Two Dynamic Methods for Efficient Large Scale Sharing

Stefanos Kaxiras
James R. Goodman

Technical Report #1351

December 1997

Two Dynamic Methods for Efficient Large Scale Sharing

Stefanos Kaxiras and James R. Goodman
{kaxiras,goodman}@cs.wisc.edu

***Abstract*—*Numerous studies have characterized the sharing patterns of programs, and it is generally believed that widely shared data occur infrequently and do not significantly affect performance. In practice, programmers are generally careful to avoid algorithms that exhibit wide sharing of data, because such algorithms are known to achieve poor performance on conventional shared-memory multiprocessors. We argue that in fact widely shared data is a more serious problem than previously recognized, and that furthermore, it is possible to provide support that actually gives an advantage to accesses to widely shared data. If a system can exploit the redundancy of the data to improve accessibility of widely shared data, programmers would find that the best algorithms make extensive use of widely shared data rather than eschewing. 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.*

Since contemporary cc-NUMA systems are build of commodity parts additional hardware support should be transparent and easily implementable without interfering with the rest of the system. The GLOW *extensions to cache coherence protocols previously proposed provide transparent support for widely shared data by defining functionality in the network domain. Modified switch nodes (*GLOW *agents) intercept requests for widely shared data and (transparently) build sharing trees that map well on the network topology. In their static version the extensions rely on the user to identify and expose widely shared data to the hardware. This approach is not appealing because: i) it requires modification of the programs, ii) it is not always possible to statically identify the widely shared data, and iii) it is incompatible with commodity hardware. To address these issues, we study two dynamic schemes to discover widely shared data at run-time. The first scheme is inspired by read-combining and it is based on observing requests in the* GLOW *agents. The agents intercept requests whose addresses have been observed recently and employ the* GLOW *extensions to handle them. We show with detailed simulations that this dynamic scheme: i) tracks closely the performance of the static* GLOW *and in some cases surpasses it, and ii) is considerably more stable than combining which is sensitive on network and application characteristics. In the second scheme, the memory directory discovers widely shared data by counting the number of reads between writes. Information about the widely shared nature of data is distributed to the nodes which subsequently request them as such. Simulations confirm that this scheme works well when the widely shared nature of the data is persistent over time.*

1 Introduction

It has been said that shared-memory multiprocessing is like virtual memory: it works very well as long as you don't really use it. This mostly facetious comment nevertheless captures the essence of a problem: shared-memory multiprocessing is only attractive if it can be supported robustly in the presence of frequent and complex memory sharing patterns. Numerous studies have characterized the sharing patterns of programs, and it is generally believed that widely shared data occur infrequently and does not significantly affect performance. In practice, programmers are generally careful to avoid algorithms that exhibit wide sharing of data, because such algorithms are known to achieve poor performance on conventional shared-memory multiprocessors. Thus one explanation for the lack of widely shared data is the self-fulfilling prophecy: hardware isn't designed to support widely shared data well because it is rarely needed, and it is rarely needed because programmers know that hardware doesn't support it well.

In this paper we argue that in fact widely shared data is a more serious problem than previously recognized, and that furthermore, it is possible to provide support that actually gives an advantage to widely shared data. The idea of read-combining [6] evolved because of the concern for network contention for widely shared data. Unfortunately, read-combining is highly dynamic, and only 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.

The presence of redundant copies of a datum in multiple caches throughout the network offers the possibility that widely shared data could actually be better supported than non-widely shared data. If many nodes in the system have a particular datum needed by node X, it is likely that a copy of the data is closer than the copy

in node Y, the home node for that datum. If a system can exploit the redundancy of the data to improve accessibility of widely shared data, programmers would find that the best algorithms make extensive use of widely shared data rather than eschewing. 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.

Several classes of sharing patterns in shared-memory applications have been identified (migratory, read-only, frequently-written sharing, etc. [3,18]). Hardware protocols (e.g, pairwise sharing and QOLB [5] in SCI) or software protocols (Munin [3], Treadmarks [25]), or application specific protocols [21] have been devised to deal with such patterns effectively. Widely shared data is a distinct sharing pattern —previously classified within other patterns— that imposes increasingly significant overhead as systems increase in size [11]: 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* [16]; 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 many systems with no provision for efficient broadcast or multicasts these invalidations consume much network bandwidth, perhaps in a wasteful manner.

When widely shared data exist they are usually a very small percentage of the dataset of a program. Studies have also shown that the average degree of sharing (the number of nodes that simultaneously share the same data) in application programs is low [18]. These observations however, do not indicate the serious performance degradation resulting from accessing such data. Even if widely shared data are a negligible percentage of the dataset they can be detrimental to performance if they are accessed frequently enough. Furthermore, the average degree of sharing is only relevant for programs that *do not* have widely shared data and it is otherwise misleading. This is because the average degree of sharing does not reflect how many accesses actually correspond to widely shared data. This number can be surprisingly high even for low degrees of sharing as we illustrate in the following example.

In Figure 1 we show a histogram (left graph) of the write-runs [29] for the GAUSS program (discussed in Section 5) running on 128 nodes. The horizontal axis shows the size of the write-runs (i.e. the number of reads between writes). The vertical axis represents the number of times that a write-run appears in the execution of the program. In this graph widely shared data appear as write-runs of size 128 (the size of the machine). The average degree of sharing for this program (computed by taking the weighted average of write-runs) is a relatively low 2.75 (i.e., on *average* there are less than three copies of shared data). However, the number of reads (or alternatively the number of invalidates) that correspond to widely shared data is 128 times the number of times the large write-runs appeared in the execution. For the system we are examining, these reads (invalidates) are about *one half* of all the reads (invalidates) in the program (right graph of Figure 1). Providing for efficient handling of such data is thus essential for scalability in cases where such data exist.

Previously, scalable coherence protocols have been proposed [8][14][15] but they 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 [2]. The GLOW extensions for cache coherence protocols were also designed exclusively to handle widely shared data on top of another cache coherence protocol [11][12]. The distinguishing characteristic of the GLOW extensions is that they create sharing trees very well mapped on top of the network topology of the system, thus exploiting "geographical locality" [12]. Bennett et al also distinguished widely shared data in their work with proxies [27]. However, in all the aforementioned work widely shared data were statically identified by the user (the programmer or, potentially, the compiler). Such *static methods* of identifying widely shared data have three major drawbacks: i) user involvement complicates the clean shared-memory paradigm, ii) it may not always be possible to identify the widely shared data statically, and most importantly iii) mechanisms are required to transfer information from the user to the hardware; these mechanisms are hard to implement when the parallel system is built with commodity parts. This last consideration is crucial since vendors must leverage existing commodity parts (e.g., processors, main-boards, and networks [7]) in order to drive development costs down and shorten the time-to-market.

