurrently with list construction.

To visualize a GLOW sharing tree in a complex topology, Figure 3.14 shows the mapping of a *k*-ary sharing tree on a 3-ary 3-cube. The GLOW agents are represented by triangles and the SCI caches by circles. For clarity the virtual tails are not shown. Lists are in hierarchical levels and the agents can have child lists in different levels. The nodes in the lists are drawn in perfect order (which is not the common case). The nodes' positions within a list depend on the timing of their requests as seen by the GLOW agent. The general structure of the tree, however, does not depend on the timing of requests.

3.5.3 Multilevel inclusion

As I described previously, when a node reads widely-shared data, it initiates a specially-tagged request addressed to the home node of the data. A GLOW agent can intercept this request at the point where it changes rings. However, the request can be completely ignored by the GLOW agent and passed toward the next hierarchical level, if there is danger of deadlock. That an agent may choose to ignore a request completely, does not affect GLOW's correctness, but it may have a negative impact on performance. When an agent ignores a request it is passed to the next hierarchical level where hopefully it will be serviced by a higher-level agent. In this case, the node that issued the original request joins an SCI list at a higher hierar-

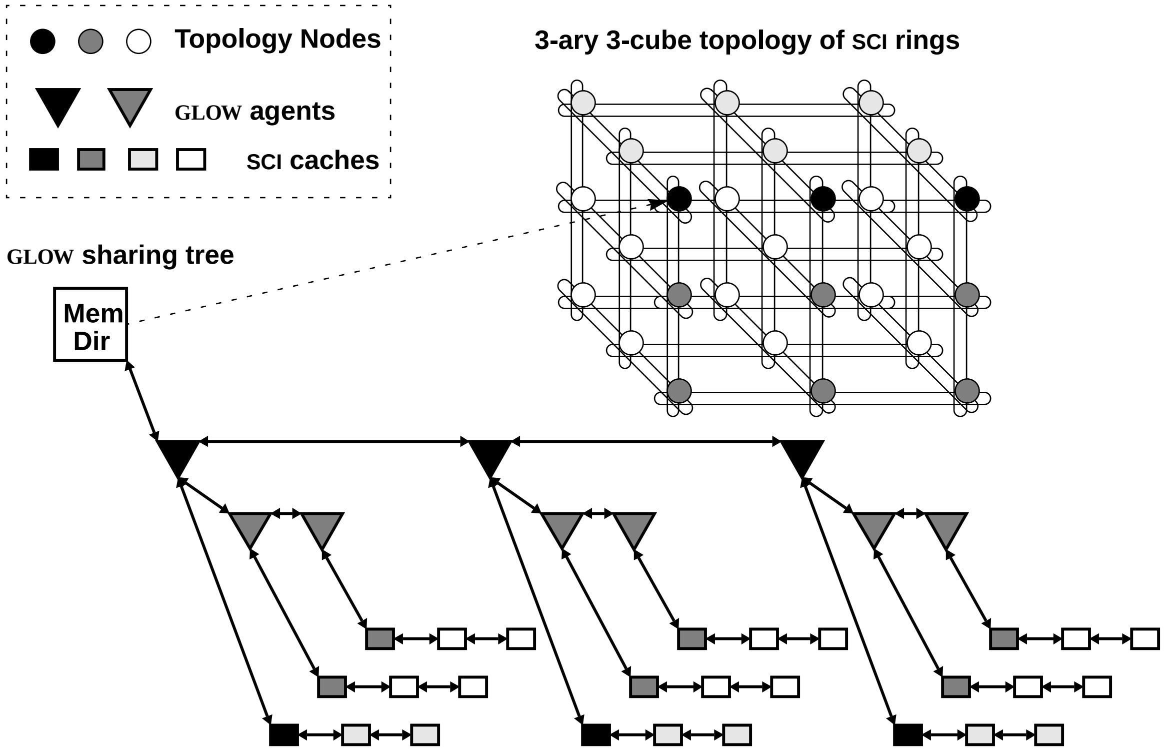


Figure 3.14. GLOW **sharing tree on a** ***3*****-ary** ***3*****-cube (virtual tails not shown).**

chical level than normally. The request may be ignored by multiple agents, all the way to the home node, where it will be serviced by the memory directory.

To avoid deadlocks because of storage conflicts, the agents may ignore certain requests. This arises only for conflicting data blocks which have different home nodes but map to the same entry in the GLOW directory caches. A request is always ignored when it requires an entry in the agent's directory cache that is currently taken by another *non-stable* sharing tree (*i.e.*, a sharing tree in a transitional state). However, if the directory entry is taken by an older, stable tree it can be evicted to make room for the new tree. A deadlock example is shown in Figure 3.15, where two sharing trees conflict in the topology shown on the left. In Figure 3.15, the top tree has advanced halfway toward its target home-node directory which is in the node designated by “M” in the lower ring. Similarly, the bottom tree has advanced halfway toward its own home-node directory (the “M” node in the top ring). To proceed, each sharing tree requires the GLOW entry occupied by the other tree. However, the top agent ignores the request of the bottom agent (and passes it directly to the top “M” node) since this request requires a GLOW entry that is *not part of a stable tree*. Similarly, the bottom agent ignores the request of the top agent.

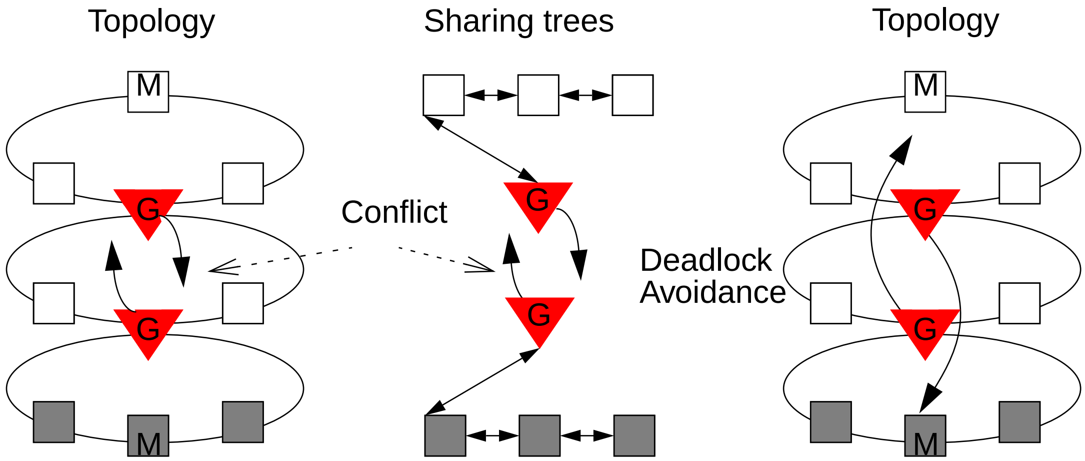


Figure 3.15. Deadlock! Two expanding sharing trees (shown in the middle) conflict in a pair of GLOW agents (topology shown in the left). Each tree needs the GLOW entry of the other tree to continue building toward its target home node memory directory. To break the deadlock each agent ignores the request of the other agent and passes it toward its destination (shown in the right).

Thus, the GLOW agents intercept requests selectively. The selective behavior of the agents means that multilevel inclusion is not imposed on GLOW sharing trees: the position of a node in the sharing tree is decided dynamically. This is a very powerful characteristic because it allows great flexibility: GLOW is not tied to any particular physical hierarchy. As described above, to avoid deadlocks agents ignore certain requests. This can be extended to other situations. For example, various agent implementations can ignore requests because they are too busy or because—somehow—they decide that some intercepts are not important. Furthermore, this flexibility allows advanced agents as described in Section 4.3.1. These agents adapt to wide sharing dynamically: they start intercepting requests only after they discover that the requests are repeated often enough.

3.5.4 Invalidation of sharing trees

A significant part of the performance benefit of GLOW comes from scalable writes. A GLOW tree can be invalidated or updated very fast when the agents work in parallel to invalidate or update their child lists. The invalidation or update starts when a node writes the data. Before a node can modify a cache block, it must first become the head of the top-level list connected directly to the memory directory of the data-block's home node. In this position, the node is the root of the sharing tree and the only node with write permission to its cache block. If a sharing node is not already the root it has to leave the sharing tree before becoming the root. An SCI node simply disconnects from a child list, as it would from any SCI sharing list. It then sends a write request, that cannot be intercepted by the GLOW agents, to the home-node directory. This guarantees that the node will become the root of the sharing tree. The node then proceeds with the actual write. After the cache block is written, invalidation messages are forwarded down the tree in parallel by the GLOW agents.

Parallelism is crucial during the invalidation or update because otherwise there is not much to be gained over the serial invalidation of the SCI sharing list (see Chapter 4, Section 4.2.3 for serial tree invalidation). However, parallelism assumes *request forwarding.* Request forwarding means a request arriving at a node is forwarded to one or more new nodes. This is a significant departure from SCI that guarantees forward progress (*i.e.*, that eventually all transactions will make progress) and deadlock absence by partitioning finite hardware resources (*i.e.*, queues and buffers) into two separate sets for requests and responses. A strict one-to-one correspondence of requests to responses and priority of response processing over request processing guarantees forward progress and deadlock avoidance. Request forwarding, however, nullifies the one-to-one correspondence of requests to responses and requires special care to avoid starvation and deadlock.

Fortunately, a solution —attributed to D. V. James and described in Johnson's thesis [44]—has been proposed to render request forwarding safe with regard to forward progress and deadlocks. Request forwarding implies that a request arriving at a node can generate multiple new requests. This can lead to deadlock by filling up the limited hardware request queues. The solution is to expand the capacity of the request queues using *virtual request queues*. A virtual request queue is built by chaining the corresponding cache entries (either node cache entries or GLOW directory entries). Every request arriving at a node corresponds to a cache entry. If the resources to forward new requests are not available (*i.e.*, the hardware request queues are full), the cache or GLOW entry is appended to a first-in-first-out (FIFO) virtual request queue comprising the cache entries waiting to finish their forwarding. While the request forwarding queue is not empty, new requests arriving at the node cause the corresponding cache entries to be appended to the virtual request queue. This guarantees that every cache entry that received a request will eventually complete its requested operation. Head and tail pointers are required to point to the first and last cache entries of the queue and a pointer in each cache entry points to the next one in the queue. The latter pointer can occupy the data space of the cache entry in cases where the data space is no longer needed (*i.e.*, in the case of invalidation where the data are discarded).

At a minimum the GLOW agents must support request forwarding. When only the GLOW agents support it I call it *partial request forwarding*. Request forwarding could be applied to the entire GLOW sharing tree (f*ull request forwarding*) if the SCI protocol in the nodes is upgraded to support it. The invalidation algorithm depends on whether partial or full request forwarding is employed:

***Partial request forwarding*—**The fully SCI compatible GLOW extensions use SCI's invalidation mechanism to invalidate their children and use request forwarding amongst them. All the GLOW agents in parallel assume a role similar to the head nodes in SCI and invalidate their child lists using the SCI invalidation [49].

The invalidation of a GLOW sharing tree is shown in Figure 3.16 (the initial tree is shown in Figure 3.16A). After the cache block is written, the node starts invalidating the highest level of the sharing tree using the SCI invalidation protocol (Figure 3.16B, transaction 1). However, the GLOW agents react differently than the SCI caches to invalidation messages. On receipt of an invalidation message the GLOW agent does the following concurrently:

1. It replies to the node that sent the invalidation message by sending a negative acknowledgment pretending that it is about to leave the tree or *rollout* (Figure 3.16B, transactions 1b and 2b). In SCI rollout, the operation of leaving the tree has precedence over invalidation. This causes the invalidating node to wait for the rollout so it can get the pointer to the

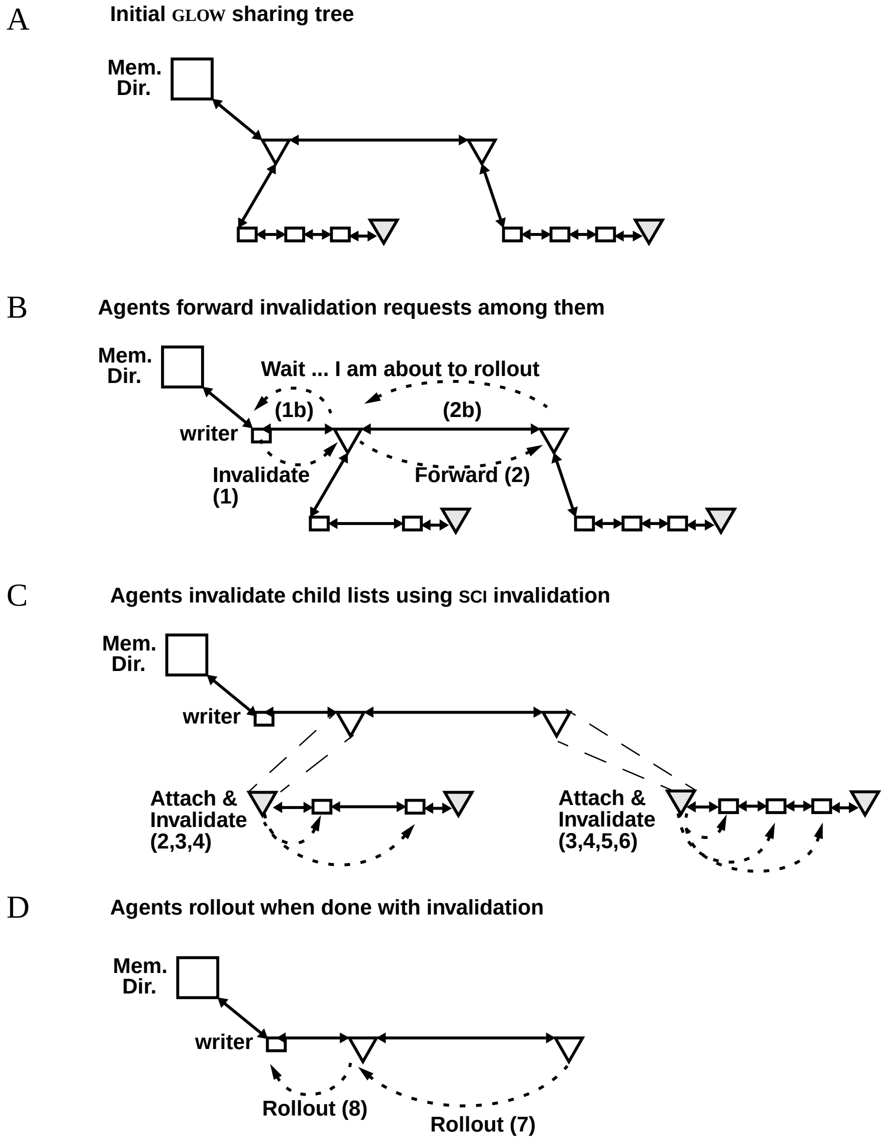

A
Initial GLOW sharing tree
Mem.
Dir.
B
Agents forward invalidation requests among them
Mem.
Dir.
Wait ... I am about to rollout
(1b)
(2b)
writer
Invalidate
(1)
Forward (2)
C
Agents invalidate child lists using SCI invalidation
Mem.
Dir.
writer
Attach &
Invalidate
(2,3,4)
Attach &
Invalidate
(3,4,5,6)
D
Agents rollout when done with invalidation
Mem.
Dir.
writer
Rollout (8)
Rollout (7)

Figure 3.16. Invalidation of a GLOW sharing tree.

downstream node and continue the invalidation. However, the GLOW agent will complete the rollout only when it becomes the tail of its list. At this point the agent will return a null pointer to the invalidator.

2. It forwards the invalidation to its downstream neighbor (Figure 3.16, transaction 2). If the downstream node happens to be an SCI node and responds with a new pointer, the agent

proceeds with invalidating the next node.

3. It attaches itself to all its child lists as a virtual head and starts invalidating them using the SCI invalidation algorithm (Figure 3.16C, transactions 2,3,4,5, and 6). The attach operation is necessary because the head nodes of the child lists—falsely—believe that they are connected directly to the actual memory directory and they only expect a new head node to invalidate them.

When the agent is done invalidating its child lists it waits until it becomes the tail of its list. This will happen because it will either invalidate all its downstream nodes, or they will rollout by themselves (if they are GLOW agents). When the agent is childless and is the tail of its list it invalidates itself and rollouts from its upstream neighbor, freeing it to invalidate itself (Figure 3.16D, transactions 7 and 8).

Full request forwarding—Assuming that all nodes in the sharing tree can forward requests, a more efficient invalidation algorithm can be applied. Full request forwarding invalidation works in two steps: (i) invalidation distribution down the tree, and (ii) acknowledgment return to the root. Both GLOW agents and SCI caches, upon receiving an invalidation message, forward it to their downstream neighbor. When the invalidation messages reach the tails of the lists, the tails invalidate themselves and send an acknowledgment back. GLOW agents wait for the acknowledgment of all the invalidation messages they forwarded, invalidate themselves, and return their own acknowledgment. In a well-structured tree, invalidation is fast because all the messages are exchanged locally, among nodes within the same ring. Note, for a scheme that can exploit broadcast effectively, confining lists to a single ring offers potentially significant reductions both in message traffic and invalidation latency. A similar algorithm can also be used to implement an update protocol where updates are serialized at the root of the tree. New values are distributed down the tree and their receipt is confirmed with acknowledgments back to the root.

3.5.5 Latency of reads and writes

GLOW is not a protocol *per se,* but rather a set of extensions to other protocols. To demonstrate how GLOW is applied to *unmodified* SCI I use partial request forwarding for invalidation as the base case for all evaluations. In this section, I use micro-benchmarks—small programs that generate controlled access patterns—to study how GLOW affects read and write latency. Furthermore, I demonstrate the performance difference of partial and full request forwarding for writes.

***Reads*—**To measure the latency of reads I use a small program in which a number of nodes are chosen at random to read a shared variable. All nodes participating in the read start at the same time. The process is repeated with many different groups of nodes reading the widely-shared data. The percentage of nodes reading varies from 100% (all nodes) to 50%, 25%, and finally 12.5% (one eighth of the nodes reading). The read latency reported in the results is the *average* latency experienced by the readers.

This micro-benchmark is nearly the worst case scenario for SCI and the case where GLOW is most effective. Nodes read simultaneously after a barrier, creating considerable contention in the memory directory and in the network. The SCI sharing list is created by nodes attaching one in front of the other and the data are distributed serially from the tail node to the head node. In contrast, GLOW creates the sharing tree in parallel and distributes the data in parallel.

Figures 3.17 and 3.18 show micro-benchmark results in 2- and 3-dimensional topologies, respectively. The top two graphs of each figure show the latency of read requests for SCI (left graph) and GLOW (right graph). In each graph, the four curves correspond to 100%, 50%, 25%, and 12.5% of nodes sharing. In SCI, read latency increases dramatically from 16 to 128 nodes and the increase is most prominent when 100% of the nodes share. In fact, for 128 node systems, the latency when all nodes are reading is more than twice the latency when half of the nodes are reading. GLOW significantly reduces the read latency, but for the 2-dimensional topology (where the degree of the GLOW trees is at most two) GLOW's read latency increases for large systems and for 100% sharing. Going to the 3-dimensional topology we observe that both SCI and GLOW show lower read latencies. However, GLOW benefits considerably more than SCI: not only is the 3-dimensional network better overall, but the GLOW trees have a degree of three, offering greater parallelism. These observations are supported by the bottom graphs of Figures 3.17 and 3.18. These graphs show the ratio of the SCI latency to the GLOW latency. For example, Figure 3.17 shows that GLOW is actually slower than SCI for 16 node systems when only 12.5% of the nodes are reading (2 nodes), while it is up to 4.4 times faster than SCI for 128 node systems when 100% of the nodes are reading. The ratio lines curve upwards, signifying that GLOW is increasingly better than SCI in larger systems.

Finally, the GLOW latency curves (and as a consequence the ratio curves) are not smooth but rather they show slight variations as the size of the system changes. This is a characteristic of GLOW that is also evident in the performance of programs. The reason for the slight variations has to do with network asymmetry (*e.g.*, the 32 node 2-dimensional topology is an asymmetric 8 by 4 mesh) and data placement, which defines the orientation of the GLOW trees in the net-

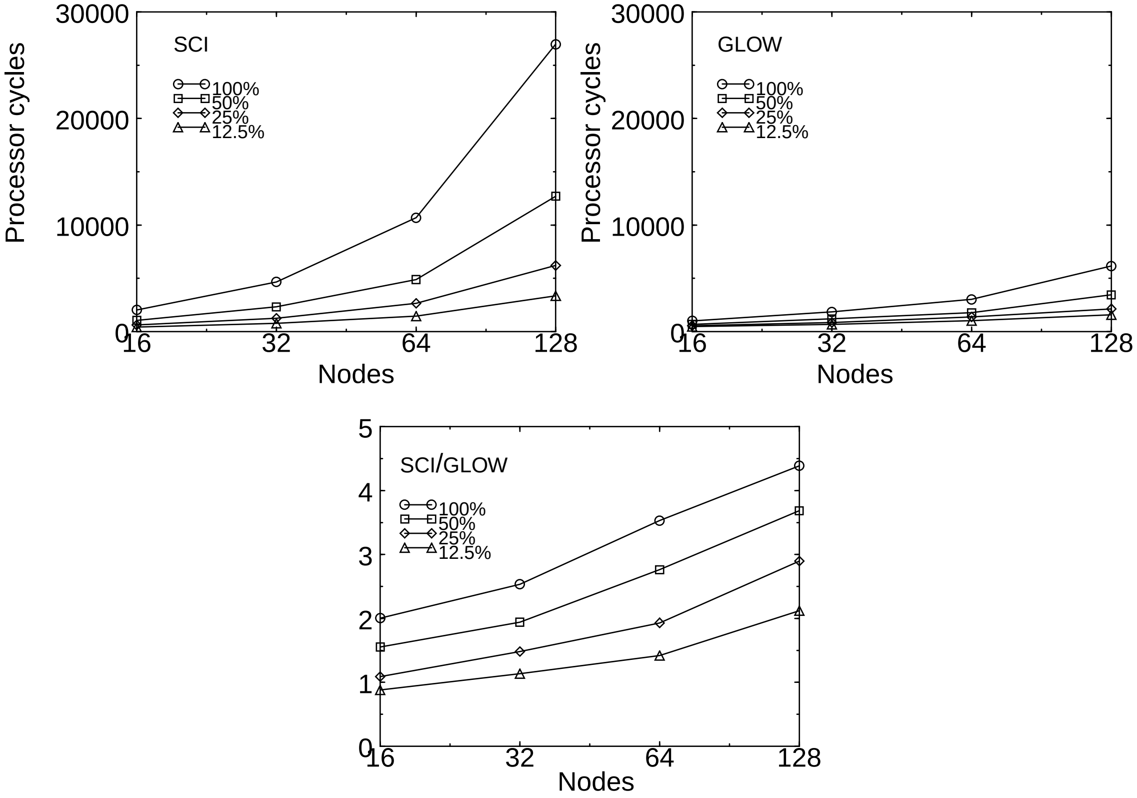


Figure 3.17. Read latency micro-benchmark results in 2-dimensional topologies.

work. In Figures 3.17 and 3.18, GLOW shows a small dip in performance for 32 node systems. This is because the widely-shared data are allocated in node 2 and the resulting GLOW trees comprise 4 agents, each with 8 children, instead of the higher-performing configuration of 8 agents, each with 4 children.

Writes**—**To measure latencies for writes I used the simple loop shown in Figure 3.21. In each iteration a different writer node writes the data. Similar to the micro-benchmark for reads, the number of nodes reading after the second barrier (and thus, the size of the sharing list or sharing tree being invalidated) varies from 100% to 50%, to 25%, and to 12.5%. Nodes are chosen at random to read the data and the process is repeated multiple times, each time with a differ-

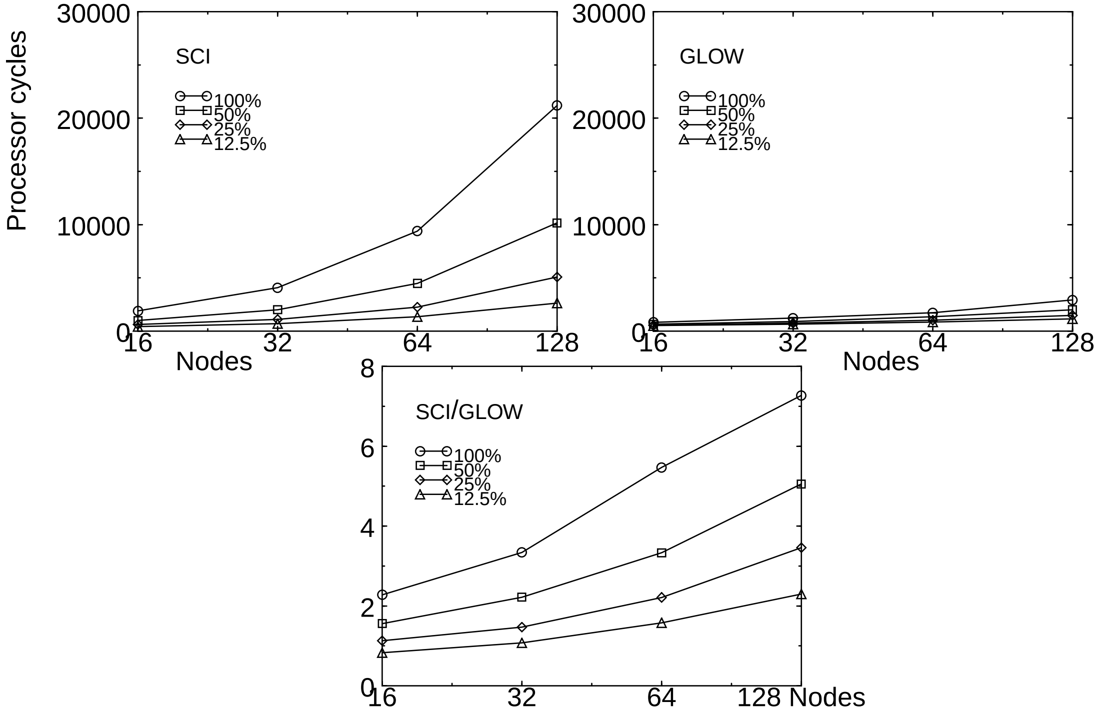


Figure 3.18. Read latency micro-benchmark results in 3-dimensional topologies.

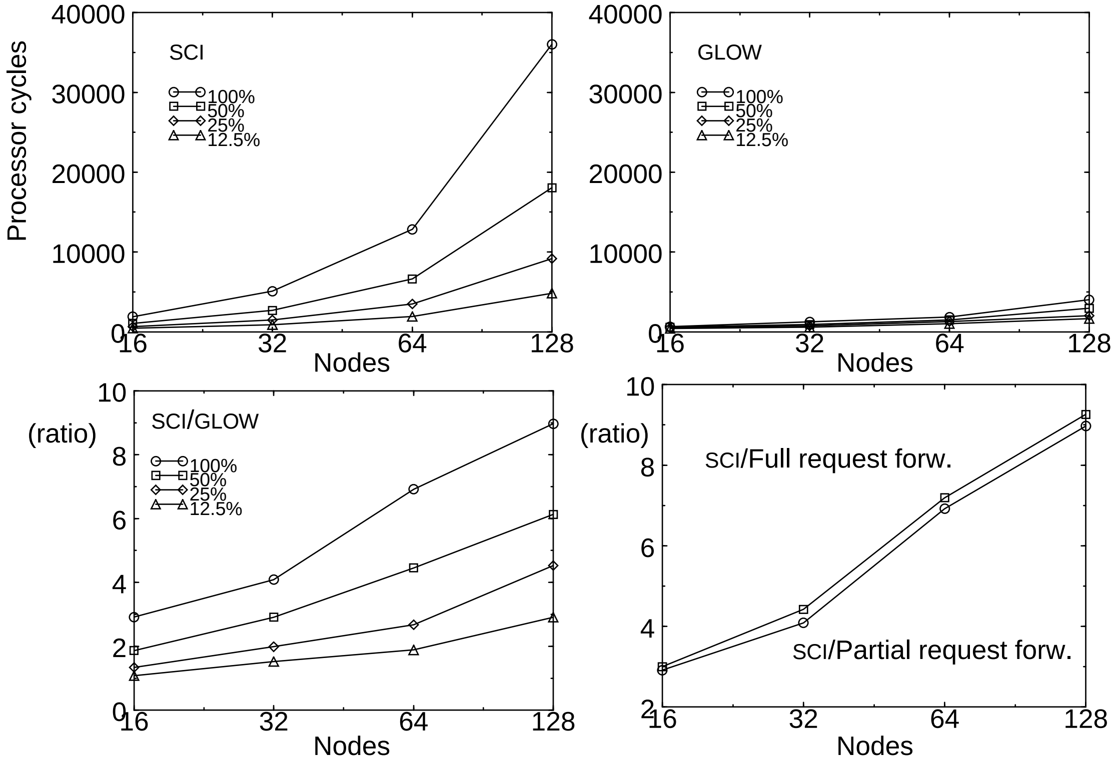


Figure 3.19. Write latency results in 2-dimensional topologies.

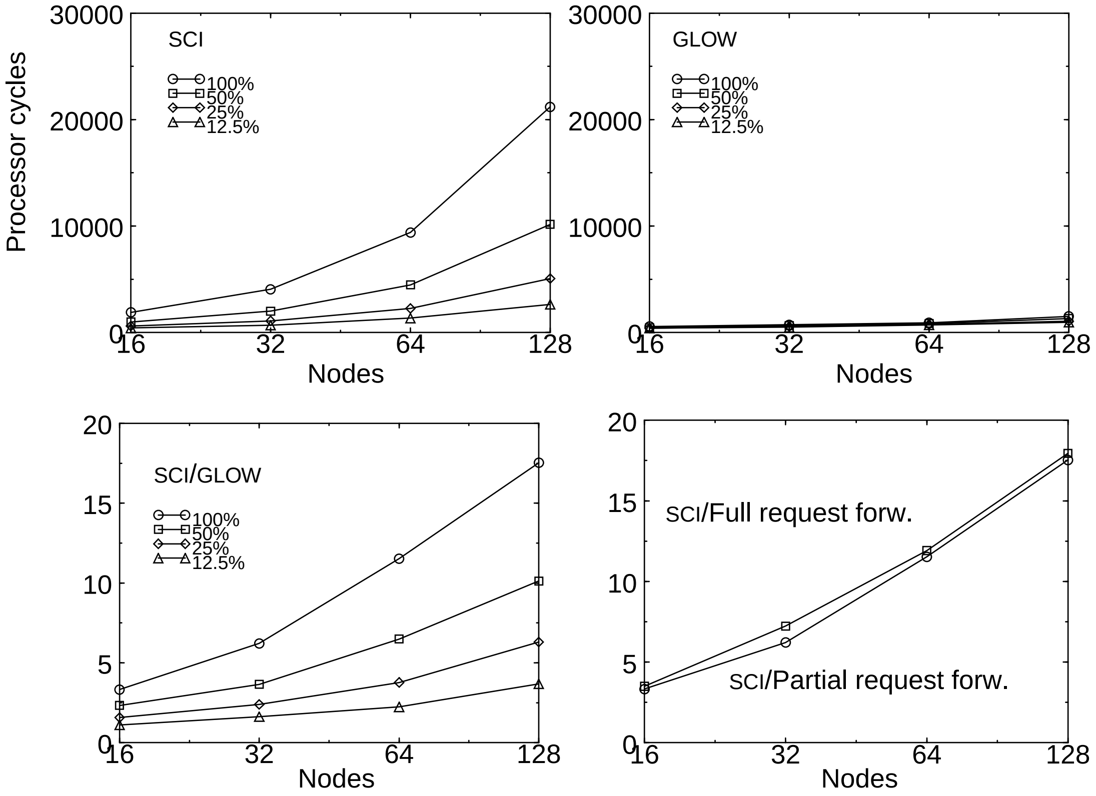


Figure 3.20. Write latency results in 3-dimensional topologies.

```
void node_code(){
  repeat{
    choose a random set of readers
    if (I belong to the readers) read widely-shared data;
    for(i=0;i<num_nodes;i++){
      barrier();
      if (I am node # i) write widely-shared data;
      barrier();
      if (I belong to the readers) read widely-shared data;
}}}
```

Figure 3.21. Micro-benchmark for writes.

ent set of random nodes. Write latency results are averaged for many different writers and many different runs.

Figures 3.19 and 3.20 show results for 2- and 3-dimensional topologies respectively. Similar to reads, average write latency for SCI grows very fast with system size, especially when all nodes read before the write. GLOW significantly reduces this latency for the 2-dimensional topology and even more for the 3-dimensional topology. The ratio of SCI latency to GLOW latency (bottom graphs of Figure 3.19 and Figure 3.20) shows that as we go to larger systems the GLOW latencies grow increasingly slower compared to the SCI latencies. For 128 node systems where all nodes share the data, the GLOW write latency is 17.5 times less than that of SCI.

Finally, the bottom-right graphs of Figures 3.19 and 3.20 demonstrate the performance difference for writes when partial or full request forwarding is used. The graphs show results for the case when all nodes are reading, which is also where the performance difference of the two schemes is most pronounced. The two curves represent the ratio of SCI latency to GLOW latency for the two schemes. Full request forwarding is faster than partial request forwarding (from 1.03 to 1.08 in 2 dimensions and from 1.02 to 1.16 times in 3 dimensions). This performance difference has little to do with the behavior of the two schemes in terms of messaging since both partial and full request forwarding generate the same number of messages. The performance difference lies in the utilization of the GLOW agents and the writer node. With Partial request forwarding, GLOW agents are responsible for sending all the invalidation messages to their children (and for receiving all responses). This drives the utilization of the GLOW servers up and creates queueing delays. In contrast, with full request forwarding, the responsibility for sending messages is distributed to the children and the response of the GLOW agents is better. In general, operations involving many massages processed by a single server lead to hot spot phenomena.

3.5.6 Replacements in sharing trees

An ordinary node leaves the tree (*rolls out)* either because of a cache block replacement or to become head-of-list for writing the data. For an SCI node the standard SCI protocol is applied for rollout. GLOW agents may roll out because of conflicts in their directory/data storage or because they are left childless. Childless agents are not permitted in the tree, unless they also cache data. The SCI protocol does not permit nodes already in the sharing list to issue a second request, since this would put them in the list twice. Thus, when the last child rolls out, the agent also rolls out. When an agent finds itself childless it is only connected to the sharing tree with its forward and backward pointers. In this case the rollout is the standard SCI rollout.

When an agent with children must roll out because of a conflict in its directory storage, it must first deal with its child lists, and then roll out as before. Two approaches for dealing with the child lists are described below. When the first is used, the rollout is called *destructive* because it destroys parts of the tree. When the second is used, the rollout is called *linearizing* because it connects the child lists into an extended linear list.

DESTRUCTIVE ROLLOUT**—**This simple solution is borrowed from hierarchical caches that enforce multilevel inclusion. When an agent rolls out, it invalidates all its descendents using the invalidation algorithm described in Section 3.5.4. When the invalidation completes, the agent is childless and it can roll out as usual. This scheme has great appeal because of its simplicity. However, it can lead to thrashing behavior when the amount of widely-shared data in the system exceeds the directory capacity of the agents. Under such circumstances, nodes that actively share a data block might be invalidated, thus immediately repeating their requests for the data. These new requests will lead to more conflicts in the GLOW agents. This behavior is difficult to avoid because agents do not have information about the access frequency of their children. Thus, it is impossible for an agent to ascertain whether its children still access the data or have long before ceased to. Any replacement strategy for the agents' directory caches is based only on information available at the time the sharing tree was created. For example, if the Least Recently Used (LRU) replacement strategy is used, the LRU information concerns only agent activity (at creation time) and not the access activity of their children. Nevertheless, this scheme is viable because in many programs the amount of widely-shared data actively accessed at any point in time is low.

***LINEARIZING ROLLOUT*—**This scheme attempts to preserve the sharing tree as much as possible, degrading the structure of the tree gracefully. This is possible because GLOW does not require multilevel inclusion. LINEARIZING ROLLOUT is based on concatenating the child lists (*i.e.*, chaining them tail-to-head into a single, linear list) and subsequently substituting the concatenated list in place of the agent in the tree. The head of the first child list connects to the tree in the position of the agent's backward pointer. The tail of the last list connects to the tree in the position of the agent's forward pointer. The chaining process involves only the head and tail nodes of the child lists. The process is shown in Figure 3.22. A part of a sharing tree is shown in Figure 3.22A. The agent in the middle rolls out. It first attaches itself in front of all its child lists as a virtual head by sending attach requests (Figure 3.22 A). Temporary pointer storage is required for the agent to appear as multiple virtual heads. However, this pointer storage is allocated only for rollouts and need not be part of every glow entry. When the agent becomes virtual head in all its child lists, the previous heads become middle nodes. All virtual nodes (virtual heads and virtual tails) are in reality only one entity (the agent itself) and there is no change in the pointers of any node. Since the virtual tail of a child list and the virtual head of the next child list are the same node (the agent) they rollout atomically, as one, and leave the two child lists connected into one (Figure 3.22 B and 3.22 C). In this way, any number of child lists can be concatenated into one list in one step. Concurrently, the agent rolls out as virtual head and virtual tail of the concatenated child lists (Figure 3.22 C). The sharing tree after the agent rollout is shown in Figure 3.22 D.

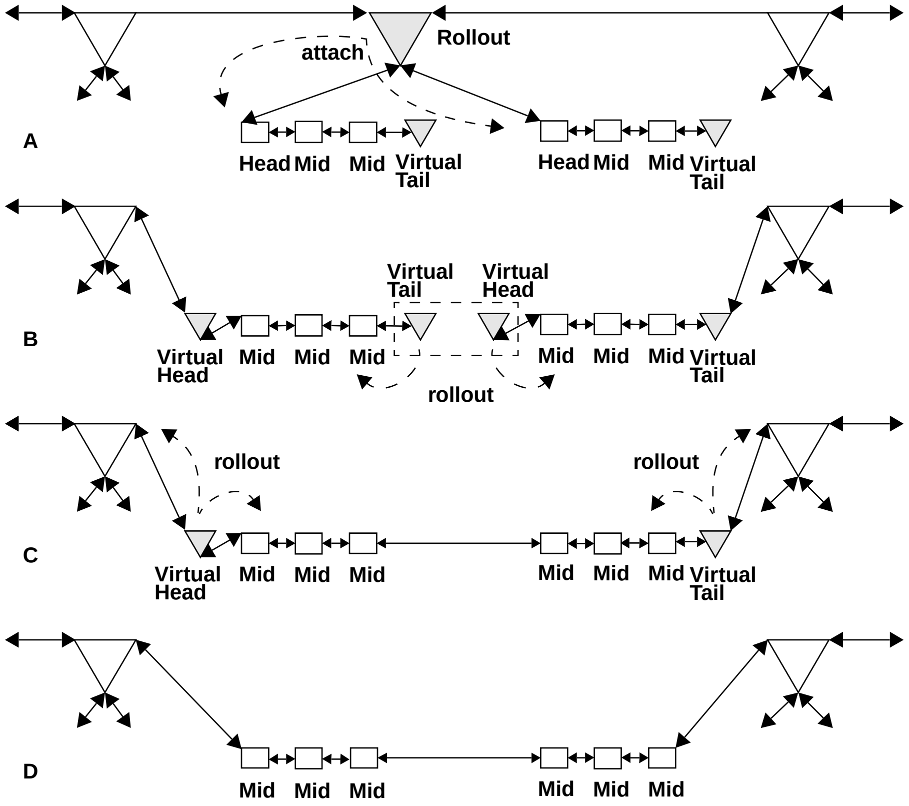


Figure 3.22. LINEARIZING ROLLOUT.

In LINEARIZING ROLLOUT, when an agent rolls out and later rejoins the tree, the relation of the agent and its children is lost at the point of the rollout. If the agent rejoins the tree, it may do so in another position in the linear list of the appropriate hierarchical level. New requesting nodes that would otherwise become heads in the old child lists will join newly created child lists under the agent. Although the sharing tree degrades over time as the agents leave and rejoin, leaving child lists scattered over multiple rings, this scheme is potentially more effective than DESTRUCTIVE ROLLOUT. Two arguments support this claim. First, there is minimal interference to other nodes: while in DESTRUCTIVE ROLLOUT many shared copies that are potentially in use are invalidated, LINEARIZING ROLLOUT requires only the participation of the head and tail nodes to complete the concatenation of the child lists. Second, the latency of invalidating a subtree can be much higher than the latency of chaining the child lists together and substituting the agent. Therefore, replacements in the agent's directory storage can be much faster. The performance difference between the two schemes becomes apparent only when thrashing is caused by very small directory caches in the GLOW agents. In such situations, DESTRUCTIVE ROLLOUT hurts performance significantly.

