Appears in Proc. of the Massively Parallel Computing Symposium, 1998.

Improving Request-Combining for Widely Shared Data in Shared-Memory Multiprocessors

Stefanos Kaxiras and James R. Goodman
{kaxiras,goodman}@cs.wisc.edu
University of Wisconsin-Madison
1210 W. Dayton St., Madison WI 53706, U.S.A.

Abstract

Widely shared data represent a serious threat to the scalability of shared-memory systems. The GLOW *extensions to cache coherence protocols were proposed to provide support for widely shared data. However, they required the user to identify the widely shared data and pass this information to the hardware. This approach is not appealing because: i) it burdens the user, 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, in this paper we propose a novel dynamic method to discover widely shared data at run-time. Our method is based on observing the request stream in network switch nodes and detect repetition of requested addresses. The switch nodes "remember" the most recent requests that passed through and detect whether the address of a new request has been seen recently. This method is a generalization on combining which restricts the observable requests to those that happen to queue simultaneously in the switch nodes. We show with detailed simulations that the dynamic scheme: i) tracks very closely the performance of the static* GLOW *and in some cases surpasses it, ii) it is considerably more robust than combining which is sensitive on network and application characteristics.*

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. However, larger non-bus-based shared-memory machines have been slow to emerge as the dominant type of parallel systems because of scalability constraints.

Because buses do not scale beyond a small number of processors (usually, up to 16), larger shared-memory machines (e.g., HP/Convex Exemplar [21], Sequent STiNG [20], SGI Origin 2000 [27]) are build by physically distributing the memory among a number of nodes connected with a network. To drive development costs down and shorten the time-to-market, distributed shared-memory systems leverage existing commodity parts such as processors, main-boards, and recently networks designed to support fine-grain communication [7]. At a high level the message-passing and shared-memory architectures share a common hardware platform. In fact, shared-memory implementations can exist purely as a software layer on top of message-passing hardware [25]. Hardware-based shared-memory, on the other hand, adds support for data coherence and support for explicit synchronization. Despite the similarities of the two architectures, shared-memory is susceptible to scalability problems for some shared-memory applications that have at least one of the following two characteristics: non-scalable sharing patterns and non-scalable explicit synchronization.

Both of these characteristics put undue pressure to the coherence and synchronization mechanisms (whether implemented in hardware or in software) that support shared-memory. In this paper we do not examine synchronization since it has been addressed in previous research. Scalable solutions to synchronization (e.g., MCS locks in software [23] and the QOLB synchronization primitive in hardware [24]) have been proposed. Instead, we concentrate on attacking the problem of the non-scalable sharing patterns.

Several classes of shared data in shared-memory

This work is supported by NSF Grant CCR-9509589, a Sun Microsystems equipment grant, and by the Intel Research Council.

applications have been identified (migratory, read-only, frequently-written, etc. [3,19]). Hardware protocols (e.g, pairwise sharing and QOLB [5] in SCI) or software protocols (Munin [3], Treadmarks [26]), or application specific protocols [22] have been devised to deal with such patterns effectively. However, widely shared data that are read by many (frequently all) processors in a system, represent the single most serious threat to the scalability of shared-memory.

Previously, scalable coherence protocols have been proposed [8][15][16] 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. Bianchini and LeBlanc distinguished widely shared data ("hot" data) from other data in their work [2]. Subsequently, we introduced the GLOW extensions for cache coherence protocols 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 [28]. 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 be always possible to statically identify the widely shared data, and most importantly iii) mechanisms are required to transfer information from the user to the hardware; these are hard to implement when the parallel system is built with commodity parts.

Because of these reasons, in this paper we introduce a dynamic scheme to identify widely shared data (a first step toward this direction was described by Tablot and Kelly [29] but their scheme does not distinguish widely shared data from other data). The main idea of our scheme is that the reference stream can be observed in the network, at the exact places where the GLOW extensions are implemented (namely at switch nodes in the network topology). Widely shared data can be identified in the reference stream and the GLOW extensions are then invoked as in the static methods. Our scheme is a generalization on combining (the well-known NYU Ultracomputer combining [6] can be thought of as a special case). With detailed simulations we show that the performance of the dynamic scheme closely tracks that of the static scheme. We also provide evidence that it is more reliable and stable than ordinary combining which is highly dependent on network timing characteristics and application characteristics.