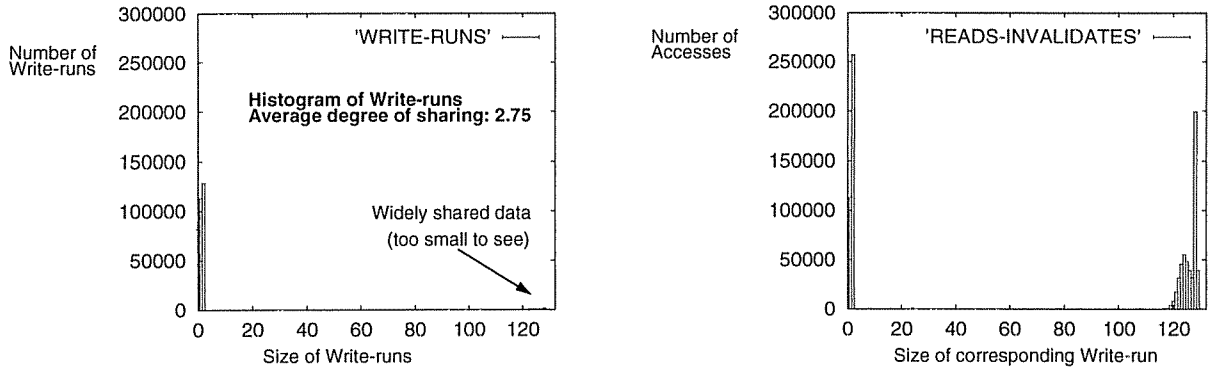

FIGURE 1. Write-run histogram and corresponding Reads/Invalidations for GAUSS **running on a 128 processor system**

Because of these reasons, we introduce two dynamic schemes to detect widely shared data that differ in where the detection takes place:

- **Agent detection**: In this scheme the request stream is observed in the network, at the exact places where the GLOW extensions are implemented (namely at GLOW agents that are switch nodes in the network topology). Changes are required only in the GLOW-specific hardware without affecting other parts of the system. Requests for widely shared data can be identified in the request stream if their addresses are seen often enough. The GLOW extensions are then invoked for such requests as in the static GLOW. This technique is similar in spirit to combining [6], but can better exploit requests scattered in time because the critical information hangs around in the combining node after a request is gone. Simulations show that the performance of this dynamic scheme closely tracks that of the static scheme. We also provide evidence that it is more stable than ordinary combining which is highly dependent on network timing characteristics and application characteristics.
- **Directory detection**: In this scheme the directory is responsible for identifying widely shared data. This scheme is similar to the behavior of limited pointer directories such as Dir_iB [30]. These directories switch from point-to-point messaging to broadcasting if the number or readers exceeds a threshold. Similarly in our scheme, the directory detects widely shared data (by keeping track of the number of readers) but it then informs the nodes in the system about the nature of the data. Subsequently, nodes use special requests for such data that are intercepted by the GLOW agents. This scheme depends on widely shared data remaining as such through multiple read-write cycles.

The rest of this paper is organized as follows: in Section 2 we describe the GLOW extensions that handle the widely shared data. In Section 3 we expand on the static methods of identifying widely shared data and their problems. We introduce the dynamic methods in Section 4. In Section 5 and Section 6 we present our evaluation and results. Finally, we conclude in Section 7.

2 GLOW extensions

The GLOW extensions provide support for widely shared data. They are independent of how the widely shared data are exposed to the hardware. For purposes of discussing the extensions we assume that special requests are used to access widely shared data. In subsequent sections we describe how to generate such special requests either statically or dynamically. GLOW extensions improve on previous efforts (EC [2], STP [15], STEM [8]) by embodying the following four characteristics:

- TRANSPARENCY. GLOW is not a protocol itself but rather a method of converting other protocols to handle widely shared data. The functionality of the GLOW extensions is implemented in selected network switch nodes called GLOW *agents* that intercept special requests for widely shared data. These nodes behave both as memory and cache nodes using the underlying cache coherence protocol recursively: toward a local cluster of nodes they service, GLOW agents impersonate remote memory nodes; toward the home node directory, agents behave as if they were ordinary caches.

- GEOGRAPHICAL LOCALITY. A sharing tree out of the GLOW 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. GLOW captures *geographical locality* so that neighboring nodes in the sharing tree are in physical proximity.
- SCALABLE READS. Since the GLOW agents intercept multiple requests for a cache line and generate only a new request toward the home node, a similar effect to read-combining is achieved, eliminating hot spots [16].
- SCALABLE WRITES. Upon a write, GLOW invokes in parallel the underlying protocol's invalidation or update mechanisms: on receipt of an invalidation (update) message, an agent starts recursively the invalidation (update) process on the other agents or nodes it services. The parallel invalidation (update), coupled with the geographical locality of the tree permits fast, scalable writes that require low bandwidth.

2.1 GLOW extensions to SCI

The first implementation of GLOW [12] is done on top of SCALABLE COHERENT INTERFACE[1] (SCI) [7]. A version of this implementation (described in [11]) defines the functionality of network switch nodes and it is fully compatible with current SCI systems.

SCI has two characteristics that make it an ideal match for GLOW. The first is that its invalidation algorithm is serial and making a tree protocol especially attractive for speeding up writes to widely shared data. The second concerns SCI topologies. SCI defines a ring interconnect as a basic building block for larger topologies. GLOW extensions can be implemented on top of a wide range of topologies constructed of SCI rings, including hypercubes, meshes, trees, butterflies[9] and many others. GLOW can also be used in irregular topologies (*e.g.*, an irregular network of workstations). In this paper, we study GLOW on highly scalable K-ary N-cube topologies [23] constructed of rings. As we mentioned in the general description, all GLOW protocol processing takes place in strategically selected switch nodes (the GLOW agents) that connect two or more SCI rings in the network topology.

GLOW agents cache directory information; caching the actual data is optional. Multilevel inclusion [1] is not enforced to avoid protocol deadlocks in arbitrary topologies. This allows great flexibility since the involvement of the GLOW agents is not necessary for correctness: it is at the discretion of the agent whether it will intercept a request or not. In the following two subsections we describe in more detail how the sharing trees are created and invalidated.

2.1.1 Creation of GLOW trees

A GLOW sharing tree is created when SCI lists form under the agent. An agent can connect multiple rings and it can accommodate (for a single cache line) multiple SCI lists, one per ring. These lists are called *child lists* and the agent is their *parent.* The child lists contain nodes whose requests are intercepted and satisfied by the agent pretending to be the remote memory locally on the ring.

Intercept of a request for widely shared data results in a lookup in the agent's directory storage. If the lookup results in a miss, the agent sends its own request for the widely shared data toward the home node. The agent inserts the requesting node in a child list and instructs it to wait for the data. As soon as the agent gets a copy of the cache line it will pass it to its child lists. If the lookup results in a hit, the requesting node is instructed to attach to the appropriate child list. The requester will get the data from either the agent (if it caches data) or the previous head of the child list. If the appropriate child list is empty and the agent does not cache the data it fetches the data from one of its non-empty child lists.

2.1.2 Invalidation of GLOW trees

Similarly to SCI, a node must be the root of a sharing tree (connected directly to the memory directory) to write a cache line. A root node writing a cache line invalidates the highest level list sending invalidation messages serially to all the nodes in that list (standard SCI invalidation protocol). Upon receiving an invalidation, an SCI node invalidates itself and returns to the writer the identity of next node in the list. However, a GLOW agent concurrently forwards the invalidation to its *downstream* (i.e., away from memory) neighbor and

[1] The ANSI/IEEE standard 1596 Scalable Coherent Interface represents a robust hardware solution to the challenge of building cache-coherent, shared-memory multiprocessor systems. It defines a network interface, a basic ring network, and a cache coherence protocol. SCI defines a distributed, directory-based cache coherence protocol. Unlike most other directory-based protocols (such as DASH [13]) that keep all the directory information in memory, SCI distributes the directory information to the sharing nodes in a doubly-linked sharing list. The sharing list is stored with the cache lines throughout the system.

invalidates its child lists appearing as a writer attached in front of them. When the agent is done invalidating its child lists it waits until it becomes tail in its list. This will happen because it will either invalidate all its downstream nodes (if they are SCI nodes) or they will delete themselves (if they are GLOW agents). When the agent finds itself childless and tail in its list, it deletes itself from the tree, freeing in turn *upstream* (i.e., toward memory) nodes to delete.