3.5.7 Replacements and GLOW cache size

In this section, I show that the performance of GLOW does not change significantly, even if the size of the GLOW caches is considerably reduced. However, for very small cache sizes, GLOW agents start to thrash and DESTRUCTIVE ROLLOUT creates havoc in the system. Although LINEARIZING ROLLOUT and DESTRUCTIVE ROLLOUT do not show a performance difference with larger GLOW caches, in the extreme case (thrashing) LINEARIZING ROLLOUT behaves much better.

The replacement rate in the GLOW agents depends on the amount of widely-shared data in a program and the size of the GLOW directory caches. Typically, at any point in time, widely-shared data are a small percentage of the data set of a program. For example, in GAUSS the whole data set is potentially widely shared, since all rows can be pivot rows but only one row is widely shared during an iteration of the algorithm. SPARSE needs at least 64 GLOW entries for the widely shared vector **X** and a few more for the widely shared variables that are frequently written. Although the widely-shared data in APSP depend on the input, its replacement rates are low. This characteristic of widely-shared data suggests small sizes are best for the GLOW directory caches.

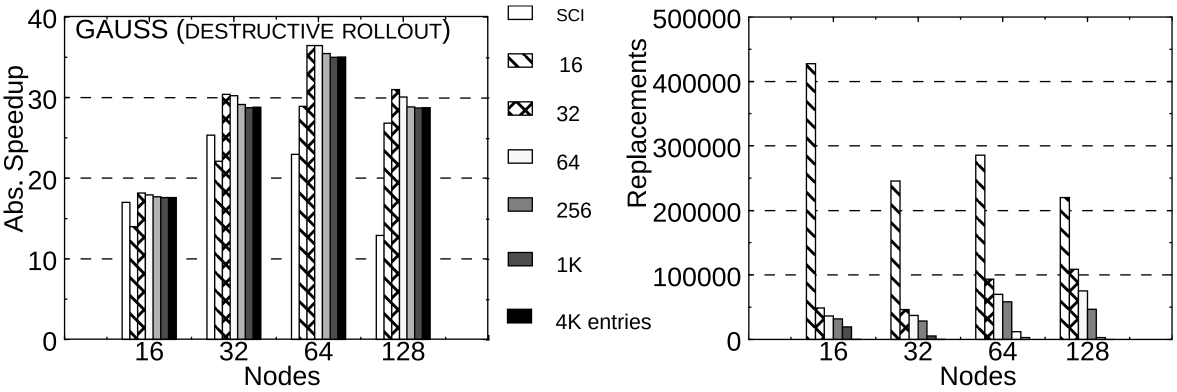


Figure 3.23. Sensitivity to directory cache size for GAUSS.

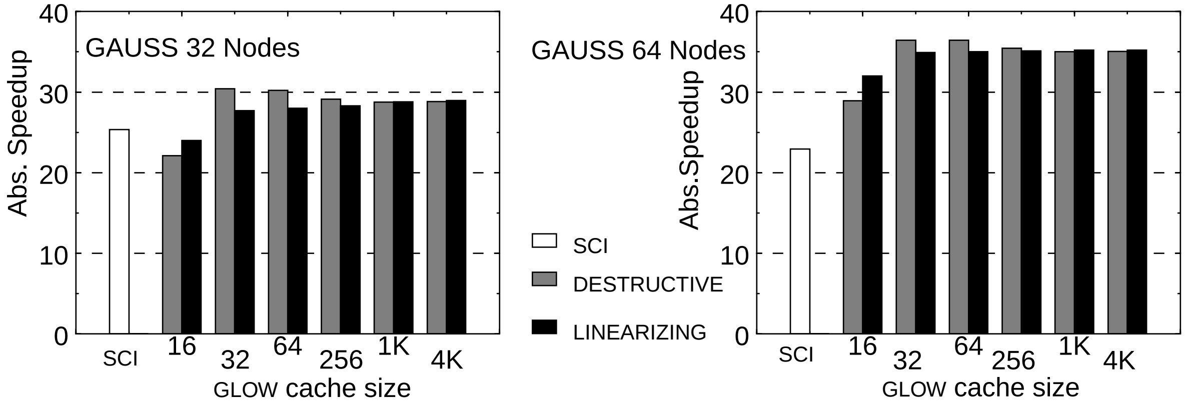


Figure 3.24. DESTRUCTIVE vs. LINEARIZING ROLLOUT for GAUSS.

Here, I show results for the DESTRUCTIVE and the LINEARIZING ROLLOUT, for three programs (GAUSS, SPARSE, APSP) running in 16-node to 128-node systems with a 2-dimensional topol-

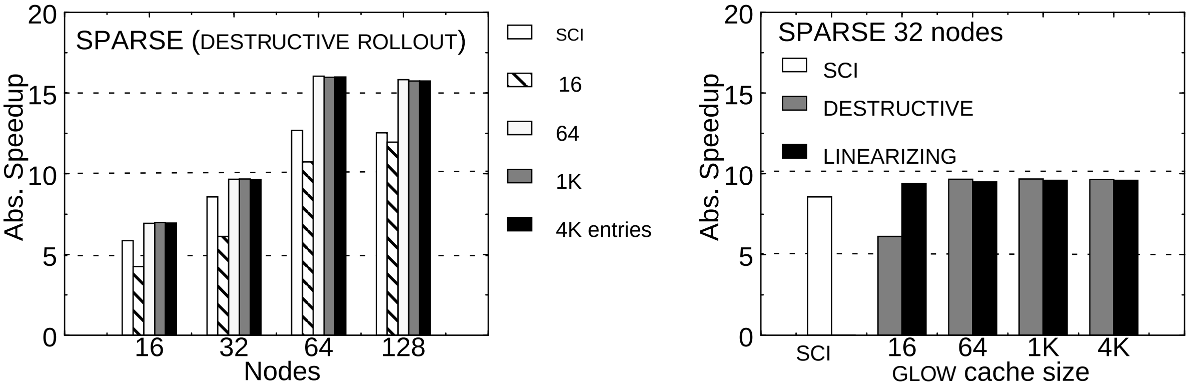


Figure 3.25. Sensitivity to directory cache size for SPARSE.

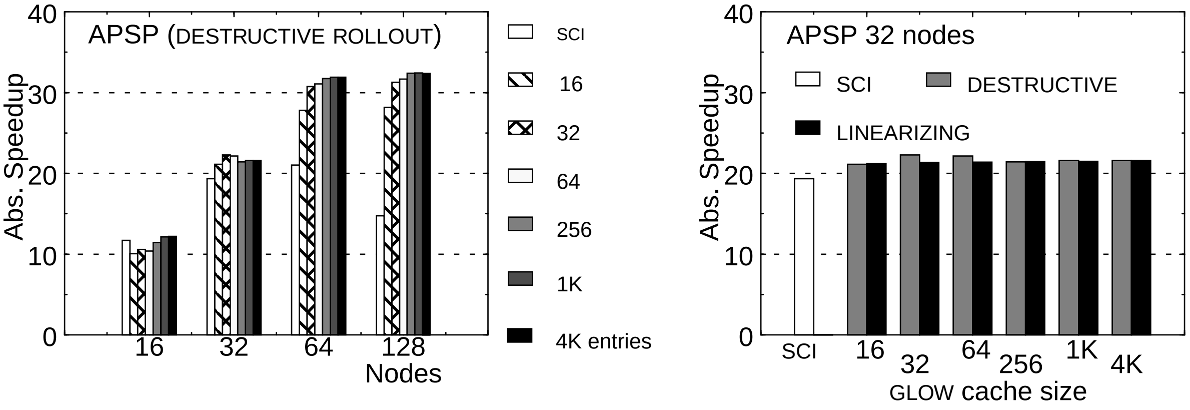


Figure 3.26. Sensitivity to directory cache size for APSP.

ogy (figures 3.23, 3.25, and 3.26). GLOW is used to access widely-shared data that are defined by directives in the source program. I examine four-way set-associative directory caches with 16, 32, 64, 256, 1024, and 4048 entries, and show the effects on performance.

DESTRUCTIVE ROLLOUT**—**The cache size seriously affects performance only if it drops under a critical size (under 32 entries for GAUSS, under 64 entries for SPARSE, and under 32 entries for APSP). At this point, the large number of replacements (which delete entries from the sharing nodes) kills performance, especially in the 16- and 32-node cases where GLOW does not offer significant performance improvements, even with large directory caches. However, in the 64- and 128-node cases, even with a very small cache (and therefore a large number of replacements), performance improvements are possible for GAUSS and APSP (figures 3.23, and 3.26). In SPARSE (Figure 3.25), DESTRUCTIVE ROLLOUT with a very small cache creates an adverse effect: when the vector ***X*** does not fit in the GLOW agents, DESTRUCTIVE ROLLOUT continuously invalidates sharing nodes forcing them to repeatedly request new copies of ***X*** with detrimental effects on performance. Without GLOW, elements of ***X*** would remain cached in the sharing nodes. An interesting effect of DESTRUCTIVE ROLLOUT is that its performance improves as the cache size decreases and suddenly (crossing a critical cache size) its performance drops tremendously. The performance increase as the cache size decreases is due to a *purging effect*. This is noticeable in all node configurations for GAUSS and for the 32-node configuration for APSP, but it is practically absent in SPARSE. The purging effect occurs when replacements in the GLOW agents destroy parts of the sharing trees that are no longer needed. When the widely-shared data are written, only a small part of the sharing tree remains to be invalidated. Thus, the purging effect manifests as a reduction in write latency.

LINEARIZING ROLLOUT**—**This scheme maintains the sharing nodes by gracefully degrading the structure of the sharing tree. Thus, it does not exhibit a purging effect similar to that of DESTRUCTIVE ROLLOUT. With very small cache sizes LINEARIZING ROLLOUT outperforms DESTRUCTIVE ROLLOUT because it does not interfere with sharing nodes. For SPARSE the performance of LINEARIZING ROLLOUT does not drop at all (Figure 3.25). Even when the read-only vector ***X*** does not fit in the GLOW agents, its copies in the sharing nodes remain valid (unlike the DESTRUCTIVE ROLLOUT case). As long as the GLOW agents can cache the few widely shared variables that are frequently written the full performance gain is obtained. For GAUSS (Figure 3.24) and APSP (Figure 3.26) the performance of LINEARIZING ROLLOUT drops slowly as the cache size decreases. In these cases, although agent rollouts are faster than those of DESTRUCTIVE ROLLOUT, the large number of replacements affects the ability of the GLOW agents to function efficiently. Furthermore, using LINEARIZING ROLLOUT, the write latency consistently worsens as the cache size decreases because sharing trees degrade to sharing lists.

Since both rollout algorithms work equally well with larger GLOW caches, I use DESTRUCTIVE ROLLOUT for all subsequent evaluations. I also use a directory cache size that accommodates the same number of entries as the number of cache blocks in an SCI cache with the same associativity. Note that there can be no more than *N* different *full* sharing trees in the system, where *N* is the number of lines in an SCI cache. Thus, for systems with 64KB SCI caches with 64-byte cache blocks, I use 1K-entry directory caches in the GLOW agents. Since a GLOW entry can be compressed to a much smaller size than a 64-byte line, the size of the GLOW caches is a fraction of the size of the SCI caches.

3.5.8 Update of sharing trees

An opportunity for more performance is to use an update protocol instead of an invalidation protocol. Update protocols are not always a clear win since they suffer from unnecessary updates [18]. These unnecessary updates can be classified into updates that are overwritten by subsequent updates without an intervening read and updates to the wrong nodes (*i.e.*, nodes that are no longer interested in the data block). Solutions that eliminate some of the unnecessary updates of the first class have already been proposed (*e.g.*, coalescing buffers or write caches [46]). For the second class of unnecessary updates, other solutions have been proposed, such as *competitive update protocols,* where cache blocks are invalidated after receiving a number of updates without an intervening read [13]. Additionally, special care must be taken to ensure that update protocols support sequential consistency [61] (discussed in the next section).

With a GLOW update protocol, a sharing tree is constructed and subsequently used to distribute updates. There are two variations of the update protocol. The first, full update, updates both the agents and the SCI nodes in the tree. The second, partial update, assumes that only the GLOW agents are updated. The SCI nodes in the sharing tree are invalidated and have to request the data again.

Full Update—The SCI coherence protocol can be extended to update rather than invalidate a sharing list. Depending on whether only the GLOW agents or all the nodes support request forwarding, the update algorithm can take one of the following two forms:

- Full update with partial request forwarding: The update algorithm resembles the standard SCI serial invalidation algorithm, but it sends update messages (carrying the new data) instead of invalidation messages. The writer sends an update message to a node in the list (starting with the first node after the writer) which, upon receiving the new data, responds with the pointer to the next node. The GLOW agents use this update algorithm on their child lists to propagate the updates down the GLOW sharing tree. Amongst themselves, the agents forward the updates as described in Section 3.5.4 for the invalidations.
- Full update with full request forwarding: A writer node becomes the root of the tree and sends updates down the tree. When updates reach the leaves of the sharing tree, acknowl-

edgments are returned to the root. The root node prohibits any other node from becoming root until it receives all required acknowledgments.

Similar to the way the invalidation algorithm is affected by full or partial request forwarding (see Section 3.5.5 for the relevant data), full update with partial request forwarding is slightly slower than full update with full request forwarding. Again this is because in the former, the GLOW agents have to send and receive many messages (half of them carrying data), while in the latter the same work is distributed over many nodes. In Chapter 4, Section 4.2.5, I present results for the full request forwarding scheme.

***Partial Update*—**A partial update of the sharing tree means that only the GLOW agents will be updated, while the SCI nodes will be invalidated. In this way, the update traffic is decreased by updating only a few GLOW agents and by bringing the data closer to nodes that might potentially access them in the future. In this scheme, update messages (carrying the new data) are used only for GLOW agents, while invalidation messages are sent to the SCI nodes. This method is complex because it requires knowing beforehand whether a node is a GLOW agent or an SCI node in order to send the correct message (update or invalidation). In effect, this requires abandoning the transparency of the GLOW agents. A similar method, *eager combining,* was studied by Bianchini and LeBlanc [17] in the context of software DSM.

3.6 Memory consistency and GLOW

In this section I discuss the relationship between the GLOW extensions and memory consistency models. In Section 3.6.1 I briefly review memory models and in Section 3.6.2 I discuss the relative importance of GLOW under different memory models. Update protocols violate the simple memory model called *sequential consistency* (SC) [61] that is typically assumed by programmers [38]. In Section 3.6.3 I describe what is needed to make GLOW update compatible with SC.

The strength of the GLOW extensions lies in their ability to reduce both read and write latencies for a particularly expensive sharing pattern. As I have shown in Section 3.5.5, the read and write latencies when accessing widely-shared data can be excessively large because of hot-spot phenomena and inefficient use of network bandwidth. The GLOW approach, as well as other work dealing with wide sharing, is a direct approach that targets the actual read and write latencies.

There are other indirect approaches that try to reduce the *apparent* read and write latencies by overlapping coherence transactions with other work. This is commonly referred to as “hiding” the latency. For reducing the apparent read latency, prefetching has been extensively researched. For reducing the apparent write latency, overlapping writes with other operations has been proposed.

3.6.1 Memory models

Lamport first described how to build a multiprocessor system that correctly executes multiprocessor programs as a system that behaves as a multiprogrammed uniprocessor. This restricts the behavior of the memory system to a specific model called *sequential consistency* (SC). In SC: “*the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified in its program*” [61].

SC poses restrictions in the design of a multiprocessor system to support it. A simple implementation of SC is to require a processor to stall until a memory operation has been completed globally. Adve and Hill showed that this is a sufficient condition to guarantee SC but not a necessary condition [5]—more aggressive implementations exist which still provide a sequentially consistent memory model. However, since the WWT allows only one outstanding coherent memory operation, I enforce SC using this simple condition.

To avoid restrictions imposed by SC on the order of memory requests, several relaxed memory models have been proposed, including: weak ordering, release consistency (RC)[32], and data-race-free-0 [6]. The relaxed memory models overlap writes with other operations, hiding the write latency. However, certain restrictions must be imposed at synchronization points to guarantee correct execution. Thus, synchronization points typically require the completion of outstanding operations (either all operations or just specific operations denoted by the user [32]), before allowing the processors to proceed.

3.6.2 GLOW and relaxed memory models

One of GLOW’s goals is to provide fast and scalable writes. Relaxed memory models also affect the (apparent) latency of writes. The issue then is whether a relaxed memory model reduces the relative value of GLOW by speeding up writes. In this thesis I argue that GLOW is useful for programs with wide sharing regardless of the memory model.

The latencies involved in accessing widely-shared data are extremely large, as shown in Section 3.5.5. No matter how aggressively the memory model can overlap writes with other operations, it is unlikely that there will be enough operations to cover latencies on the order of tens of thousands processor cycles (when an ordinary memory transaction is on the order of a few hundred processor cycles). In this case, benefit could come from overlapping multiple writes *by the same processor* to widely-shared data, each of which would incur a significant latency. Because in SCI it is the writer that is responsible to invalidate the sharing list, allowing multiple writes to widely-shared data to proceed concurrently makes sense only when the node can handle the protocol processing overhead (sending and receiving many messages). For the systems I simulate and for the six programs I use, I have found that this is not the case. For other protocols such as $\mathrm{Dir}_n\mathrm{NB}$ or DASH, allowing multiple writes (to widely-shared data) by the same processor to proceed concurrently would be a win, if different home-node directories were involved for each invalidation (thus achieving a significant overlap). Additionally, results in Section 4.2.3 indicate that read latency is very important for overall performance and GLOW can offer significant performance improvement under any memory model.

In Section 4.2.4 I present simulation results using a relaxed memory model for both SCI and GLOW. The results show that a relaxed memory model helps both SCI and GLOW. However, when it comes to wide sharing, a relaxed memory model cannot compete with the overall performance improvement offered by GLOW. Thus, GLOW with sequential consistency is significantly better than SCI with a relaxed memory model. Since SC is the natural memory model for programmers, the combination of GLOW and SC is very appealing because of its transparency. This result supports Hill's view that multiprocessor systems should implement sequential consistency [38].

3.6.3 Update protocols and memory models

Update protocols can violate SC because different processors can observe writes in different order. A technique to make an update protocol sequentially consistent, is to use a two-phase commit algorithm so that all nodes are stalled until an update is performed globally. Intuitively, this imposes an order among concurrent updates. In GLOW this means that when a writer wants to update a sharing tree it must initiate an operation to lock all the caches and GLOW agents in the sharing tree so that read requests cannot be satisfied. In the meantime the writer does not let any new node read or write the data by refusing to allow a prepend in front of the sharing tree. After the writer receives acknowledgment that all caches and GLOW agents have been locked it can update the data. The caches and GLOW agents are unlocked and freed to process requests as the acknowledgments are returned to the root (the writer). This process would significantly complicate the update algorithm. Corner cases are especially troublesome. The locking phase must take into account that GLOW agents may have already accepted a request and cannot be locked until it is satisfied. The performance of such an update is also questionable because the sharing tree must be traversed twice.

Because of the complexity of a sequentially consistent update protocol and because of its questionable performance benefit, I did not implement it. Instead, I implemented the update algorithm described in Section 3.5.8 to conform to a relaxed memory model. The implementation stalls a writer at synchronization points until an outstanding update has been acknowledged by all nodes in the sharing tree. This implementation works for programs which implement correct synchronization and are data-race-free. In Section 4.2.5, I report results for this update implementation. The results show that an update protocol offers little performance improvement over the invalidation protocol. This further enhances the argument that it is unlikely that a sequentially-consistent update protocol will be cost-effective.

3.7 Other possible GLOW implementations

In this section I discuss possible applications of the GLOW extensions to other cache coherence protocols besides SCI. In Section 3.7.1 I describe how GLOW can be applied to full map protocols such as Dir_nNB [8] and DASH [65]. Although these protocols do not suffer from a serial invalidation algorithm such as that of SCI, GLOW can offer performance improvements because of its scalable reads and because of better utilization of network bandwidth for writes. In Section 3.7.2 I discuss how GLOW can transparently offer pointer storage for limited pointer protocols, thus offering the potential to improve their performance for wide sharing considerably. In Section 3.7.3 I discuss the relation of pruning caches [88] and GLOW. Finally, in Section 3.7.4 I discuss how GLOW can be applied in software-based shared memory.

3.7.1 Full-map directory protocols

GLOW can implemented on top of full-map cache coherence protocols (*e.g.*, Dir_nNB, or DASH) but some constraints are required to satisfy the different node addressing methods of these protocols. The node addressing problem and the constraints it imposes are discussed in detail below. With respect to performance, the full-map protocols do not face a performance degradation comparable to that of SCI for writes, where the invalidation algorithm is serial. However, GLOW is also applicable to these protocols since it can offer significant benefits from request combining and reduction of the load in the network by using the tree structure for invalidations. This claim is supported in Chapter 4, Section 4.2.3, where I present data indicating that the performance improvement from GLOW's scalable reads is more important for overall performance than the performance improvement from scalable writes.

3.7.1.1 The agent addressing problem in full-map protocols

In any GLOW implementation, the GLOW agents must be uniquely identified, so direct messages can be sent to them (*e.g.*, invalidation or replacement messages). In SCI this is easily solved by assigning the GLOW agents their own unique *node addresses* or *node identifiers* (IDs). This is transparent, since SCI uses pointers (not bit-maps as the full-map protocols do) that can address up to 65536 nodes. A simple partitioning of this address space is sufficient to distinguish simple nodes from agents. When, in an SCI implementation, only a few pointer bits are implemented instead of the full 16 bits, as defined by the standard [41], the overhead required to accommodate GLOW agents is minimal. An additional bit in every pointer automatically doubles the node address space.

The situation is different with full-map protocols, where each directory entry must include at least as many bits as the number of nodes in the system. In the general case, increasing the node address space to accommodate agents with their own unique IDs is prohibitive in terms of storage, unless techniques such as dynamic allocation of directory entries—proposed by Simoni and Horowitz [90]—are used. If such techniques are an option and the increased storage requirements to accommodate unique IDs for the agents are not a significant problem, the GLOW extensions for the full-map protocols are similar to the SCI GLOW extensions. Here, however, I will discuss the case where the node address space cannot be increased (no more bits can be added to the directory bit-maps) and I will describe in detail how GLOW can be implemented under this constraint.

3.7.1.2 A sample GLOW design for Dir_nNB

One method to solve the GLOW agent addressing problem in Dir_nNB-type protocols is to unify the GLOW agents with the nodes so that they share the same IDs. This is a compromise that negates one of the characteristics of GLOW: transparency. A $\mathrm{Dir}_n\mathrm{NB}$ system must be designed from the start to accommodate GLOW.

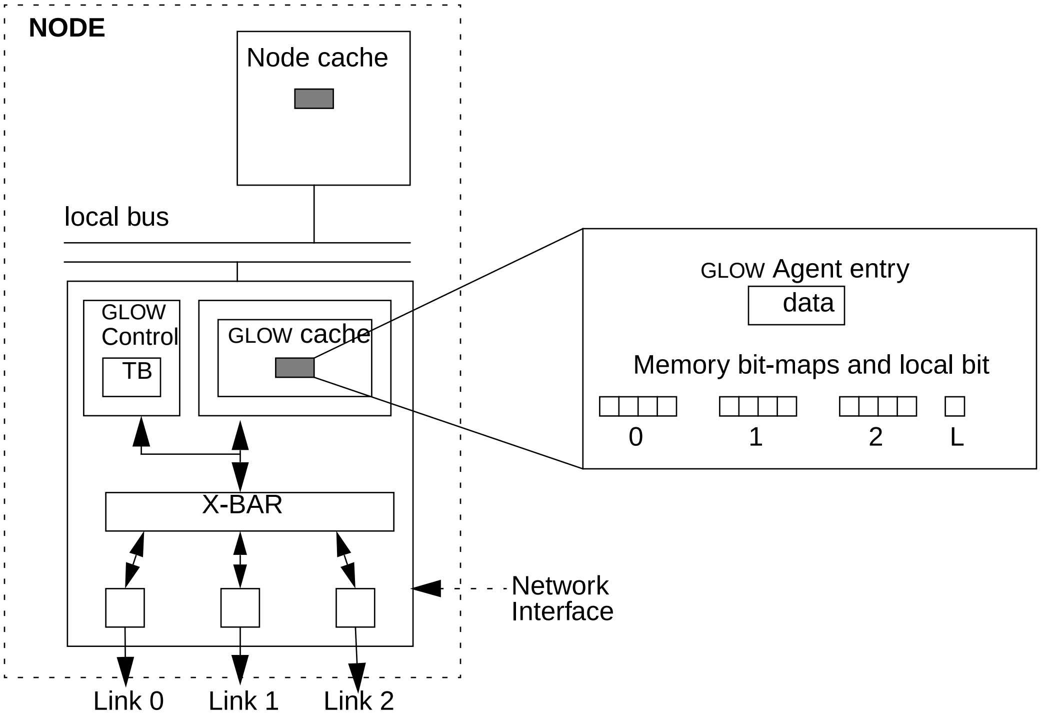


Figure 3.27. Internals of a GLOW agent in a $\mathrm{Dir}_n\mathrm{NB}$-based system. The right scheme is a representation of a GLOW entry in the directory cache.

Figure 3.27 shows the sample design of a $\mathrm{Dir}_n\mathrm{NB}$ GLOW node. The GLOW agent is the switch node that routes messages among the three—in this example—network links. A GLOW entry contains data storage, necessary bit-maps, and state to impersonate three memory directory entries (one for each link—although only two can be active at any time). There is also a bit, L-bit,[2] that indicates that the node cache itself has a copy of the data. Since the agent takes over

[2] The L-bit is actually redundant (it substitutes one bit of one of the GLOW entry's bit-maps) but it is used for algorithmic clarity.

the ID of the node for the specific memory address, this bit is required to indicate whether the node also has a copy of the data (see also the agent replacement algorithm below). The agent must snoop all incoming and outgoing node traffic. In addition, the agent intercepts special GLOW requests that pass through the switch—but not necessarily all of the GLOW requests.

An alternative implementation is to split the GLOW entry into two parts: (i) the part that contains the directory bit-maps and state and, (ii) the part that resides in the node cache and contains the data (in which case no L-bit would be needed). In this implementation, each GLOW entry corresponds to a node cache entry (but the reverse does not necessarily hold). Although this implementation does not replicate the data in two places inside a node, it requires far more transactions between the GLOW logic and the node cache over the node's local bus. This could potentially degrade performance.

Subsequently, I will describe how reading, writing, and agent replacement are performed with $\mathrm{Dir}_n\mathrm{NB}$ GLOW using the agent implementation depicted in Figure 3.27.

Reading—When a GLOW request, passing through an agent, is intercepted, or when the local node sends a GLOW request through the node's agent, the agent allocates an entry (if possible) and sends its own request—using the node ID—toward memory (as in the example of Figure 3.28 A). The agent may ignore any read request to avoid possible deadlocks, if the entry required is taken by another sharing tree and it is in a transitional state (see Section 3.5.3 for the analogous situation in SCI GLOW). In the time it takes for a response to the agent's own request, more requests may be intercepted (as in the example of Figure 3.28 B). Upon receiving the data, the agent satisfies the request of the corresponding nodes indicated in its bit-maps (as in example of Figure 3.28 C).

The local bit, when set, indicates that the local node also has the data (or has requested the data). Because the node cache can obtain a copy of the data in other ways (*e.g.*, using a non-GLOW request), the agent must perform a lookup on the local node cache when it allocates a new entry. *If the data are found locally no new request is sent.* Unless the allocation of GLOW entries is very frequent, the lookup in the node cache—a relatively inexpensive operation—is not going to affect significantly the performance of GLOW, nor the performance of the node's processors.

Using the above technique we guarantee the following invariant for the sharing tree: each node ID is registered in exactly one bit-map (either the home-node's bit map or an agent's bit-map). In other words, the node and the agent that share the same ID are never allowed to be registered in two different places in the sharing tree.

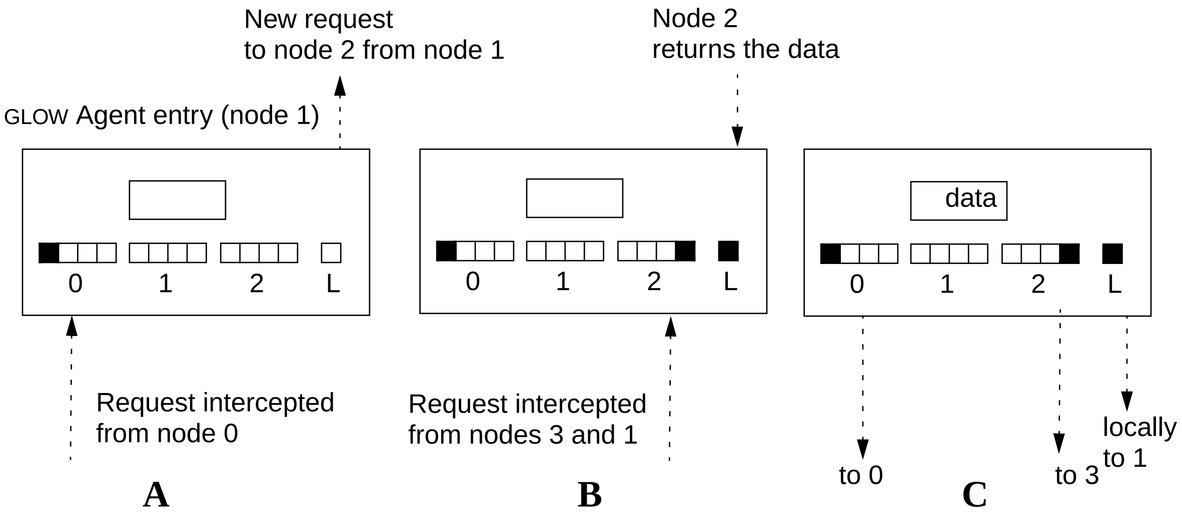
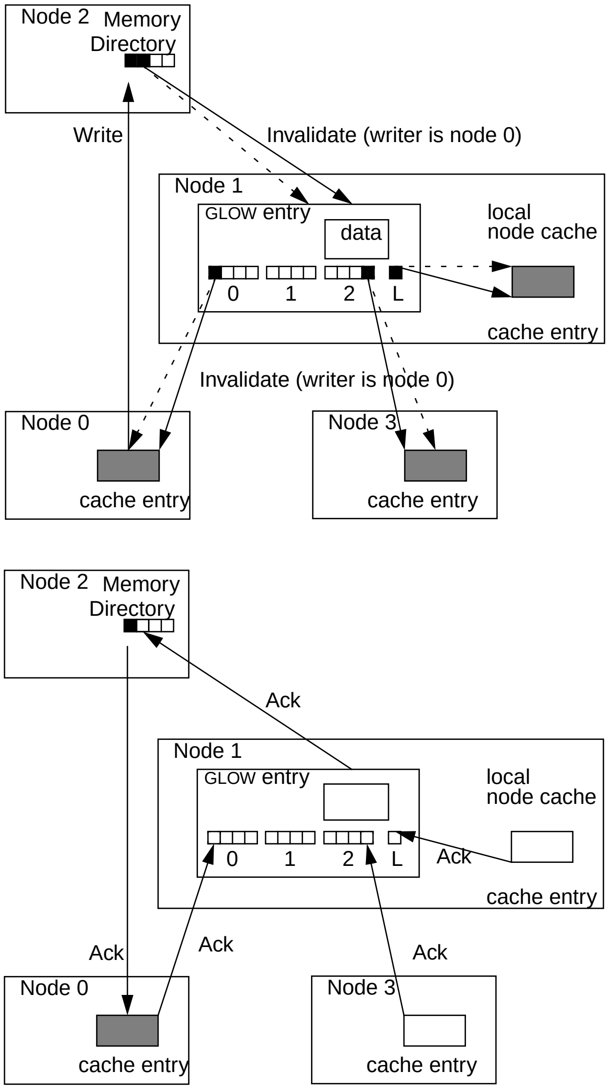


Figure 3.28. Hypothetical example of a four node system. A GLOW agent assumes the identity of node 1 for widely-shared data and intercepts requests from nodes 0 and 3 as well as locally from node 1.



Dotted lines show the sharing tree.
Solid lines denote messages.

Node 0 sends a write request to the home node (node 2) which, in turn, sends an invalidation message to node 1. Node 1 is actually a GLOW agent that has serviced nodes 0,1, and 3. The agent sends invalidation messages to all its children.

The invalidation messages carry the identity of the writer. Node 0 receives an invalidation message and discards it since it is the writer.

Acknowledgments are returned back. The agent sends its acknowledgment to the home directory when it receives acknowledgments from all its children.

Figure 3.29. Invalidation of a GLOW sharing tree in a Dir$_n$NB system. The hypothetical four node example of figure (3.28) is used.

Writing—A writer node sends a non-interceptable request to the home node which starts the invalidation process. The home node sends invalidation messages to *all* the nodes designated in its bit-map, *possibly including the writer node*. Agents that receive an invalidation message forward it to all the nodes indicated in all their bit-maps and wait for acknowledgments. The invalidation messages can reach the writer node either directly from the home node or from an agent, if the writer is registered in an agent's bit-map. Thus, the writer node must be prepared to ignore its own invalidation request. To avoid problems when multiple nodes write the same data block, the identity of the writer must be carried with the invalidation messages so that the writers recognize their own invalidation requests. Even if a GLOW agent recognizes the writer node from the invalidation message, it will still send the writer node an invalidation message exactly as the home node would do. This is necessary because the writer node may be an agent itself. For example, in Figure 3.30 node 1 is the writer node and is also an agent. The home node sends the invalidation message back to node 1, where it is intercepted by the agent. The agent sends its own invalidation messages which include an invalidation message to the local node (because of the set L-bit). This invalidation message is a result of the node writing and it is ignored.

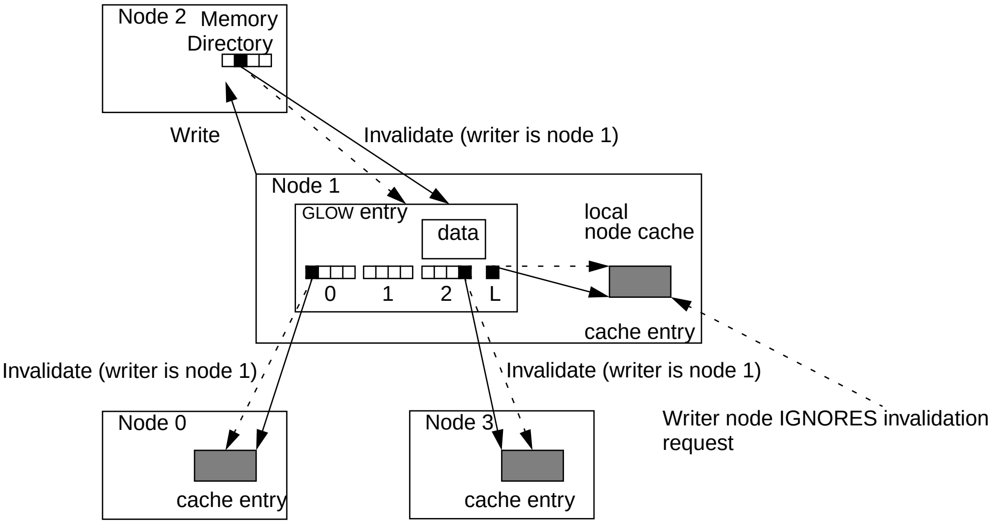

Node 2
Memory
Directory
Write
Invalidate (writer is node 1)
Node 1
GLOW entry
data
local
node cache
0
1
2
L
cache entry
Invalidate (writer is node 1)
Invalidate (writer is node 1)
Writer node IGNORES invalidation request
Node 0
cache entry
Node 3
cache entry

Figure 3.30. Invalidation message returns to the writer node which is also a GLOW agent.

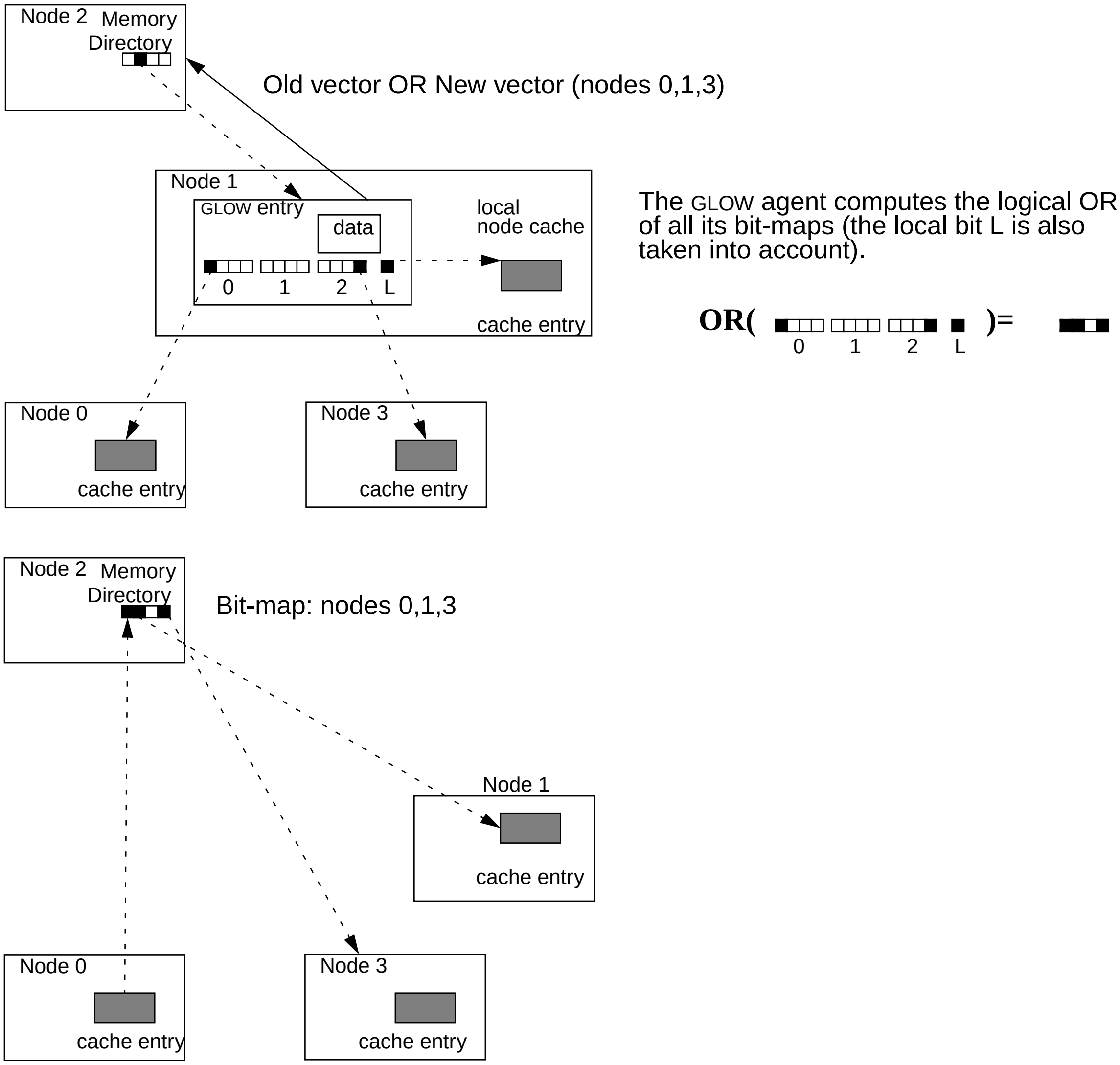


Figure 3.31. Replacement of a GLOW agent in a Dir$_n$NB system. The hypothetical four node example of figure (3.28) is used.

***Replacements*—**In Dir$_n$NB, when a node evicts a cache block, it notifies the home node directory. When a sharing tree exists for the relevant data-block there are three cases:

1. The cache block is a child of the GLOW agent that coincides in the same node. In this case, since the agent monitors the outgoing traffic of the node, the replacement message will be intercepted and the agent will reset the L-bit of the corresponding GLOW entry.

2. The cache block is a child of the home node directory (*i.e.*, the node ID is registered directly in the home node bit-map). In this case, the replacement message must reach the home node. Even if it is intercepted in intermediate agents it will be re-routed toward the home node since the node ID is only registered in the home node.
3. The cache block is a child of an agent in a different node. This means that a local GLOW entry does not exist. In this case, the replacement message must arrive at the correct agent. This can only be guaranteed, if the agents intercept all replacement messages without exception and there is a unique path to the home node. Adaptive routing techniques are incompatible with this algorithm. However, adaptive routing partially attacks the same problems as GLOW (*e.g.*, network congestion) and because of complexity of the two solutions it may be not be cost-effective to use them simultaneously.

Agent replacement, on the other hand, is straightforward and it is depicted in Figure 3.31. The agent computes the union of its bit-maps (using a logical OR function) and sends this bit-map toward the home node in a replacement message. This message may be intercepted by a higher-level agent or it may arrive at the home node. Regardless of the actual destination, the bit-map is merged with the appropriate bit-map at the node that services the replacement message (again a logical OR function). In Figure 3.31, the agent's bit-map arrives at the home node and its is merged with the home-node bit-map.