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 method in Section 4 and in Section 5 and Section 6 we present our evaluation and results. Finally we conclude in Section 7.

2 GLOW extensions

GLOW is not a protocol itself but rather a method of converting other protocols to handle widely shared data. GLOW itself does not provide transactions for reading and writing shared blocks from scratch. Instead, since it works as an enhancement to another protocol, it borrows its mechanisms. The functionality of the GLOW extensions is implemented in selected network switch nodes called GLOW *agents*. These agents intercept read requests for widely shared data.

The idea behind GLOW is that these selected switch nodes in the topology behave both as memory and cache nodes. The GLOW agents can impersonate the remote memory, however far away it is, on a local cluster of nodes and thus satisfy their requests locally. Toward the home node directory, an agent behaves as if it were an ordinary cache sharing the data block.

A sharing tree can be 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 memory node where the widely shared data reside. GLOW captures *geographical locality* by mapping the sharing trees on top of the trees formed from the natural traffic patterns. 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 request combining is achieved, eliminating hot spots [17] and providing scalable reads. GLOW invokes in parallel the underlying protocol's invalidation or update mechanisms for the writes to widely shared data. On receipt of an invalidation (or update) message, an agent starts the invalidation (or update) process on the other agents or nodes it services. This parallel invalidation or update of the sharing tree permits fast, scalable writes.

2.1 GLOW extensions to SCI

The ANSI/IEEE standard 1596 Scalable Coherent Interface (SCI) [7] represents a 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. SCI defines a distributed, directory-based cache coherence protocol. Unlike most other directory-based protocols (such as DASH [14]) 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.

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

SCI has two characteristics that make GLOW an ideal match for it. The first is that its invalidation algorithm is serial and this makes a tree protocol especially welcome for speeding up writes to widely shared data. The second concerns SCI topologies. SCI defines a ring interconnect which is 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, butterfly topologies [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 K-ary N-cube topologies constructed of rings because they are highly scalable (see Figure 1).

As we mentioned in the general description, all GLOW protocol processing takes place in strategically selected *bridges* (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 in the discretion of the agent whether it will intercept a request or not. In the following three sections we describe in more detail sharing tree creation, invalidation and agent replacement.

Creation of GLOW trees. A GLOW sharing tree is created when SCI lists form under the agent (see Figure 1). An agent (bridge) 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. The agent links the requesting nodes in the small child lists. Without GLOW, these requests would go all the way to the remote memory and would join a global list.

Interception 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.

We depict a GLOW sharing tree on a 4-ary 3-cube system in Figure 1. The sharing caches at the bottom of the tree are represented by small rectangles while the GLOW agents occupying the higher levels of the tree are represented by triangles. Note that this is a perfectly formed GLOW tree. However, since we do not impose multilevel inclusion any combination of caches and agents is permitted at any level of the tree.

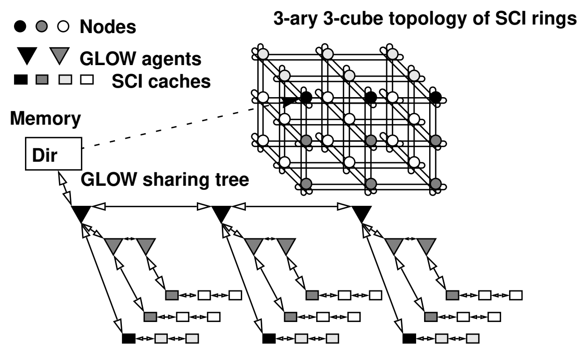

FIGURE 1. GLOW sharing tree on a *3*-ary *3*-cube

Invalidation of GLOW trees. A node must be the root of the sharing tree to write a cache line. As root of the sharing tree it starts invalidating 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, on receipt of an invalidation message a GLOW agent concurrently forwards the invalidation to its *downstream* (i.e., away from memory) neighbor and starts invalidating its child lists as if it were 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 themselves.