3 Statically identifying and exposing widely shared data

In the previous section we described the GLOW mechanisms to handle requests for widely shared data. These mechanisms are independent of how the widely shared data are distinguished from other data. Here, we describe the static methods to define the widely shared data (also discussed in [12]).

The main characteristic of the static methods is that the user identifies either the widely shared data or the code that accesses widely shared data. As of yet it has not been investigated whether this can be done automatically by a compiler. In cases where identification is difficult, profiling tools can possibly help.

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 and invoke the GLOW agents. We divide the static methods depending on whether the programmer identifies the actual data that are widely shared or the instructions that access such data. The following two subsections describe the two alternatives.

3.1 Identifying addresses of widely shared data

This is the simplest method to implement and we have used it for the evaluations in later sections. A possible implementation of this method uses *address tables*, structures that store arbitrary addresses (or segments) of widely shared data. The address tables can be implemented in the network interface or as part of the cache coherence hardware. In both cases the user must have access to these tables in order to define and "undefine" widely shared data. Implementing such structures, however, is not trivial because of problems relating to security, allocation to multiple competing process, and address translation. 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.

3.2 Identifying instructions that access widely shared data

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 WIDELY SHARED DATA requests. We have proposed 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. The main drawback of this scheme is that it requires external hardware close to the processor.
- 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.

3.3 Disadvantages of the static methods

All the static methods have three serious disadvantages:

1. Involvement of the programmer (and/or possibly the compiler) is required. This puts a certain burden on the user that contradicts our 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 or the instructions that access such data. Especially in cases when the nature of data changes frequently and unpredictably, the static approaches may be inadequate.
3. Implementation difficulties: Both alternatives (identifying addresses or identifying instructions) have serious implementation problems. 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.

4 Dynamically identifying widely shared data

Because of the problems of the static methods it is highly desirable to provide hardware support that is transparent from the user and the rest of the system. For this we need to dynamically detect widely shared data and in this section we describe two schemes to accomplish this. The first scheme relies exclusively on the GLOW agents for the detection while the second scheme relies on the memory directories.

4.1 Agent detection of widely shared data

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. The dynamic scheme described here is intended to perform such filtering.

Agents observe the request traffic and detect addresses that are repeatedly requested. Requests for such addresses are then intercepted in the same way as the special requests in the static methods. In an implementation of this scheme each agent, besides its ordinary message queues, keeps a small queue (possibly implemented as circular queue) of the last N read requests it has observed. The queue contains 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 it channels through its ports.

When a new request arrives at the agent, its address is compared to those previously stored in recent-addresses queue. If the address is found in the queue the request is immediately intercepted by the agent as a request for widely shared data[2]. Otherwise, the request is forwarded to its destination. In both cases its address is inserted in the queue. This method results in some lost opportunities: for example we do not intercept the first request of an address that it is later repeated in other requests. Also, if a stream of requests for the same address is diluted sufficiently by other intervening requests we fail to recognize it as a stream of widely shared data requests. This scheme might also be confused by a single node repeatedly making the same request frequently enough to appear more than once within the agent's observation window (this could happen in producer-consumer or pairwise sharing). A safeguard to protect against this is to avoid matching requests from the same node against each other.

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 fairly fast. Of course, the minimum latency through the switch will dictate the maximum size of the queue. For the switches we model in our simulations we expect that a size of up to 128 entries to be feasible.

When the agents observe the reference stream only when there is congestion (in other words when multiple requests are queued in the agent's message queues) our method defaults to read-combining as was proposed for the NYU Ultracomputer [6]. In this case, the observable requests are only the ones delayed in the message queues. The problem with such combining (that our method effectively solves) 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. In the Section 6 we show that we can effectively discover widely shared data using a sliding window whereas combining fails in most cases.

4.2 Directory detection of widely shared data

In this scheme the memory directory is responsible to discover widely shared data. In contrast to the previous scheme that is transparent to the rest of the system this scheme requires some modifications to the coherence 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 they can be upgraded (e.g., STiNG [19]).

The directory is a single point in the system that can observe the request stream for its data blocks. It is therefore in a position to distinguish widely shared data. In directories such as Dir_iX [30] the number of readers is readily available. However, in SCI where the directory keeps a single pointer to the head of the sharing list, the directory must count the number of reads between writes. A counter, associated with each data block, counts up for each read and it is reset with a write. Data

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

blocks for which the corresponding counter reaches some threshold value are deemed widely shared. In SCI this is a heuristic since 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_iX). However, this would be an expensive addition to the SCI directory and we do not examine it further. In our evaluations we extended the SCI directory tag with a small 2-bit saturating counter.

The first time a data block is widely accessed all the read requests reach the directory without any intervention from the GLOW agents. If the directory finds that the data are widely shared it notifies the nodes in the system so the next time they access the block they will use special requests that can be intercepted by the GLOW agents. This information 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. Only the nodes that participated in the first read will learn about the data block. This information is stored in each node in the invalidated cache tag which we call "hot tag."[3] If a node tries to access a "hot tag" it will send a request for widely shared data which will be handled by the GLOW agents.

This scheme is based on the premise that data blocks are widely shared for many read-write cycles. Since the opportunity to optimize the first read-write cycle is 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.

A further consideration about this scheme is that it is easier to adapt from non-widely shared to widely shared than 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 not widely shared anymore or because the GLOW extensions absorb most of the requests in the network. Even if the directory recognized a transition to a non-widely shared state, it would have to notify again the nodes in the system about this change. Fortunately, the "hot tag" concept provides a natural way to adapt from widely shared to non-widely shared. If the data block is not widely accessed the "hot tags" around the system will be replaced —they are invalid tags after all— and the nodes will lose the information that the block was widely shared[4].

5 Experimental evaluation

A detailed study of the methods we propose requires execution driven simulation because of the complex interactions between the protocols and the network. The Wisconsin Wind Tunnel [17] is a well-established tool for evaluating large-scale parallel systems through the use of massive, detailed simulation. 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 takes the appropriate actions defined by the simulated protocol. The WWT keeps track of virtual time in processor cycles. SCI has previously been simulated extensively under WWT [10] and the GLOW extensions have been applied to this simulation environment. We simulated systems that resemble SCI systems made of readily available components such as SCI rings and workstation nodes.

We have simulated K-ary N-cube systems from 16 to 128 nodes in two and three dimensions. The nodes comprise a processor, an SCI cache, memory, memory directory, a GLOW agent, and a number of ring interfaces. Although we 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 to service the processors inside the node. The processors run at 400MHz 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 line size of 64 bytes. The cache size of 64K is intentionally small to reflect the size of our benchmarks. Processor, memory and network interface (including

[3] Alternatively, the information about which data blocks have been found to be widely shared can be kept in address tables similar to those described for the static GLOW. However, address tables would make the whole scheme more difficult to implement and are not examined further.

[4] The only pathological case that can result from the inability of this scheme to adapt quickly from widely shared to non-widely shared is when the data block becomes migratory after it was widely shared. In this case many subsequent reads will incur the overhead of widely shared data because it will take many read-write cycles to erase the information about the nature of the data block from all the nodes.