3.7.1.3 Summary of GLOW on $\mathrm{Dir}_n\mathrm{NB}$

Assuming that in a $\mathrm{Dir}_n\mathrm{NB}$ system we cannot provide the GLOW agents with their own unique IDs, we can integrate them with other nodes. With this compromise, GLOW maintains most of its characteristics described in Section 3.4.1, some with modifications:

1. Simplicity: The Dir_nNB protocol is replicated in the GLOW agents similar to SCI.
2. Network locality: the sharing trees created in Dir_nNB GLOW map on the network topology. However, topologies are restricted to ones where switches coincide with nodes.
3. No multilevel inclusion is enforced: the GLOW agents can ignore any read request *even requests coming from the same node*. However, before an agent allocates a new GLOW entry it must perform a lookup in the node cache (in case the data are already there and the node ID is registered in the bit-map of the home node or another agent).
4. Scalable reads. Similar to SCI.
5. Scalable writes. Although in Dir_nNB the invalidation algorithm is not serial as in SCI, the network locality of the sharing tree reduces the bandwidth requirements by sending a comparable number of messages in *much shorter distances*.

Transparency is compromised because of the unification of the GLOW agents and the nodes. GLOW is not transparent since the system must be designed to accommodate it: network switches must be integrated with nodes and network topologies where switches do not coincide with nodes are excluded. In the sample design presented above, the main differences from the SCI version of GLOW are:

1. The GLOW agents are enhanced nodes and cannot be stand-alone entities (because of lack of node addressing space).
2. GLOW agents must always carry data. This is because, in contrast to SCI where nodes can actually get the data from other caches, in Dir_nNB nodes always get the data from the memory node.
3. GLOW agents must snoop at all incoming and outgoing traffic of the corresponding node. A

GLOW agent assumes the identity of the corresponding node for the cache blocks that are active in its (GLOW) cache. Messages (*e.g.*, invalidation messages) addressed to the node ID may be destined either for the node itself or for the GLOW agent, which is given priority. The replacement algorithm also requires monitoring of the outgoing traffic. This requirement does not compromise the policy of not enforcing multilevel inclusion, which relates to the action taken upon intercepting a read request.

4. Modifications to the base protocol (Dir_nNB) are required: invalidation messages must carry the identity of the writer.

3.7.2 Limited pointer directory protocols

A consideration with full-map protocols is the memory requirement for directory storage. A thousand-node Dir_{1024}NB (1024 pointers, No Broadcast) machine requires 1Kbit of tag for every data block in memory. Proposed solutions are to implement a limited pointer directory (*e.g.*, Dir_iB, which has *i* pointers and broadcasts invalidations for more that *i* nodes sharing), LimitLESS directories [24], where additional pointers beyond what the hardware provides are handled in software, or sparse directories that hold sharing information for a limited number of data blocks. GLOW, however, can provide an alternative solution offering not only support for widely-shared data but also additional pointer storage in its agents. The agents are unified with nodes and are implemented as described in the previous section. A well designed Dir_iB system with GLOW would have at most *i* GLOW agents in the first level after memory. The memory directory would rarely see more than *i* requests for any data block (regardless of whether it is widely-shared or not).

GLOW attacks different weaknesses in different environments. SCI is scalable with respect to the memory required for directory information by distributing this information in lists. Because of this, SCI pays the price of serial invalidations. GLOW provides a solution to this weakness. Dir_nNB suffers from scalability problems because of directory memory requirements. Limited pointer variations attack this problem, but create a new one: exceeding the pointer capacity. These protocols pay the price of invalidation broadcasts, even for the cases where broadcasts are an overkill. LimitLESS directories pay a severe penalty whenever the pointer capacity is exceeded. GLOW provides a solution to this situation by providing dynamic pointers, making solutions such as dynamic pointer allocation [90] unnecessary.

3.7.3 GLOW and pruning caches

Pruning caches, proposed by Scott and Goodman [88,86] are hierarchical directories based on the multilevel inclusion property. Pruning caches allow a directory entry in a sharing tree to be replaced without invalidating its subtrees. However, for any subsequent operation (*e.g.*, invalidation) the conservative assumption must be made that the subtrees of the replaced entry are fully populated [86].

The concept of pruning caches can also be applied to GLOW potentially offering higher performance. The scheme depends on the observation that entire child lists in SCI GLOW are typically (but not always) contained within a single ring. This fact, and the expectation that a modest number of nodes will be attached to a single ring, suggests the possibility of storing directory information related to a single list not in the form of a doubly-linked list, but as a bitmap with local scope (restricted to the nodes of a ring). Storing the information as a bitmap opens up a range of alternative algorithms, using broadcast.

The pruning cache concept depends on the observation that if the bitmap information is lost, the tree can be reconstructed conservatively, that is, by including all possible siblings in the child list. While nodes not in the tree may be inadvertently inserted into the tree, they need not contain data, so no harm is created by adding them (except that the tree is unnecessarily larger than it need be, and extra traffic may be generated in destroying it).

One of the benefits of the bitmap scheme is that a node can leave the tree for free. Note that this feature meets a basic requirement of the bit-mapped scheme: all lists must be contained within a single ring. This scheme therefore always forms the perfect tree, and avoids the problems of patching ill-formed trees. However, this also means that multilevel inclusion is enforced in the tree: nodes or agents cannot occupy any arbitrary level of the tree but strictly the one necessary by the topology. In this case, special care must be employed to avoid deadlocks in arbitrary topologies (see Section 3.5.3).

3.7.4 Software GLOW (SOFTGLOW)

GLOW can be applied on software-based shared-memory platforms (shared virtual memory implementations such as Munin [22], IVY [67], Treadmarks [10], etc. are prime examples). A class of machines exemplified by the Typhoon [80] and FLASH [60] architectures provide necessary hardware mechanisms to assist the implementation of shared memory, but they also implement all their protocols in software. GLOW can be implemented in software (called SOFTGLOW) without hardware GLOW agents. In this case the GLOW agents exist only as abstractions and are implemented as software handlers, along with the rest of the protocols.

The importance of SOFTGLOW is justified as follows:

1. It applies to a large class of shared-memory platforms that are considered cost-effective.
2. There is no direct hardware cost associated with implementing GLOW agents; instead cost appears as resources occupied in the nodes.
3. It can offer considerable benefits to these machines.

One way GLOW can benefit such systems is by reducing the network load. More importantly, GLOW may alleviate hot spots, which is especially relevant for the classes of machines mentioned above. In these machines the network interface (NI) is possibly a significant bottleneck. GLOW may be able to reduce the load on the NIs by distributing widely-shared data accesses more evenly across the system.

SoftGLOW can be implemented in any platform that provides both:

- the ability to implement the GLOW agent abstraction in software in any node (*i.e.* the ability to use intermediate nodes as replicas of home nodes);
- and the ability to direct accesses to GLOW home-node replicas instead of the actual home nodes.

3.7.4.1 SOFTGLOW and software combining

Since SOFTGLOW is a group of software handlers for widely-shared data, it is related to software combining techniques [106,107] (software trees etc.). Software combining is implemented at the application level, thus changing the program itself and introducing substantial software overhead. SoftGLOW is implemented in a lower level where it is transparent to the application and the overhead is small and hidden from the application level. The program need not execute more instructions other than the single load or store to the widely-shared data, in contrast to the software combining techniques. Thus, SOFTGLOW provides for efficient handling of widely-shared data in a manner that is transparent to the application, leading to cleaner, simpler and more elegant shared-memory programs.

3.7.4.2 Related work

Systems similar to SOFTGLOW have already been implemented. In particular, Bianchini and LeBlanc have proposed *eager combining* (EC) [17] and Bennett, Kelly, Refstrup and Talbot have proposed *proxies* [96], that are similar to the concept of GLOW home-node replicas. EC uses specified nodes as servers for widely shared pages or *hot* pages. These pages are updated in the server nodes (using *eager sharing*). Clients request the data from the servers, rather than the actual home node. EC is similar to GLOW in that the authors only use it for widely-shared data. However EC does not take into account network topology, it uses partial update for the servers, and caches the actual data, which is optional in GLOW. The authors report speedup over DASH in the range of 2 to 3 for up to 128 processors. Proxies are another approach to software support for wide sharing, similar to EC. While EC employs a partial update protocol (only the server nodes are updated and not the sharing nodes), proxies employ an invalidation protocol. Many of the SOFTGLOW's features are covered by the proxies work.

3.8 Summary

In this chapter I argued that widely-shared data are a more serious problem than previously recognized, and that furthermore, it is possible to provide transparent support that actually gives an advantage to widely-shared data by exploiting their redundancy to improve accessibility. The GLOW extensions to cache-coherence protocols provide such support for widely-shared data by defining functionality in the network domain.

A GLOW implementation described in detail in this thesis, encompasses six characteristics that are found together for the first time in a single proposal. These are:

1. TRANSPARENCY. GLOW is not a protocol, but a transparent extension to other protocols. However, for some GLOW implementations, this characteristic is compromised (Section 3.7.1).
2. SIMPLICITY. GLOW agents behave both as memory directories and cache nodes, using the underlying cache coherence protocol recursively.
3. NO MULTILEVEL INCLUSION. GLOW does not impose *multilevel inclusion* in its sharing trees. The implication is that GLOW builds *logical* sharing trees on top of any physical topology, efficiently avoiding deadlocks.
4. NETWORK (GEOGRAPHICAL) LOCALITY. The logical GLOW trees follow—to the extent possible, but not necessarily—the tree that fans-in from all the sharing nodes to the actual home node of the widely-shared data.
5. SCALABLE READS. GLOW offers long-lived directory storage in the network thus achieving request combining and elimination of hot spots.

6. SCALABLE WRITES. GLOW employs parallel invalidation (update) which, coupled with the network locality of the sharing tree, offers fast and scalable writes making efficient use of network bandwidth.

To demonstrate GLOW, I successfully implemented it transparently on top of a complex coherence protocol (SCI). The main part of this chapter describes this implementation. The policy of not enforcing multilevel inclusion allows two algorithms (DESTRUCTIVE ROLLOUT and LINEARIZING ROLLOUT) to handle replacements in the sharing tree. I have shown that both perform equally well over a wide range of GLOW directory cache sizes. In the extreme cases where the directory storage in the GLOW agents cannot accommodate the small amount of widely-shared data (actively used at any point in time), DESTRUCTIVE ROLLOUT suffers because it interferes negatively with sharing nodes.

Finally, I have discussed invalidation and update protocols, the effects of relaxing the memory model for both SCI and GLOW (and argued that GLOW with sequential consistency is better than SCI with a relaxed consistency memory model), and proposed GLOW implementations on top of other hardware and software coherence protocols.

The GLOW extensions are an optimization for wide sharing. Because the GLOW extensions incur overhead for building a sharing tree (which is not leveraged when very few nodes are sharing), they should not be applied indiscriminately on all data (or for all accesses). Doing so results in performance loss. The GLOW extensions are independent of methods to define (statically) or discover (dynamically) widely-shared data or instructions that access widely-shared data. In the next chapter I discuss several such methods.

4 Static and Dynamic Optimizations for Wide Sharing

In the previous chapter I described the GLOW mechanisms to handle widely-shared data. The mechanisms are engaged *only* for special requests that indicate wide sharing. However, the GLOW mechanisms are independent of how these special requests are generated, *i.e.*, they are independent of how widely-shared data are distinguished from other data. In this chapter I describe both static [50,51] and dynamic methods [52,53] to generate special requests for wide sharing. Table 5 is an overview of all the methods I propose and study. The methods are divided into static (compile-time) and dynamic (run-time), address-based (*i.e.*, widely-shared data are distinguished by their addresses) and instruction-based (*i.e.*, widely-shared data are distinguished by the instructions that access them). Related work (EC, proxies, STEM, combining) appears with references in the shaded cells.

WIDE SHARING	**Static**	**Dynamic**
Address-based	EC [17], PROXIES [15][96]	STEM [44] COMBINING [36]
	Static GLOW address-based	Agent-Detection Directory-Detection
Instruction-based	Static GLOW instruction-based	Instruction-based Prediction 1) Latency 2) Directory-feedback

Table 5. Wide sharing: static and dynamic, address-based and instruction-based methods to apply the GLOW optimizations. Related work is shown in the shaded cells.

Section 4.1 presents the static (address-based and instruction-based) methods to generate GLOW requests. I consider the static address-based method as the base case for the performance of GLOW and I compare all dynamic methods to this case. Using the static address-based method I examine various aspects of GLOW performance in Section 4.2.

In the next two sections I present the dynamic methods to apply the GLOW extensions. In Section 4.3 I present two address-based methods. The first is called AGENT DETECTION and it is inspired by request combining. The second is called DIRECTORY DETECTION and in this method the directory is responsible for discovering wide sharing. Section 4.4 presents two instruction-based methods in which, when a load instruction misses in the coherent cache, we can predict (based on its past history) whether it will access widely-shared data or not.

In the two sections where I describe the address-based and instruction-based dynamic methods, I present specific results to illustrate various aspects of each. A full-blown comparison of all the methods (static and dynamic) is presented in the last section of this chapter (Section 4.5). In this section, to interpret the results, I examine how the various methods affect the read-runs of the programs.

4.1 Static optimizations for wide sharing

In the static methods the *user* identifies, at compile-time, either the widely-shared data (address-based) or the code that accesses widely-shared data (instruction-based). Conceptually, this can be done automatically by a compiler (this is beyond the scope of this thesis). In cases where identification is difficult, profiling could provide the necessary information.

Identifying the widely-shared data in the source program is only the first step. The appropriate information must then be passed to the hardware so special requests for widely-shared data can be generated to invoke the GLOW agents. Thus, the static methods require an interface to pass information from the program to the hardware. Only when such an interface is possible and has an acceptable implementation do the static methods make sense. Otherwise, the dynamic methods presented later in this chapter must be used to transparently provide detection of widely-shared data.

4.1.1 Static address-based GLOW

Conceptually, address-based identification of widely-shared data is simple: directives in the source code define and un-define widely-shared data. During run-time, data that are defined as widely shared are accessed using GLOW. A possible hardware implementation of this method uses *address tables*—structures that store arbitrary addresses (or segments) of widely-shared data. Address tables can be implemented in the network interface (NI) or as part of the cache-coherence hardware. In both cases the user must have access to these tables in order to define and un-define widely-shared data. Implementing these structures, however, is not trivial. The main issues are:

- **Security**: Access to hardware resources must be protected by the operating system. The operating system must support these tables with a special application interface (API).
- **Allocation:** Multiple competing processes may want to access the tables concurrently. In this case the operating system must partition the tables to multiple processes or in some way multiplex their accesses.
- **Address translation**: Another consideration is that translation from virtual to physical addresses may be needed for these address tables. Schoinas and Hill have studied the general problem of address translation in NIs and discuss several solutions [85].

The address tables could be virtualized by the operating system, but this solution is also unsatisfactory since (i) it requires operating system support and (ii) it will slow down access to these tables.

Alternatively, address-based identification can be accomplished by mapping widely-shared data to a predefined memory area (to a pre-defined set of pages if virtual addresses are visible at the GLOW agents, or to predefined physical frames if only physical addresses are available). The NI could generate GLOW requests for addresses that fall in some predefined area. The drawback of this method is that it is not flexible enough to handle data that change behavior dynamically (without resorting to data movement in memory). For example, in the GAUSS benchmark, memory mapping of widely-shared data is not convenient because the pivot row changes with every iteration.

4.1.2 Static instruction-based GLOW

The observation that led to all the instruction-based schemes is that the nature of the data may change dynamically, but the nature of the code often does not. For example, in the GAUSS program, the pivot row (*i.e.*, widely-shared data) is accessed at a very specific point. Finding the instructions that access the pivot row is enough to generate wide-sharing requests, even if the pivot row changes in every iteration.

If specific code is used to access widely-shared data, the programmer can annotate the source code and the compiler can generate memory operations for this code that are interpreted as wide-sharing requests. I propose the following implementations:

- COLORED OR FLAVORED LOADS: The processor is capable of tagging load and store operations explicitly. Currently this method enjoys little support from commercial processors.
- EXTERNAL REGISTERS: A two-instruction sequence is employed. First, a special store to an uncached, memory mapped, external register is issued, followed by the actual load or store. This special store sets up external hardware that will tag the following memory operation as a widely-shared data operation. While it is desirable that the two operations be performed atomically, this is not a requirement for correctness, so no extraordinary measures need be employed to handle discontinuities caused by exceptions. The main drawback of this scheme is that it requires external hardware close to the processor, which may not be feasible in a system that is built with commodity main-boards.
- PREFETCH INSTRUCTIONS: If the microprocessor has prefetch instructions they can be used to indicate to the external hardware which addresses are widely shared. Again, external hardware is required close to the processor, making this a “custom hardware” approach.
- ADDRESS MANIPULATION: An instruction accessing widely-shared data could be preceded by code that sets an unused bit (*e.g.*, the most significant bit) in the address accessed. The

GLOW agents can be set to intercept addresses with the specific bit set. The drawback of this method is that it reduces the address space available to the program by half, in order to avoid address conflicts. Appropriate mappings in the virtual system must also be set to avoid bogus page faults.

The performance of static instruction-based GLOW is discussed in connection with that of its dynamic counterpart *instruction-based prediction* (Section 4.4).

4.2 Performance of static optimizations

The static address-based GLOW is the base case for the GLOW optimizations, against which all other static and dynamic GLOW methods are compared. In this section, I study aspects of the performance of the static address-based GLOW:

1. In Section 4.2.1: Basic performance compared to SCI. Effect of caching data in the GLOW agents. Effects of 2- and 3-dimensional topologies.
2. In Section 4.2.2: Effect of the input size on scalability of programs.
3. In Section 4.2.3: Analysis of read and write performance.
4. In Section 4.2.4: Effect of relaxing the memory model.
5. In Section 4.2.5: Update protocols.

I examine six programs (GAUSS, SPARSE, APSP, TC, BARNES, CG) in various system configurations (2-dimensional and 3-dimensional networks with 16 to 128 nodes). I included directives in the source code to define and undefine widely-shared data. I model a very small overhead of 6 processor cycles for defining and undefining widely-shared data at run-time, using address tables. Since changes to the address tables are infrequent for all programs, execution time is affected minimally.

I use the 3-dimensional topologies to show how network scalability affects the GLOW extensions. In general the results show that GLOW offers greater performance advantage with higher dimension networks, because it can create shorter trees with larger fan-out.

I measure execution time for each program and present speedups normalized to a base case. I selected the base case to be SCI on 16 nodes with the appropriate 2- or 3-dimensional network (shaded rows in Table 6). This presentation of the results achieves two goals: (i) since all simulations start at 16 nodes (SCI on 16 nodes has a normalized speedup of 1.00) the normalized speedup directly gives the performance gain when the number of processors is increased two-, four-, and eight-fold, and (ii) the normalized speedup curves are the same as the speedup-over-one-node curves but scaled by a constant factor. In contrast, GLOW speedup over SCI *for the same number of nodes* does not provide scalability information.

The programs I use do not scale beyond 32 or 64 nodes for SCI (Table 6). The GLOW extensions allow these programs to scale to 128 nodes but the performance difference between 64 and 128 nodes is generally small. A limitation of the simulation methodology is that the input size of the programs is kept constant and relatively small. This is necessary because of practical constraints pertaining to simulation time and memory limitations of the host system (CM-5). With larger datasets these programs scale to more nodes for the base SCI case and the GLOW extensions yield increasingly higher performance improvements, as I will show in Section 4.2.2.

	Nodes	GAUSS	SPARSE	APSP	TC	BARNES	CG
2-D	16	16.57	5.86	11.70	14.45	7.22	15.99
	32	25.34	8.57	19.36	20.10	8.56	24.24
	64	22.95	12.68	21.02	19.27	12.69	37.26
	128	12.92	12.53	14.74	13.20	—	—
3-D	16	16.77	6.09	11.77	14.51	7.23	—
	32	26.33	10.01	19.88	20.60	8.54	—
	64	25.27	15.73	21.93	20.14	12.56	—
	128	16.18	18.54	15.94	14.28	—	—

Table 6. Actual speedups (over a single node) for SCI. The shaded rows are speedups for the 16-node systems with 2- and 3-dimensional topologies that are used as the base cases for normalizing GLOW speedups.

4.2.1 Basic performance: GLOW with and without data cache

Nodes		GAUSS	SPARSE	APSP	TC	BARNES	CG (128)
2-D	16	1.04	1.19	1.04	1.04	0.99	1.40
	32	1.14	1.13	1.11	1.14	1.00	1.52
	64	1.53	1.26	1.52	1.55	1.00	1.58
	128	2.22	1.26	2.20	2.22	—	—
3-D	16	1.03	1.16	0.93	1.04	0.99	—
	32	1.15	1.24	1.17	1.19	1.00	—
	64	1.58	1.38	1.62	1.67	1.00	—
	128	2.44	1.31	2.59	2.64	—	—

Table 7. Speedup of static GLOW (S) over SCI on 16 to 128 nodes, 2 and 3 dimensions.

In this section I present results that show: (i) how the programs scale using SCI and using GLOW, (ii) the effects of the dimensionality of the network in performance, and (iii) the effect of caching data in the GLOW agents. Table 7 shows the speedup of GLOW—without data storage—over SCI for 2- and 3-dimensional networks.

***GAUSS*—**GAUSS (Figure 4.1) on SCI does not scale beyond 32 nodes, showing serious performance degradation with higher numbers of nodes. For GLOW I define the unknown vector and the pivot row as widely-shared data (see Section 2.3.1 for a description of the widely-shared data in the GAUSS program). The pivot row is re-defined for every iteration. The GLOW extensions scale to 64 nodes in 2 dimensions and to 128 nodes in 3 dimensions (although the additional speedup is negligible). Static, address-based GLOW (without data storage) is up to 2.22 times faster than SCI in 2 dimensions and up to 2.44 times faster in 3 dimensions (speedups over SCI are shown in Table 7). Data storage in the GLOW agents provides marginal performance improvement.

SPARSE*—**SPARSE (Figure 4.2) scales to 128 nodes for both 2- and 3-dimensional networks (although the increase in performance from 64 to 128 nodes in 2 dimensions is negligible). For GLOW I define vector ***X as widely-shared data (see Section 2.3.2). GLOW is up to 1.26 and up to 1.36 times faster than SCI in 128 nodes in 2 and 3 dimensions (Table 7). Data storage in the GLOW agents offers 5% improvement in 128 nodes in 3 dimensions and less for the other configurations.

APSP and TC**—**With SCI, APSP (Figure 4.3) does not scale beyond 64 nodes and TC (Figure 4.4) does not scale beyond 32 nodes. Since it is impossible to determine at compile-time the widely-shared data for both programs, the whole array that represents the graph is defined as widely-shared data. GLOW scales to 128 nodes but the performance increase from 64 to 128 nodes in 2 dimensions is negligible (for TC performance drops). For APSP, static GLOW is up to 2.20 times faster than SCI in 2 dimensions and up to 2.59 times faster in 3 dimensions (see Table 7). Similarly, for TC, static GLOW is up to 2.22 and up to 2.64 times faster than SCI for 2 and 3 dimensions, respectively (Table 7). Data storage offers negligible performance improvements.

BARNES**—**BARNES (Figure 4.5) scales to 64 nodes for both SCI and GLOW even with the very small input set used. For BARNES, static address-based GLOW does not offer performance improvements. This is because, I defined only the root of the oct-tree as widely-shared data and I do not dynamically define other levels as widely-shared data. The dynamic GLOW schemes presented in subsequent sections are able to detect widely-shared data effectively and show good performance improvements over SCI.

CG**—**CG (Figure 4.6) scales well for both SCI and GLOW with a 128x128 input. The main array of CG is defined as widely shared. GLOW offers good performance improvement that increases with system size. Data storage offers negligible performance improvement.

To summarize:

- GLOW does not show performance improvement over SCI for less than 16 nodes because widely-shared data only become detrimental to scalability for larger system sizes.
- Even with the small inputs used, GLOW offers good performance improvement over SCI and allows scaling to larger systems.
- The performance improvements are higher for the 3-dimensional topologies where GLOW trees are better.
- Data storage in GLOW agents does not significantly help performance. This is expected since the only case where data storage makes a difference is when a GLOW agent with child lists creates a new child list. Without data storage the agent would have to fetch the data from one of its older child lists. This is not the common case. Typically, a node gets the data directly from the head of the child list.

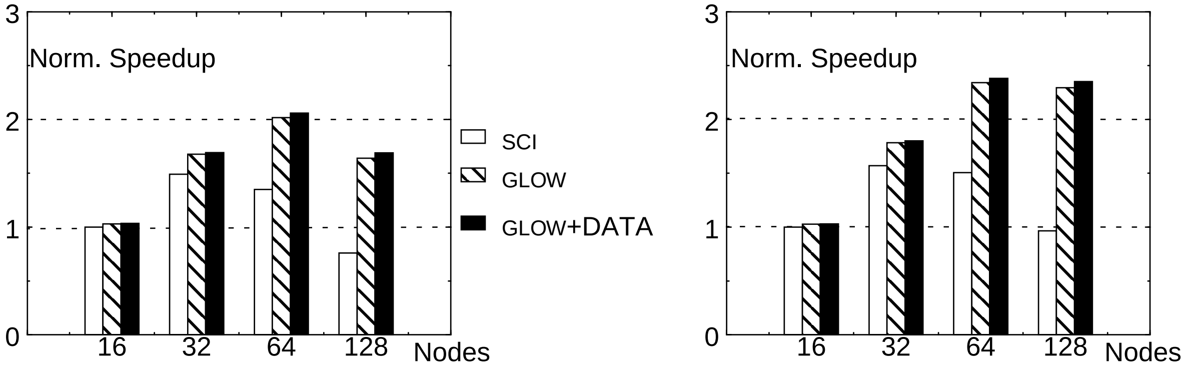


Figure 4.1. GAUSS: Normalized speedup (base system is SCI on 16 nodes) for 2- and 3-dimensional networks. GLOW with and without data cache.

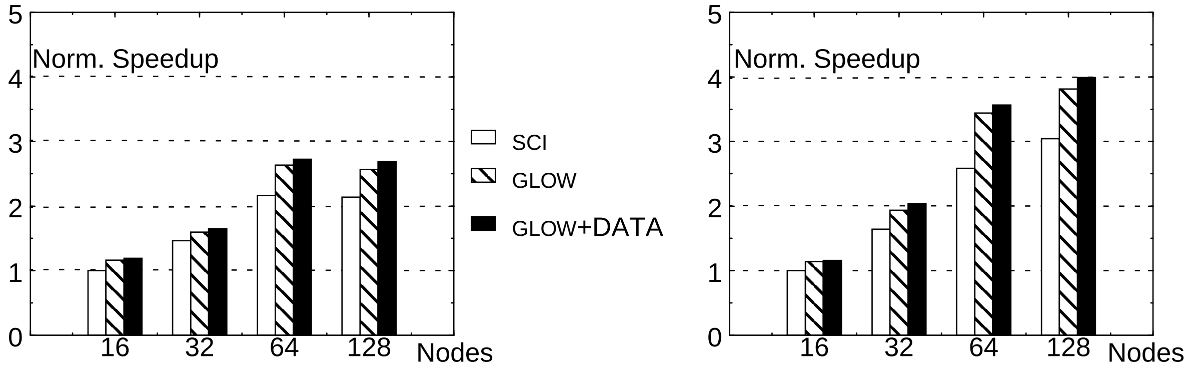


Figure 4.2. SPARSE: Normalized speedup (base system is SCI on 16 nodes) for 2- and 3-dimensional networks. GLOW with and without data cache.

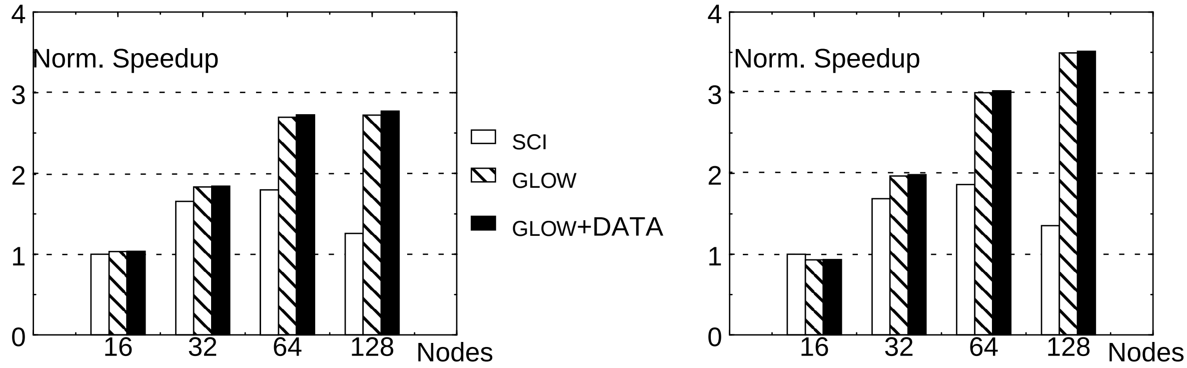


Figure 4.3. APSP: Normalized speedup (base system is SCI on 16 nodes) for 2- and 3-dimensional networks. GLOW with and without data cache.

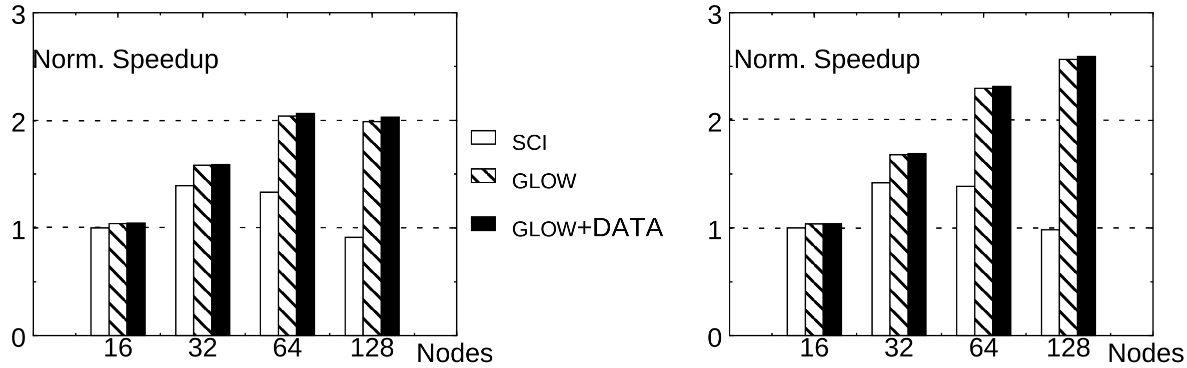


Figure 4.4. TC: Normalized speedup (base system is SCI on 16 nodes) for 2- and 3-dimensional networks. GLOW with and without data cache.

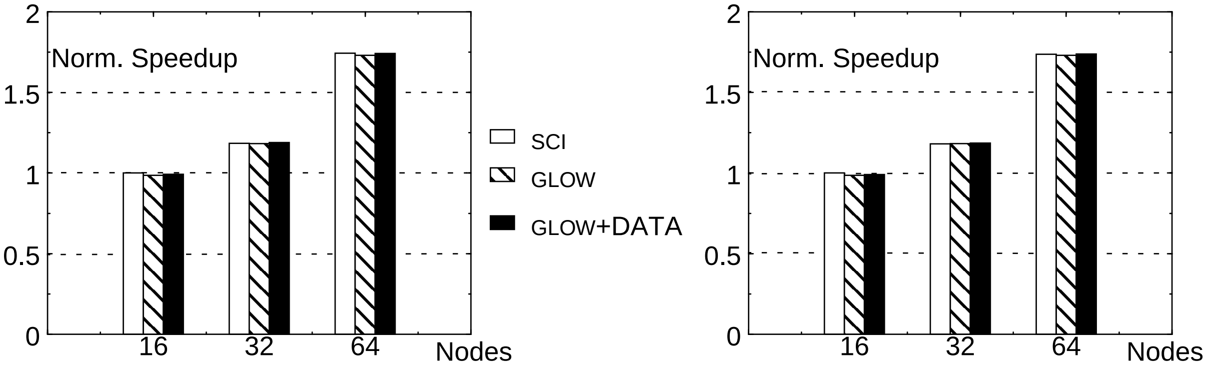


Figure 4.5. BARNES: Normalized speedup (base system is SCI on 16 nodes) for 2- and 3-dimensional networks. GLOW with and without data cache.

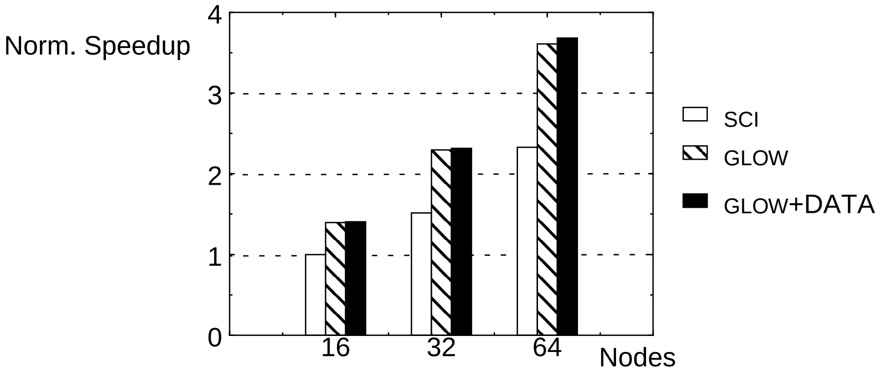


Figure 4.6. CG: Normalized speedup (base system is SCI on 16 nodes) for 2-dimensional network. GLOW with and without data cache.

4.2.2 Dataset

Because of practical simulation constraints, I keep the input size constant regardless of the system size. Thus, the programs usually do not scale beyond 32 (or at most 64) nodes. The reason is that there is not enough work to be distributed among the nodes in the large systems and the communication to computation ratio in such cases is very high. In addition, SCI is slowed down by widely-shared data (GLOW is always faster than SCI in the larger systems).

	256x256	**512x512**	**768x768**
16	1.06	1.04	1.02
32	1.32	1.14	1.07
64	2.14	1.53	1.49
128	3.81	2.22	2.69

Table 8. GLOW speedup over SCI for (running on the same number of nodes) for GAUSS for three inputs.

Using larger datasets, GLOW scales much better than SCI. This is evident for GAUSS when it is run with larger data sets. Figure 4.7 shows actual speedup (over a single node) for SCI and GLOW for three different datasets:

- 256x256: With the small dataset, SCI does not scale (the performance improvement from 16 to 32 nodes is negligible). Its performance is also diminished by wide sharing. GLOW's speedup increases very slowly up to 64 nodes and then drops. However, GLOW remains significantly faster than SCI (Table 8).
- 512x512: With the medium dataset, SCI still does not scale beyond 32 nodes but GLOW scales better. GLOW's speedup over SCI is not as high as in the previous case especially for the small systems. This dataset is used for all further evaluation.

- 768x768: With the large dataset, the picture changes: SCI scales up to 64 nodes and GLOW scales up to 128 nodes. However, widely-shared data only become significant for the 64- and 128-node systems. Thus, GLOW's speedup over SCI is less than that the previous case (medium dataset) for the smaller systems but it is higher for the 128-node system.

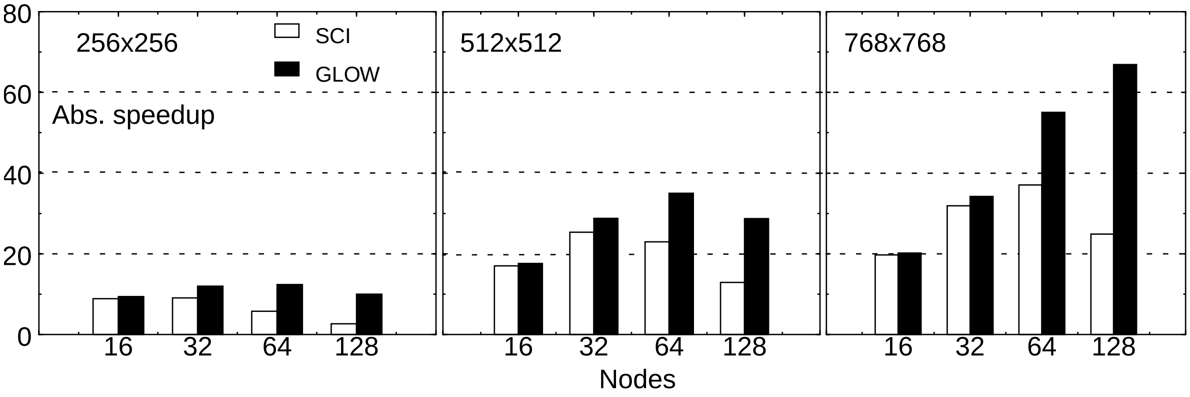


Figure 4.7. Absolute speedup (over a single node) for GAUSS in 2 dimensions with three input set sizes: 256x256, 512x512, 768x768.

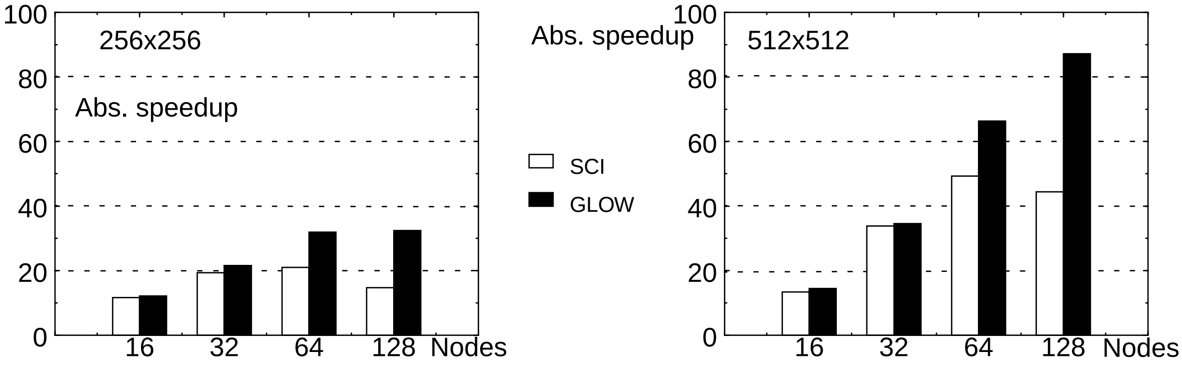


Figure 4.8. Absolute speedup (over a single node) for APSP in 2 dimensions with two input set sizes: 256x256 and 512x512.

Similar observations hold for APSP and TC when run with two input set sizes: 256x256 which is the default dataset for all further evaluation, and 512x512. Figures 4.8 and 4.9 show the actual speedups (over a single node) for APSP and TC. With larger data sets GLOW scales much

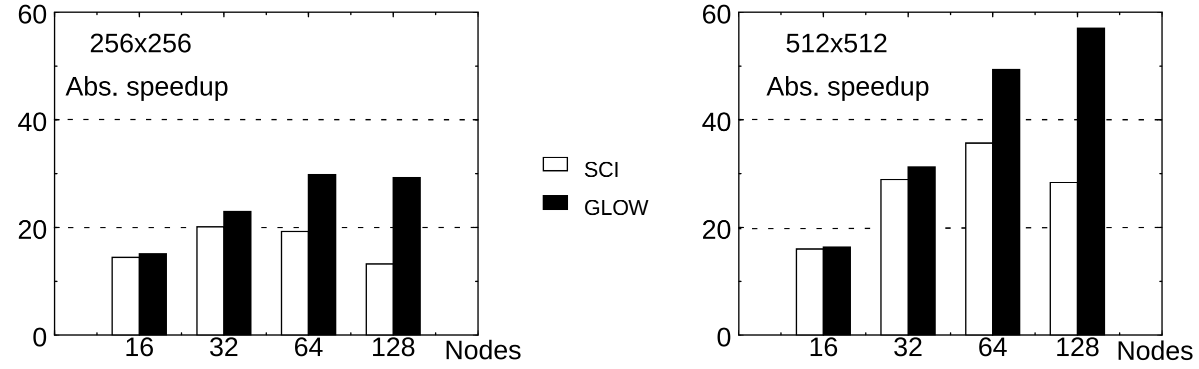


Figure 4.9. Absolute speedup (over a single node) for TC in 2 dimensions with two input set sizes: 256x256 and 512x512.

better but it only becomes significantly faster than SCI for large systems. BARNES and CG also show the same trends.

4.2.3 Analysis of read and write performance

GLOW provides both scalable reads and scalable writes. The purpose of this section is to attempt to separate the performance components attributed to reads and to writes. This is difficult to do by simply examining how GLOW affects read and write latencies (as it was presented in Appendix 3.5.5 for micro-benchmarks) because of the complex overlap of operations in a program.