GLOW agents) communicate through a 133 MHz 64-bit bus. The SCI K-ary N-cube network of rings uses a 400 MHz clock; 16 bits of data can be transferred every clock cycle through every link. We simulate contention throughout the network but messages are never dropped since we assume infinite queues. Each GLOW agent is equipped with a 1024-entry directory cache and 64K of data storage. In order to minimize conflicts the agent's directory it is organized as a 4-way set-associative cache.

To evaluate the performance of GLOW we used five benchmark programs: GAUSS SPARSE, All Pairs Shortest Path, Transitive Closure, and Conjugate Gradient (CG). Although these programs are not in any way representative of a real workload, they serve to show that GLOW can offer improved performance. Additionally, these programs represent the core of many scientific applications used for research in many engineering and scientific disciplines. We did not consider programs without widely shared data because such programs would hardly activate the GLOW extensions.

The GAUSS program solves a linear system of equations using the well known method of Gaussian elimination. Details of the shared-memory program can be found in [12]. A coefficient matrix $N \times N$ is filled with random numbers and then the linear system is solved using a known vector (N is 512 for our simulations). In every iteration of the algorithm a *pivot* row is chosen and read by all processors while elements of previous pivot rows are updated. Potentially every row of the coefficient matrix can be widely shared. For the static methods, we define a pivot row as widely shared data for the duration of the corresponding iteration.

The SPARSE program solves $\boldsymbol{AX=B}$ where $\boldsymbol{A}$ and $\boldsymbol{B}$ are matrices ($\boldsymbol{A}$ being a sparse matrix) and $\boldsymbol{X}$ is a vector. The main data structures in the SPARSE program are $\boldsymbol{A}$, the $N \times N$ sparse matrix and $\boldsymbol{X}$, the vector that is widely shared (N is 512 for our simulations). In the static methods we define vector $\boldsymbol{X}$ as widely shared data.

The All Pair Shortest Path (APSP) and the Transitive Closure (TC) programs solve classical graph problems. For both programs we used dynamic-programming formulations, that are special cases of the Floyd-Warshall [4] algorithm. In the APSP, an N vertex graph is represented by an $N \times N$ adjacency matrix. The input graph used for the simulations is a 256 vertex dense graph (most of the vertices are connected). In the TC program an $N \times N$ matrix represents the connectivity of the graph with ones and zeroes. The input is a 256 vertex graph with a 50% chance of two vertices being connected. For both programs and for the static methods the whole main matrix is defined as widely shared data.

In the CG program the conjugate gradient method is applied to solve a Poison equation. The code is structured similarly to the NAS CG benchmark [31]. CG is computationally intensive and we were only able to simulate it with a small input matrix of 64x64. Because of the small dataset we did not examine CG in systems with more than 64 nodes.

6 Results

In this section we present simulation results for the five programs and for the various system configurations (2-dimensional and 3-dimensional networks, 16 to 128 nodes). We use the 3 dimensional topologies to show how network scalability affects the GLOW extensions. In general our results show that GLOW offers greater performance advantage with higher dimensionality networks because it can create shorter trees with larger fan-out.

We compare SCI, static GLOW, and three versions of the dynamic GLOW. The first version of the dynamic GLOW observes only requests delayed in the message queues because of congestion and it is therefore equivalent to read-combining (we simply refer to this as COMBINING). The second version employs a 128-entry recent-addresses queue to discover repetition in the addresses (we refer to this as REACTIVE AGENTS). In the third version the directory discovers the widely shared data and we refer to this as REACTIVE DIRECTORIES.

We measure execution time, and for each program we present speedup normalized to a base case. We selected the base case to be SCI on the appropriate number of nodes and with the appropriate 2- or 3-dimensional network. The actual speedups over a single node for the base cases are shown in Table 1. Note here that the programs we are using do not scale beyond 32 or 64 nodes for SCI. The GLOW extensions allow these programs to scale to 128 nodes but the performance difference between 64 and 128 nodes is small. A limitation of our simulation methodology is that we keep the input size of the programs constant and —because of practical constraints— relatively small. With larger datasets these programs scale to more nodes for the base SCI case and the GLOW extensions yield performance improvements for even larger numbers of nodes.

	Nodes	GAUSS	SPARSE	APSP	TC	CG
2-D	16	16.57	5.86	11.70	14.45	3.27
	32	25.34	8.57	19.36	20.10	3.26
	64	22.95	12.68	21.02	19.27	3.78
	128	12.92	12.53	14.74	13.20	—
3-D	16	16.77	6.09	11.77	14.51	3.31
	32	26.33	10.01	19.88	20.60	3.87
	64	25.27	15.73	21.93	20.14	4.66
	128	16.18	18.54	15.94	14.28	—

Table 1: Actual speedups of the base cases (SCI on 16 nodes)

Figure 2 shows the normalized speedups for the GAUSS program. The two graphs present results for the 2- and 3-dimensional networks. GAUSS on SCI does not scale beyond 32 nodes, showing serious performance degradation with higher numbers of nodes. The GLOW extensions, however, scale to 64 nodes in 2 dimensions and to 128 nodes in 3 dimensions (although the additional speedup is negligible).

Static GLOW outperforms the other alternatives and is up to 2.22 times faster than SCI in 2 dimensions and up to 2.44 times faster in 3 dimensions. COMBINING reaches about halfway the performance improvement of static GLOW while the REACTIVE AGENTS remain within 5% of the performance of static GLOW. The REACTIVE DIRECTORIES scheme also works well staying within 10% of the static GLOW.

To explain the behavior of the various GLOW schemes we examine how they appear to change the write-runs of the program from the directories' point of view. Specifically, for each scheme we plot the reads that correspond to write-runs of different sizes. The GLOW schemes "compress" these accesses toward the small write-run sizes. Each scheme's "compression" relates to its performance improvement.

Figure 3 plots the number of reads that correspond to write-runs ranging in size from 1 to 128. The data are for GAUSS on 128 nodes on a 2 dimensional network. These data correspond to what the SCI directories observe and as such they are not an entirely accurate picture of the write-runs of the program. The difference is that SCI directories count the same node multiple times in the same write-run if —because of replacements— it sent multiple requests. Because these graphs contain very large and very small numbers we use a logarithmic scale in the vertical axis. This tends to emphasize the small numbers that would otherwise be invisible as in Figure 1.

The graph for SCI shows a large number of accesses corresponding to large write-runs. COMBINING, REACTIVE AGENTS, static GLOW, and REACTIVE DIRECTORIES 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 write-runs of size 9. 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, manages to compress the write-runs from a size of 128 down to a size of 40. REACTIVE AGENTS manage to compress most of the large write-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. REACTIVE DIRECTORIES eliminate the largest write-runs and convert them to write-runs of size 9 (similarly to the static GLOW). However, they still leave a significant number of accesses corresponding to large write-runs unaffected.

SPARSE 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 this program the REACTIVE AGENTS *outperform* the 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 and REACTIVE AGENTS can handle them at run-time. REACTIVE AGENTS perform 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 REACTIVE DIRECTORIES perform on a par with static GLOW.