Instead, I modified the invalidation algorithm to operate serially (without any request forwarding). Thus, the sharing trees are built as before, obtaining the full read performance, but they are invalidated similarly to a sharing list. Although the serial sharing tree invalidation algorithm is slower, it is only a **rough approximation** to the serial sharing list invalidation because of three significant differences: (i) in the sharing tree, more nodes (the GLOW agents) are invalidated and this slows the invalidation, (ii) the sharing tree has more network locality

—invalidation transactions among agents and SCI nodes are usually confined to single rings—and this speeds up the invalidation, and (iii) in the sharing tree invalidation (serial or parallel), no single node is responsible for all invalidation transactions, as opposed to a sharing list where only the head node sends all invalidation messages. This may speed up the invalidation because no single server is busy for the duration of the invalidation.

Figure 4.10 compares the write latency of SCI, GLOW with serial invalidation, and GLOW with parallel invalidation (the last graph gives the ratio of the SCI latency to the latency of serial GLOW). As is evident from Figure 4.10 the serial sharing tree invalidation is considerably slower than the parallel sharing tree invalidation, but not quite as slow as the sharing list invalidation. When all nodes are involved (100%) SCI write latency can be as high as 1.6 times the serial GLOW latency. When fewer nodes are reading (25% and 12.5%) the extra GLOW agents in the sharing tree slow down the serial GLOW invalidation to the point where it becomes slower than the SCI invalidation.

Given this behavior of serial GLOW invalidation, a comparison with parallel GLOW invalidation for four benchmarks[3] shows the following:

- For GAUSS the reduction of the write latency is a significant component of the overall GLOW performance since the serial GLOW performs considerably worse than the parallel GLOW.
- For SPARSE, however, the serial GLOW invalidation does not affect performance. This is because only three widely shared variables are written frequently.

[3] BARNES is not included because STATIC GLOW does not offer any performance over SCI.

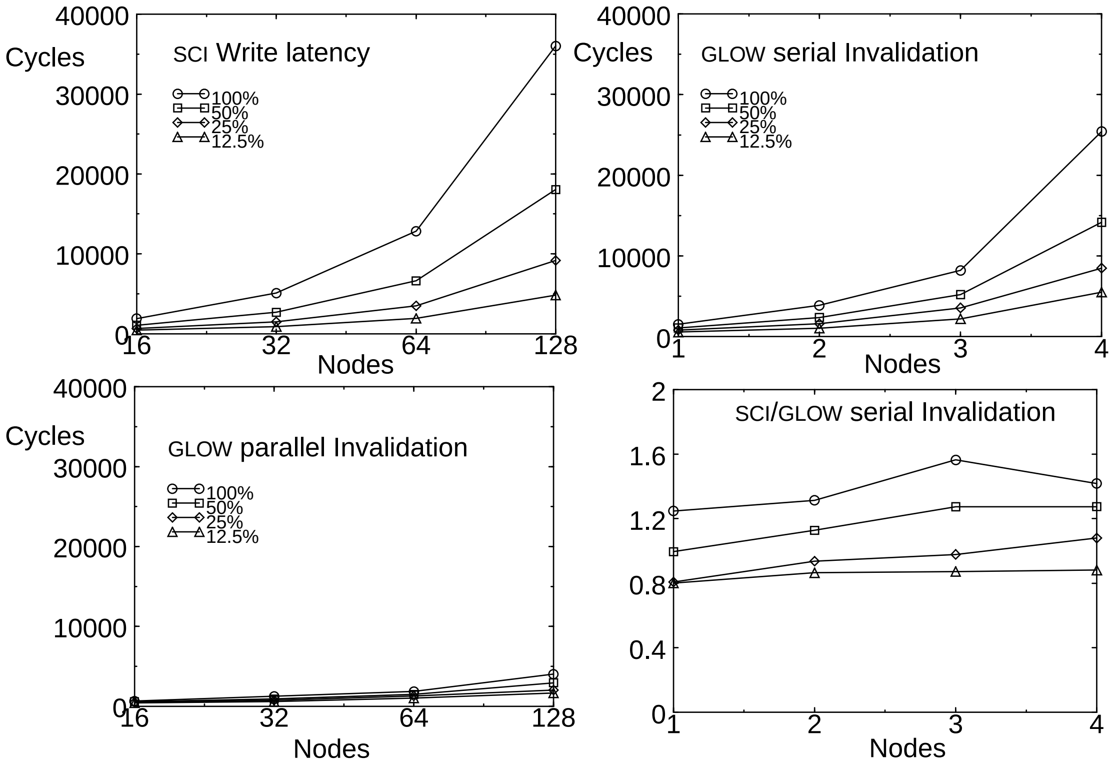


Figure 4.10. Write latency micro-benchmark results for GLOW with serial invalidation.

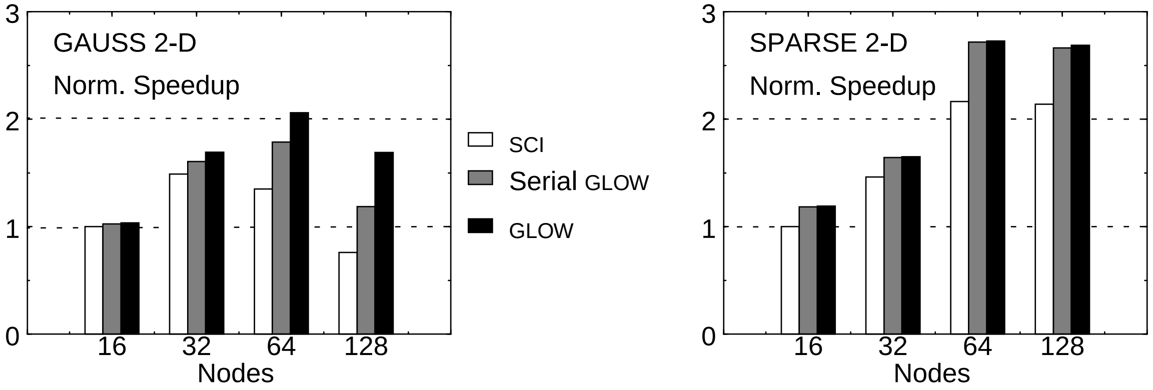


Figure 4.11. Normalized speedup for GAUSS and SPARSE with serial GLOW. Base system is SCI on 16 nodes.

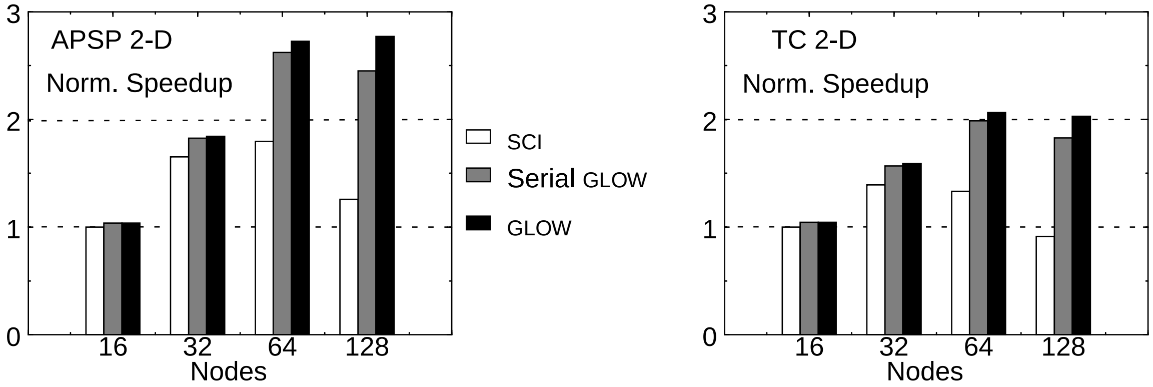


Figure 4.12. Normalized speedup for APSP and TC with serial GLOW. Base system is SCI on 16 nodes.

- For APSP and TC the write latency matters only for large systems where the it becomes significant; otherwise, the reduction of the read latency is responsible for most of the performance benefit.

These results indicate that scalable reads are more important for overall GLOW performance than scalable writes, which become important only in the largest systems. The implication is that GLOW is useful even in protocols that have better-than-serial invalidation algorithms (*e.g.*, Dir$_n$NB, or DASH-like) and do not suffer from write latency nearly as much as SCI.

4.2.4 Relaxed consistency

The results presented so far are for a sequentially-consistent memory system. Relaxing the memory model provides the opportunity to overlap write operations, thereby achieving greater concurrency.

I used the relaxed consistency model (henceforth called **r1**) developed by Kägi et al. [47]. This model allows the processor to continue past writes after it becomes the head of the sharing list, thus overlapping the invalidation of the sharing list or sharing tree—by far the most expensive part of the write—with computation or other writes. More aggressive schemes are possible but Kägi et al. [47] have shown that although they offer additional performance improvement they also impose a significant increase in complexity over r1. Synchronization operations implement *memory fences* that stall the processor until *all* outstanding memory operations have been completed. This model is not as aggressive as relaxed consistency [32], which blocks the processor on synchronization operations until *properly labeled* memory operations have been globally performed.

The results for all the benchmarks (GAUSS, SPARSE, APSP, TC and BARNES) are shown in Figures 4.13, 4.14, and 4.15. Relaxed consistency helps both SCI and GLOW since *all* writes are relaxed. A significant result is that GLOW with SC is significantly better than SCI with r1. This is because the latencies involved in wide sharing are very large and cannot be hidden with r1. In contrast, GLOW attacks the problem directly, reducing the latency of reads and writes. This result supports Hill's argument for implementing a simple memory model such as SC [38].

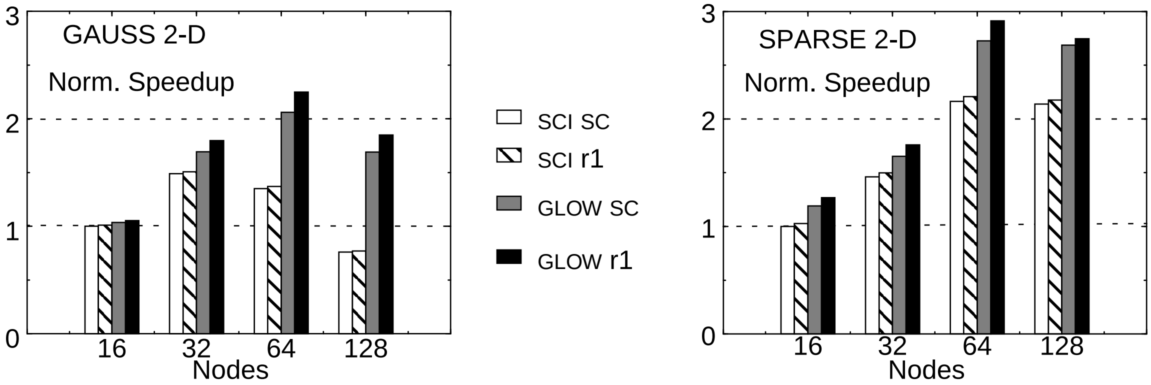


Figure 4.13. Normalized speedup for GAUSS and SPARSE with relaxed-consistency memory model. Base system is SCI on 16 nodes.

An additional advantage of GLOW is that is that it shows slightly higher performance improve-

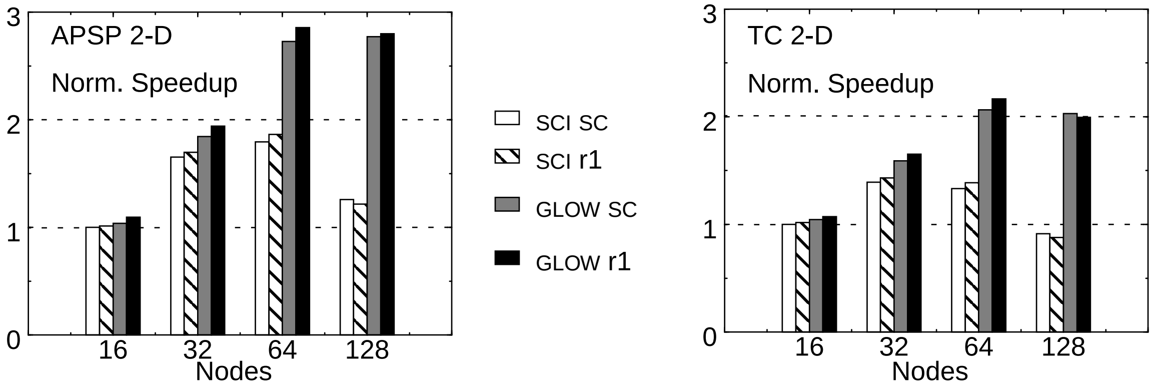


Figure 4.14. Normalized speedup for APSP and TC with relaxed-consistency memory model. Base system is SCI on 16 nodes.

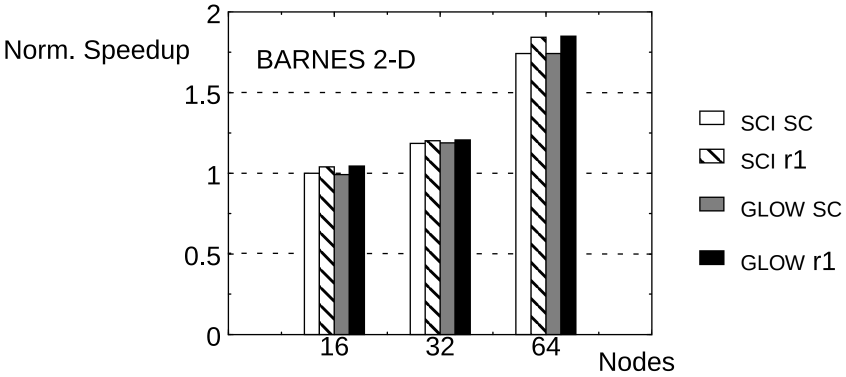


Figure 4.15. Normalized speedup for BARNES with relaxed-consistency memory model. Base system is SCI on 16 nodes,

ment with r1 than SCI. There are two effects at play here. The first is that GLOW makes better use of the network bandwidth because of its network locality. Therefore, more bandwidth remains available for relaxing the memory model and allowing more memory requests in the network concurrently. The second effect has to do with server utilization in the writer nodes. Recall that operations involving many messages tend to create hot spots. Since in SCI the writers invalidate the sharing list, relaxing the memory model allows more requests to be issued from the writer node before the invalidation of the sharing list has completed. This increases the utilization of all resources in the writer node. In contrast, when GLOW is in use, writer nodes invalidate only the highest level of the sharing tree and the rest of the invalidations are distributed to the GLOW agents. In APSP and TC the performance of r1 drops in the 128-node cases for both SCI and GLOW because of the pressure in the network and the cache servers of the writer nodes.

4.2.5 Update protocol

I also extended the GLOW protocol to do updates rather than invalidates. This extension is full update with full request forwarding and does not assume sequentially consistent memory. However, I have been able to ascertain that there are no data races in the benchmarks.

The update protocol shows considerable performance improvement for SPARSE, while for the other benchmarks the performance improvement is either small or negative. This is because only in SPARSE do the widely-shared data remain the same (and do not change behavior). In all other benchmarks the widely-shared data change overtime. Thus, in SPARSE we can update the widely-shared data without fear of sending bogus updates. However, for the other benchmarks, updating a sharing tree (*i.e.*, all the nodes) when the data are not widely shared carries a penalty that increases with system size.

Figure 4.16 compares invalidation and update for GLOW for SPARSE in 2- and 3-dimensional topologies. The update protocol is up to 1.20 times faster than the invalidation protocol in 2 dimensions and up to 1.16 times in 3 dimensions. In contrast, the update protocol performs worse than the invalidation protocol for GAUSS (Figure 4.17). In larger systems the performance of the update protocol worsens since the number of superfluous update messages increases. The effect is most pronounced in the 2-dimensional topology where bandwidth is precious.

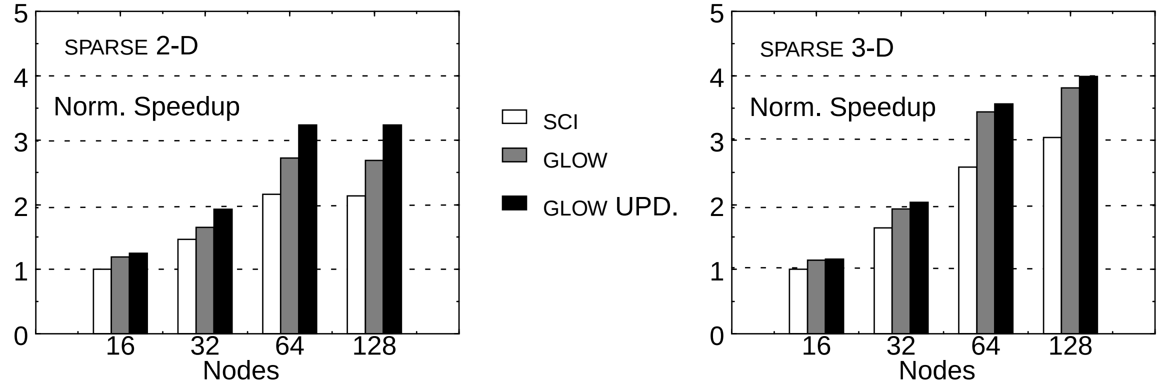


Figure 4.16. Normalized speedup for full update in SPARSE. Base system is SCI on 16 nodes.

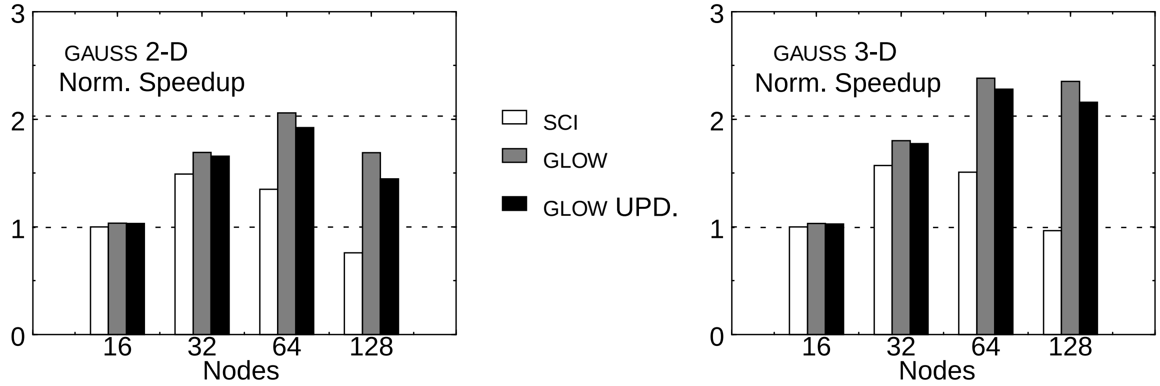


Figure 4.17. Normalized speedup for full update in GAUSS. Base system is SCI on 16 nodes.

4.3 Dynamic address-based optimizations for wide sharing

Shared-memory computers will only be used if they are inexpensive, fast and easy to program. Even if efficient handling of widely-shared data can be achieved using static identification methods, such methods might put so much burden on the programmer and the computer design, that they turn out to be too expensive. The static methods have three serious disadvantages:

1. Involvement of the programmer (and/or possibly the compiler) is required. This puts a burden on the user that contradicts the desire to keep the shared-memory paradigm simple while increasing its efficiency.
2. It is not always trivial to determine the addresses of widely-shared data statically, nor the instructions that access such data. In so called data-driven programs, the access patterns are decided at run time, according to the data values used. Also, if for other reasons the nature of data changes frequently and unpredictably, the static approaches may be inadequate.
3. An interface is required to transfer information from the program to the hardware. This leads to implementation difficulties. Both static alternatives (identifying addresses or identifying instructions) have implementation problems, especially in systems built of commodity parts. Address tables may require operating system support for their virtualization. Identifying instructions that access widely-shared data requires custom hardware, unless the processor itself provides appropriate support in the form of flavored loads. Address manipulation halves the available address space of the program.

For situations where the above three disadvantages are intolerable, I propose dynamic methods to discover widely-shared data or the instructions that access them. The three approaches I propose are:

- Congestion-based detection of widely-shared data which is inspired by request combining (Section 4.3.1).
- Directory detection of widely-shared data which is an address-based scheme where the directory is responsible to discover the widely-shared data (Section 4.3.2).
- Instruction-based prediction which detects instructions that access widely-shared data (Section 4.4). This is the dynamic counterpart of the instruction-based scheme described in Section 4.1.2.

4.3.1 Congestion-based detection of widely-shared data

The appeal of the methods described in this section lies in that they are strictly confined to the network domain and specifically to the GLOW agents themselves. Thus, only the GLOW hardware need change, without affecting other commodity parts in the system.

Conceptually, a GLOW agent could intercept every request that passes through and do a lookup in its directory cache. This would result in slowing down the switch node, polluting the directory caches with non-widely-shared data, and incurring the overhead of building a sharing tree for non-widely-shared data. Instead, we want to filter the request stream and intercept only the requests that are likely to refer to widely-shared data. Detection of such requests can be assisted by the congestion they create in the network.

4.3.1.1 Request-combining in GLOW

The same technique as request-combing (proposed for the NYU Ultracomputer [36]) can be used to detect requests for widely-shared data. The idea is that requests for widely-shared data that are initiated at the same time are likely to create congestion in the network and collide in the GLOW agents as the requests change rings on their way to the home node. Thus, when an agent detects that its message queues are filling up with requests for the same data, it can intercept them, take them out of the queues, and deliver them to the GLOW hardware.

The problem with such combining is that it is based too much on luck. Requests combine only if they happen to be in the same queue at the same time, which might happen only in the presence of congestion. Combining is highly dependent on the network timing and queuing characteristics as well as the congestion characteristics of the application (Section 4.3.1.3).

4.3.1.2 AGENT DETECTION of widely-shared data

Since combining is not guaranteed to work successfully, I propose a new method (henceforth called AGENT DETECTION) to discover widely-shared data. The idea of AGENT DETECTION is that the GLOW agents can artificially increase and keep constant the window of observable requests, regardless of congestion, by remembering the last requests that passed through. Thus, the agents observe the request traffic and detect addresses that are repeatedly requested. Requests for such addresses are then intercepted.

The GLOW agents are switch nodes in the network and their main responsibility is to channel traffic. To implement AGENT DETECTION, besides its ordinary message queues, each agent keeps a small queue (possibly implemented as a circular queue) of the last N read requests it has observed. The actual contents of the queue are the target addresses of the requests, hence its name: *recent-addresses queue*. Using this queue each agent maintains a sliding window of the request stream channeled through its ports.

When a new request arrives at the agent, its address is compared to those previously stored in the recent-addresses queue (which can be searched associatively). If the address is found in the queue the request is immediately intercepted by the agent as a request for widely-shared data.[4] Otherwise, the request is forwarded to its destination. In any case its address is also inserted in the queue.

This method results in some lost opportunities. For example, the first request for an address that is later repeated in other requests is not intercepted. Also, if a stream of requests for the same address is diluted sufficiently by other intervening requests, AGENT DETECTION fails to recognize it as a stream of widely-shared data requests. Finally, AGENT DETECTION does not intercept all requests for widely-shared data and this is possible because GLOW does not impose the multilevel inclusion property (disused in Section 3.5.3).

In the absence of congestion (*i.e.*, when the agent's message queues are empty) we need to search the recent-addresses queue in slightly less time than it takes for a message to pass through the agent. Since the recent-addresses queue is a small structure located at the heart of the switch it can be searched quickly. The minimum latency through the switch will dictate the maximum size of the queue. For the switches I model in the simulations a size of 128 entries could be feasible.

[4] A small threshold can be applied requiring an address to be present in the queue more than once for a request to be intercepted.

4.3.1.3 Performance of congestion-based optimizations

In this section I compare SCI, static address-based GLOW, combining, and AGENT DETECTION. Combining only observes requests delayed in the message queues because of congestion, while AGENT DETECTION employs a 128-entry recent-addresses queue to discover repetition in the addresses. The results in this section show that combining is sensitive to application and network characteristics, while AGENT DETECTION always works better and, in addition, it is insensitive to the size of the recent-addresses queue. Table 9 shows the speedup of the three schemes over SCI running on the same number of nodes. Results for BARNES and CG are presented in a later section (Section 4.5).

		GAUSS			SPARSE			APSP			TC		
	Nodes	**C**	**A**	**S**	**C**	**A**	**S**	**C**	**A**	**S**	**C**	**A**	**S**
2-D	16	1.01	1.01	1.04	1.00	1.11	1.19	0.98	0.99	1.04	1.00	1.01	1.04
	32	1.08	1.11	1.14	1.00	1.10	1.13	0.99	1.08	1.11	1.00	1.09	1.14
	64	1.19	1.45	1.53	1.00	1.29	1.26	1.00	1.40	1.52	1.01	1.42	1.55
	128	1.61	2.01	2.22	1.01	1.29	1.26	1.01	1.97	2.20	1.01	1.96	2.22
3-D	16	1.01	1.00	1.03	1.00	1.08	1.16	0.98	0.98	0.93	1.00	1.01	1.04
	32	1.08	1.13	1.15	0.99	1.16	1.24	0.99	1.10	1.17	1.00	1.13	1.19
	64	1.32	1.52	1.58	1.00	1.25	1.38	1.01	1.51	1.62	1.01	1.53	1.67
	128	1.80	2.31	2.44	1.01	1.33	1.31	1.03	2.38	2.59	1.04	2.39	2.64

Table 9. Speedup using combining (C), AGENT DETECTION (A), and static GLOW (S) over SCI on 16 to 128 nodes, 2 and 3 dimensions.

***GAUSS*—**Figure 4.18 shows the normalized speedups (with regard to SCI on 16 nodes) for the GAUSS program. The two graphs present results for the 2- and 3-dimensional networks. Combining reaches about half the performance improvement of static GLOW while AGENT DETECTION remains within 5% of the performance of static GLOW.

***SPARSE*—**For this program (Figure 4.19) AGENT DETECTION with recent-addresses queues *out-*

performs static GLOW (in the 64- and 128-node systems in 2 dimensions and in the 128-node system in 3 dimensions). This is because SPARSE actually contains more widely-shared data than just the vector ***X*** described previously (Section 2.3.2) and AGENT DETECTION can handle them effortlessly and better than static GLOW. AGENT DETECTION performs up to 1.29 times faster than SCI in 2 dimensions and up to 1.33 times faster in 3 dimensions. Combining fails to provide any significant performance improvement.

APSP and TC—APSP (Figure 4.20) and TC (Figure 4.21) show similar behavior. For both programs combining again fails to show any performance improvement while AGENT DETECTION works satisfactorily compared to the static, address based GLOW.

To summarize the results: AGENT DETECTION consistently tracks the performance of static GLOW while combining only works for some program (GAUSS and BARNES). These results show that combining is sensitive to the congestion characteristics of the application. The behavior of combining also changes depending on the network characteristics (*e.g.*, link or switch latency), while the behavior of AGENT DETECTION is largely unaffected by the number of intercepted requests.

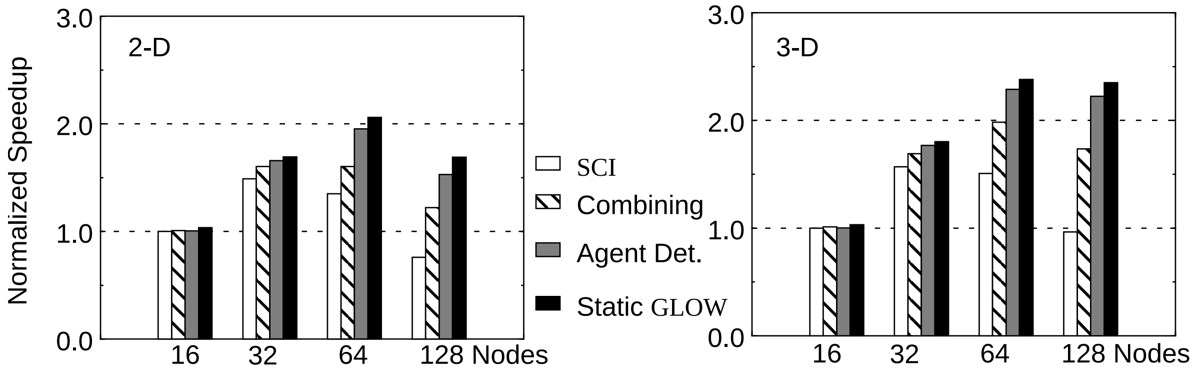


Figure 4.18. Speedup for GAUSS (with respect to SCI on 16 nodes) in 2 and 3 dimensions (16 to 128 nodes).

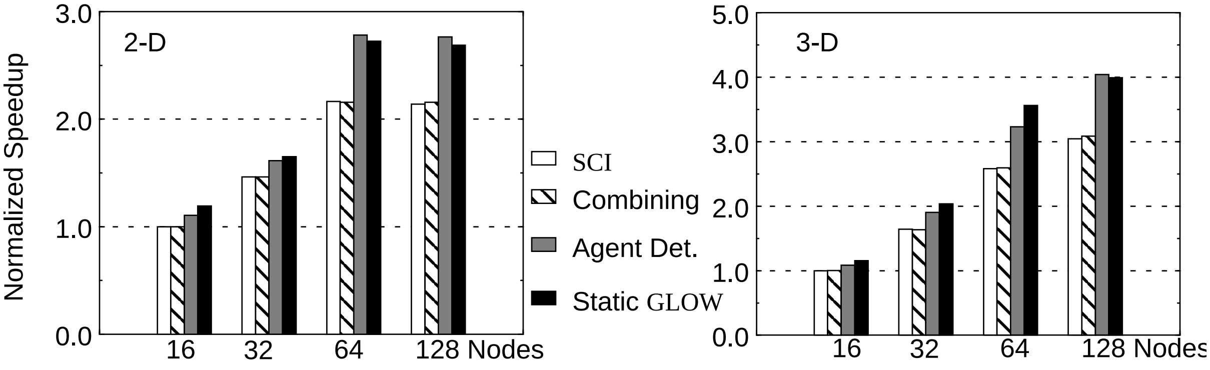


Figure 4.19. Speedup for SPARSE (with respect to SCI on 16 nodes) in 2 and 3 dimensions (16 to 128 nodes).

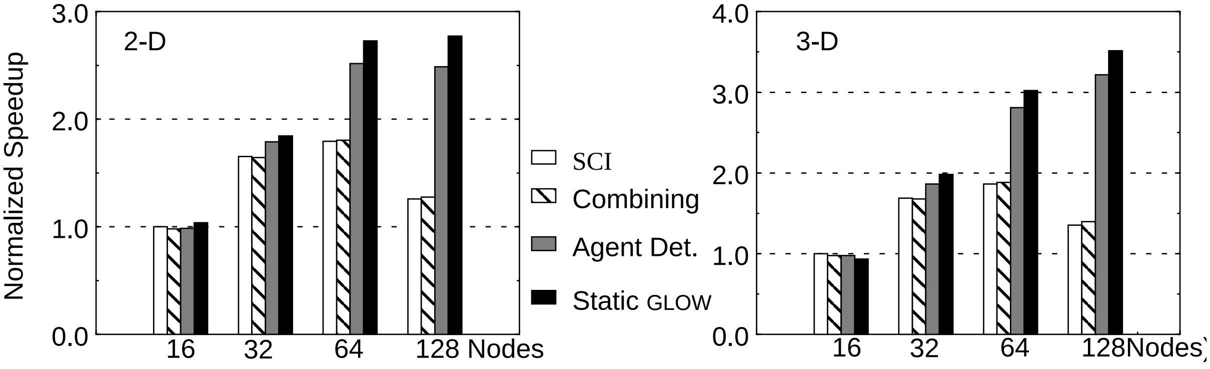


Figure 4.20. Speedup for APSP (with respect to SCI on 16 nodes) in 2 and 3 dimensions (16 to 128 nodes).

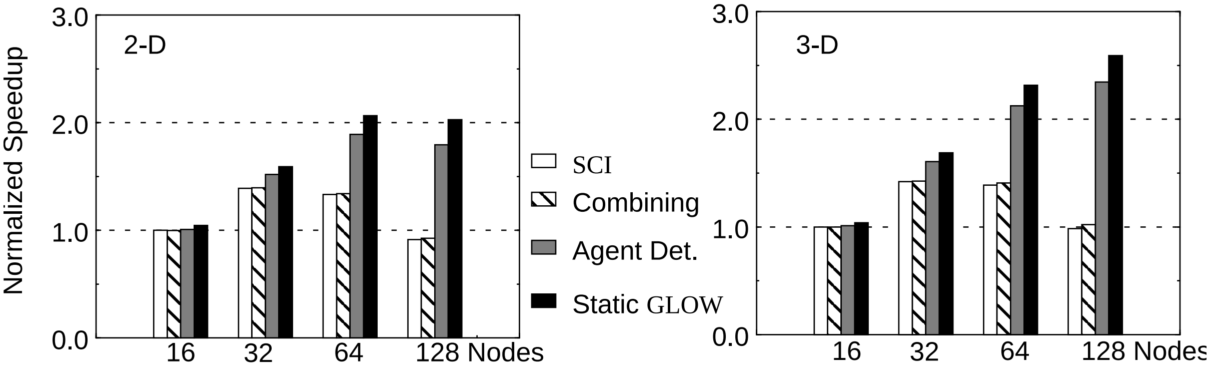


Figure 4.21. Speedup for TC (with respect to SCI on 16 nodes) in 2 and 3 dimensions (16 to 128 nodes).

Sensitivity to window size**—**One interesting result is that AGENT DETECTION is largely insensitive to the size of the recent-addresses queue, especially for the larger systems. However, the programs I use in this work are small scientific codes and other workloads may need large queues to capture repetition in requests.

Figure 4.22 shows the performance of the GAUSS and APSP programs for four different sizes of the recent-addresses queue: 8, 32, 128 and 256 entries. Other benchmarks exhibit similar behavior. It is evident in Figure 4.22 that the window size does not seriously affect the performance of AGENT DETECTION. In fact, for GAUSS, the smallest window of size 8 performs slightly better than the larger windows. This is because with larger windows there is the possibility of intercepting requests for non-widely-shared data (thus incurring the overhead of the extensions when there is no benefit), simply because they are repeated often. The implication of the insensitivity to the window size is that the recent-addresses queue can be made small and fast (*i.e.*, without unwanted side-effects in the performance of the switch nodes) while still performing well.

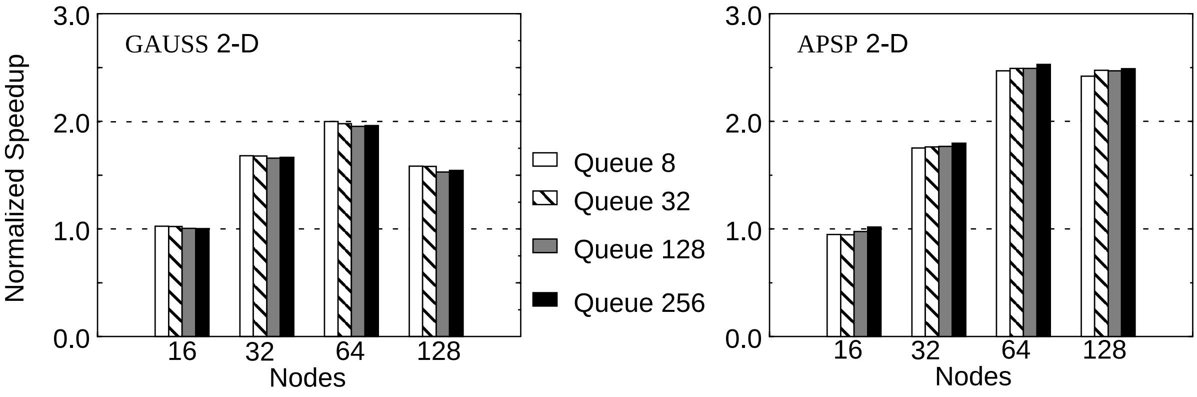


Figure 4.22. Sensitivity analysis for the size of the recent-addresses queue (speedup of AGENT DETECTION with respect to SCI on 16 nodes).

Sensitivity to switch latency**—**That AGENT DETECTION works well for all benchmarks examined in this thesis, while combining works partially for only one of them, suggests the latter is sensitive to the congestion characteristics of applications. To confirm that combining is also sensitive to network parameters (while AGENT DETECTION is not) I perform a sensitivity study on network parameters. In particular, I examine what happens when the switches are slowed down by increasing the latency to transfer a message from one ring to another through the switch.

Other network parameters include the latency of the point-to-point links that comprise the rings and end-point latencies at the node ring interfaces. However, increasing these latencies compared to the switch latency actually decreases congestion, because this tends to space messages farther apart. In contrast, increasing the switch latency creates more congestion.

The results presented in the previous sections assume very aggressive switches whose latencies are equal to the point-to-point link latencies (10 processor cycles). The minimal congestion we observe is mainly a result of multiple messages from different rings being routed to the same destination. To observe significant congestion I increase the switch latency eight-fold.

Figure 4.23 shows results for the four benchmarks for a system with slow switches. I use the same base case as before, to assess the effect of the slow switches on performance. Thus, I derive speedups by dividing the execution time of SCI on 16 nodes with fast switches by the execution time of SCI, combining, AGENT DETECTION and static GLOW with slow switches (for 16 to 128 nodes).

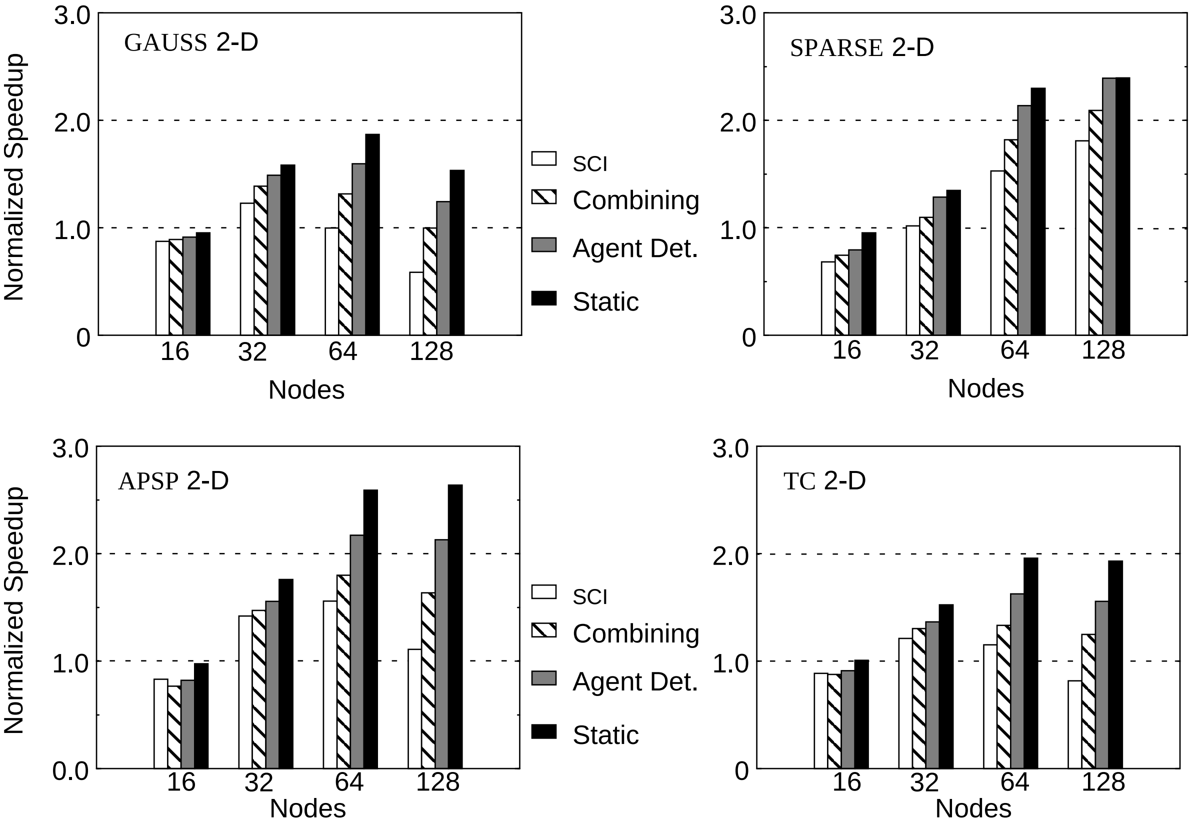


Figure 4.23. Speedup results with slower switches. Contention in the switch nodes makes ordinary combining competitive but it also slows down the whole system. Speedups for SCI, combining, dynamic and static GLOW are shown with respect to SCI on 16 nodes with fast switches.

Because of the slow switches, SCI exhibits lower speedups than before (see Figure 4.18). However, static and AGENT DETECTION are affected less than SCI. This is because the GLOW extensions considerably reduce the number of ring crossings —the switches are used less to transport messages across rings. Table 10 summarizes the speedups for fast and slow switches for the four programs (for 16 to 128 nodes in 2 dimensions). Comparing the speedups we observe that the greater the GLOW benefit for systems with fast switches, the less is the performance hit using slow switches. In other words, when GLOW works well the importance of the switch latency diminishes.