Figure 5 shows the compression of write-runs for SPARSE (again on 128 nodes with a 2 dimensional net-

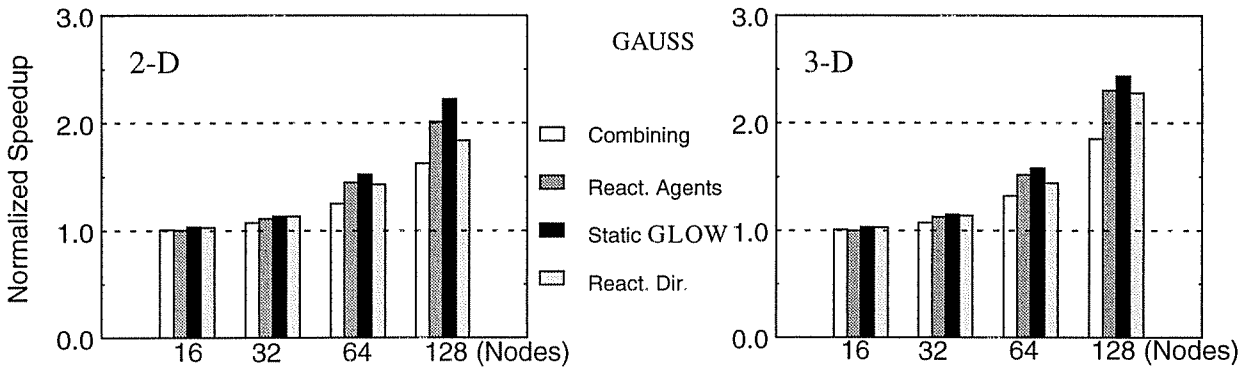

FIGURE 2. Normalized speedup (over SCI) for GAUSS in 2 and 3 dimensions (16 to 128 nodes).

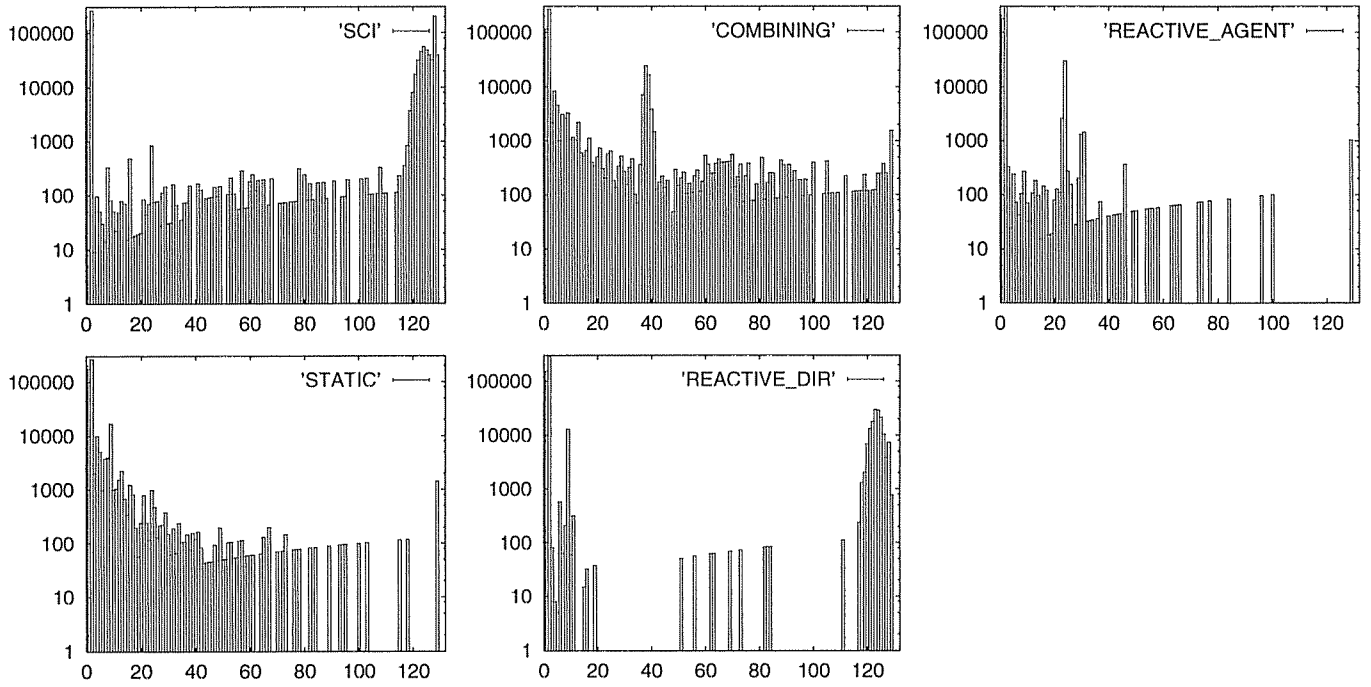

FIGURE 3. Write-run compression for GAUSS (128 nodes 2-dimensions). Accesses corresponding to large write-runs are shifted toward smaller write-runs using GLOW extensions.

work). COMBINING does not perform very well for SPARSE and this is also evident in its failure to affect the large write-runs. REACTIVE AGENTS spread the largest write-runs all over the write-run spectrum. This means that the number of requests that slip through the agents before they detect widely shared data has significant variance. Static GLOW again performs very well, most of the time allowing the directories to see only 9 requests (8 GLOW agents and the local node).

APSP and TC show similar behavior. With SCI APSP does not scale beyond 64 nodes and TC does not scale beyond 32 nodes. For APSP, static GLOW is 2.20 times faster than SCI in 2 dimensions and 2.59 times faster in 3 dimensions. Similarly, for TC static GLOW is 2.22 and 2.64 times faster than SCI for 2 and 3 dimensions respectively. For both programs COMBINING and REACTIVE DIRECTORIES fail to show any performance improvement while REACTIVE AGENTS perform closely to the static GLOW.

APSP and TC exhibit similar behavior so we only demonstrate write-run compression for APSP (Figure

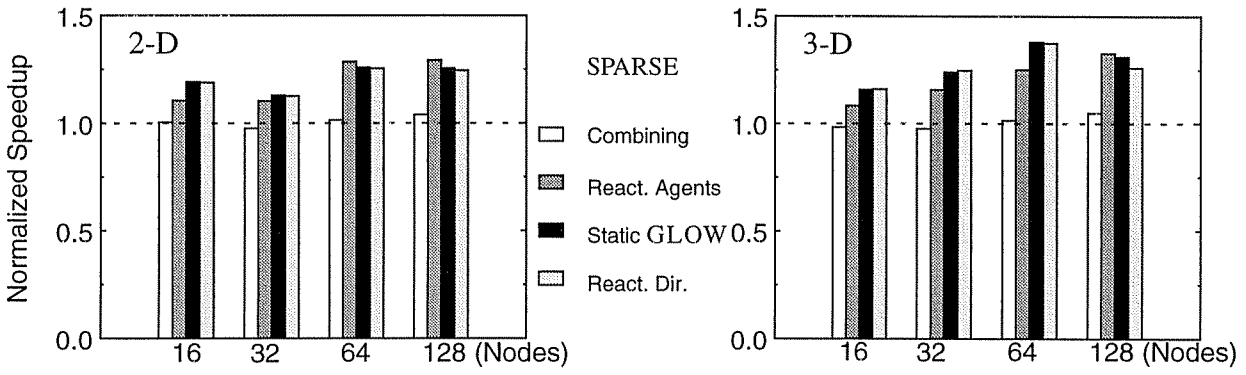

FIGURE 4. Normalized speedup (over SCI) for SPARSE in 2 and 3 dimensions (16 to 128 nodes).

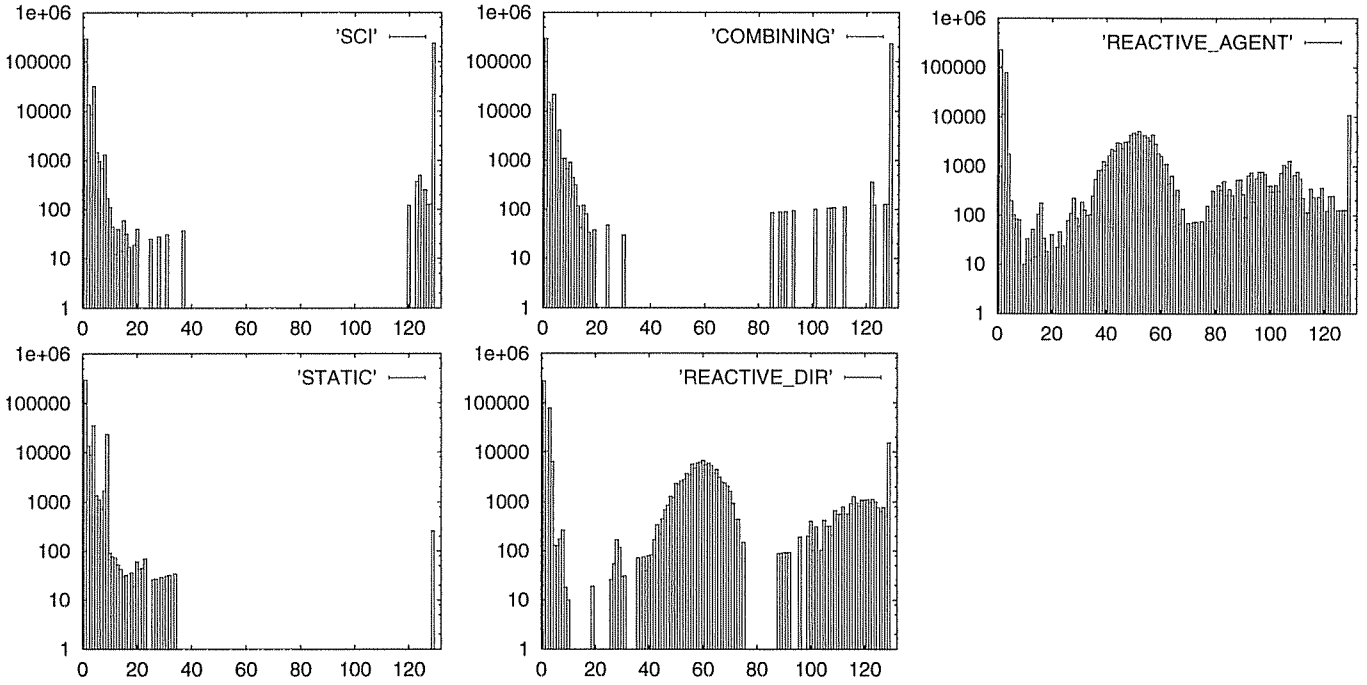

FIGURE 5. Compression of write-runs for SPARSE (128 nodes, 2 dimensions). Accesses corresponding to large write-runs are shifted toward smaller write-runs using GLOW extensions.

8). A significant percentage of the reads of the program correspond to large write-runs. As expected COMBINING is not successful in hiding accesses from the directories. Although it shifts accesses to smaller write-runs, it is not enough to make a difference in performance. REACTIVE AGENTS are quite successful compressing the write-runs 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 write-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 REACTIVE DIRECTORIES fail to change the write-runs of the program.

The behavior of the CG program is somewhat peculiar, mainly because we run it with a very small dataset. GLOW extensions exhibit the largest speedups over SCI in 16 nodes (1.9 and 2 times faster than SCI in 2 and 3 dimensions). REACTIVE DIRECTORIES perform very well and actually take the performance lead from the

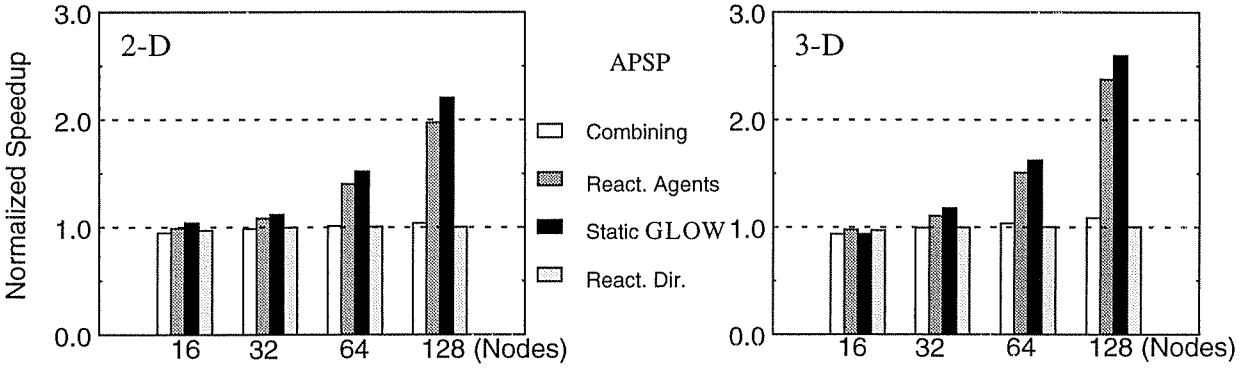

FIGURE 6. Normalized speedup (over SCI) for APSP for 2 and 3 dimensions (16 to 128 nodes).

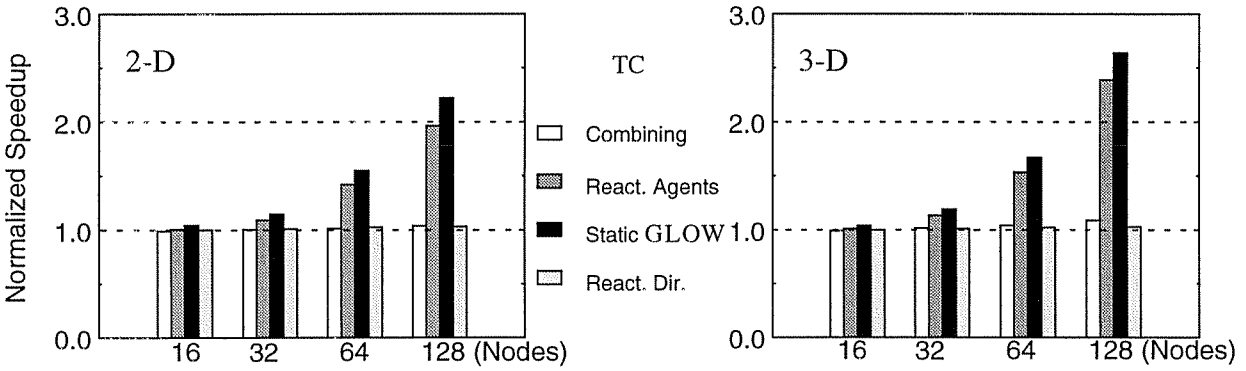

FIGURE 7. Normalized speedup (over SCI) for TC in 2 and 3 dimensions (16 to 128 nodes)

static GLOW in 32 and 64 nodes in 2 dimensions. COMBINING does not provide any performance improvement while REACTIVE AGENTS are competitive (within 25% of the performance of the static GLOW)

Figure 10 shows the write-run compression for CG: COMBINING does not affect accesses corresponding to the largest write-runs; REACTIVE AGENTS shift many accesses to write-runs of size 23 but still leave some accesses corresponding to the large write-runs unaffected; static GLOW again eliminates large write-runs leaving only those of size 9 and REACTIVE DIRECTORIES are also very successful shifting many accesses to write-runs of size 9.