	GAUSS (FAST/ SLOW)				SPARSE(FAST/ SLOW)					APSP (FAST/SLOW)				TC (FAST/SLOW)			
N.	SCI	**C**	**D**	**S**	SCI	**C**	**D**	**S**	N.	SCI	**C**	**D**	**S**	SCI	**C**	**D**	**S**
16	1.00/0.87	1.01/0.89	1.00/0.91	1.03/0.95	1.00/0.68	1.00/0.72	1.10/0.74	1.19/0.79	16	1.00/0.83	0.95/0.76	0.98/0.82	1.04/0.97	1.00/0.88	0.99/0.87	1.00/0.91	1.04/1.00
32	1.49/1.23	1.60/1.38	1.66/1.49	1.69/1.58	1.46/1.01	1.43/1.10	1.61/1.29	1.65/1.35	32	1.65/1.42	1.63/1.47	1.79/1.56	1.84/1.76	1.39/1.21	1.40/1.30	1.52/1.36	1.59/1.52
64	1.35/0.99	1.69/1.31	1.95/1.60	2.06/1.87	2.16/1.52	2.20/1.82	2.78/2.14	2.73/2.30	64	1.79/1.56	1.81/1.80	2.51/2.17	2.73/2.59	1.33/1.15	1.35/1.33	1.89/1.63	2.06/1.96
128	0.76/0.58	1.23/1.00	1.53/1.24	1.69/1.53	2.14/1.80	2.18/2.09	2.76/2.39	2.69/2.39	128	1.26/1.11	1.30/1.63	2.48/2.13	2.77/2.64	0.91/0.82	0.95/1.25	1.79/1.56	2.03/1.93

Table 10. Speedup using fast and slow switches for SCI (SCI), combining (C), AGENT DETECTION (D), and static GLOW (S). The base case is SCI on 16 nodes with fast switches.

The performance of combining relative to AGENT DETECTION and static GLOW improves for all benchmarks and especially for the three benchmarks whose performance was previously unaffected by combining (SPARSE, APSP, and TC). Combining reaches at least half the performance benefit of AGENT DETECTION. The performance of AGENT DETECTION is still higher than combining but drops relative to the performance of static GLOW. Again this has to do with the utilization of the switches. Static GLOW makes the least use of the switches to transport messages across rings and therefore is able to maintain higher performance.

4.3.2 DIRECTORY DETECTION of widely-shared data

In this scheme, henceforth called DIRECTORY DETECTION, the directory is responsible for identifying widely-shared data. DIRECTORY DETECTION is the address-based dynamic scheme for GLOW. It resembles limited pointer directories such as Dir$_i$B [8], which switch from point-to-point messaging to broadcasting if the number of readers exceeds some threshold. Similarly, DIRECTORY DETECTION identifies wide sharing by keeping track of the number of readers and informs the nodes in the system about the nature of the data. After the nodes learn that a data block is widely shared they use the GLOW extensions to access it.

DIRECTORY DETECTION works when data blocks are widely shared for many read-write cycles. Since the opportunity to optimize the first read-write cycle may be lost, this scheme does not provide any performance improvement when data blocks are widely shared only once. Furthermore, it may degrade performance by incorrectly treating such data blocks as widely shared when they are not.

In contrast to AGENT DETECTION that is transparent to the rest of the system, DIRECTORY DETECTION requires modifications to the directory protocols. This is feasible in many commercial or research systems where the cache coherence protocols are implemented as a combination of software and hardware and can be upgraded (*e.g.*, STiNG [68], Typhoon [80], Flash [60]).

Since the home-node directory is a single point in the system that can observe the request stream for its data blocks, it is in a position to detect wide sharing. In directories such as Dir_nNB [8] the number of readers is readily available. However, in SCI, where only a single pointer to the head of the sharing list is kept, the directory must count the number of reads between writes. A counter associated with each data block counts up for each read and is reset with a write. Data blocks for which the corresponding counter reaches some threshold value are deemed widely shared. In SCI this is a heuristic since:

- The directory in SCI does not always see the write-faults. In SCI, when the writer is the head node it has permission to write the cache block (provided that it will invalidate the sharing list). Therefore, write-faults of the head nodes are not communicated to the directory. In contrast, when the writer is a middle or tail node, it has to roll out of the sharing list and re-attach in front of the sharing list as a head node. This makes the write-fault visible to the directory. For widely-shared data, where the sharing list is very large, it is unlikely the writer will be the head of the list. If the writer does not change frequently, it is most likely the tail node. In the random case, a writer has only *1/N* probability to be the head of a sharing list of *N* nodes. To address this problem, I use a *sticky* bit in every directory entry to indicate wide sharing. This bit is set when the counter exceeds the predefined threshold and is reset by writers who verify the directory's claim that a data block is widely shared.

- The directory might incorrectly deem a data block as widely shared just by seeing multiple reads from the same node. Determining whether read requests actually come from different nodes is possible if we keep a bitmap of the readers (similarly to Dir_nNB [8]). However, this would be an expensive addition to the SCI directory and I do not examine it further.

In the evaluation I extended the SCI directory tag with a small 2-bit saturating counter and a wide sharing bit. I have tested this scheme and found that the performance of DIRECTORY DETECTION does not change for threshold values above four.

When the directory discovers a data block to be widely shared this information must be transmitted to all nodes in the system so that the GLOW extensions will be used to access this data block in the future (via GLOW requests). To transfer this information from the directory to the nodes I propose three methods: (i) direct-notification, (ii) broadcast, and (iii) invalidation carried notification:

- **Direct notification**: Upon detecting widely-shared data, the directory returns responses to new requests indicating (with a single set bit in the response) that the data accessed are widely shared. Nodes will use GLOW the next time they access this data block.
- **Broadcast**: Upon detecting widely-shared data, the directory broadcasts the address of the data block so that all nodes learn about it. Such broadcast consumes considerable bandwidth and typically, by the time it reaches the nodes they have already sent their requests without using GLOW. If the data block is not widely shared beyond this read cycle, the opportunity to optimize wide sharing is lost. The performance of this method in tests was not satisfactory and I have not examined it further.
- **Invalidation carried notification**: Information about a widely shared block is transferred to the nodes when the data block is written. Upon a write, the directory (or the writer in SCI) sends invalidation messages that notify the nodes about the nature of the data block. Since the information is carried with the invalidation messages it does not consume extra bandwidth. However, with this method only the nodes that participated in the first read will learn about the data block and the opportunity to optimize the first read cycle is lost since the information is carried to the nodes after the data block is written.

When information about data blocks is transferred to the nodes, it needs to be stored for future reference. The nodes need to remember what data blocks are widely shared so they can use GLOW to access them. I propose two solutions:

- **Address Tables**: Information about which data blocks have been found to be widely shared is kept in address tables similar to those described for the static address-based GLOW (Section 4.1.1).

- **Hot Tags**: Information about wide sharing is stored in the cache tags themselves. However, this information is not thrown away when the cache tags are invalidated. Instead the tags are kept in the cache like other valid tags and are not available for immediate replacement as junk.[5] I call these tags *hot tags*. If a node tries to access a hot tag it will experience a miss but it will send a request for widely-shared data which will be handled by the GLOW agents.

4.3.2.1 Adapting back

A consideration about DIRECTORY DETECTION is that it adapts easily from non-wide sharing to wide sharing but it is very hard to adapt the other way around. If a data block is widely accessed only once, the directory will observe very few read requests between writes after the first read-write cycle. However, it cannot determine whether it sees very few requests because the data block is no longer widely shared, or because the GLOW extensions absorb most of the requests in the network. On the other, hand the directory might see many read requests without an intervening write simply because sometimes write-faults are not visible to the directory.

To solve these problems, once a directory detects a large number of reads for a data block it assumes that it is widely shared and continuously indicates this in its responses until it is directed to do otherwise. The sticky wide-sharing bit in the directory tag is set to indicate wide sharing. The writers are responsible for verifying—and if necessary, correcting with an extra transaction—the directory's claim that a data block is widely shared, by counting the number of nodes they invalidate (in SCI the writer node invalidates all other sharing nodes rather than

[5] But they are candidates under the cache's replacement algorithm (*e.g.*, LRU, random, etc.).

the directory). To get an accurate count of the nodes in the sharing tree, the GLOW agents recursively count the number of SCI nodes they have invalidated. They return this count, added to the counts they receive from any downstream neighbors, to the node that sent the invalidation message. The recursive invalidation algorithm (Section 3.5.4) guarantees that the writer will eventually learn the total number of nodes in the sharing tree.

With this scheme, DIRECTORY DETECTION works as follows:

1. The first read cycle requests go to the directory. If the counter exceeds a pre-defined threshold the sticky wide-sharing bit is set. Responses carry this bit to the nodes.
2. Nodes which receive responses with the wide-sharing bit set store the information in address tables or in the cache tags themselves (which might become hot tags when invalidated).
3. When the data block is written, the writer gets the wide-sharing bit from the directory and checks to see whether the data block was actually widely shared. If the number of invalidated nodes does not exceed the pre-defined threshold, the writer resets the wide-sharing bit (but not the sharing counter) in the directory.
4. In the next read cycle the data block might remain widely shared or change to a different (non-wide sharing) access pattern. In any case, the nodes believe the data block to be widely shared and will use GLOW to access it. The responses of the GLOW agents to the nodes carry the wide-sharing bit received from the directory. If the wide-sharing bit has been reset the nodes erase the relevant information from the address tables or the cache tags. However, the wide-sharing bit might be set anew in the directory if the request counter exceeds the threshold.

5. Goto 3.

This scheme is complex and adapts slowly, requiring a full read-write cycle to detect a change in the behavior of the data block. A pathological case that can result in performance loss occurs when the data block becomes migratory after being marked widely shared. In this case, many subsequent reads will incur the overhead of invoking GLOW agents because it will take many read-write cycles to erase the wide sharing information from all the nodes in the system. However, I have not encountered the transition from wide sharing to migratory sharing in any of the programs I use in this work.

The complexity of the scheme to adapt back makes DIRECTORY DETECTION unattractive. Fortunately, the hot tag concept provides a natural way to adapt from wide sharing to non-wide sharing. If the data block is not widely accessed, the hot tags in the system will be replaced and the nodes will lose the information that the block was widely shared.

Alternatively, wide-sharing information in the directory can be deleted randomly using a counter per directory.[6] The counter is decremented with every access to data blocks that have the wide-sharing bit set. When the counter crosses zero, the wide-sharing bit of the currently accessed data block is reset. If the wide-sharing bit of a data block is wrongly reset the data block has to be tagged widely shared anew the next time it is accessed. The appeal of this adapt-back technique is that it is easy to implement.

To evaluate DIRECTORY DETECTION, I use direct notification to transfer information from the directories to the nodes. This information is stored in the cache tags (which become hot tags

[6] This technique was suggested by M. D. Hill.

upon invalidation). Hot tags are easier to implement than address tables and they provide an easy way to adapt back. Results are presented in Section 4.5.

4.4 Dynamic instruction-based optimizations for wide sharing

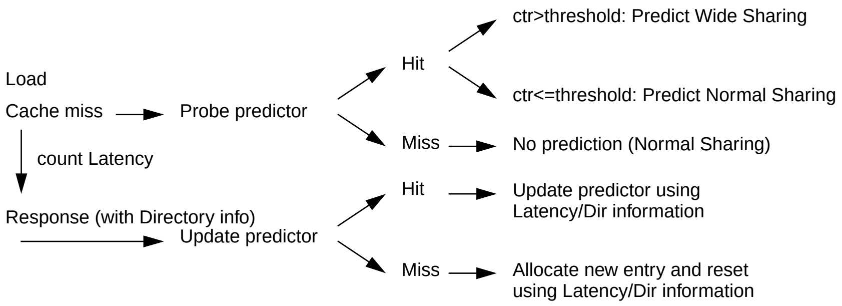
The congestion-based and the directory-based schemes detect wide sharing by examining addresses. In this work, I propose a novel approach to identify and optimize sharing patterns. This approach is based on dynamically examining instructions rather than addresses; in essence examining what the program is trying to do, rather than what happens to the data. In Section 4.1.2, I have proposed a static approach where the user identifies instructions that access widely-shared data. In this section, I propose its dynamic counterpart. Specifically, I propose a mechanism to predict which load instructions are likely to access widely-shared data.

The prediction is based on previous history. If a load accessed widely-shared data in the past then it is likely to access widely-shared data in the future. This behavior can be traced to the way parallel programs are structured. For example, in Gaussian elimination the pivot row is widely shared and it is accessed in a specific part of the program. Therefore, once the load instruction that accesses the pivot row has been identified it can be expected to continue to access widely-shared data. I found that this prediction is very strong for all the benchmarks examined in this thesis.

For wide sharing, a simple predictor works well. The predictor is a small fully-associative cache with LRU replacement. A predictor entry comprises an address tag for the program counter (PC) of the load instruction and a small (*e.g.*, 2-bit) saturating counter used to make predictions. To reduce the size of the predictor entries only a few PC bits (*e.g.*, a few low order bits) can be stored in the predictor entry. Alternatively, the address field can be omitted altogether. In this case, the predictor is a directly indexed column of counters. Although truncating or eliminating the address field may introduce mistakes in accessing or updating the predictor, the smaller size of each predictor entry translates to many more entries for a fixed transistor budget. This may be preferable for large programs.

The first few times a load misses and we discover that it is accessing widely-shared data its PC is inserted in the predictor. In subsequent misses, this small prediction cache is probed using the PC of the load. If the probe results in a hit and the corresponding counter indicates that the load accessed widely-shared data a few times in a row, GLOW is used for the new access. I have identified two citeria to determine whether a load accessed widely-shared data:

- **latency**: If the miss latency exceeds a threshold value the load is considered to have accessed widely-shared data. The latency scheme is discussed in detail in Section 4.4.1.
- **directory feedback**: The directory is responsible to determine whether a load accesses widely-shared data. This scheme is discussed in detail in Section 4.4.2.



ctr>threshold: Predict Wide Sharing
Hit
Load
Cache miss
Probe predictor
ctr<=threshold: Predict Normal Sharing
Miss
No prediction (Normal Sharing)
count Latency
Hit
Update predictor using Latency/Dir information
Response (with Directory info)
Update predictor
Miss
Allocate new entry and reset using Latency/Dir information

Figure 4.24. Instruction-based prediction for wide sharing.

Figure 4.24 depicts a working example of instruction-based prediction for wide sharing. Upon a load-miss the predictor is probed for information. At first the predictor is empty (predictor miss). The load-miss generates a coherent read. The response to this read will update the predictor according to the criterion used (latency or directory-feedback). A new entry is allocated in the predictor and its counter is reset to 0. This will happen two more times before the counter exceeds the threshold. At this point a predictor probe returns a positive prediction for wide sharing and instead of an ordinary read, a special GLOW read is issued. The GLOW read will trigger the creation of sharing trees in the network.

4.4.1 Latency

Whether a load accessed widely-shared data can be judged by its miss latency. A very large miss latency suggests an access to widely-shared data. Using latency as the basis for the prediction is not as farfetched as it sounds: access latency of widely-shared data is significantly larger than the average access latency of non-widely-shared data. This is because of network contention and most importantly because of contention in the home-node directory, which becomes a hot spot [78]. For example, the latency of 128 requests going to the same node is much higher that the latency of 128 requests going to 128 different nodes. Microbenchmark results discussed earlier (see Section 3.5.5) confirm this observation. The latency threshold for widely-shared data is a *tuning* parameter that can be set independently for different applications.

This scheme does not guarantee that it will detect wide sharing and only wide sharing; it can be invoked by misses that have long latency for other reasons (*e.g.*, the home node is a hot spot for other data). In these situations invoking GLOW agents unnecessarily does not have a significant impact on performance (the miss latency was very large to begin with). In fact, in some situations invoking a GLOW agent even for non-wide sharing may alleviate performance problems.

***Adapting back*—**The simple instruction-based prediction described above adapts easily to wide sharing but it is not trivial to adapt the other way around. If a load ceases to access widely-shared data and GLOW extensions are used for non-widely-shared data, very few nodes incur all the overhead of building scalable sharing trees in the network without other nodes benefiting. Thus, we need to detect when wide sharing has ceased and refrain from using GLOW. Here, I describe a scheme to adapt back.

It is virtually impossible to obtain reliable feedback for the latency-based prediction because a low miss latency can be attributed either to *lack* of wide sharing *or success* of the GLOW extensions in handling wide sharing. To adapt back in this situation we can use an expiration counter (I call it *poison* counter) for each predictor entry. The counter is set to a non-zero value and each time the predictor entry is used the counter is decremented. When the poison counter hits zero the predictor entry is deleted. This scheme can also be used in the directory-feedback prediction scheme described below.

4.4.2 Directory feedback

This scheme is the result of combining the instruction-based and address-based techniques: it is instruction-based prediction, but it uses DIRECTORY DETECTION (the dynamic address-based scheme presented earlier in Section 4.3.2) to detect when instructions access widely-shared data.

In DIRECTORY DETECTION, information about the nature of the data is supplied by the directory. If the number of reads between writes exceeds a certain threshold, the directory’s responses indicate that the data block is widely shared. The threshold is again a *tuning* parameter and for this work I set it to 4 (*i.e.*, more than 4 out of the 32 nodes reading is considered wide sharing). This scheme is more focused on wide sharing than the latency-based scheme, which could be fooled by random long-latency operations. However, in this scheme, a few first nodes will receive responses from the directory claiming that data are not widely shared and the rest of the nodes will receive responses claiming that data are widely shared. Thus, it may be possible that initially the same load instruction is treated differently in various nodes. Eventually, if the load instruction is executed repeatedly, all nodes will learn about the nature of the accessed data. This is because, the order in which the nodes read the data varies considerably, especially when the same load instruction reads data in different home nodes.

***Adapting back*—**The problem with directory-feedback prediction is that the actual number of sharers cannot be reliably tracked by the directories when GLOW is in use, because of GLOW's read-combining. To solve this problem the same mechanism for adapting back as the one discussed in Section 4.3.2 is used. To summarize: once a directory discovers a widely shared block, it continuously indicates this in its responses until it is directed to do otherwise. The writers are responsible to verify (and correct if necessary) the directory's claim that a data block is widely shared, by counting the number of nodes they invalidate. The method is based on sticky wide-sharing bits in the directory that are carried to the nodes with every directory response. In the instruction-based prediction scheme, these bits—when set—are the confirmation that an instruction accesses widely-shared data and increase the corresponding predictor counter. When these bits are not set the predictor counter is decremented.

For the benchmarks that *do have* widely-shared data, I found no benefit in using the schemes for adapting back. In fact the performance benefit diminishes slightly. These schemes are intended as a safety device for situations where incorrect wide-sharing prediction can have harmful effects on performance (see Section 4.4.4).

4.4.3 Implementation issues

Contrary to the uniprocessor/serial-program context where predictors are updated and probed continuously, instruction-based prediction for wide sharing *only* updates the prediction history and probes the predictor in the case of a miss. This makes the prediction mechanism much less frequently accessed. Furthermore, its latency is not in the critical path since we only need its prediction on misses, which are of significant latency anyway. Thus, it is not a potential bottleneck nor does it add any cycles to the critical path.

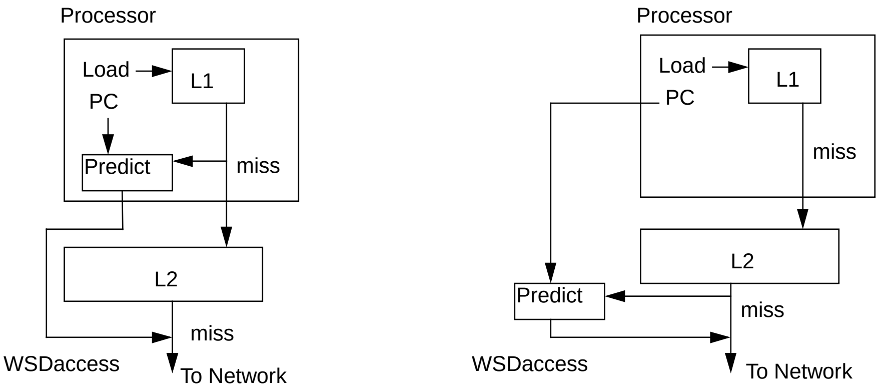


Figure 4.25. Datapaths for the prediction mechanisms (according to its location).

The prediction mechanism needs input from both the processor core (the PC of the load that missed) and the coherence mechanisms (signaling of coherence events, latency information or feedback from the directory). There are two choices for the location of the prediction mechanism: inside or outside the processor. Figure 4.25 shows the two cases. When the mechanism is inside the processor it is probed and possibly updated when a load misses in the L1 cache. Care must be taken for the update since the information (*i.e.*, latency information or directory feedback) from the coherence mechanisms is only correct when a miss also occurs in the L2 cache and it is routed toward the home node. When the prediction cache returns a positive prediction a special request for widely-shared data is issued outside the processor. Since the type of the request only matters when we also have a miss in the L2 cache, the request may lag a cycle behind the L2 access without degrading performance. The mechanism operates similarly when it is implemented outside the processor with the assumption that the PC of the corresponding load instruction is available outside the processor on an L2 cache miss. The resulting request could be delayed one cycle until a prediction is obtained. However, this cycle can be hidden by cache coherence protocol or network access latencies. I do not distinguish between the two implementations since I model a processor with a single cache and a prediction cache that does not delay the corresponding requests.

Wherever this mechanism is implemented it necessitates a custom approach. If it is inside the processor it requires a custom designed core, and if it is outside the processor it requires that the PC of a load that misses be known outside the processor (I am not aware of any commercial processor with this feature). Despite this drawback, two arguments support this method: (i) it is highly successful in the context of wide sharing, and (ii) such prediction mechanisms can be used to optimize not only wide sharing but also other access patterns (such as migratory sharing discussed in Chapter 5 and producer-consumer sharing discussed in Chapter 6), thus the cost of the prediction hardware will be amortized over multiple optimizations.

4.4.4 Results

To study the performance of instruction-based prediction for wide sharing I present results for the two prediction schemes (latency-based and directory-feedback) and the DIRECTORY DETECTION scheme described earlier (Section 4.3.2). Table 11 shows the speedups in 32- and 64-node systems for five benchmarks with wide sharing and for two control benchmarks that do not exhibit wide sharing. The two instruction-based prediction schemes perform almost identically, yielding speedups of up to 1.30 for BARNES in 32 nodes and up to 1.54 for TC in 64 nodes. They both outperform DIRECTORY DETECTION in all benchmarks (in the case of APSP and TC by a significant margin). Only the performance of one of the control benchmarks (CHOLESKY) suffers because of instruction-based prediction optimizations and in particular from the directory-feedback scheme. However, when the mechanism to adapt back is enabled the negative performance impact is reduced. The other control benchmark (OCEAN) is affected negatively by the address-based scheme.

Table 12 contains predictor statistics for the directory-feedback scheme (results for the latency-based scheme are similar). For these schemes—because they do not adapt back—the number of predictor hits is approximately equal to the number of predictor probes and the number of optimizations is approximately equal to the number of predictor hits. A striking result is the amazingly small number of predictor entries allocated for each benchmark.

Figures 4.26, 4.27, and 4.28 compare SCI, static address-based GLOW and the two instruction-based prediction schemes (latency and directory-feedback). The instruction-based schemes perform comparably, but in some instances one is slightly faster than the other. They both fare well against the static, address-based GLOW and in GAUSS and BARNES they actually out-per-

		32 nodes			64 nodes		
		Instruction-based prediction		Address-based prediction	Instruction-based prediction		Address-based prediction
		Latency	Directory-feedback / adapt-back	Adaptive Directory-detection	Latency	Directory-feedback / adapt-back	Adaptive Directory-detection
Wide Sharing Benchmarks	GAUSS	1.20	1.19	1.13	1.66	1.64	1.43
	SPARSE	1.06	1.04	1.13	1.32	1.28	1.25
	APSP	1.11	1.12	1.00	1.53	1.52	1.00
	TC	1.14	1.14	1.01	1.53	1.54	1.02
	BARNES	1.30	1.29	1.27	1.13	1.14	1.13
Control Benchmarks	OCEAN	1.00	1.00	0.91	1.00	1.00	0.95
	CHOLESKY	1.00	0.91 / 0.99	0.96	1.00	0.92 / 0.99	1.02

Table 11. Results for wide sharing optimizations (speedup over SCI).

Nodes	Statistics	GAUSS	SPARSE	APSP	TC	BARNES	CHOLESKY
32	Total loads allocated in all predictors	249	643	94	91	1555	349
	average per node	8	20	3	3	49	11
	maximum	9	25	3	3	54	35
64	Total loads allocated in all predictors	557	1238	180	185	3251	711
	average per node	9	20	3	3	51	11
	maximum	11	24	3	3	56	39

Table 12. Statistics for wide sharing prediction (directory-feedback scheme).

form the static scheme (in BARNES by a large margin).

4.4.5 Static instruction-based optimizations for widely-shared data

Although for many programs (*e.g.*, GAUSS, SPARSE, BARNES) it is straightforward to point out instructions that access widely-shared data, I have not studied this manual technique further. Since such manual techniques are highly subjective, results may vary depending on factors

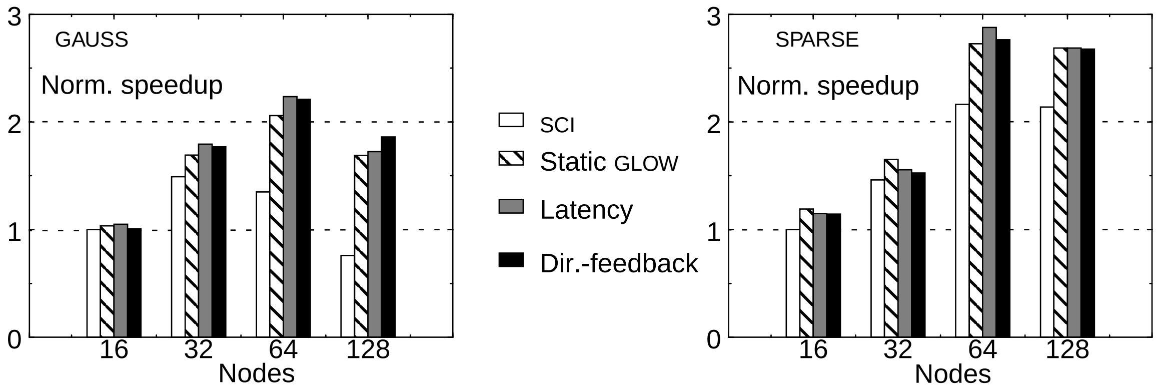


Figure 4.26. Comparison of SCI, static GLOW, and the two dynamic instruction-based schemes for GAUSS and SPARSE. Base system is SCI on 16 nodes.

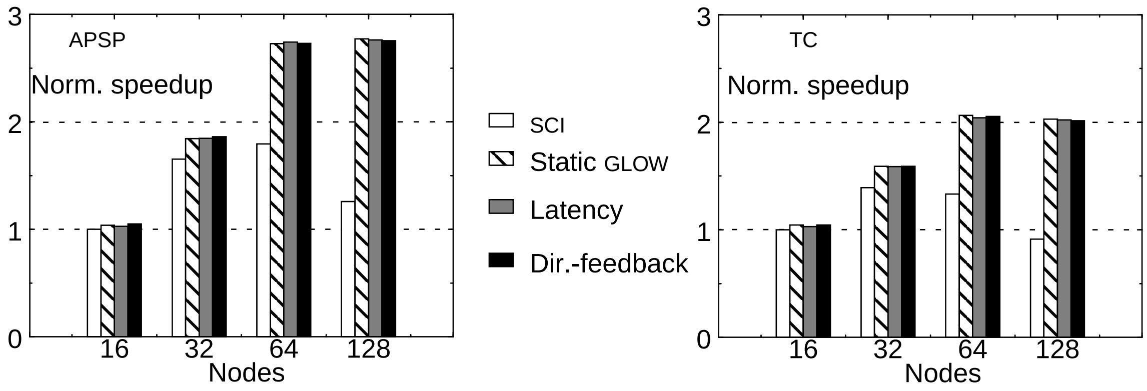


Figure 4.27. Comparison of SCI, static GLOW, and the two dynamic instruction-based schemes for APSP and TC. Base system is SCI on 16 nodes.

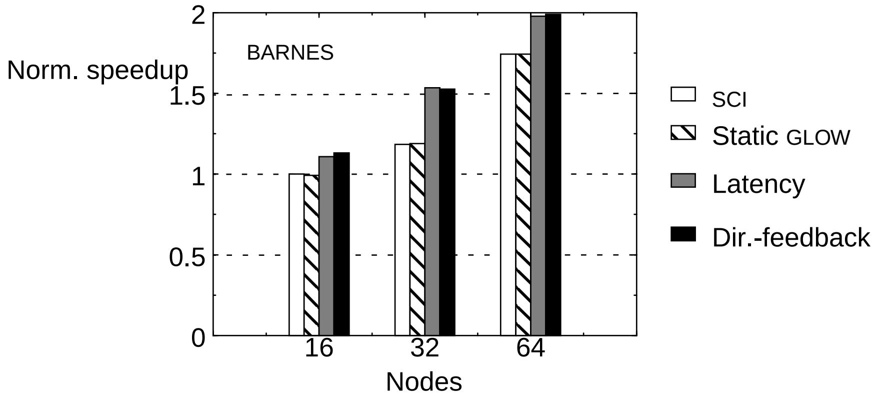


Figure 4.28. Comparison of SCI, static GLOW, and the two dynamic instruction-based schemes for BARNES. Base system is SCI on 16 nodes.

such as readability of the source code, etc. Instead, I have used profiling to automatically obtain a list of the loads that access widely-shared data. For profiling I have used the simulations of the dynamic instruction-prediction schemes. At the end of each simulation run, the contents of the prediction cache are dumped and used as the list of instructions that access widely-shared data. This is possible since, for the small programs used, the contents of this cache are few (there are no replacements in a 64-entry prediction cache—see Table 12) and very stable, *i.e.*, the behavior of the loads does not change. Not surprisingly, the performance of the static instruction-based scheme is identical to the dynamic scheme used for the profiling.

It is easy to do profiling in a simulator, but in a real system, lacking support for dynamic optimizations (*i.e.*, lacking the prediction cache, the directory-detection of widely-shared data, etc.), it is impossible to do profiling as described above. Other techniques such as binary rewriting could be used to profile a program for widely-shared data. Since both manual source code annotations and the profiling in a real system may not be as successful as profiling in simulations, the performance of the dynamic schemes can be considered to be an upper bound for the performance of the static schemes.

4.5 Performance comparisons and read-run analysis

In this section I present comparisons for the six programs that exhibit wide sharing and for the various system configurations (2- and 3-dimensional networks with 16 to 128 nodes). In contrast to the previous sections, here the speedup results for each system are not normalized to the speedup of the 16-node SCI system, but to the SCI system with the same number of nodes. This gives the relative speedup of the various glow schemes over SCI, but does not provide any indication of scalability (which was explored previously). Recall that most of the programs with relatively small inputs do not scale beyond 64 nodes, so the very good GLOW speedups for the large systems correspond to cases where SCI is doing badly.

I compare SCI, static address-based GLOW, congestion-based read-combining (COMBINING) and three versions of dynamic GLOW. The first version, AGENT DETECTION, employs a 128-entry recent-addresses queue to discover repetition in data addresses. I assume that the switch latency is not affected by the additional combining or agent detection hardware present in the switches. However, this may not be the case in a realistic implementation. Results for slower switches are presented in Section 4.3.1.3 and the performance of COMBINING and AGENT DETECTION can be adjusted accordingly. In the second version, DIRECTORY DETECTION with a threshold of 4, the directory discovers the widely-shared data. In the third version, INSTRUCTION-PREDICTION, instruction-based prediction is used with the latency criterion. A miss latency threshold of 1000 cycles determines which misses correspond to widely-shared data. As a reference, in most benchmarks the average latency of all accesses is around 600 cycles.

To explain the behavior of the various GLOW schemes, I examine how they appear to change the read-runs[7] of the program from the directories' point of view. Specifically, for each scheme I plot the number of reads that correspond to read-runs of different sizes. The GLOW schemes compress these accesses toward the small read-run sizes. Each scheme's compression relates to its performance improvement. Since the directory can observe large read-runs, because either write-faults are not communicated or because a single node requests the same data multiple times because of replacements, sometimes the large write runs appear to be more that what they are in reality.

***GAUSS*—**Figure 4.29 shows the normalized speedups for the GAUSS program. Static GLOW is up to 2.22 times faster than SCI in 2 dimensions and up to 2.44 times faster in 3 dimensions. COMBINING achieves about half the performance improvement of static GLOW, while AGENT DETECTION remains within 5% of the performance of static GLOW. The DIRECTORY DETECTION scheme also works well, staying within 10% of static GLOW. Using INSTRUCTION-PREDICTION results in the largest speedups over SCI (up to 2.27 times in 2 and 2.82 times in 3 dimensions).

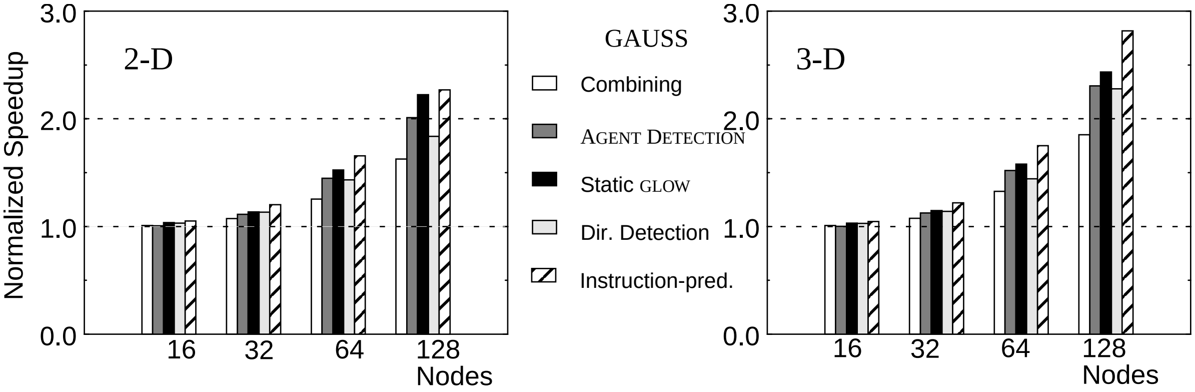


Figure 4.29. Normalized speedup (over SCI) for GAUSS in 2 and 3 dimensions (16 to 128 nodes).

[7] Defined in Section 3.1.

Figure 4.30 plots the number of reads that correspond to read-runs ranging in size from 1 to 128. The horizontal axis is the size of the read-run and the vertical axis is the number of accesses (reads/invalidates). The data are for GAUSS on 128 nodes on a 2 dimensional network. These data represent what the SCI directories observe and the correspondence of the read-run size and the degree of sharing is not exact. Because these graphs contain very large and very small numbers, I use a logarithmic scale in the vertical axis. This tends to emphasize the small numbers that would otherwise be invisible as in Figure 3.1.

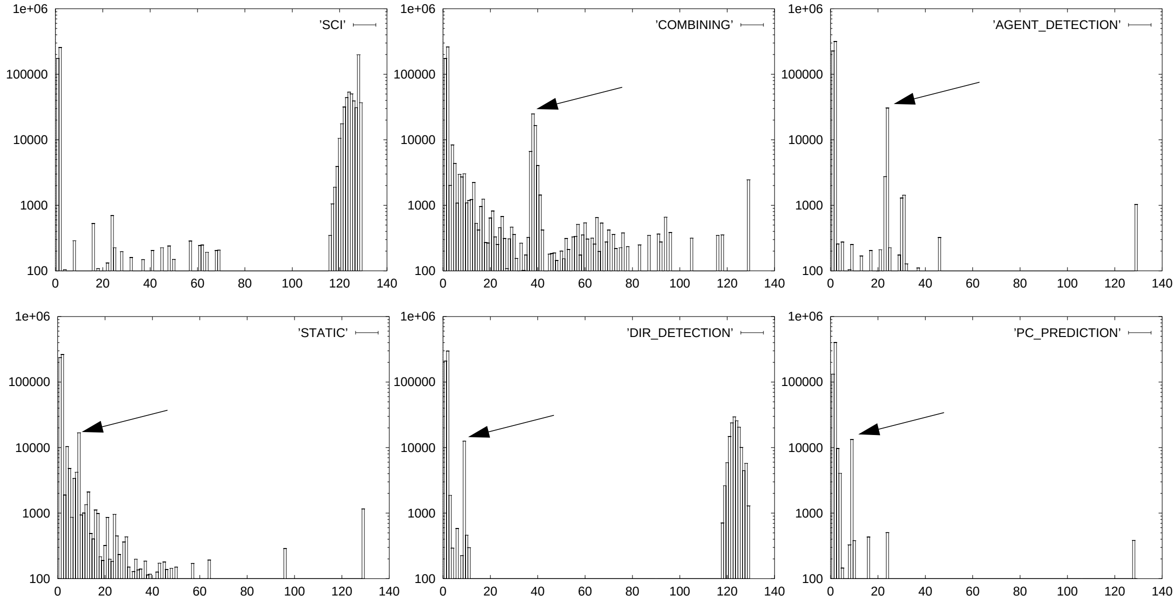


Figure 4.30. Read-run compression for GAUSS (128 nodes 2-dimensions). Accesses corresponding to large read-runs are shifted toward smaller read-runs using GLOW extensions.

The SCI graph shows a large number of accesses corresponding to large read-runs. COMBINING, AGENT DETECTION, static GLOW, DIRECTORY DETECTION and INSTRUCTION-PREDICTION all absorb a large number of requests in the network and, as a result, the directories see fewer requests between writes. Static GLOW compresses many accesses to read-runs of size 9 (pointed out in the graph). This number corresponds to 8 GLOW agents plus an extra node: for any widely-shared data block in the 2-dimensional 128-node system (8x16 nodes) there are 8 agents covering all nodes except the data block's home node. In contrast, COMBINING, which does not perform as well, compresses the read-runs from a size of 128 down to a size of about 38. AGENT DETECTION compresses most of the large read-runs to a size of 24. This means that before all 8 GLOW agents are invoked for a widely shared block, 16 requests slip by and reach the directory. DIRECTORY DETECTION eliminates the largest read-runs, converting them to read-runs of size 9 (similarly to the static GLOW). INSTRUCTION-PREDICTION gives the cleanliest spectrum of read-runs pushing most of the large ones down to a size of 9.

***SPARSE*—**For this program (Figure 4.31) AGENT DETECTION *outperforms* static GLOW (in the 64- and 128-node systems in 2 dimensions and in the 128-node system in 3 dimensions). AGENT DETECTION performs up to 1.29 times faster than SCI in 2 dimensions and up to 1.33 times faster in 3 dimensions. COMBINING fails to provide any significant performance improvement and DIRECTORY DETECTION performs on par with static GLOW. INSTRUCTION-PREDICTION is again the most successful (speedups of up to 1.33 and 1.50 for 2 and 3 dimensions respectively).

Figure 4.32 shows the compression of read-runs for SPARSE (again on 128 nodes with a 2 dimensional network). COMBINING does not perform well for SPARSE. This is also evident in its failure to affect large read-runs. AGENT DETECTION spreads the largest read-runs over the read-run spectrum, with a center around 60. This means that the number of requests that slip through the agents before they detect widely-shared data has significant variance. Static GLOW

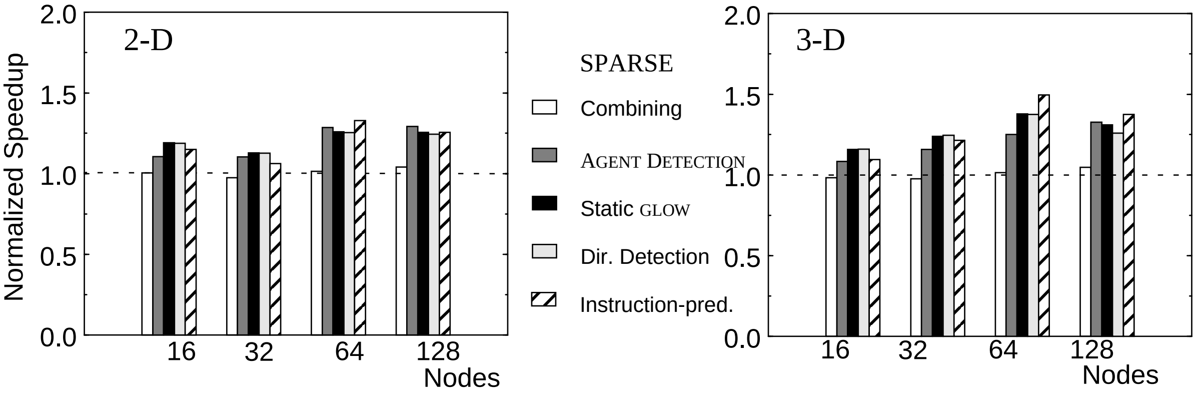


Figure 4.31. Normalized speedup (over SCI) for SPARSE in 2 and 3 dimensions (16 to 128 nodes).