To summarize the results: REACTIVE AGENTS consistently track the performance of the static GLOW while COMBINING only works for one program (GAUSS). The results show that COMBINING is indeed sensitive to the congestion characteristics of the application. From limited experimentation we have indications that the behavior of COMBINING also changes depending on the network characteristics (e.g., link and switch latency, bandwidth) while the behavior of the REACTIVE AGENTS with regard to the number of intercepted requests remains largely unaffected. REACTIVE DIRECTORIES give mixed results working only for three of the five programs.

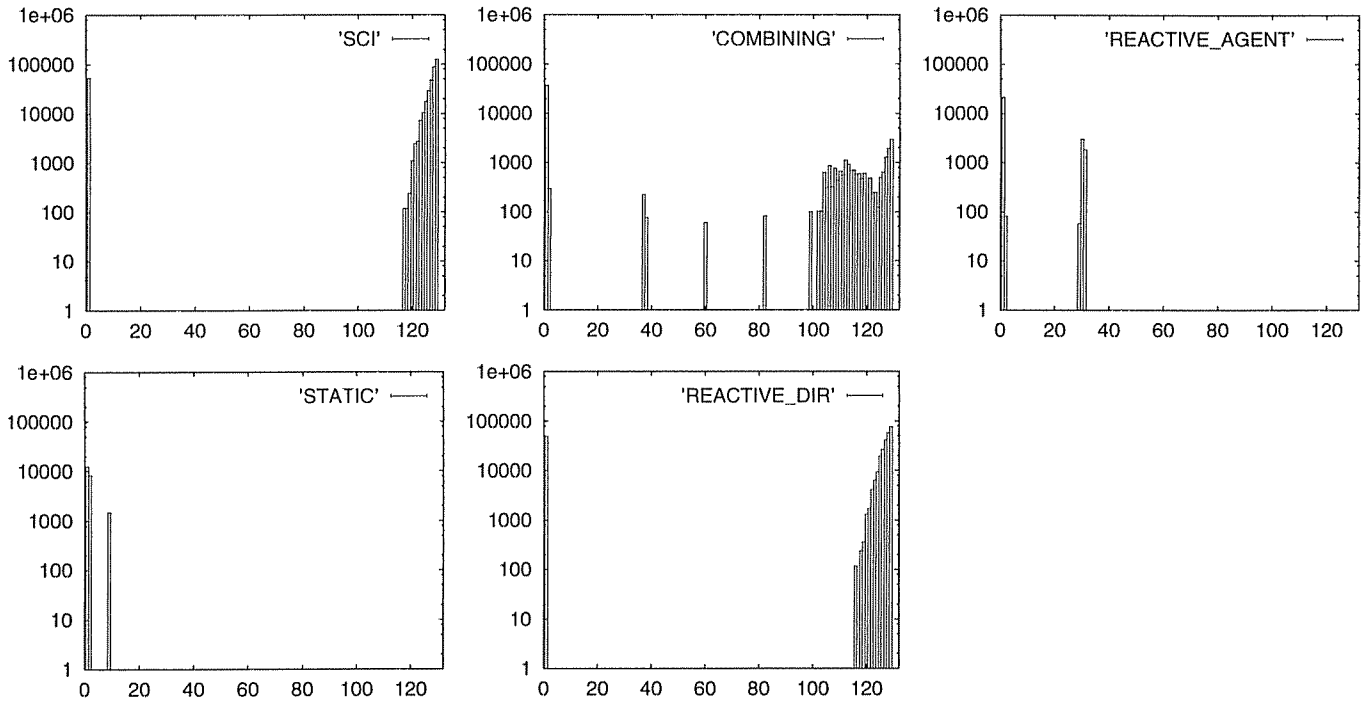

FIGURE 8. Write-run compression for APSP (128 nodes, 2 dimensions). Accesses corresponding to large write-runs are shifted toward smaller write-runs using GLOW extensions.

6.1 Sensitivity of Agent detection to Window Size

An interesting result is that the REACTIVE AGENTS scheme is largely insensitive to the size of the recent-addresses queue especially for the larger systems for the five programs we examined. In Figure 11 we show the performance of the GAUSS and APSP programs for four different sizes of the recent-addresses queue: 8, 32, 128 and 256. The other three programs exhibit similar behavior. It is evident in Figure 11 that the window size does not seriously affect the performance of the REACTIVE AGENTS. In fact for GAUSS the smallest windows of size 8 perform 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

7 Conclusions

In this paper we argued that accesses that correspond to widely shared data can be excessively many even if the widely shared data in programs are not. 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. The GLOW extensions to cache coherence protocols are designed with transparency in mind: they are implemented in the network domain, outside commodity workstation boxes, and they 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 writes. However, in their static form they require the user to define the widely shared data and issue special requests that can be intercepted by GLOW agents. This is undesirable for various reasons including implementation difficulties that inhibit transparency. In this paper we propose and study two schemes that can detect widely shared data at run-time and obtain performance comparable to the static version of GLOW.

The first scheme 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.

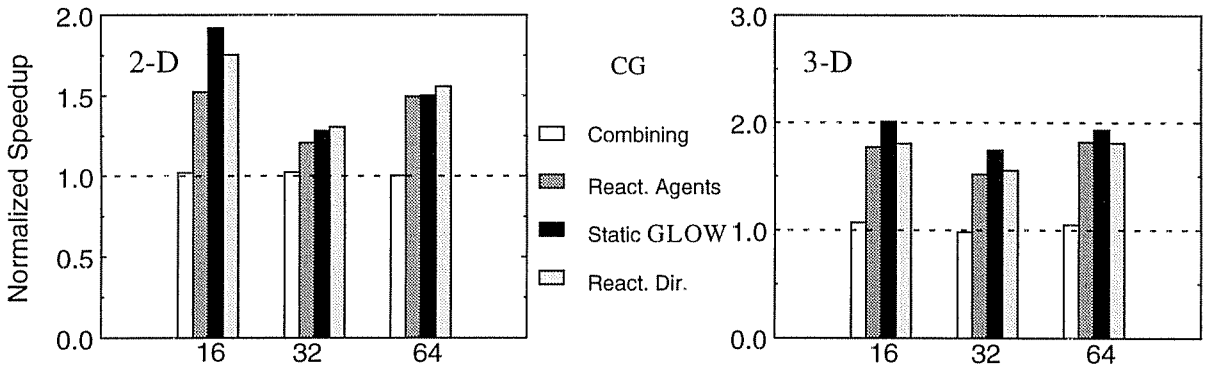

FIGURE 9. Normalized speedup (over SCI) for CG in 2 and 3 dimensions (16 to 64 nodes)