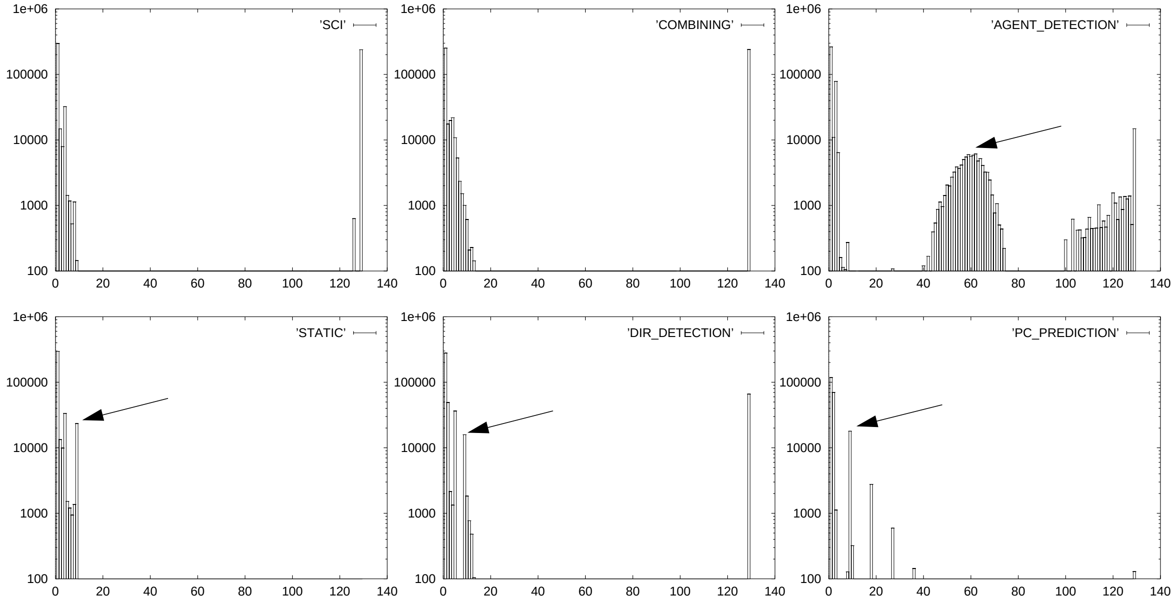


Figure 4.32. Compression of read-runs for SPARSE (128 nodes, 2 dimensions). Accesses corresponding to large read-runs are shifted toward smaller read-runs using GLOW extensions.

and INSTRUCTION-PREDICTION perform very well, most of the time allowing the directories to see only 9 requests (8 GLOW agents and the local node).

APSP and TC—For both programs, COMBINING and DIRECTORY DETECTION fail to show any performance improvement, while AGENT DETECTION performs close to static GLOW (Figure 4.33 and Figure 4.34). INSTRUCTION-PREDICTION performs almost as well as static GLOW.

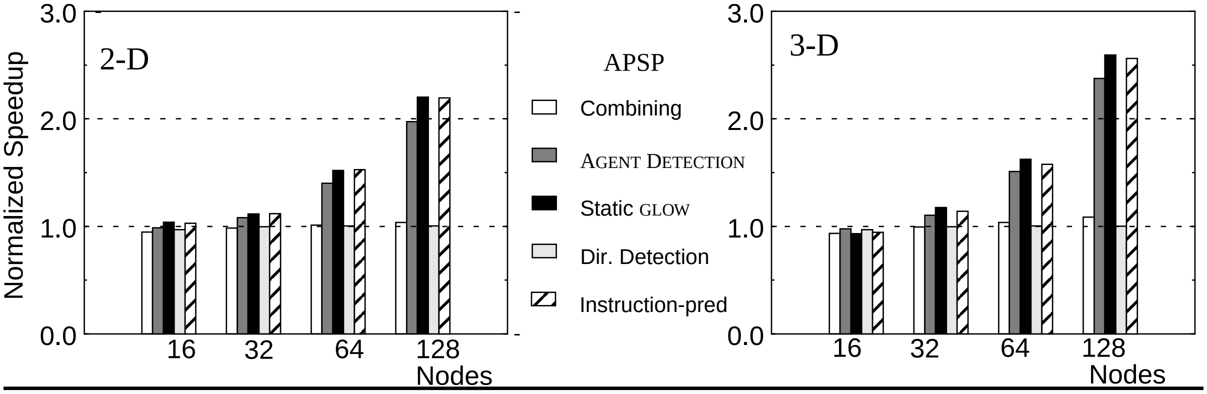


Figure 4.33. Normalized speedup (over SCI) for APSP for 2 and 3 dimensions (16 to 128 nodes).

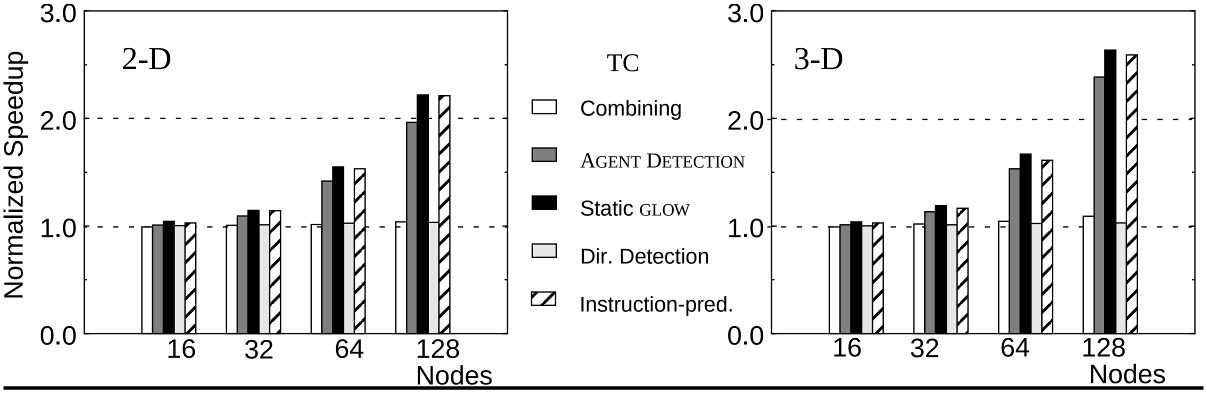


Figure 4.34. Normalized speedup (over SCI) for TC in 2 and 3 dimensions (16 to 128 nodes).

Since APSP and TC exhibit similar behavior, I only demonstrate read-run compression for APSP (Figure 4.35). A significant percentage of the reads in the program correspond to large read-runs. As expected, COMBINING is not successful in hiding accesses from the directories. Although it shifts accesses to smaller read-runs, it does not shift them far enough to make a difference in performance. AGENT DETECTION is more successful, compressing the read-runs

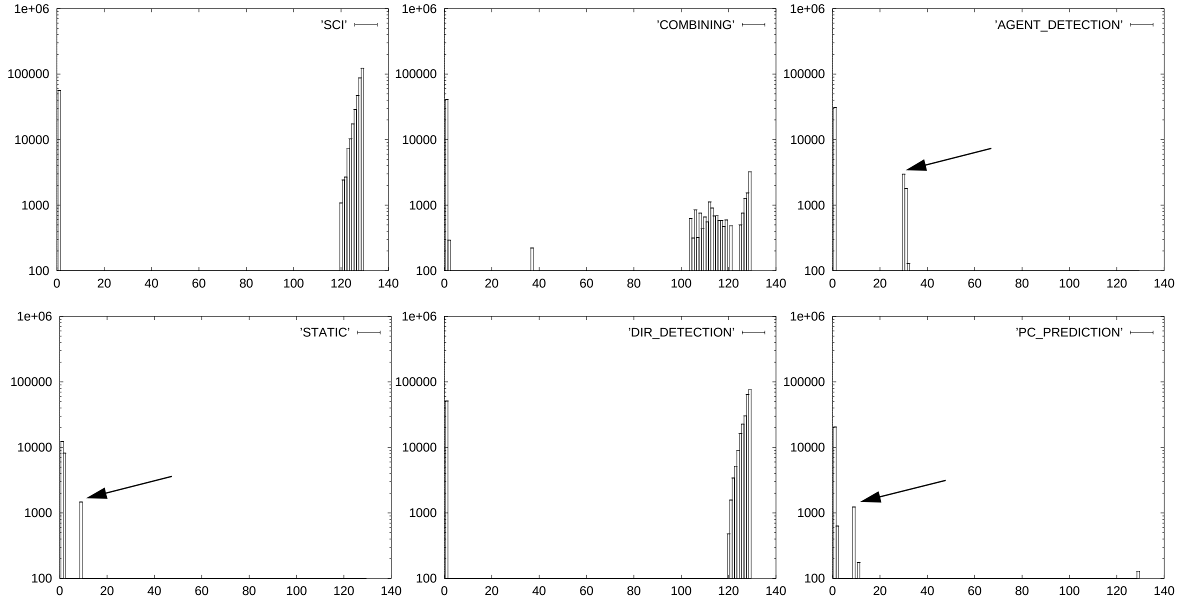


Figure 4.35. Read-run compression for APSP (128 nodes, 2 dimensions). Accesses corresponding to large read-runs are shifted toward smaller read-runs using GLOW extensions.

to a size of around 30 (this translates to about 22 requests slipping trough 8 GLOW agents while the rest are intercepted). Static GLOW works very well, leaving only read-runs of size 9 (similarly to the previous two programs). A common characteristic of the APSP and TC programs is that their data blocks are widely shared only once. Not surprisingly, DIRECTORY DETECTION fails to compress the read-runs of the program. The read-run histogram for INSTRUCTION-PREDICTION is almost exactly the same as static GLOW'S. This explains their almost identical performance.

***CG*—**The behavior of the CG program is somewhat peculiar, because of the very small dataset (64x64) used here (Figure 4.36). GLOW extensions exhibit the largest speedups over SCI in 16 nodes (1.9 and 2 times faster than SCI in 2 and 3 dimensions). DIRECTORY DETECTION performs very well and actually takes the performance lead from the static GLOW in 32 and 64 nodes in 2 dimensions. COMBINING does not provide any performance improvement while AGENT DETECTION is competitive (within 25% of the performance of the static GLOW). INSTRUCTION-PREDICTION performs similarly to static GLOW, surpassing it in some cases and lagging behind in others.

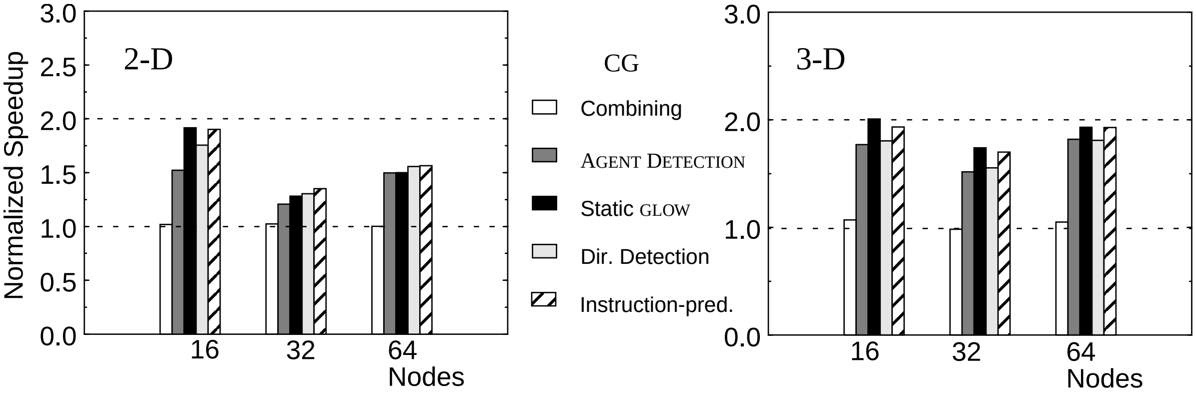


Figure 4.36. Normalized speedup (over SCI) for CG in 2 and 3 dimensions (16 to 64 nodes).

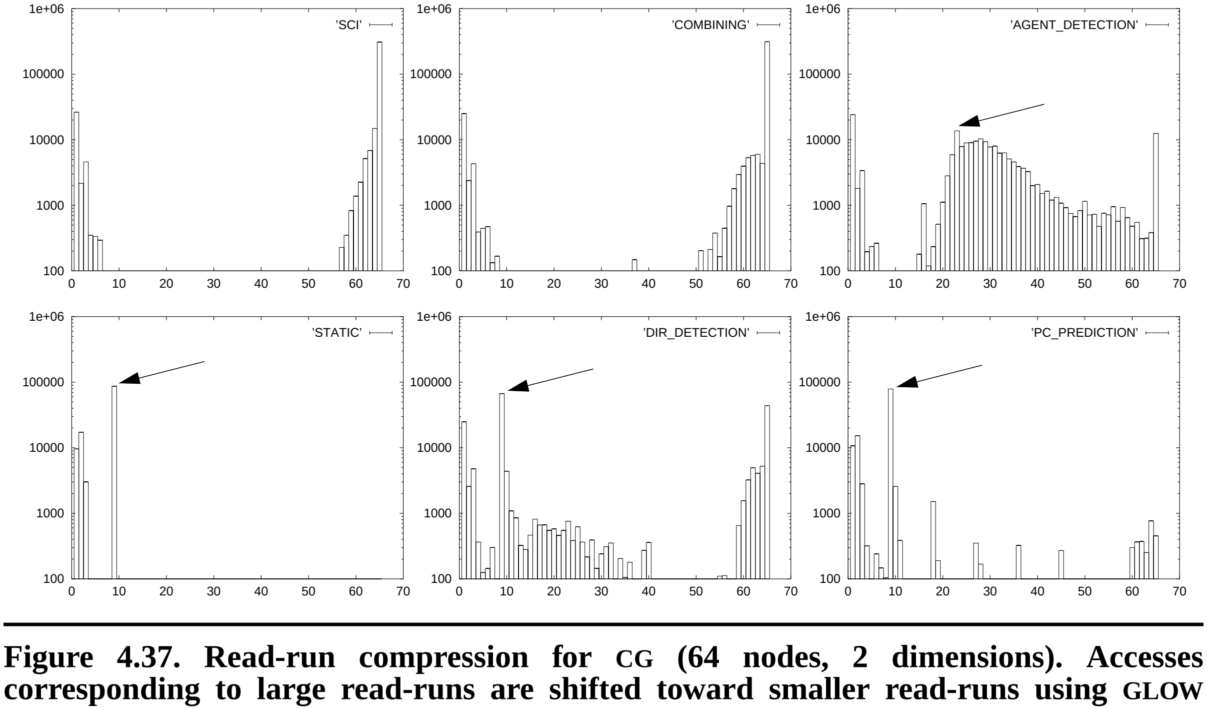


Figure 4.37. Read-run compression for CG (64 nodes, 2 dimensions). Accesses corresponding to large read-runs are shifted toward smaller read-runs using GLOW extensions.

Figure 4.37 shows the read-run compression results for CG: COMBINING does not affect accesses corresponding to the largest read-runs; AGENT DETECTION shifts many accesses to read-runs of size 23; static GLOW again eliminates large read-runs leaving only those of size 9; and DIRECTORY DETECTION is also very successful, shifting many accesses to read-runs of size 9. The read-run compression of INSTRUCTION-PREDICTION is second to only static GLOW's. The largest read-runs are mostly gone but there still is some "noise" in the middle of the spectrum.

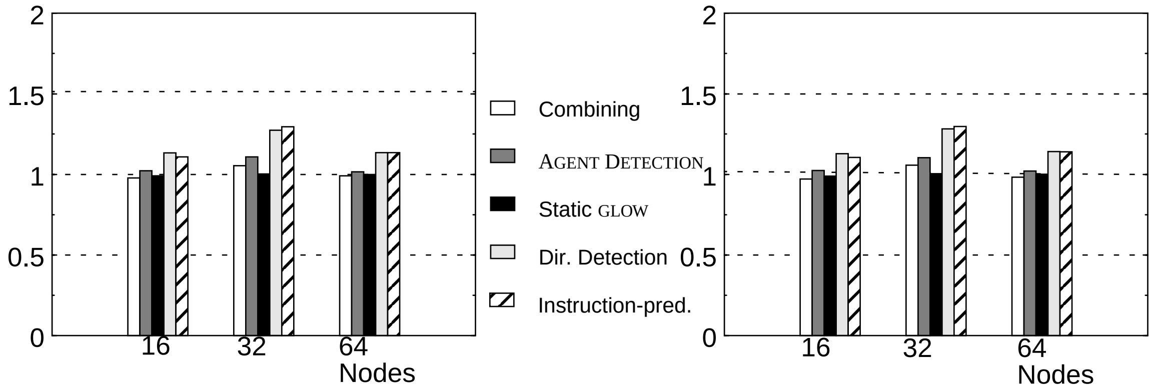


Figure 4.38. Normalized speedup (over SCI) for BARNES for 2 and 3 dimensions (16 to 128 nodes).

BARNES**—**BARNES is not affected much by the dimensionality of the network and does not speedup considerably with larger number of processors (Table 6). This is due to the very small dataset I was able to simulate with the WWT (4K particles). With larger datasets BARNES should exhibit better scaling. Nevertheless, the schemes I propose show speedups over SCI (Figure 4.38)—as much as 1.3 for 32 nodes. COMBINING and AGENT DETECTION, as well as static GLOW, do not show significant speedups over SCI. However, DIRECTORY DETECTION and PC-PREDICTION work very well, the former recognizing the top of the tree as widely shared and the latter identifying the instructions that access the top of the tree. The best speedups are

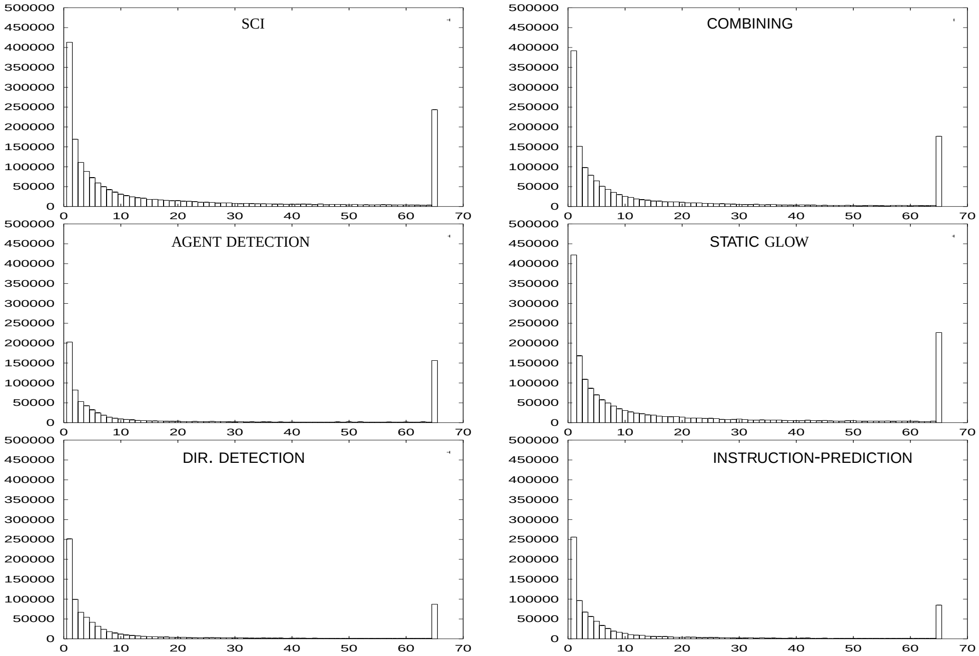


Figure 4.39. Read-run compression for BARNES (64 nodes, 2 dimensions). Y-axis (no. of accesses) *not* in logarithmic scale.

1.27 for DIRECTORY DETECTION and 1.3 for INSTRUCTION-PREDICTION, both for 32 nodes. Read-run analysis shows that all schemes exhibit different behavior for BARNES than for the other benchmarks (Figure 4.39). All schemes redistribute both large and small read-runs among the smaller read-runs, but without any particular peaks and in various degrees of success. DIRECTORY DETECTION and INSTRUCTION-PREDICTION do best at reducing large read-runs.

To summarize the results: AGENT DETECTION consistently tracks the performance of static GLOW, while COMBINING only performs well for one program (GAUSS). The results show that COMBINING is indeed sensitive to the congestion characteristics of the application. The behavior of COMBINING also changes depending on the network characteristics (*e.g.*, link and switch latency, bandwidth), while the behavior of AGENT DETECTION remains largely unaffected by the number of intercepted requests. DIRECTORY DETECTION gives mixed results, performing well only for three of the five programs. INSTRUCTION-PREDICTION proved to be not only the most successful dynamic scheme but also better than the static scheme in many cases.

4.6 Summary

I opened this chapter by discussing static (compile-time) approaches to define widely-shared data or instructions that access widely-shared data. These approaches require an interface to pass information from the program to the hardware and they make sense only in situations where such an interface can be provided. Because these approaches are reasonably good for most programs I consider them to be the base case for GLOW and use them to study various aspects of the behavior of the GLOW extensions. I studied: data caching in GLOW agents, effects on performance for 2- and 3-dimensional networks, effects of the input size on the scalability of programs (showing that, with larger datasets, program scalability with GLOW improves considerably faster than with SCI), read and write performance, effects of relaxing the memory model, and performance of an update protocol.

In this chapter I also proposed three schemes that can detect widely-shared data at run-time. The advantage of these schemes is that they apply the GLOW extensions *transparently* without need to change the software/hardware interface. I compare them against SCI, static GLOW and NYU Ultracomputer-style combining.

The first scheme, AGENT DETECTION, discovers widely-shared data more reliably than combining, by expanding the window of the observable requests. Switch nodes remember recent requests even if these have long before left the switch. Requests whose addresses have been seen in the window are intercepted (as requests for widely-shared data) and passed to the GLOW extensions for further processing. The interesting characteristic of this scheme is that in large systems even a small window performs very well. This scheme achieves a significant percentage of the performance improvement of static GLOW and has the potential to outperform the static version in programs where it is difficult for the user to define the widely-shared data. Since AGENT DETECTION requires modification only in the switch nodes, I believe it is the least intrusive of all the schemes. Finally, congestion-based combining (COMBINING), which is slightly simpler than AGENT DETECTION, is highly dependent on the congestion characteristics of the applications and the network.

In the second scheme, DIRECTORY DETECTION, the directories are modified to discover the widely-shared data by counting reads between writes. When a directory finds a data block to be widely shared it notifies the nodes in the system to subsequently request this data block as widely-shared. The applicability of this scheme is limited: it works well only when data blocks are widely accessed more than once.

The third scheme, INSTRUCTION-PREDICTION, is the most successful and is based on predicting which load instructions are going to access widely-shared data. Although its implementation is intrusive to the processor, it offers the best performance. The potential for further optimizations based on INSTRUCTION-PREDICTION increases its value.

Finally, in this chapter I used read-run analysis to gain insight on how these schemes affect accesses to widely-shared data. This tool enables visualization of these effects and allows us to reason about the behavior of the various schemes.

5 Migratory Sharing

In this chapter I address optimizations for migratory sharing. First, I present motivation for optimizing migratory sharing. Subsequently, I discuss previously proposed static and dynamic address-based methods to apply a migratory sharing optimization. Finally, I propose instruction-based methods to apply this optimization.

Migratory data, as defined by Weber and Gupta [100], are accessed by one processor at a time. Typically, these data are protected by locks and are accessed inside critical sections. In invalidation-based cache-coherence protocols the optimization is to “migrate” the data whenever a new processor accesses them by giving the processor both read and write permissions even if it first accesses the data with a read. To give the new processor exclusive access to the data the previous copy (in the processor that last accessed them) is invalidated. The optimization works well because it collapses two coherent transactions into one.

Figure 5.1 shows the optimization applied to a typical directory-based coherence protocol (similar to Dir_nNB) with a 4-message read protocol. On the left side, in the un-optimized case, a new writer wishes to modify the migratory data. The new writer experiences a load-miss and communicates with the directory to read the data. The directory then fetches the data from the last writer and gives it back to the new writer. In the mean time the last writer is left with a read-only (RO) copy of the data. Subsequently, the new writer experiences a store-write-fault when it first tries to write the data. A new set of transactions involving the directory is needed to invalidate the RO copy owned by the last writer and upgrade the RO copy owned by the

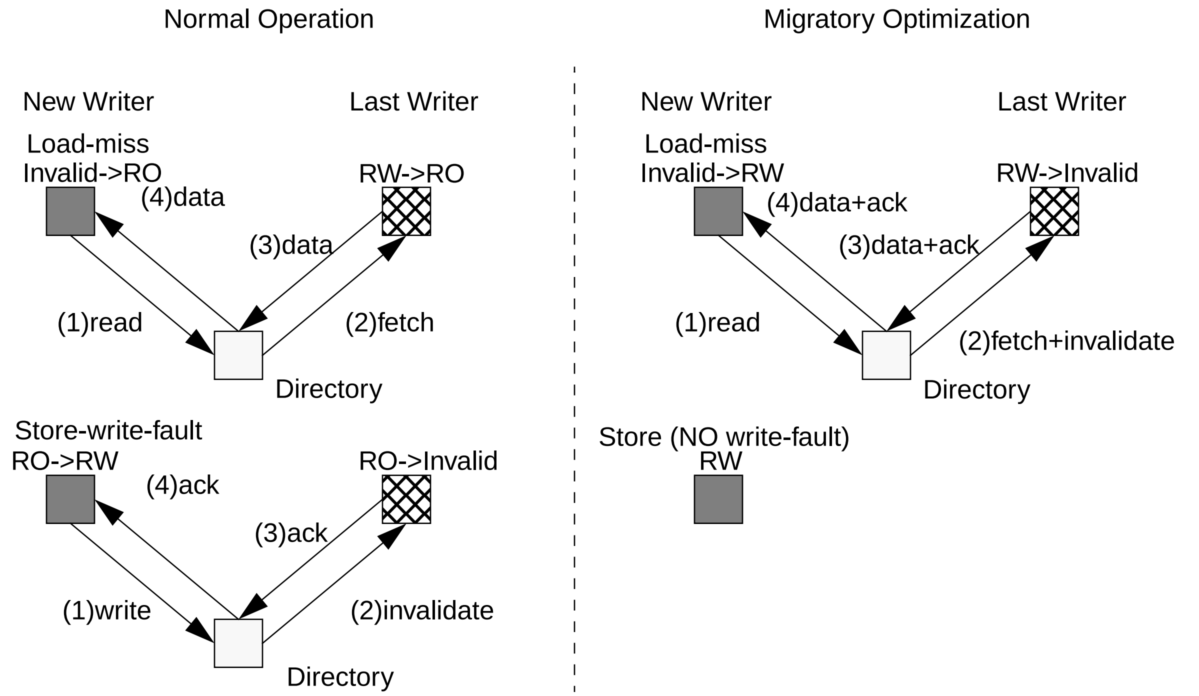


Figure 5.1. Migratory sharing optimization.

new writer to a read-write (RW) copy. In contrast, when we collapse the two transactions into one in the optimized case (at the right of Figure 5.1), the latency of the store-write-fault is completely eliminated and the transaction traffic is reduced by half (in DASH-like protocols which have 3-message read protocols the traffic is reduced by 60%). The collapse of the two transactions can be initiated either by the home node directory when it decides to return a RW cache block in response to a read request [28,93], or by the processor when, upon a read, it asks for a RW block knowing that it accesses migratory data (as in Munin [22]).

However, indiscriminate application of this optimization is harmful to performance (as I will show in Section 5.2.6). Thus, the problem is to detect migratory sharing or to distinguish the migratory data from other data and apply the optimization selectively.

As with the wide sharing optimization, the migratory optimization can be applied statically (*i.e.*, decided at compile time) or dynamically (*i.e.*, decided at run time). Additionally, migratory sharing can be optimized either by detecting the migratory data themselves (address-based) or by detecting instructions that access migratory data (instruction-based). Both static and dynamic address-based optimizations have been previously proposed [22,28,93]. In this thesis I introduce instruction-based optimizations. Table 13 shows the four combinations. Previously proposed optimizations are shown in the shaded cells.

MIGRATORY SHARING	**Static**	**Dynamic**
Address-based	MUNIN [22]	ADAPTIVE-MIGRATORY [28][93]
Instruction-based	CRITICAL SECTION GUIDED	INSTRUCTION-BASED PREDICTION (CRITICAL SECTION DETECTION, Appendix 3)

Table 13. Techniques to apply the migratory sharing optimization.

The static instruction-based scheme I propose is comparable to the static address-based scheme. In both, the user must determine either the addresses of the migratory data or the instructions that access migratory data. Implementation issues are similar: in both address-based and instruction-based schemes the information supplied by the program must be communicated to the hardware so the appropriate cache-coherence optimizations can be applied. The techniques that can be used in a hardware-based shared-memory environment are discussed previously for static address-based and static instruction-based wide sharing optimizations (Section 4.1.1 and Section 4.1.2, respectively). The new dynamic instruction-based scheme I propose does not involve the user and, although it requires additional hardware, it uses fewer resources that the dynamic address-based optimization previously proposed.

In the rest of this chapter I review the static and dynamic address-based optimizations and propose new instruction-based optimizations. Results show that the dynamic instruction-based techniques work comparably to their address-based counterparts, while using very few hardware resources.

5.1 Address-based optimizations for migratory sharing

Weber and Gupta’s paper on sharing patterns [100] prompted the design of address-based optimizations. Carter, Bennet and Zwaenepoel took into consideration migratory data in the design of the Munin Distributed (Virtual) Shared Memory system [22]. Munin introduced in the same system multiple coherence protocols, tailor-made for specific types of data. For example Munin supported different coherence protocols for read-only, migratory, write-shared, producer-consumer, reduction, result and conventional data objects [22]. However, the user (programmer) had to define statically the sharing pattern for every data object in the system, using source-code annotations. For the migratory data, they applied the standard optimization of returning—on first read—the data object to a new processor with read and write permissions, invalidating the copy of the previous processor. They do not report performance numbers for the migratory optimization.

Subsequently, two groups, Cox and Fowler [28], and Stenström, Brorsson, and Sandberg [93], independently proposed a dynamic address-based approach to discover migratory data at run-time and apply the aforementioned optimization. Although, the proposed techniques were for hardware based cc-NUMA, they can be applied equally well to software distributed shared memory [58]. In the adaptive protocols the directory discovers migratory data and returns RW cache blocks (instead of RO) whenever a new processor reads the data. The directory itself, upon detecting migratory sharing, collapses two coherent transactions (read and write) into one. The cost associated with these adaptive protocols is additional storage *per directory entry* to maintain the identity of the last writer [28,93]. Furthermore, the directory protocol requires more states to detect migratory sharing [28,93].

5.2 Instruction-based optimizations for migratory sharing

Previously proposed address-based optimizations were based on observing what happens to the data. In this thesis, I propose optimizations derived from observing instructions, *i.e.*, the behavior of the program itself. I will briefly discuss static and—in greater length—dynamic instruction-based optimizations for migratory sharing.

5.2.1 QOLB

Using the QOLB synchronization primitive [35] can be considered a static instruction-based scheme for migratory sharing. QOLB provides both synchronization for the critical section that protects the migratory data and automatic transfer of the data—*i.e.*, QOLB is a migratory optimization on its own. However, data must be *collocated* with the lock that protects them [48], for QOLB to provide the desired functionality in this context.

5.2.2 Static instruction-based optimizations

The goal in static instruction-based migratory-sharing optimization is to discover read-modify-write operations inside critical sections (protected by locks) and to annotate the loads so that the coherence protocol will treat them as writes. For many codes, this is straightforward. Consider the critical sections shown in Figure 5.2. These are typical critical sections found in the MP3D and OCEAN programs. The read-modify-write operation is denoted by the ++ operator on a shared variable. A simple annotation on this operation could generate a special first load that the coherence protocols translates to a coherent write. The static instruction-based method can be very successful because of its simplicity. Although compiler optimizations are beyond the scope of this thesis, I believe that this optimization for migratory data is easy to implement in a compiler.

This static instruction-based optimization suffers from the same implementation problems as the corresponding optimization for wide sharing (discussed in Section 4.1.2), namely how to pass information about requests from the program to the cache-coherence hardware. As with the static instruction-based wide-sharing optimization, I did not directly study manual annotation of the source program. Instead, in Appendix 3, I describe the dynamic counterpart of this method, which can also provide profiling information for the static instruction-based approach.

```
LOCK(Global->num_mol_lock);
      local_num_mol = Global->num_mol++;
UNLOCK(Global->num_mol_lock);


LOCK(Global->next_res_lock);
        local_next = Global->next_res++ % num_res;
UNLOCK(Global->next_res_lock);


LOCK(locks->psibilock);
   global->psiai = global->psiai + psiaipriv;
UNLOCK(locks->psibilock)
```

Figure 5.2. Typical critical sections in MP3D and OCEAN.

5.2.3 Dynamic instruction-based optimizations

As an alternative to the address-based or the static instruction-based optimizations I propose a dynamic instruction-based optimization that can handle migratory sharing patterns. The idea is to detect when a load-miss is followed by a store-write-fault on the same cache block. If such a load/store pair is recurring often we can predict, upon seeing the load-miss, that a write-fault is soon to follow. This mechanism can be classified as **cache block anti-depen-**

dence prediction since it detects write-after-read on the same cache block. This work is influenced by similar uniprocessor predictions [72]. Alternatively, we can detect entry to and exit from critical sections and apply optimizations only within those critical sections. This method is not researched further in this thesis and is sketched in Appendix 3.

Let us examine why this optimization is related to migratory sharing patterns. Migratory data are continuously read-modified-written, but each time by a different processor [100]. Each processor brings them into its cache as a RO cache block, tries to modify them, generates a write fault, converts them to a RW cache block, writes them, and subsequently loses them to another processor that will go through the same cycle. The connection to the instruction-based prediction is straightforward: migratory data are likely to generate load-misses closely followed by store-write-faults.

The optimization is to convert the coherent read to a coherent write ending up with a RW cache block and thus avoiding the write fault. The inspiration for the optimization comes from Carter et al. (Munin)[22] and the adaptive CC-protocols proposed independently by Cox and Fowler [28], and by Stenström, Brorsson, and Sandberg [93].

In the following section I discuss in more detail a prediction scheme that successfully detects anti-dependences on cache blocks. Subsequently, I evaluate this scheme and present the results for seven benchmarks.

5.2.4 Cache-block anti-dependence prediction

The idea of this scheme is simple: if we observe a load-miss/store-write-fault pattern a few times, then every time we encounter the load-miss we will request a RW cache block to prevent the write-fault. The predictor is a small fully-associative table indexed using the load PC. Each predictor entry contains the PC of the load, the address of the last cache block on which the load missed, a small n-bit saturating counter used to make predictions, and a prediction bit (P-bit) that indicates unconfirmed predictions and provides the means for adapting back. The size of each entry is as small as 9 bytes assuming 32-bit addressing and that we store the full addresses.

However, storing the full addresses is not necessary. A predictor of a certain size, for example, can be directly indexed with just a small number of PC bits. A field to store additional PC bits (similar to a tag in caches) is optional. The address of the cache block can also be truncated to a small number of bits. Although truncating or eliminating address fields allows the possibility for mistakes, *i.e.*, updating the wrong entry, using the wrong entry to make a prediction, and mistaking the address block for another, the trade-off is that we can implement a predictor with many more entries with the same transitor budget. Many more entries tend to decrease the rate of mistakes, while at the same time, considerably more information can be stored. In large programs, such as database codes, having many entries may be more important than accuracy for each individual entry. For convenience, in the rest of this thesis I use the fully-expanded version of the predictor entries with full address fields.

Only the load PC is used in the predictor entries and not the store PC. This means that unique load/store pairs are not tracked, but all pairs that have a common load PC are lumped together. A predictor entry, therefore, refers to a single load, but can be affected by multiple distinct stores. I examined alternative implementations where the predictor entry contains both the load PC and the store PC but I did not find enough evidence for their usefulness.

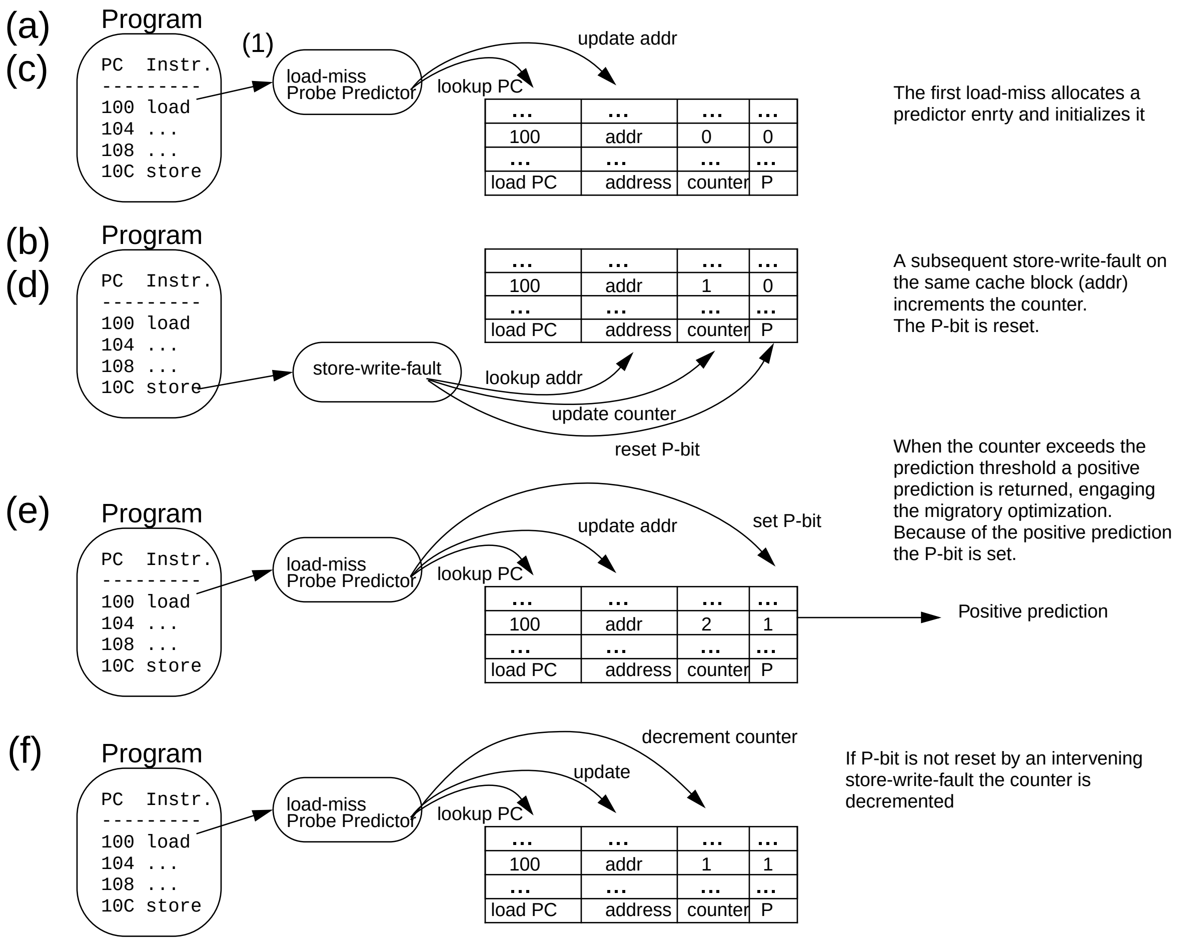

(a)
(c)
Program
PC Instr.
100 load
104 ...
108 ...
10C store
(1)
load-miss
Probe Predictor
lookup PC
update addr
100 addr 0 0
load PC address counter P
The first load-miss allocates a predictor enrty and initializes it
(b)
(d)
Program
store-write-fault
lookup addr
update counter
reset P-bit
100 addr 1 0
load PC address counter P
A subsequent store-write-fault on the same cache block (addr) increments the counter.
The P-bit is reset.
(e)
Program
load-miss
Probe Predictor
lookup PC
update addr
set P-bit
100 addr 2 1
load PC address counter P
Positive prediction
When the counter exceeds the prediction threshold a positive prediction is returned, engaging the migratory optimization.
Because of the positive prediction the P-bit is set.
(f)
Program
load-miss
Probe Predictor
lookup PC
update
decrement counter
100 addr 1 1
load PC address counter P
If P-bit is not reset by an intervening store-write-fault the counter is decremented

Figure 5.3. Cache block anti-dependence prediction mechanism.