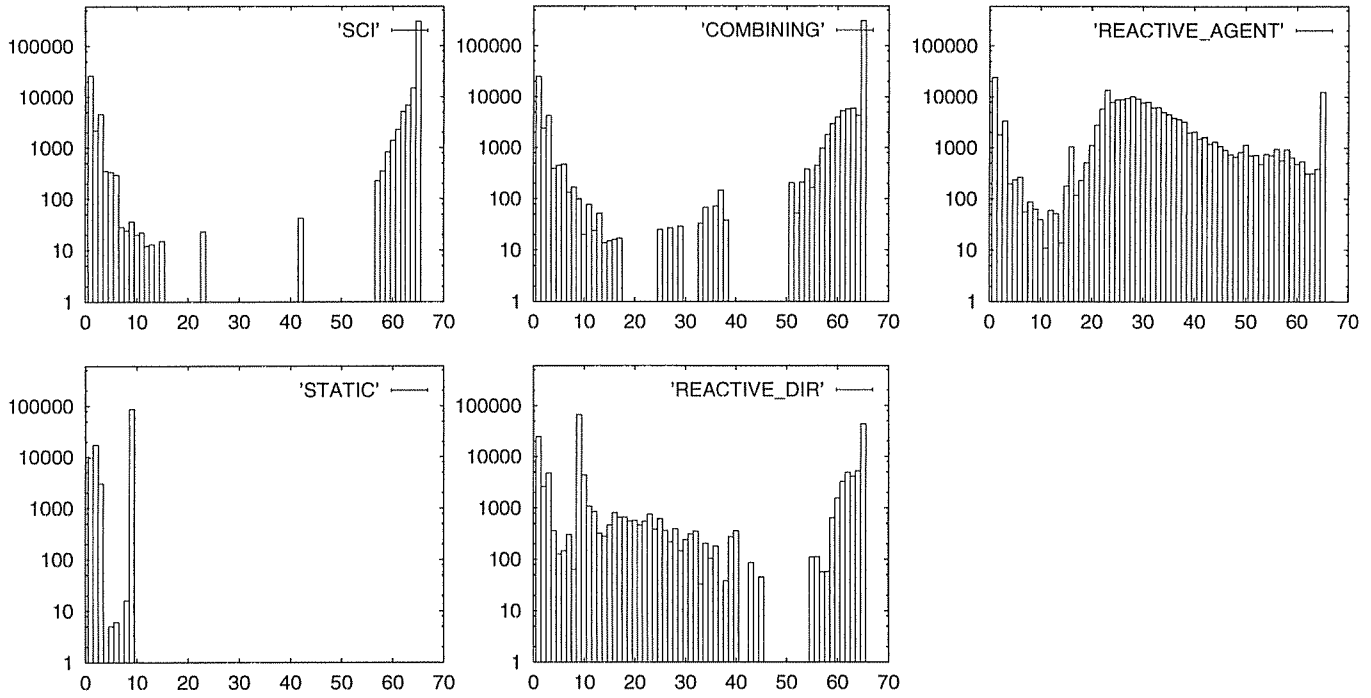

FIGURE 10. Write-run compression for CG (64 nodes, 2 dimensions). Accesses corresponding to large write-runs are shifted toward smaller write-runs using GLOW extensions

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.

In the second scheme the memory directories 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 data. This scheme works well when data blocks are widely accessed more than once.

Finally, we have used the changes of the write-runs of a program as seen by the memory directories to explain the behavior of the different schemes.

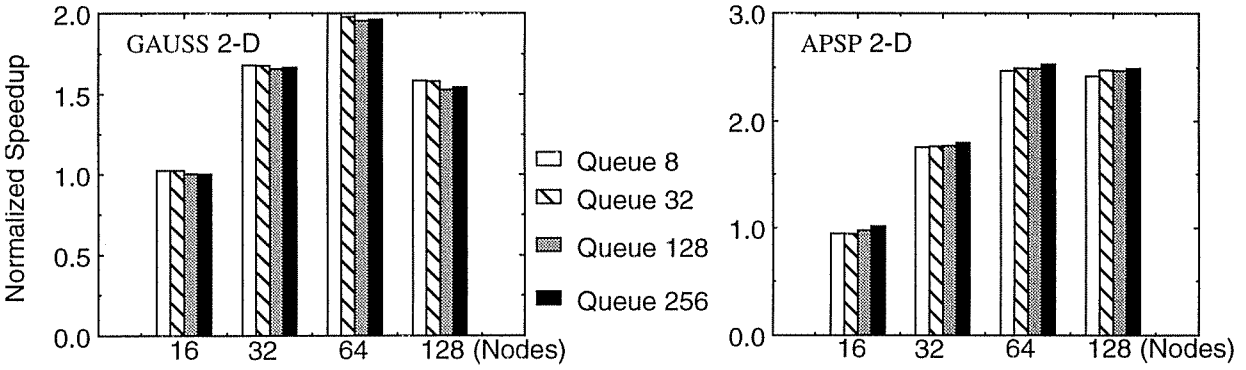

FIGURE 11. Sensitivity analysis for the size of the recent-addresses queue (speedup of the dynamic GLOW **with respect to** SCI **on 16 nodes).**

8 References

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

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

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

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

[5] J.R. Goodman, Mary K. Vernon, Philip J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache Coherent Multiprocessors." *Proc. of the 3rd Int. Conf. on Architectural support for Programming Languages and Operating Systems (ASPLOS-III)*, April 1989.

[6] 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.

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

[8] Ross E. Johnson, "Extending the Scalable Coherent Interface for Large-Scale Shared-Memory Multiprocessors." PhD Thesis, University of Wisconsin-Madison, 1993.

[9] Ross E. Johnson, James R.Goodman, "Interconnect Topologies with Point-to-Point Rings," *Proc. of the International Conference on Parallel Processing*, August 1992.

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

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

[12] S.Kaxiras and J. R. Goodman "The GLOW Cache Coherence Protocol Extensions for Widely Shared Data." *International Conference on Supercomputing*, May 1996.

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

[14] 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.

[15] 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.

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

[17] 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." *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pp. 48–60, May 1993.

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

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

[20] Convex Computer Corporation, "The Exemplar System," 1994.

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

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

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

[24] Ioannis Schoinas et al., "Implementing Fine-Grain Distributed Shared Memory On Commodity SMP Workstations", *Technical Report TR-1307*, University of Wisconsin-Madison, Mar. 1996

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

[26] James Laudon, Daniel Lenoski. "The SGI Origin: A cc-NUMA Highly Scalable Server," in Proceedings of the 24th International Symposium on Computer Architecture, June 1997.

[27] A. J. Bennett, P. H. J. Kelly, J. G. Refstrup, S. A. M. Talbot. "Using Proxies to Reduce Controller Contention in Large Shared-Memory Multiprocessors" EURO-PAR 96, Lyon, France, August 1996, pages 445-452, volume 1124 of Lecture Notes in Computer Science, Springer-Verlag.

[28] Sarah A. M. Talbot, Paul H. J. Kelly. "Reducing Controller Contention in Shared-Memory Multiprocessors Using Combining and Two-Phase Routing". Technical Report. Department of Computing, Imperial College of Science, Technology and Medicine, University of London.

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

[30] A. Agarwal, M. Horowitz and J. Hennessy, "An Evaluation of Directory Schemes for Cache Coherence." Proceedings of the 15th International Symposium on Computer Architecture, pp. 280-289, June 1988.

[31] David H. Bailey et al., "The NAS parallel benchmark: Summary and Preliminary Results." IEEE Supercomputing '91, pp 158-165. Nov., 1991.