A working example—Figure 5.3 shows how the predictor is updated by load-misses, stores and store-write-faults, and by external events. On a load-miss the predictor is probed and at first it is empty. A new entry is allocated, and includes the load PC and the address of the cache block (Figure 5.3a). The counter is initialized to zero and the prediction bit (P-bit) is reset since no positive prediction has been made yet (Figure 5.3a). When a store generates a write-fault the predictor is searched (associatively) using the address of the cache block, to find the corresponding load (Figure 5.3b). Note that the store PC is not needed. A store-write-fault increments the counter of the corresponding entry (Figure 5.3b). This load-miss/store-write-fault scenario repeats until the counter exceeds a prediction threshold (Figures 5.3c and 5.3d). The next time a load-miss occurs (Figure 5.3e) a positive prediction is made and the coherent read is converted to a coherent write.

***Adapting back*—**The prediction bit (P-bit) provides the method for adapting back. Its purpose is to confirm the prediction that the cache block will be written by an ensuing store. When a positive prediction has been made the P-bit is set. To confirm the prediction, the P-bit must be reset when a store writes the block. The next time the same load probes the predictor it will find the P-bit either set or reset. If it is set the cache block was not written. In this case (shown in Figure 5.3f), the prediction counter is decremented and the P-bit is reset (if, however, the counter did not fall below the threshold a new positive prediction will set the P-bit again). On the other hand, if a load finds the P-bit reset a previous positive prediction has been confirmed.

Special care is needed to enable a store to reset the P-bit. Because of the positive prediction, no write-fault will occur to trigger a predictor update. Various solutions exist:

- We can update the predictor with any dynamic store instance (regardless of whether it generates a write-fault or not). The disadvantage of this method is increased pressure on the predictor.
- We can trap stores that write cache blocks brought in with positive predictions. This can be accomplished by bringing in the cache block to an intermediate state which enjoys write privileges but denotes unmodified data. Such a state exists in many protocols and commonly referred to as “Exclusive.” Thus, the transition from “Exclusive” to “Modified” (which can be considered as a soft write-fault) triggers a predictor update and resets the P-bit. This is the scheme we implemented for the evaluation.

***External events*—**A simple-minded implementation of this instruction-based prediction scheme can be fooled by non-migratory-sharing patterns (*e.g.*, when more than one node read the data block before it is written). To avoid such complications, I only consider read-modify-write operations on cache blocks that are the only copies in the system (*i.e.,* exclusive) and are not affected in other way throughout the operation. If the cache block is read by another node between the time of the load-miss and the time of the store-write-fault we disable the update of the predictor. This is accomplished by deleting the cache block address field of the corresponding predictor entry. Thus, the correspondence of a load-miss and a store-write-fault cannot be established through the address of a cache block that was externally read. Cache block replacements and invalidations also have a similar effect because they lead to subsequent (load or store) misses—not store-write-faults.

***Prediction threshold*—**The prediction threshold provides hysteresis in adapting to migratory sharing and back. A low threshold allows the optimization to be applied soon after the load-miss store-write-fault behavior is detected, but it delays adapting back in the face of unconfirmed predictions when the saturating counter has reached its highest value. The opposite behavior is obtained by using a high threshold. In this work we used a prediction threshold of 1 with a 2-bit saturating counter.

***Implementation details*—**In the above description each predictor entry has a cache-block address field (the meeting point between a load-miss and a subsequent store-write-faults).

Depending on where we put this meeting point, we can change the way the predictor operates, specifically the way new entries are allocated. I have identified three options for the meeting point that perform comparably but differ in implementation cost:

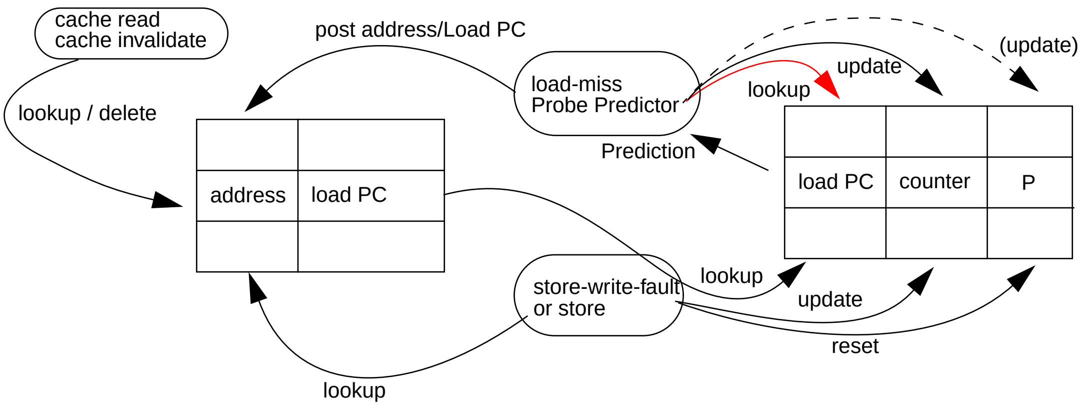


Figure 5.4. Alternative implementation for the prediction mechanism: Correspondence of a load and a store is established via an external structure.

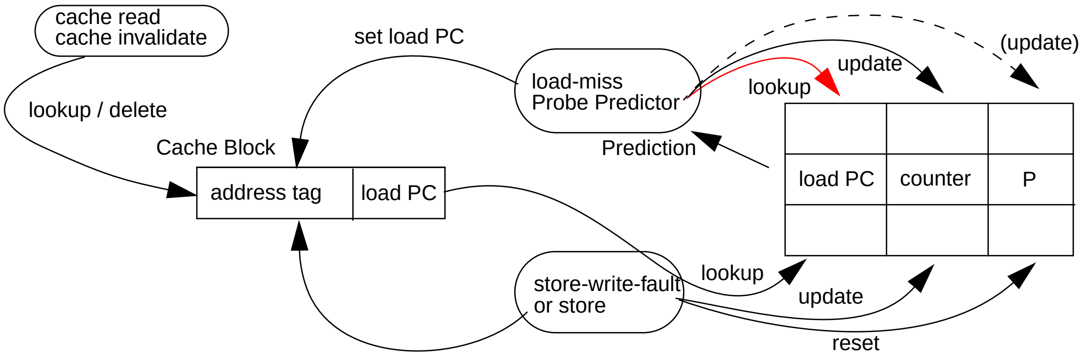


Figure 5.5. Alternative implementation for the prediction mechanism: Correspondence of a load and a store is established via the cache block.

1. The predictor entry contains the address of the last block loaded (Figure 5.4). On a store-write-fault an associative search is performed on the block addresses to find the corresponding load. The problem with this method is that more than one load may have missed (at different times) on the same cache block whose address then exists in multiple predictor entries. The solution is to allow only one instance of an address in the predictor and specifically only for the load that missed last.

2. Alternatively, a small *detection* structure (*e.g.*, fully-associative cache) can be used to keep

track of the correspondence of addresses to load PCs (or addresses to predictor entries)—shown in Figure 5.4. This structure is updated on a load-miss with the cache-block address and the load PC and is probed on a store-write-fault. When a probe hits, the corresponding entry is deleted. This structure need not be large since the information needs to be kept for short periods of time (from the time of the load-miss to the time of the store-write-fault). If this information is prematurely lost, the store-write-fault will be unable to update the predictor, but this is not fatal.

3. Finally, the correspondence of loads and stores (*detection*) can be established via the cache block itself (Figure 5.5). When a load misses it tags the newly brought-in cache block with its PC (or, alternatively, with a pointer to the predictor entry). When a store write-faults on a cache block it uses the PC tag of the cache block to find the corresponding load in the predictor. External events on the cache block delete the PC tag. Although this is an elegant method, it requires storage for the PC tag for every cache block and it is not as cost-effective as the previous methods.

With modern ILP processors, multiple instances of the same instruction can be simultaneously in the processor’s instruction window. Thus, there is the possibility that a predictor can be accessed multiple times by different instances of the same load before the corresponding stores come into play. In such cases, a split detection-predictor mechanism (cases 2 and 3 above) is required so that a single predictor entry represents the unique static load instruction, but multiple detection entries deal with multiple instances of the instruction. Split detection-prediction mechanisms were introduced for dependence prediction by Moshovos et al. [72].

In addition, because in ILP processors multiple instances of the same load can access the detection-prediction structure before any of the corresponding stores appear, care must be taken for the update of the P-bit. When corresponding stores are delayed with respect to multiple instances of the same load, it can appear as if wrong predictions are made. Thus, we must always update the P-bit in *program order.* This means that P-bit updates take place when load instructions are retired. However, this introduces a delay in adapting back: because updating the P-bit trails behind accessing the predictor, additional false predictions can be issued before the saturating counter is decremented below the threshold.

5.2.5 Migratory sharing optimization on SCI

The ideas in this chapter are independent of the cache-coherence protocol used. Throughout this thesis I use SCI as the platform to examine various optimizations to sharing patterns. Although, in the introduction of this chapter I have illustrated the migratory sharing optimization on full map directory protocols (such as $\mathrm{Dir_nNB}$ or DASH), here I describe how this optimization applies to SCI.

In SCI, upon a miss a node communicates with the home-node directory and attaches itself in front of the sharing list as the new head node. In this position, the node is the only one that can write the cache block. Thus, it can initiate an invalidation of the sharing list without communicating with the home-node directory. In the case of migratory data, there is only one other node in the sharing list—the previous writer. Because of the unique position of the head node, SCI requires only two messages for the invalidation, in contrast to other protocols which require four messages.

In the migratory sharing optimization we collapse the two transactions into one. Although

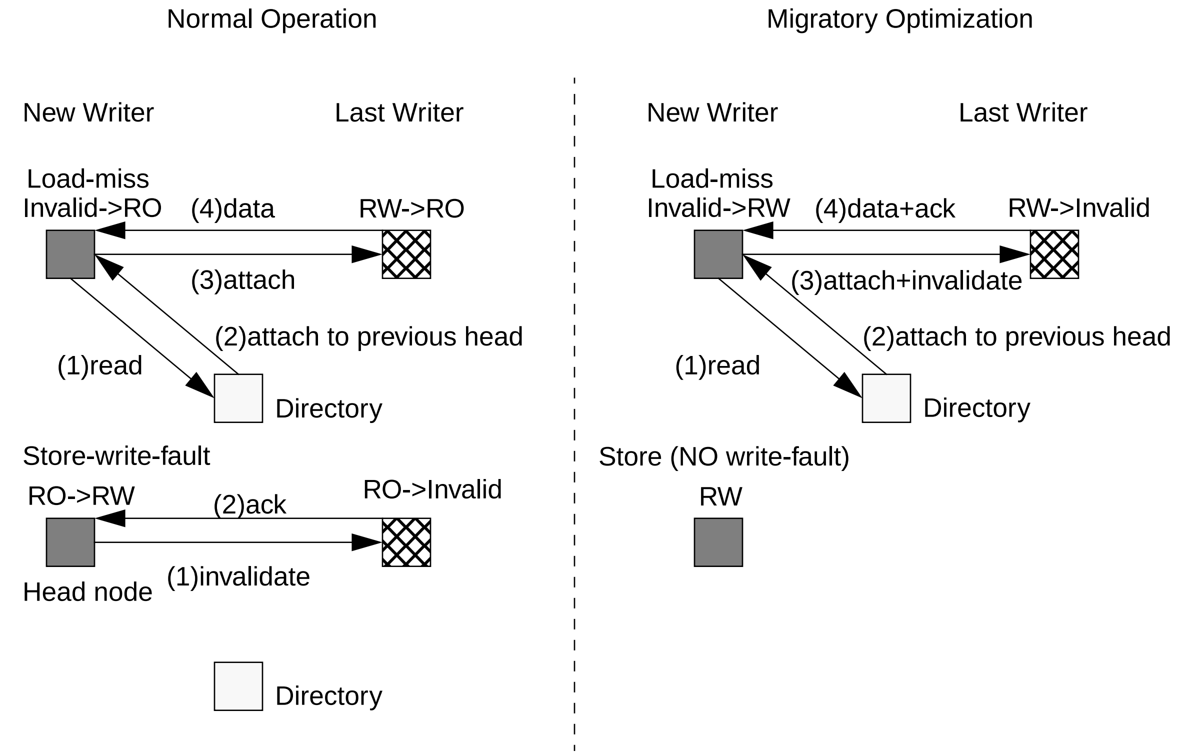


Figure 5.6. Migratory sharing optimization on SCI.

converting the coherent read to a coherent write eliminates the write latency, in SCI it actually increases the read latency and does not reduce traffic. The problem is that in SCI a coherent write requires six messages in three non-overlapping transactions (Figure 5.6): (i) the node communicates with the directory, (ii) attaches itself to the sharing lists, and (iii) invalidates the previous writer. I examined instruction-based prediction for migratory sharing using SCI unmodified, and I found that even in this case there is performance improvement for programs that have migratory sharing [54]. In this thesis, I modified SCI to support a combined attach and invalidate operation (Figure 5.6). Although, with this modification the read latencies remain unchanged, the SCI migratory optimization achieves at most a 30% reduction in traffic while in other protocols the traffic reduction is 50% or 60%.

5.2.6 Results

I studied the instruction-based prediction optimizations on CHOLESKY, MP3D and PTHOR, which exhibit migratory sharing, and on four control benchmarks (GAUSS, APSP, BARNES and OCEAN). Table 14 shows the speedups obtained over SCI with the instruction-based and address-based schemes. In addition, I show results applying the migratory optimization for all data (denoted by "Always" in Table 14). Using the migratory optimization indiscriminately (instead of selectively as the other schemes do) ranges from positive (CHOLESKY), to harmless (MP3D), to disastrous (all other programs).

Benchmark	SC				Relaxed model r1[47]	
	SCI	MIG Instruction-based	MIG. Addres-based	Always	SCI	MIG-Instruction
CHOLESKY	1.00	1.13	1.12	1.08	1.01	1.14
MP3D		1.16 (1.30 with no adapt-back)	1.21	1.02	1.22	1.29
PTHOR		1.02	1.00	0.69	1.07	1.10
GAUSS		1.00	1.00	0.42	1.01	1.01
APSP		1.03	1.00	0.45	1.03	1.04
BARNES		0.99	1.00	0.73	1.06	1.05
OCEAN		1.02	1.00	0.83	1.05	1.06

Table 14. Simulation results for migratory sharing optimizations (32 nodes, speedup over SCI).

Results show that instruction-based prediction works better than the address-based scheme for CHOLESKY (speedup of 1.13 vs. 1.12). For MP3D, the instruction-based method lags behind the address-based method (speedup of 1.16 vs. 1.21) but I discovered that disabling the adapt-back mechanism increases the speedup to 1.30. In MP3D some migratory accesses (to the particle array) are *not* protected by locks. Thus, although they are migratory "in spirit" in reality, the adapt-back mechanism rules out quite a few. For the rest of the programs which do not have migratory sharing (PTHOR, GAUSS, APSP, BARNES, and OCEAN), instruction-based prediction provides small performance improvements.

Under a relaxed memory model that hides write latency, instruction-based prediction still offers performance improvements, albeit smaller than the sequentially-consistent memory model case. I have used the relaxed memory model for SCI, called **r1**, described by Kägi et al. [47]. These results are consistent to the results reported by Stenström, Brorsson and Sandberg. With a relaxed memory model the argument for the migratory optimization (regardless of how it is applied) becomes primarily an argument of traffic: the migratory optimization reduces coherent traffic in proportion to the amount of migratory sharing in the program [93].

The results are comparable to those reported previously for address-based prediction [28,93], given the differences in the simulated systems and in particular the cache-coherence protocols, the number of nodes and the larger block size. Cox and Fowler reported that the block size has significant effects on the performance of their adaptive protocol for migratory data: increasing block size leads to smaller performance improvements. They reported speedups of 1.23 for CHOLESKY and 1.11 for MP3D in 16 nodes and with a block size of 16 bytes. Similarly, Stenström, Brorsson and Sandberg report good speedups (1.54 for MP3D and 1.25 for CHOLESKY), again in 16 nodes and for a small block size (16 bytes). WWT does not allow a block size smaller than 32 bytes, but I briefly examined larger blocks (64 bytes) and observed similar effects for the instruction-based prediction scheme: large block sizes (64 bytes) reduce the performance benefit. The effects of the block size on prediction mechanisms merit further investigation (beyond the capacity of this paper). The instruction-based prediction mechanism could be protected from the adverse effects of large block sizes by using the exact word addresses of the load and store instructions rather than the coarse-grain cache block address.

The most striking results, however, are presented in Table 15 (for the migratory sharing benchmarks). The number of predictor entries allocated is very low. On average, 19 predictor entries are needed for CHOLESKY, 24 for MP3D and 53 for PTHOR. Note however, that these are very small scientific programs and other workloads such as databases may have considerably higher requirements. In general, the number of predictor entries depends on the number of critical sections or the number or read-modify-write code instances rather than the actual size of the code. In comparison, the adaptive protocols for migratory data (*i.e.*, address-based prediction) require storage in proportion to the size of the directories. The maximum number of predictor entries was allocated in node 0 (which also executes initialization code) for all three benchmarks. Table 15 also contains statistics about the behavior of the predictors. Even though the predictors are not tuned individually for each benchmark, they do provide performance improvements, even for small rates of positive predictions.

Statistics		**CHOLESKY**	**MP3D**	**PTHOR**
Static loads considered	all 32 nodes	1844	2264	6309
	average per node	58	71	197
Active predictor entries (loads followed by store-write-faults)	all 32 nodes	597	751	1687
	average per node	19	24	53
	maximum	46	34	71
Total predictor probes		283473	569012	984330
Hits in the predictor		159018	553510	598100
Hits as % of total probes		56%	97%	61%
Positive Predictions	Number	143337	339866	221548
	% of predictor hits	90%	61%	37%
	% of predictor probes	50%	60%	23%

Table 15. Statistics for instruction-based prediction.

5.3 Summary

In this section I discussed optimizations for migratory sharing. I presented previous work on static and dynamic, address-based schemes to distinguish migratory data. A contribution of this thesis is a novel instruction-based prediction to optimize migratory sharing. The idea is to predict whether a load-miss will be followed by a store-write-fault. This prediction/optimization works well for three benchmarks (CHOLESKY, MP3D, and PTHOR) that exhibit migratory sharing and is competitive with previously proposed address-based adaptive protocols. When equipped with safeguards to avoid applying the optimization to non-migratory sharing it shows no negative performance impact on four control benchmarks (gauss, apsp, barnes, ocean). For the programs used (admittedly small programs) less than 100 predictor entries were used in any node's prediction table.

6 Producer-Consumer Sharing

In this chapter I discuss optimizations for producer-consumer sharing and techniques to selectively apply these optimizations. I propose a novel dynamic approach to apply producer-consumer optimizations using instruction-based prediction and a novel optimization for producer-consumer sharing that uses speculative execution. Because of the limitations of the simulation environment, I present only a preliminary investigation of these schemes. Further research is needed to assess their performance.

Conceptually, most sharing in a program is producer-consumer sharing: a node creates a value that is used by another node. Such a wide definition of producer-consumer sharing makes it difficult to reason about it and design effective optimizations. Therefore, of interest are *stable* producer-consumer relationships where the producer and consumer(s) remain the same over time. In such cases, the common thread of the optimizations is to try to get newly created values to the consumer(s) as soon as possible so that the consumer(s) are saved from communicating requests for the new value.

PRODUCER-CONSUMER SHARING	**Static**	**Dynamic**
Address	MUNIN [22] UPDATE PROTOCOLS	COMPETITIVE UPDATE
Instruction	DATA FORWARDING [59] FINE-GRAIN PRODUCER-INITIATED COMMUNICATION[3]	IINSTRUCTION-BASED PREDICTION SPECULATIVE PRE-SEND [54]

Table 16. Producer-consumer sharing optimizations

Analogous to wide-sharing and migratory-sharing optimizations, the producer-consumer optimization should be applied selectively and not indiscriminately, otherwise performance loss is likely. The methods to apply a potential optimization are divided into static and dynamic, and address-based and instruction-based. Table 16 shows the four possible combinations. Three out of the four cases (shaded cells in Table 16) have been explored in previous work. The contribution of this thesis is to propose a transparent dynamic instruction-based technique to apply producer-consumer optimizations, as well as a new optimization based on speculative execution.

6.1 Related work

In this section I present previous work on optimizing producer-consumer sharing.

6.1.1 Address-based optimizations

The static address-based optimization was proposed for the Munin software DSM system [22]. Data objects involved in stable producer-consumer relationships were defined in the source code and a specialized protocol (producer-consumer optimizations) was used for such data. Again, the disadvantages of this approach lie in its need for an interface to transfer information from the program to the hardware and the need to involve the user to identify producer-consumer data.

Competitive update protocols can be considered dynamic address-based optimizations for producer-consumer sharing.

6.1.2 Instruction-based optimizations

A different approach to optimize producer-consumer sharing is called *data forwarding* and it appears in DASH in the form of the *deliver* instruction [65], and has been studied by Koufaty, Chen, Poulsen and Torellas [59] and in different form by Abel-Shafi, Hall, S.V. Adve, and V.S. Adve [3]. In data forwarding, store instructions which generate values needed in other consumer nodes are identified in the source program. If the set of consumer nodes can be accurately identified either by the programmer and/or the compiler, then a special form of the store instruction is used both to update the producer's cache and to send the data to the consumer(s). The consumer(s) can use the data sent in this way because they are by definition valid —

because the program says so. Thus, the burden of correctness lies with the programmer or compiler.

Data forwarding or producer-initiated communication requires explicit identification of producer-consumer relationships. This, however, is also a characteristic of the message-passing programming paradigm, which makes it much less appealing than shared-memory. The shared-memory programming paradigm abstracts and hides producer-consumer relationships by using a shared name space. In effect, the use of global addresses is a way to hide explicit communication. In this respect, user involvement in data forwarding reduces the appeal of shared memory. On the other hand, since compiler identification of producer-consumer relations ships is not trivial (because conservative assumptions must be made for correctness), this may restrict both the application and the potential benefit of an otherwise transparent scheme [59].

QOLB can also be considered as an static instruction-based optimization for producer-consumer sharing, if it is used to provide both the synchronization among producer and consumer(s) and the data transfer. Here also, the optimization works when data are collocated with the lock that protects them [48].

6.2 Dynamic instruction-based optimizations

As an alternative to previous methods to apply producer-consumer optimizations, I propose dynamic instruction-based prediction to optimize producer-consumer sharing. The advantage of instruction-based prediction compared to static instruction-based methods is that it is completely *transparent* (it does not require an interface or involvement of the user/compiler).

In instruction-based prediction, a store instruction that generates misses or write-faults is a (potential) producer. Its consumer(s) are tracked using information from the CC-protocol. The prediction can take one of the following two forms: i) a binary prediction for the existence of stable producer-consumer sharing and, ii) prediction of the identity of potential consumers. Using the first form, we can invoke a pairwise sharing optimization or switch to an update protocol. Using the second form we can switch to an update protocol or *pre-send* data *speculatively* to consumers.

An update protocol would not constitute a *transparent* optimization in the case of a sequentially-consistent memory system because such protocols can violate sequential consistency and therefore need support from the programmer/compiler to guarantee correctness. Because of this reason and because SCI does not yet support an update protocol, I did not study this optimization. However, I believe that it is an interesting future direction for instruction-based prediction.

6.2.1 Pairwise-sharing prediction

This scheme predicts whether there is a stable producer-consumer relationship with a unique consumer. A simple predictor tracks for each store its last known consumer (see also "In search of the consumers" in the next section). The predictor is similar to those proposed in the previous chapter for migratory sharing but with the addition of an extra field per predictor entry, to store the identity of the last consumer (a total of about 13 bytes per entry). A 2-bit saturating counter is used to indicate whether the last consumer remains the same. When the predictor entry is updated with a different consumer the counter is decremented; otherwise it is incremented. The identity of the consumer can be changed while the counter remains below the threshold.

If a store (producer) always has the same consumer we can optimize their communication. SCI provides such an optimization called *pairwise sharing* [41,47]. Pairwise sharing allows the head and the tail nodes of a two-node sharing list to communicate without going to the home-node directory. I applied this prediction/optimization scheme on OCEAN and achieved a speedup of 1.07 (1.10 with 256KB caches) using on average 77 predictor entries per node. The pairwise-sharing optimization is very well implemented in SCI and even if it is heavily misused it does not affect performance much. Thus, turning this optimization on for all data leads to comparable results (1 or 2 percentage points lower than the instruction-based prediction optimization).

6.2.2 Producer-consumer prediction with speculative-execution optimization

The most advanced prediction/optimization scheme I propose predicts the identity of the consumer(s) and uses speculative execution to optimize producer-consumer sharing. This scheme is prompted by previous work, including: Moshovos and Sohi's [73] and Tyson and Austin's

[97] work on optimizing producer-consumer communication in uniprocessors, Koufaty, Chen, Poulsen and Torellas' work on data forwarding [59], Abel-Shafi, Hall, S.V. Adve, and V.S. Adve's evaluation of producer-initiated communication [3], and was influenced by Hill's views on speculative execution in shared-memory [38]. Evaluation of this scheme presents considerable difficulties because WWT does not support speculative execution. Thus, I am unable to provide execution-time measurements. Instead—analogous to studying branch prediction—I study this scheme by presenting prediction accuracies and hit rates for the speculative pre-sends. Finally, I discuss possible implementations of this scheme, including how to read speculative data external to the processor.

6.2.3 In search of the consumers

Before I describe the prediction scheme, I will explain how to identify possible consumers. The following discussion is dependent on the idiosyncrasies of SCI—in other directory-based CC-protocols the directory itself is an excellent source of information about consumers. In SCI, a node that wishes to write a cache block is responsible for invalidating the sharing list. Thus any nodes that are invalidated by the producer are considered consumer nodes. Additionally, any node that at a later point attaches in front of the producer (*i.e.*, reads the producer's cache block) is considered to be a consumer. Since the attach operation is unrelated to any particular store instruction the correlation of external events and instructions is established through an address filed in the prediction mechanisms (similar to that discussed for the migratory sharing prediction mechanisms).

6.2.4 Prediction

I use a predictor structure similar to the pairwise predictor, but in each prediction entry, instead of the field that holds the identity of a single consumer, I use a bit-map to track multiple consumers (a total of about 13 bytes per entry). Again as with the migratory sharing predictors (Section 5.2.4), it is not necessary to have all the address fields or the full address fields in an implementation. Address fields can be truncated or eliminated. For example, the predictor can be directly indexed with a few PC bits. Storing additional PC bits in the predictor entry is optional. For a fixed transistor budget, this would give many more predictor entries but also introduce the possibility of mistakenly accessing them or updating them. A large number of predictor entries may be preferable for realistic—large—workloads. For the small programs used in this thesis, I studied the fully expanded—and accurate—version of the predictor entries.

I examine two simple predictor schemes and propose a third: (i) LAST-PREDICTION that predicts the last set of consumers to be the new set, (ii) INTERSECTION-PREDICTION that predicts the intersection of the last two sets of consumers to be the new set, and (iii) two-level adaptive prediction inspired by the analogous scheme in branch prediction [105] and by Mukherjee and Hill's work on protocol prediction [75]. The three schemes work as follows:

1. **LAST-PREDICTION**: The predictor is both updated and probed on a store-miss or a store write fault. The predictor is updated when the producer node invalidates a sharing list. The update collects the identities of the invalidated nodes on a temporary bit-map and compares it to the bit-map stored in the predictor entry. If there is significant overlap between the bit-maps the entry's 2-bit saturating prediction counter is incremented; otherwise it is decre-

mented. The temporary—new—bit-map is then installed in the predictor entry. The predictor is then probed and if the counter exceeds a threshold, the bit-map containing the possible consumers is returned. This predictor has two tuning parameters: the counter threshold and a parameter that defines what is "significant overlap" between bit-maps. In this work the threshold is 1 and the overlap parameter requires at least 2 common consumers (in bit-maps that have 2 or more consumers).

2. **INTERSECTION-PREDICTION**: The predictor is updated when the producer invalidates a sharing list and the identities of the consumers are collected on a temporary bit-map. The logical `AND` of the temporary bit-map and the predictor entry bit-map (that contains the consumers of the previous store-miss or store-write-fault) constitutes the prediction bit-map. After the prediction bit-map is calculated, the temporary bit-map is installed over the predictor entry's bit-map.

3. **Two-level adaptive prediction:** This scheme is based on the two-level adaptive prediction techniques proposed by Yeh and Patt [105]. The predictor is updated when the producer invalidates a sharing list. The bit-map (or a hash thereof) of the invalidated nodes is used as a symbol for a *History Register (HR)* [105]. The HR is an index to a *Pattern History Table (PTH)* [105]. Each PTH entry contains a prediction bit-map and a saturating counter to control updates to the prediction bit-map. The prediction bit-map can use the LAST-PREDICTION scheme as described above, or the INTERSECTION-PREDICTION scheme, or any other suitable scheme. I have not examined the two-level adaptive scheme further. I believe that because of its cost the two-level adaptive scheme warrants further investigation only when the performance benefit of the Instruction-based prediction for producer-consumer sharing has been established. However, the simulation tool used in this thesis (WWT) does not pro-

vide the necessary capabilities for such a determination.

6.2.5 A novel optimization: speculative *pre-send*

After predicting the identity of the consumers we can send them the data on the condition that they use them speculatively, until they verify the data's correctness through the coherence protocol. I call this *speculative pre-send*[8]. The hope is that the data will arrive at the consumer(s) before they even ask for them. Speculative pre-send is not an update because: (i) it is *outside the coherence domain*, (ii) the set of the predicted consumers is not cumulative (as in the update) but it changes dynamically, and (iii) it allows feed-back through the coherence protocol. Since everything has to be verified through the CC-protocol, speculative pre-sends affect only performance but not correctness.

There are two questions concerning pre-sends: what to send and when to send.7 Regarding the first question we must decide whether to send just the new value written by the store or the whole cache block, while for the second question we must decide whether to send it immediately (at the end of the write fault) or wait until a later time. In this work pre-sends are accumulated in a special buffer (similar in functionality to a write cache used to enhance update protocols) which is flushed on synchronization operations (*i.e.*, barriers and unlocks) and the whole cache block (if it is available at the time of the actual send)[9] is forwarded to the consumer(s). In contrast to a write cache for an update protocol, the pre-send buffer does affect the choice of a memory model because of the speculative nature of the technique. On the con-

[8] Inspired by "*pre-fetch*."

[9] Since this is the initial exploratory work in this area, the full design space needs to be studied in future work.

sumer side, pre-sends are accumulated in the cache by taking advantage of invalid cache blocks. A speculative pre-send is only accepted if an invalid cache block with the same address exists in the consumer's cache. The reasoning for this restriction is that a correct pre-send is likely to encounter a corresponding invalid cache block since the producer previously invalidated all the consumers. No additional storage in the consumer nodes is needed for the pre-sends.

In reality this optimization trades bandwidth for latency. Speculative pre-sends consume additional bandwidth in hope that they will reduce the apparent read latency. I have not been able to investigate this trade-off with WWT since speculative execution is not supported. Results (Section 6.2.7) indicate that the number of speculative pre-sends is low (and therefore their bandwidth requirements are low). If the reduction of the critical path for the parallel computation is significant then this technique could provide considerable performance improvements.

6.2.6 How can a processor read external speculative data?

To make a convincing argument for the feasibility of the speculative schemes I propose, I sketch a method for a processor to read speculative data from outside. This proposal is compatible (at a high level) with existing memory speculation mechanisms in advanced processor designs.

In modern microprocessors that support speculative execution, loads can speculatively bypass stores that issued earlier and whose target address is unknown. If at a later time the address of the store is resolved and there is no dependence to the speculative load then the latter is committed; otherwise, if there is a dependence, the speculative load is "squashed" along with all speculative instructions that followed (or in the case of *selective invalidation* along with all speculative instructions dependent on the speculative load).

To read speculative data from the outside world, the processor creates a hypothetical *shadow store* whose address is unknown. The purpose of this shadow store (which never really executes) is to control the fate of the load that reads the external data. This load is executed speculatively, pending confirmation of absence of dependence on the shadow store. After the load reads the external speculative data, the address of the shadow store remains to be resolved. The outside mechanisms control the speculative execution by supplying the appropriate address for the shadow store. Eventually, the validity of the speculative data will be verified by the CC-protocol. If the data were correct the outside mechanisms supply to the shadow store an irrelevant address (*e.g.* 0x0000). If, however, the data were found to be wrong, their address is supplied to the shadow store, thereby squashing all incorrect execution. Note that there is some sort of random value speculation involved in these schemes: even if there never was a producer-consumer relationship but the data just happened to be correct, the speculative execution is committed. Unless the dynamic instruction window overflows with speculative instructions the processor will race ahead while coherence enforcement will follow behind (otherwise it will stall, waiting for coherence enforcement). The extent that this is possible remains to be investigated with more powerful simulators that implement speculative execution.

Other approaches include the recently introduced speculative loads of the Intel’s EPIC^{TM} architecture (and its implementation IA64^{TM}). In such architectures a speculative load can be issued to read speculative data. A subsequent check instruction will test for the validity of the load. The check instruction can receive information from the coherence mechanisms about the validity of the data. If the check instruction receives a confirmation from outside that the data are correct, execution continues. Otherwise, the check instruction branches to routines that undo (in software) the effects of the speculative execution and restart the faulting load and subsequent instructions.

6.2.7 Results

In this section I present preliminary results for the producer-consumer prediction using the pre-send optimization. I implemented all mechanisms described in the preceding sections in the WWT, except speculative execution. Thus, the producers use the predictors to send cache blocks to the consumers; the pre-send messages are accepted in the consumer nodes only if there is available space in their cache in the form of invalid cache blocks; the consumers, upon a miss, access their caches to read speculative data. However, they cannot execute speculatively so they wait until they obtain a coherent cache block through the CC-protocol. When the coherent cache block is brought into the cache the consumers compare the speculative data to the coherent data to determine mis-speculations. Table 17 and Table 18 show statistics gathered using this setup for four benchmarks (OCEAN, BARNES, GAUSS, and SPARSE).

Table 17 shows the results using the LAST-PREDICTION scheme and Table 18 using the INTERSECTION-PREDICTION scheme. Both tables list the number of static stores that generated misses or write-faults and the number of entries in the prediction tables. These two numbers are the same since all stores encountered are tracked. Similar to the other two instruction-based predictions described in previous sections, the number of predictor entries required is very low for all programs. The total number of predictor probes gives an indication of the

Statistics	LAST PREDICTION			
Benchmark	**OCEAN**	**BARNES**	**GAUSS**	**SPARSE**
Static Stores considered (all nodes)	2499	1636	471	310
average per node	79	51	15	10
Predictor entries allocated (all nodes)	2499	1636	471	310
average per node	79	51	15	10
Total number of predictor probes (all nodes)	1378698	106535	175564	813815
% of probes that return a prediction	56%	63%	54%	3%
Total pre-send messages sent	402404	100696	87930	81756
% of non-null predictions	52%	150%	93%	335%
Pre-sends as % of data-carrying messages	10%	3%	12%	1%
pre-sends rejected at consumers	168463	60106	2466	24085
% of total pre-sends	42%	60%	3%	29%
pre-send accessed in consumers	158103	24984	85183	56421
% of total sent	39%	25%	97%	70%
Accessed pre-sends verified as correct	135866	19439	85123	55032
% of total accessed	86%	78%	100%	98%
% of total sent	34%	19%	97%	67%
Accessed pre-sends failed to verify	22237	5545	60	1390
% of accessed	14%	22%	0%	2%
% of total sent	5%	6%	0%	3%

Table 17. Statistics for producer-consumer LAST-PREDICTION with speculative pre-send (32 nodes).

usage of the predictors (equivalent to the number of coherence events generated by stores). The percentage of the probes that return a prediction is a metric that depends on the prediction scheme employed. For the first scheme, LAST-PREDICTION, a saturating counter is used to indicate whether the successive store instances have common consumers. When the predictor is probed and the counter is below the threshold a null prediction is returned. The percentage of non-null predictions ranges from 3% for SPARSE to 63% for BARNES. This percentage can be changed by tuning the threshold of the saturating counter and the parameter that defines the overlap in the consumer bit-maps. For the second scheme, INTERSECTION-PREDICTION, we do

Statistics	INTERSECTION PREDICTION			
Benchmark	**OCEAN**	**BARNES**	**GAUSS**	**SPARSE**
Static Stores considered (all nodes)	2500	1644	471	310
average per node	78	51	15	10
Predictor entries allocated (all nodes)	2500	1644	471	310
average per node	78	51	15	10
Total number of predictor probes (all nodes)	1378658	106263	175564	813960
% of probes that return a prediction	100%	97%	100%	100%
Total pre-send messages sent	247428	36530	87809	39480
% of non-null predictions	18%	34%	50%	5%
Pre-sends as % of data-carrying messages	7%	1%	12%	1%
pre-sends rejected at consumers	57268	15636	2032	10480
% of total pre-sends	23%	43%	2%	26%
pre-send accessed in consumers	129465	14142	85612	28025
% of total sent	52%	39%	97%	71%
Accessed pre-sends verified as correct	109501	11227	85567	26745
% of total accessed	85%	79%	100%	95%
% of total sent	44%	31%	97%	68%
Accessed pre-sends failed to verify	19964	2915	45	1280
% of accessed	15%	21%	0%	5%
% of total sent	8%	8%	0%	3%

Table 18. Statistics for producer-consumer INTERSECTION-PREDICTION with speculative pre-send (32 nodes).

not employ a saturating counter. A null prediction is returned the first two times a store is encountered. The non-null predictions can generate from zero to 32 pre-send messages (depending on the number of consumers predicted). However, a prediction may be nullified if the cache block is not available at the time of the pre-send. Because the pre-send can be delayed until a synchronization point, cache blocks are often lost before they can be sent. Under these conditions the total number of pre-sends is shown in the corresponding rows of the two tables. The total number of pre-sends is also expressed as a percentage of the non-null predictions in the same row.

Some of these pre-sends are rejected in the consumer nodes because there is no free space in their caches in the form of invalid cache-blocks. This number is high (*e.g.*, 60% for BARNES). A possible solution is to implement a speculative pre-send cache that holds the pre-sends that do not fit in the main cache. To read speculative data, the processor would access this cache in parallel with its main cache. Such a cache is additional hardware but would increase the number of pre-sends accessed and verified as correct.

Finally, the numbers of interest are the number of pre-sends accessed in the consumer nodes and the percentage of them verified as correct. For OCEAN, and GAUSS these two numbers are comparable for the two prediction schemes. For BARNES and SPARSE the second prediction scheme (INTERSECTION-PREDICTION) performs better. The percentage of pre-sends accessed ranges from 25% (BARNES) to 97% (GAUSS) for the first scheme and from 39% (BARNES) to 97% (GAUSS) for the second scheme. The percentage of the accessed pre-sends that are verified through the CC-protocol as correct (a direct measure of mis-speculations) is high: for LAST-PREDICTION it ranges from 78% for BARNES (22% mis-speculations) to 100% for GAUSS (0% mis-speculations); for INTERSECTION-PREDICTION it ranges from 79% for BARNES to 100% for GAUSS. For all programs the percentage of correct pre-sends is comparable for the two schemes, with values ranging from 19% to 97% for LAST-PREDICTION and 31% to 97% for INTERSECTION-PREDICTION. More advanced prediction schemes such as two-level adaptive prediction have potential to increase this percentage.

The optimization based on speculative execution is a tradeoff between bandwidth and latency. In hope of reducing the apparent latency of reads, we send more data that, nevertheless, will be re-sent for verification. The results show that: (i) a significant number of pre-sends will allow the processors to go ahead and execute useful work and (ii) the pre-send traffic is low ranging from 1% to 12% of the total data traffic (see Table 17 and Table 18, “pre-sends as % of data-carrying messages”). If these pre-sends are in the critical path of the program execution, then we could considerably reduce latency, consuming a small amount of bandwidth.

6.3 Predictor interactions

In this thesis, I have not examined the combined effects of the three prediction mechanisms (for migratory sharing, wide sharing, and producer-consumer sharing). The predictors for each sharing pattern can be integrated into a generalized structure (since predictor entries of one scheme can be embedded in the predictor entries of another). Integrating predictor structures, however, is not trivial and may lead to instability, if different sharing patterns conflict in predictor entries.

The predictor entries for the producer-consumer sharing refer to store instructions and thus they are easily distinguished from wide-sharing entries or migratory-sharing entries that refer to load instructions. However, the wide-sharing entries and migratory-sharing entries both refer to load instructions. One approach to resolve conflicts would be to tag predictor entries with the type of sharing involved at allocation time and use them as such until they are replaced. For example, we can allocate an entry as a wide-sharing entry, if at the time of the load-miss we detect wide sharing, otherwise we can allocate it as a migratory-sharing entry. Once an entry is allocated, we can only update it with the update mechanisms of the corresponding sharing pattern. Inactive entries (with a prediction counter set to zero) could be replaced to compensate for possible errors in the sharing type at allocation time. Additionally, we can apply similar rules for the store instructions. For example, store instructions that correspond to producer-consumer entries cannot be used to update migratory entries and vice-versa.

Further research is required to understand the interaction of sharing patterns in a common prediction structure and, in general, the combined effects of the three prediction schemes. However this is beyond the capacity of this thesis.

6.4 Summary

In this chapter I proposed instruction-based prediction to optimize producer-consumer sharing. The idea is to predict which node is going to consume a value generated by a store (similarly to uniprocessor dependence prediction and synchronization [72]). I examine store instructions that generate write-faults and keep track of the potential readers of the newly written cache-blocks using very few resources.

Using simple predictors we can predict, upon seeing a store-write-fault, whether there is a stable producer-consumer relation and apply a pairwise sharing optimization with direct cache-to-cache transfers without involving the home-node directory. Using more advanced predictor structures, we can predict the identity of the consumer(s). In this case we can *speculatively pre-send* the newly created values to the predicted consumers, who can use these values *speculatively* at miss time, but they have to *verify* them through the normal cache coherence protocol.

For the pairwise sharing prediction/optimization, I found that it does not offer significant advantages over SCI's default pairwise sharing optimization. For the producer-consumer optimization based on speculation, I found that for four programs more than 78% of the speculative pre-sends can be successful. For this prediction, no more than 105 9-byte predictor entries were ever needed in any node.

7 Summary and Future Directions

The contributions of this thesis are twofold:

1. I propose two novel optimizations to optimize wide sharing and producer-consumer sharing. The first optimization is called GLOW and it is the major thrust of this thesis. To a lesser extent, I examine the novel producer-consumer-sharing optimization and a migratory-sharing optimization previously proposed.
2. I classified the techniques to identify sharing patterns and selectively apply the appropriate optimizations. I proposed various such techniques for GLOW. Furthermore, I introduced a novel dynamic instruction-based technique which is based on prediction. I have demonstrated how this instruction-based prediction technique can be applied for wide sharing, migratory sharing, and producer-consumer sharing.

In the rest of this chapter, I summarize GLOW (Section 7.1), instruction-based prediction (Section 7.2), and conclude with notes for future directions (Section 7.3).

7.1 The GLOW optimization for wide sharing

In this thesis I have shown that the number of accesses to widely-shared data can be large, even if the amount of widely-shared data is small. The time spent in accessing such data can be considerable, hence there is considerable benefit in providing transparent hardware support for widely-shared data. This benefit increases with system size, since large systems suffer the most from widely-shared data.

For economic reasons, hardware support for specific sharing patterns must be transparent and non-intrusive to the commodity parts of the system. I propose the GLOW extensions to cache coherence protocols which are designed with transparency in mind: they are implemented in the network domain, outside commodity workstation boxes and are transparent to the underlying coherence protocol. The GLOW extensions work on top of another cache-coherence protocol, by building sharing trees mapped well on top of the network topology, thus providing scalable reads and scalable writes. However, in their static form they require the user to define the widely-shared data and issue special requests that can be serviced by GLOW. This is undesirable for various reasons, including implementation difficulties that inhibit transparency. In this thesis I propose and study three schemes that can detect widely-shared data at run-time and I compare them against SCI, static GLOW and combining. Each scheme exhibits different performance and cost characteristics, hence it is valuable to explain why they perform as they do. To this end, I examined each scheme's effects on the read-runs of six programs.

The first scheme, AGENT DETECTION, discovers widely-shared data more reliably than read-combining, by expanding the window of the observable requests. Switch nodes remember recent requests even if these have long left the switch. Requests whose addresses have been seen in the window are intercepted (as requests for widely-shared data) and passed to the GLOW extensions for further processing. The interesting characteristic of this scheme is that, in large systems, even a small window performs very well. This scheme achieves a significant percentage of the performance improvement of the static GLOW and has the potential to outperform the static version in programs where it is difficult for the user to define the widely-shared data. Since it requires modification only in the switch nodes, I believe it is the least intrusive of all the schemes. As for congestion-based combining (COMBINING,) which is slightly simpler, I found that it is highly dependent on the congestion characteristics of the applications.

In the second scheme, DIRECTORY DETECTION, the directories are modified to discover the widely-shared data by counting reads between writes. When a directory finds a widely-shared data block, it notifies the nodes in the system to subsequently request this data block using GLOW. The applicability of this scheme is limited. It works well when data blocks are widely accessed more than once.

The third scheme, INSTRUCTION PREDICTION, is the most successful and is based on predicting which load instructions are going to access widely-shared data. Although its implementation is intrusive to the processor itself, it offers the best performance. Finally, in this thesis, I used read-run analysis to gain insight on how these schemes affect accesses to widely-shared data. This tool enables us to visualize these effects and to reason about the behavior of the various schemes.

7.2 Instruction-based prediction

In this thesis I also explore instruction-based prediction, to transparently optimize hardware shared-memory. Instruction-based prediction is well established in the uniprocessor world, but novel in the world of parallel shared-memory architectures. The compelling advantage of instruction-based prediction, compared to address-based prediction, is that it requires *very few* prediction resources (for the programs examined).

I propose and study instruction-based prediction that *logically* stands halfway between the processor and the cache-coherence protocol mechanisms. It requires two streams of information to converge to the prediction structures: from the processor it requires the PC of the load and store instructions that generate coherence events; from the cache-coherence mechanisms it requires coherence information. Thus, we can track the history of loads and stores in relation to coherence, events such as cache misses or write-faults. Subsequently, each time a known load or store generates a new coherence event we can take action to optimize it. In contrast, uniprocessor instruction-based prediction is deeply embedded in the processor and is invoked with every dynamic instruction instance.

To make the case that instruction-based prediction is a serious competitor—not only in terms of resource usage but also in terms of performance—to previously proposed address-based prediction mechanisms, I proposed optimizations for three different sharing patterns:

- Wide sharing (also discussed in the previous section). This prediction/optimization works very well and consistency outperforms DIRECTORY DETECTION (address-based scheme) on six benchmarks which exhibit wide sharing (GAUSS, SPARSE, APSP, TC, CG, and BARNES).

With appropriate mechanisms for adapting back to non-wide sharing, there is no negative performance impact on two control benchmarks (CHOLESKY and OCEAN). No more that 56 (5-byte) predictor entries were ever needed in any node's prediction table for the small scientific programs used.

- Migratory sharing. This prediction/optimization works well for three benchmarks (CHOLESKY, MP3D, and PTHOR) that exhibit migratory sharing and is competitive to previously proposed address-based adaptive protocols. Equipped with safeguards to avoid applying the optimization to non-migratory sharing, it shows no negative performance impact on four control benchmarks (GAUSS, APSP, BARNES, OCEAN). No more than 71 (9-byte) predictor entries were ever needed in any node's prediction table for the small scientific programs used.
- Pairwise sharing and producer-consumer sharing. For the pairwise sharing prediction/optimization, I found that it does not offer significant advantages over the default pairwise optimization of SCI. For the producer-consumer optimization that is based on speculation, I found a significant number of speculative pre-sends can be successful. For this prediction, no more than 51 predictor entries were ever needed in any node for the small scientific programs used.

7.3 Future directions

I believe that this work will be a starting point for novel and better instruction-based prediction optimizations. Similarly to work that examined the coherence behavior of data [100], we need to examine the behavior of the instructions in relation to the cache-coherence protocol events, taking into account hardware parameters such as cache size and block size. Also, research is needed to examine how instruction-based prediction can be applied to bus-based shared-memory systems and even software-based shared-memory. Yet another direction is to examine hybrid prediction schemes that use both instruction-based and address-based prediction. This is especially interesting with the advent of generalized address-based prediction [75].

I consider the instruction-based sharing-pattern identification techniques of this thesis, a starting point. There are many possible applications of instruction-based prediction. For example, Dynamic Self-Invalidation (DSI) which was proposed as an address-based technique by Lebeck and Wood [64] may be achieved with instruction-based prediction (suggested by Sohi). I have identified yet another technique: *consumer-producer* instruction-based prediction, which is—in some sense—the opposite of the producer-consumer instruction-based prediction technique described in Chapter 6. In the consumer-producer scheme, upon encountering a load instruction—*i.e.*, a consumer—we can make a prediction about the identity of the producer of the data being accessed. Potentially, we could prefetch the value directly from the predicted producer—again using speculative execution as in the producer-consumer scheme (Chapter 6)—or we can coherently access the desired data directly from the predicted producer without going to the directory, if this is supported by a specialized cache coherence protocol such as GLOW. Other interesting applications of instruction-based prediction may emerge with future research.

Finally, I subscribe to Hill’s opinion that speculation will play an increasingly important role in transparently optimizing shared-memory. In this work I propose such an optimization and I have performed a preliminary evaluation constrained by the limits of the WWT. Future work is needed to research in depth speculation in shared-memory.

8 References

[1] Gheith A. Abandah and Edward S. Davidson, “Effects of Architectural and Technological Advances on the HP/Convex Exemplar’s Memory and Communication Performance.” In *Proceedings of the 25th ISCA*, June 1998.

[2] Gheith Abandah, “Characterizing Shared-Memory Applications: A Case Study of the NAS Parallel Benchmarks.” Hewllet-Packard Labs *Technical Report HPL-97-24*, January 1997.

[3] Hazim Abdel-Shafi, Jonathan Hall, Sarita V. Adve, and Vikram S. Adve, “An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors.” In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, February 1997.

[4] Santosh G. Abraham, Rabin A. Sugumar, Daniel Windheiser, B. R. Rau and Rajiv Gupta. “Predictability of Load/Store Instruction Latencies.” In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, November 1993.

[5] Sarita V. Adve, Mark D. Hill, “Implementing Sequential Consistency In Cache-Based Systems.” *International Conference on Parallel Processing*, August 1990.

[6] Sarita V. Adve, Mark D. Hill, “Weak Ordering - A New Definition.” In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA)*, June 1990.

[7] A. Agarwal et al., “The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor.” In *Dybous and Thakkar (eds.), Scalable Shared Memory Multiprocessors*, Kluwer Academic Publishers, Boston, MA, 1992.

[8] A. Agarwal, M. Horowitz and J. Hennessy, “An evaluation of Directory schemes for Cache Coherence.” In *Proceedings of the 15th International Symposium on Computer Architecture*, pp. 280-289, June 1988.

[9] George S. Almasi, Alan Gottlieb, “Highly Parallel Computing.” Benjamin/Cummings Publishing Company Inc., Redwood City, California, 1994.

[10] C. Amza et al., “TreadMarks: Shared Memory Computing on Networks of Workstations.” *Computer*, Vol. 29, No. 2, pp. 18-28, Feb. 1996.

[11] Tom Anderson, David Culler, David Patterson, “A Case for Networks of Workstations: {NOW}.” *Technical Report*, University of California Berkeley, Jul. 1994.

[12] J. L. Baer and W. H. Wang, “Architectural Choices for Multi-Level Cache Hierarchies.” In *Proceedings 16th International Conference on Parallel Processing*, pp. 258-261, 1987.

[13] Jean-Loup Baer and Wen-Hann Wang, “On the Inclusion Properties for Multi-Level Cache Hierarchies.” In *Proceedings of the 15th Annual Symposium on Computer Architecture*, May 1988.

[14] David H. Bailey et al., “The NAS Parallel Benchmarks: Summary and Preliminary Results.” *IEEE Supercomputing ‘91*, pp 158-165. November 1991.

[15] Andrew J. Bennett, Paul H. J. Kelly, Jacob G. Refstrup and Sarah A. M. Talbot, “Using Proxies to Reduce Controller Contention in Large Shared-Memory Multiprocessors.” In *Luc Bouge et al, editors, EURO-PAR 96*, Aug. 1996, pp. 445-452, Vol. 1124 Lecture Notes in Computer Science, Springer-Verlag.

[16] R. Bianchini and T.J. LeBlanc, “Software Caching on Cache-Coherent Multiprocessors.” In *Proc. of the 4th Symposium on Parallel and DIstributed Processing*, Dec. 1992.

[17] R. Bianchini and T. J. LeBlanc, “Eager Combining: A Coherency Protocol for Increasing Effective Network and Memory Bandwidth in Shared-Memory Multiprocessors.” In *Proceedings of the 6th Symposium on Parallel and Distributed Processing*, October 1994.

[18] R. Bianchini, T.J. LeBlanc, and J.E. Veenstra, “Categorizing Network Traffic in Update-Based Protocols on Scalable Multiprocessors.” In *Proceedings of the 10th International Parallel Processing Symposium*, April 15, 1996.

[19] D.C. Burger and D.A. Wood, “Accuracy vs. Performance in Parallel Simulation of Interconnection Networks,” In *Proceedings of the 9th International Parallel Processing Symposium (IPPS)*, April, 1995.

[20] D.C. Burger and J.R. Goodman, “Simulation of the SCI Transport Layer on the Wisconsin Wind Tunnel,” *2nd International Workshop on SCI-based High-Performance Low-Cost Computing*, pp. 57-66, March, 1995.

[21] CRAY Research Inc., CRAY T3D System Architecture Overview, 1993.

[22] John Carter, John Bennett and Willy Zwaenepoel, “Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence.” In *Proceedings of the Conference on the Principles and Practices of Parallel Programming*, 1990.

[23] Lucien M. Censier and Paul Feautrier, “A New Solution to Coherence Problems in Multicache Systems.” *IEEE Trans. Computers,* Vol. 27, No. 12, pp. 1112-1118, Dec. 1978.

[24] David Chaiken, John Kubiatowicz, Anant Agarwal. “LimitLESS Directories: A Scalable Cache Coherence Scheme.” In *Proc. of the 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 224-234, April 1991.

[25] Satish Chandra, James R. Larus, Anne Rogers, “Where is Time Spent in Message-Passing and Shared-Memory Programs?” In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 61-73, October 1994.

[26] Convex Computer Corporation, “The Exemplar System,” 1994.

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, “Introduction to Algorithms.” MIT Press, Cambridge, MA, 1990.

[28] Alan L. Cox Robert J. Fowler, “Adaptive Cache Coherency for Detecting Migratory Shared Data.” In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993.

[29] Susan J. Eggers and Randy H. Katz, “A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation.” In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 373-382, Jun. 1988.

[30] Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, Ioannis Schoinas, Mark D. Hill James R. Larus, Anne Rogers, David A. Wood, “Application-Specific Protocols for User-Level Shared Memory.” *Supercomputing* ‘94, Nov. 1994.

[31] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill, “Programming for Different Memory Consistency Models.” *Journal of Parallel and Distributed Computing*, 15(4), 1992.

[32] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy, “Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors.” In *Proceedings of the 17th International Symposium on ComputerArchitecture*, pages 15-26, May 1990.

[33] A. Gonzalez, C. Aliagas, M Valero. “A Data Cache with Multiple Caching Strategies Tunned to Different Types of Locality.” In *Proceedings of the International Conference on Suprecomputing*, 1997.

[34] J. R. Goodman, P. J. Woest, “The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor.” In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, May 1988.

[35] J. R. Goodman, M. K. Vernon, P. J. Woest, “A Set of Efficient Synchronization Primitives for a Large-Scale Shared-Memory Multiprocessor.” In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (APLOS III)*, April 1989.

[36] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, M. Snir, “The NYU Ultracomputer Designing a MIMD Shared-Memory Parallel Computer.” *IEEE Transactions on Computers*, Vol. C-32, no 2, pp. 175-189, February 1983.

[37] Eric Hagersten, Anders Landin, and Seif Haridi, “DDM — A Cache-Only Memory Architecture.” *IEEE Computer*, Vol 25. No 9, September 1992.

[38] Mark D. Hill, “Multiprocessors Should Support Simple Memory Consistency Models.” *IEEE Computer*, Vol. 31, No 8, pp 28-34, Aug. 1998.

[39] Mark D. Hill, James R. Larus, David A. Wood, “Tempest: A Substrate for Portable Parallel Programs.” *COMPCON* Spring 95, 1995.

[40] W. D. Hillis and Lewis W. Tucker. “The CM-5 Connection Machine: A Scalable Supercomputer.” *Communications of the ACM*, Nov. 1993.

[41] IEEE Standard for Scalable Coherent Interface (SCI) 1596-1992, IEEE 1993.

[42] Intel Corporation, “Paragon Technical Summary.” Intel Supercomputer Systems Division, 1993.

[43] David V. James and Stefanos Kaxiras, “IEEE Standard for Cache Optimization for Large Numbers of Processors using the Scalable Coherent Interface (SCI) Draft 0.35/0.36,” 1995.

[44] Ross E. Johnson, “Extending the Scalable Coherent Interface for Large-Scale Shard-Memory Multiprocessors.” Ph.D. Thesis, University of Wisconsin-Madison, 1993.

[45] Ross E. Johnson, James R.Goodman, “Interconnect Topologies with Point-to-Point Rings.” In *Proceedings of the International Conference on Parallel Processing*, August 1992.

[46] N. P. Jouppi, “Cache Write Policies and Performance.” In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. May 1993.

[47] Alain Kägi, Nagi Aboulenein, Douglas C. Burger, James R. Goodman, "Techniques for Reducing Overheads of Shared-Memory Multiprocessing." In *Proceedings of the International Conference on SuperComputing*, July 1995.

[48] Alain Kägi, Doug Burger, and James R. Goodman, "Efficient Synchronization: Let Them Eat QOLB." In *Proceedings of the 24th Annual International Symposium on Computer Architecture,* June 1997.

[49] Stefanos Kaxiras, "Kiloprocessor Extensions to SCI." In *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

[50] Stefanos Kaxiras and J. R. Goodman, "The GLOW Cache Coherence Protocol Extensions for Widely-shared Data." In *Proceedings of the International Conference on Supercomputing*, May 1996.

[51] Stefanos Kaxiras and J. R. Goodman, "The GLOW Cache Coherence Extensions for Widely-shared Data." *University of Wisconsin-Madison, C.S. Dept., Technical Report 1305,* March 1996.

[52] Stefanos Kaxiras and James R. Goodman, "Improving Request-Combining for Widely-shared Data in Shared-Memory Multiprocessors." In *Proceedings of the Third International Conference on Massively Parallel Computing Systems (MPCS), April 1998.*

[53] Stefanos Kaxiras, Stein Gjessing, and James R. Goodman, "A Study of Three Dynamic Approaches to Handle Widely-shared Data in Shared-Memory Multiprocessors." In *Proceedings of the International Conference on Supercomputing,* July 1998.

[54] Stefanos Kaxiras. "The Use of Instruction-Based Prediction in Hardware Shared-Memory." *University of Wisconsin-Madison Computer Sciences Dept. TR-1368,* April 1998.

[55] Stefanos Kaxiras, Rabin Sugumar, James Schwarzmeier. "Distributed Vector Architectures: Beyond a Single Vector-IRAM." *24th ISCA Workshop on Processor-Memory Integration: Chips that Compute and Remember,* June 1997.

[56] Stefanos Kaxiras and Rabin Sugumar. "Distributed Vector Architectures: Fine Grain Parallelism with Efficient Communication." *University of Wisconsin-Madison CS Dept. TR-1339*, February 1997. Also available from CRAY Research.

[57] Kendall Square Research. "Kendall Square research technical Summary," Waltham, MA, 1992.

[58] Jai-Hoon Kim and Nitin H. Vaidya, “Adaptive Migratory Scheme for Distributed Shared Memory.” *11th ACM International Conference on Supercomputing (ICS)*, Vienna, Austria, July 1997.

[59] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas, “Data Forwarding in Scalable Shared-MemoryMultiprocessors.” In *Proceedings of the International Conference on Supercomputing*, July 1995.

[60] 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, April 1994.

[61] Leslie Lamport, “How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs.” *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[62] James Laudon and Daniel Lenoski, “The SGI Origin: A cc-NUMA Highly Scalable Server.” In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[63] Alvin R. Lebeck and G. S. Sohi, “Request Combining in Multiprocessors with Arbitrary Interconnection Networks.” In *IEEE Transactions on Parallel and Distributed Systems*, November 1994.

[64] Alvin R. Lebeck and David A. Wood, “Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors.” In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, June 1995.

[65] Daniel Lenoski *et al*., “The Stanford DASH Multiprocessor.” *IEEE Computer*, Vol. 25 No. 3, pp. 63-79, March 1992.

[66] Kai Li, Paul Hudak, “Memory Coherence in Shared Virtual Memory Systems.” *ACM Transactions on Computer Systems*, Vol. 7, No. 4, pp. 321-359, November 1989.

[67] Li, K. and Hudak, P., “Memory Coherence in Shared Virtual Memory Systems.” In *Proc. of the 5th Annual ACM Symp. on Principles of Distributed Computing (PODC’86)*, pp. 229-239, August 1986.

[68] Tom Lovett, and Russell Clapp, “STiNG: A CC-NUMA Computer System for the Commercial Marketplace.” In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[69] Yeong-Chang Maa, Dhiraj K. Pradhan, Dominique Thiebaut, “Two Economical Directory Schemes for Large-Scale Cache-Coherent Multiprocessors.” *Computer Architecture News*, Vol 19, No. 5, pp. 10-18, September 1991.

[70] John M. Mellor-Crummey, Michael L. Scott, “Synchronization Without Contention.” In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*. April 8, 1991.

[71] Haim E. Mizrahi, Jean-Loup Baer, Edward D. Lazowska, John Zahorjan, “Introducing Memory into Switch Elements of Multiprocessor Interconnection Networks.” In *Proceedings of the 16th Annual Symposium on Computer Architecture*, May 1989.

[72] A. Moshovos, S. E. Breach, T. N. Vijaykumar, G. S. Sohi, “Dynamic Speculation and Synchronization of Data Depenences.” In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[73] A. Moshovos and G. S. Sohi, “Streamlining Inter-operation Memory Communication via Data Dependence Prediction.” In *Proceeding of the 30th Annual Symposium on Microarchitecure*, Dec. 1997.

[74] Shubhendu S. Mukherjee, Mark D. Hill, “An Evaluation of Directory Protocols for Medium-Scale Shared-Memory Multiprocessors.” *International Conference on Supercomputing*, 1994.

[75] Shubhendu S. Mukherjee and Mark D. Hill “Using Prediction to Accelerate Coherence Protocols.” In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA)*, July 1998.

[76] Håkan Nilsson, Per Stenström, “The Scalable Tree Protocol—a Cache Coherence Approach for Large-Scale Multiprocessors.” *4th IEEE Symposium on Parallel and Distributed Processing*, pp. 498-506, 1992.

[77] David Patterson et al., “The case for Intelligent RAM.” *IEEE Micro*, Vol 17, No. 2, March/April 1997.

[78] Gregory F. Pfister and V. Alan Norton, “‘Hot Spot’ Contention and Combining in Multistage Interconnection Networks.” In *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 790-797, August 20-23, 1985.

[79] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood, “The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers.” In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pp. 48–60, May 1993.

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

[81] Ashley Saulsbury, Fong Pong, and Andreas Nowatzyk, “Missing the Memory Wall: The Case for Processor/Memory Integration.” In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA)*, May 1996.

[82] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin, “An Argument for Simple COMA.” In *Proceedings of the 1st International Symposium on High-Performance Computer Architecture*, January 1995, pp. 276--285

[83] Ioannis Schoinas et al., “Fine-grain Access Control for Distributed Shared Memory.” In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pp. 297-307, Oct. 1994.

[84] Ioannis Schoinas et al., “Implementing Fine-Grain Distributed Shared Memory on Commodity SMP Workstations.” *Technical Report TR-1307*, University of Wisconsin-Madison, March 1996

[85] Ioannis Schoinas and Mark D. Hill, “Address Translation Mechanisms in Network Interfaces.” In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[86] Steven L. Scott, “Toward the Design of Large-Scale, Shared-Memory Multiprocessors.” Ph.D. Thesis, University of Wisconsin-Madison, August 1992.

[87] Steven L. Scott, “Synchronization and Communication in the T3E Multiprocessor.” In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, October 1996.

[88] Steve L. Scott and James R. Goodman, “Performance of Pruning-Cache Directories for Large-Scale Multiprocessors,” *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, no. 5, May 1993.

[89] Steven L. Scott, James R. Goodman, Mary K. Vernon, “Performance of the SCI Ring.” In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 403-414, May 1992.

[90] R. Simoni and M. Horowitz, “Dynamic Pointer Allocation for Scalable Cache Coherence Directories.” In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, April 1991.

[91] J.P. Singh, W. -D. Weber, A. Gupta, “SPLASH: Stanford Parallel Applications for Shared Memory.” *Computer Architecture News*, March 1992.

[92] James E. Smith, “A Study of Branch Prediction Strategies.” In *Proceedings of the 8th Annual International Computer Archtecture Symposium*, 1981.

[93] Per Stenström, Mats Brorsson, Lars Sandberg, “An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing.” In *Proceedings of the 20th Annnual International Symposium on Computer Architecture*, 1993.

[94] P. Stenström, T. Joe, and A. Gupta, “Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures.” In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992, pp. 80--91.

[95] H. Sullivan and L. Cohn, “Shared Memory Computer Method and Apparatus,” U.S. Patent 4,707,781 , November 1987.

[96] Sarah A. M. Talbot and Paul H. J. Kelly, “Stable Performance for cc-NUMA Using First Touch Page Placement and Reactive Proxies.” In *High Performance Computing Systems and Applications*, Kluwer Academic Publishers.

[97] Gary S. Tyson and Todd M. Austin, “Improving the Accuracy and Performance of Memory Communication Through Renaming.” In *Proceedings of the 30th Annual Symposium on Microarchitecure,* December 1997.

[98] G. Tyson et al., “A New Approach to Cache Management." In *Proceedings of the 28th Annual Symposium on Microarchitecure*, Nov. 28 - Dec. 1, 1995.

[99] Anujan Varma and C.S. Raghavendra, “Interconnection Networks for Multiprocessors and Multicomputers, Theory and Practice.” IEEE Computer Society Press, 1994.

[100] Wolf-Dietrich Weber and Anoop Gupta, “Analysis of Cache Invalidation Patterns in Multiprocessors.” In *Proceedings of the 3rd International Conference on Architectural support for Programming Languages and Operating Systems*, pp. 243-256, April 1989.

[101] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta, “The SPLASH-2 Programs: Characterization and Methodological Considerations.” In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[102] David A. Wood et al., “Mechanisms for Cooperative Shared Memory.” In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[103] David A. Wood and Mark D. Hill, “Cost-Effective Parallel Computing.” *IEEE Computer*, 28(2):69–72, February 1995.

[104] Chuan-lin Wu, Tse-yun Feng, “Tutorial: Interconnection Networks for Parallel and Distributed Processing.” IEEE Computer Society Press, 1984.

[105] T-Y Yeh and Yale Patt, “Alternative Implementations of Two-Level Adaptive Branch Prediction.” In *Proceedings of the 19th Annual International Symposium on Computer Archtecture*, 1992.

[106] P.-C. Yew, N. -F. Tzeng, D. H. Lawrie, “Distributing Hot-Spot Addressing in Large-Scale Multiprocessors.” *IEEE Transactions on Computers*, Vol C-36, No. 4, April 1987.

[107] P.-C. Yew and P. Tang, “Software Combining Algorithms for Distributed Hot-Spot Addressing,” *Journal of Parallel and Distributed Computing*, October 1990.

Appendix 1: SCI Cache Coherence

The ANSI/IEEE Standard 1596 Scalable Coherent Interface represents a thorough and robust hardware solution to the challenge of building cache-coherent shared-memory multiprocessor systems. It defines both a network interface and a cache-coherence protocol. The network interface section of SCI defines a 1Gbyte/s ring interconnect, and the transactions that can be generated over it. A performance analysis by Scott, Vernon, and Goodman [89] showed that an SCI ring can accommodate small numbers of high-performance nodes, in the range of four to eight. To build larger systems, networks constructed out of smaller rings must be used (*e.g.*, *k*-ary *n*-cubes [45], multistage topology networks etc.). In such complex topology networks GLOW can take advantage of network locality to improve the performance of the network. Larger systems can also be built using centralized switches, each connecting many rings. However, in systems with large centralized switches, GLOW may be of little benefit and other schemes such as STEM [44] are more appropriate.

SCI also defines a distributed directory-based cache-coherence protocol. In contrast to most other directory-based protocols (*e.g.*, DASH [65]), that keep all directory information in memory, SCI distributes the directory information among the sharing nodes in a doubly-linked sharing list. The sharing list is stored with the cache blocks throughout the system. The memory directory has a pointer to the last node that requested the data. In a stable list —where data distribution has ceased— this node is called *head.* As the head of the sharing list, a node has both read and write permissions to the cache block; all other nodes in the sharing list only have read permission (except in *pairwise-sharing mode* where the write permission is transferred back-and-forth between the two nodes of the pairwise-sharing list). However, when many nodes are trying to join the list concurrently, a singly-linked *prepend* queue is formed in front of the head. The head may be asked to link itself to its prepending successor at any time, but it may briefly postpone its response until it is ready to give up its write permission. In Figure 8.1, an SCI sharing list is depicted. Caches, represented by small rectangles, are connected in linked lists with their *forward* and *backward* pointers. There are two directions defined for the SCI lists:

- *upstream*: toward the home-node directory, in the direction of the backward pointers
- *downstream*: toward the tail of the list, in the direction of the forward pointers

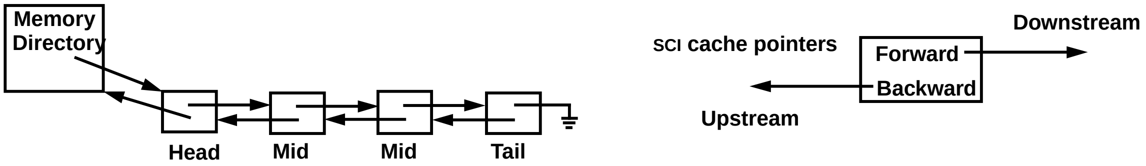


Figure 8.1. SCI sharing list.

In the following sections I will describe the three basic operations of SCI: construction of sharing lists, node removal from a sharing list or *rollout*, and invalidation of sharing lists. In general, latency of the sharing-list construction may be affected by the number of sharing nodes (due to contention, data propagation delays, etc.). Addition of a single node to a stable SCI list, as well as, node removal from a sharing list (rollout), does not depend on the number of nodes in the list. However, when *N* nodes join a prepend list at approximately the same time, the propagation delay of the data through the list is *O(N)*. The latency of the invalidation of an *N*-

node sharing list is also *O(N)*.

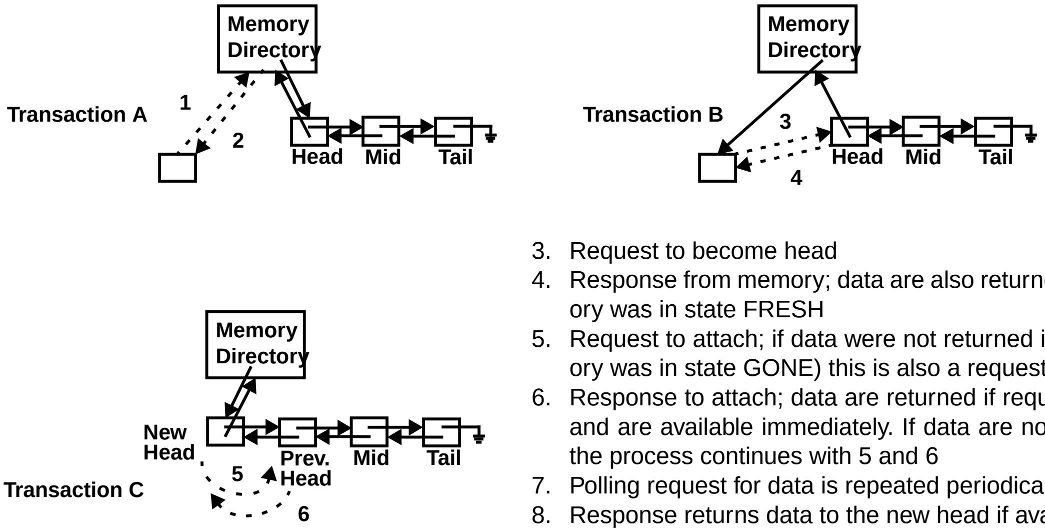
SCI sharing list construction**—**An SCI sharing list is comprised of nodes that accessed the same data block. The first node that requests data from memory triggers the creation of the sharing list (Figure 8.2). This node becomes the head and only node of the list. The memory directory points to this node and the node points back to the memory directory by means of the memory address.

Memory Directory
Transaction A
1
2
Memory Directory
Head

Figure 8.2. Creation of a sharing list.

Subsequently, a node can join the sharing list by asking the memory directory for data, thus joining a prepend queue (Transaction A, subactions 1 and 2 in Figure 8.3). The node attaches to their previous head (Transaction B, subactions 3 and 4 in Figure 8.3). It might get the data from the memory itself or from the previous head of the list, depending on whether memory has a valid copy of the cache block (denoted by the memory state FRESH) or the memory copy is potentially stale (denoted by the memory state GONE). In the second case, data distribution in the prepend queue proceeds in the upstream direction, from the oldest head node toward the memory directory (Transaction C, subactions 5 and 6 in Figure 8.3).

Rollout**—**A node can also leave the sharing list by communicating with its neighbors. This operation is called *rollout* in the SCI terminology, and it takes place in two situations: (i) when

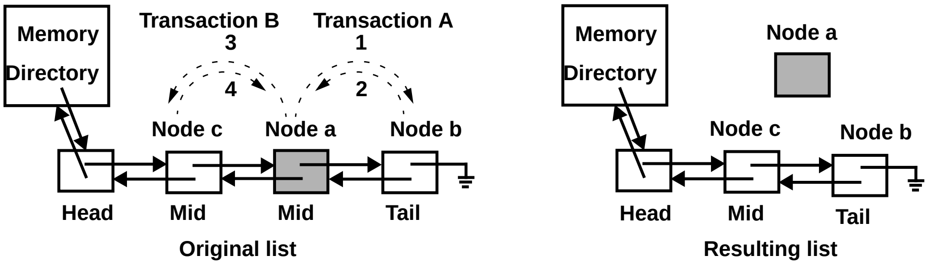


3. Request to become head
4. Response from memory; data are also returned if memory was in state FRESH
5. Request to attach; if data were not returned in 2 (memory was in state GONE) this is also a request for data
6. Response to attach; data are returned if requested in 3 and are available immediately. If data are not available the process continues with 5 and 6
7. Polling request for data is repeated periodically
8. Response returns data to the new head if available

Figure 8.3. Additions to a SCI sharing list.

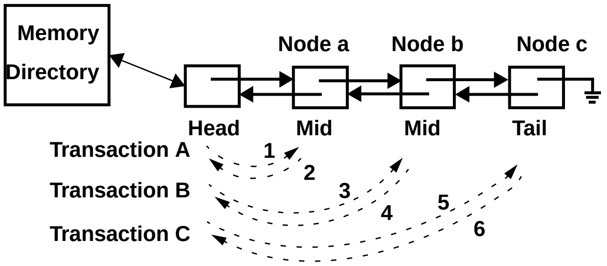
there is a conflict in a cache and the cache block has to be replaced, the node rolls out of the sharing list, and (ii) a node that is not currently the head of the sharing list has to roll out and become the new head in order to obtain write permission and write the cache block. In Figure 8.4, node **a** leaves the sharing list. With subaction 1 of Transaction A it notifies its downstream neighbor node **b** to point to node **c**; subaction 2 is the positive acknowledgment. Similarly, it notifies node **c** to point to node **b**. The resulting sharing list is shown in the right of Figure 8.4. Notice that the order of Transactions A and B is critical, to allow concurrent rollouts in the sharing list. In case of conflicts the downstream node (the one closer to the tail) has priority.

SCI sharing list invalidation—As the head of a sharing list, a node has to invalidate, or *purge,* the rest of the sharing list upon writing the cache block. This will force the rest of the processors that share the data to re-read them to get the new value. The head sends invalidation messages serially to the other nodes in the sharing list. Each of these nodes acknowledges the



1. Rollout; change backward pointer to **c**
2. Response
3. Rollout; change forward pointer to **b**
4. Response

Figure 8.4. SCI **cache rollout.**



1. Invalidation
2. Response with pointer to **b**
3. Invalidation
4. Response with pointer to **c**
5. Invalidation
6. Response with NULL pointer

Figure 8.5. SCI **sharing list invalidation.**

invalidation by returning its *forward* pointer. In Figure 8.5, the head of the list invalidates the other nodes. For Transaction A, subaction 1 invalidates node **a** and subaction 2 returns the pointer to **b** to the head; similarly for the rest of the transactions.

Appendix 2: Enhancements for the Construction of GLOW Trees

In Section 3.5.2 I have described the creation of the GLOW trees in a manner that is fully compatible with the current SCI cache-coherence protocol. However, by changing the way data are distributed in the tree we can further optimize the creation of the GLOW trees. There are three options for the distribution of data in the tree:

- **Tail-to-head**: As is currently defined in SCI and described previously. The agent instructs the node to prepend itself to the child list and request the data from the old head (exactly as the SCI memory would do). The first node to request the data attaches to the virtual tail of the child list. It will eventually get the data from the agent disguised as the virtual tail. This scheme requires the agent to maintain two pointers per child list: (i) the forward pointer of its memory personality and the backward pointer of its virtual tail personality.
- **Head-to-tail**: This is the reverse of the SCI data distribution. If a requesting node prepends to a child list, where the old head is waiting for the data, then it assumes responsibility to forward the data to the old head. When the agent gets the data it will pass it to the waiting heads of its child lists. The head nodes then forward the data toward the waiting tail nodes. This scheme does not require the agent to appear as a virtual tail at the end of its child lists. However, this scheme increases the variance of the latency experienced by the requesting nodes.
- **Broadcast on the ring**: This is a broadcast confined within a single ring. All nodes that are waiting will receive the broadcast and consequently the data. This scheme results in better performance than the base case, but is hard to implement on top of the current SCI

protocol. The reason is that the SCI protocol is based on request-response transactions and it uses polling in situations where nodes have to wait. For example, nodes in the prepend queue poll their predecessor until the data arrive from the directory and are distributed through the sharing list.

Appendix 3: Critical Section Detection

Another instruction-based optimization for migratory sharing is based on the observation that migratory data are typically accessed inside critical sections. Thus, all that is needed is to detect entry to and exit from critical sections. The standard optimization (*i.e.*, modifying the first load of migratory data into a coherent write) can be applied indiscriminately or in a controlled fashion inside critical sections. The appeal of this method lies in its simplicity: very small hardware structures are needed to support it.

To detect entry to and exit from critical sections we must recognize LOCK and UNLOCK operations. Since some sort of synchronization or atomic primitive is always used to implement the LOCK operation we only need to detect when such primitive is executed *successfully*. Figure 8.6 shows an MP3D critical section written by Alain Kägi. A number of synchronization primitives can be used to implement the entry in this critical section: Swap, Test&Swap (analogous to Test&Set), and QOLB [35]. Writes implementing memory fences (required for relaxed memory models) or the QOLB release primitive are used to exit the critical section. Detection of the entry in a critical section is to detect a successful execution of the atomic primitives of the QOLB primitive. Detection of the exit from the critical section requires remembering the address of the lock and detecting a write on this address. Thus a single address buffer is all that is needed to support detection of critical sections. On a successful execution of an atomic/synchronization primitive the address buffer is loaded with the address of the lock (used in the primitive). Subsequent writes are checked against the address in this buffer to detect the exit of the critical section. In the case of nested critical sections, the single address buffer must be upgraded to a small stack (supporting a fixed nesting depth).

When a critical section is detected, there are two methods to apply the migratory optimization:

1. Indiscriminately, treating all load-misses as coherent writes. This method requires no additional hardware (*e.g.*, a predictor) beyond what is necessary for critical section detection.
2. Controlled by instruction-based prediction, treating load-misses followed by store-write-fault as coherent writes. This method requires all the mechanisms described for instruction-based prediction of migratory sharing.

```
#if defined(USE_MCS)
            LOCK(glob->DeadLock);
#elif defined(USE_TSWAP)
            while (glob->DeadLock || wwt_swap_acq(&dead->DeadLock, 1));
#elif defined(USE_SWAP)
            while (wwt_swap_acq(&glob->DeadLock, 1));
#elif defined(USE_QOLB)
            _qolb_acquire_acq(&glob->DeadLock);
#elif defined(USE_WITH)
            with_annot(&glob->DeadLock) {
#endif USE_WITH

            glob->util[glob->DeadCount] += glob->clktemp-glob->lastclock;
            glob->lastclock=glob->clktemp;
            glob->DeadCount++;

#if defined(USE_MCS)
            UNLOCK(glob->DeadLock);
#elif defined(USE_TSWAP)
            _signal_write(&glob->DeadLock, 0);
#elif defined(USE_SWAP)
            _signal_write(&glob->DeadLock, 0);
#elif defined(USE_QOLB)
            _qolb_release_rel(&glob->DeadLock);
#elif defined(USE_WITH)
            }
#endif USE_WITH
```

Figure 8.6. Example of an MP3D critical section with migratory data. Entry is through a number of different synchronization algorithms based on different atomic/ synchronization primitives: Swap, Test&Swap, and QOLB. The MCS algorithm is implemented using Test&Swap. The USE_WITH algorithm is based on QOLB. Special writes (implementing memory fences) or the QOLB release primitive are used to exit the critical section.