GLOW agent replacement. An ordinary SCI node deletes from the tree (rolls out) because of a replacement or as a prerequisite for writing the data. A deleting SCI node connected in some child-list informs its two neighbors as it leaves the list (standard SCI protocol). Since the GLOW agents are caches (albeit directory caches) there is the possibility of conflicts and replacements in their storage. In such cases, an agent with child lists deletes from the sharing tree by chaining all its child-lists into one long child list and substituting this child-list in its place in the sharing tree. This method permits the structure of long-lived trees to degrade gracefully. Alternatively, the subtree beneath the agent (all the child lists) could be invalidated. This method, however, results in higher deletion latencies and may affect active sharers.

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 in the source code or the code that accesses widely shared data. We have not yet 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 the requests for widely shared data can be tagged as such and in turn allow the GLOW agents to intercept them. 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 sections 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 hold arbitrary addresses (or address ranges) of widely shared data. If we constrain the widely shared data to specific pages (so all data within these pages are treated as widely shared data) we can simplify the address tables to page table -like structures.

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 "un-define" widely shared data. The implementations, however, are not trivial because of security problems and allocation to multiple competing process problems, as well as address translation problems. The address tables could be virtualized by the operating system, but this solution is also unsatisfactory since it requires operating system support and 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. The 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

To avoid the above problems we introduce a dynamic method to transparently detect widely shared data. In this paper we concentrate only on methods strictly confined to the network domain and specifically to the GLOW agents themselves. In this way, we change only the GLOW hardware we introduced (the GLOW agents) without affecting other commodity parts in the system.

The general idea of the dynamic method is that the GLOW 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 specially tagged requests in the static methods.

The GLOW agents are switch nodes in the network and their main responsibility is to channel traffic. To implement the dynamic method 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 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 (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[1]. 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 we do not intercept the first request for an address that 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.

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 around 128 entries to be entirely 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 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 attacks) 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 result section we show that we can effectively discover widely shared data using a sliding window whereas combining fails in most cases.

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 (WWT) [18] 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. The Scalable Coherent Interface 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. 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 64KB 4-way set-associative cache with a cache line size of 64 bytes. The cache size of 64KB is intentionally small to reflect the size of our benchmarks. Processor, memory and network interface (including 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. A discussion of the network model can be found elsewhere [13]. Each GLOW agent is equipped with a 1024-entry directory cache and 64K of

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

data storage. In order to minimize conflicts the agent's directory it is organized as a 4-way set-associative cache.

5.1 Benchmarks

To evaluate the performance of GLOW we used four benchmark programs: GAUSS, SPARSE, All Pairs Shortest Path, and Transitive Closure. Although these programs are not in any way representative of a real workload, they serve to show that GLOW can offer improved performance. We did not consider programs without widely shared data because such programs would never 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 Pairs 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.

6 Results

In this section we present simulation results for the four programs and for the various system configurations (2-dimensional and 3-dimensional networks, 16 to 128 nodes). We compare SCI, static GLOW, and two versions of the dynamic GLOW. The first version of the dynamic GLOW observes only requests delayed in the packet queues because of congestion and it is therefore equivalent to 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 dynamic GLOW).

We measure execution time, and for each program we present speedups normalized to a base case. We selected the base case to be SCI on 16 nodes (with the appropriate 2- or 3-dimensional network). The actual speedups over a single node for the base cases are shown in Table 1.

	GAUSS	SPARSE	APSP	TC
2-D	16.57	5.86	11.70	14.45
3-D	16.77	6.09	11.77	14.51

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). A limitation of our simulation methodology is that we keep the input size constant but with larger data sets GAUSS could scale to more processors. GLOW does not show performance improvement over SCI for less than 16 nodes because widely shared data only start becoming detrimental to scalability for larger system sizes. This is also true for the rest of the programs.

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 (speedups over SCI are shown in Table 2). Combining reaches about half the performance improvement of static GLOW while dynamic GLOW employing the recent-addresses queue remains within 5% of the performance of static GLOW.

SPARSE (Figure 3) 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 dynamic GLOW with recent-addresses queues *outperforms* 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 $\boldsymbol{X}$ described previously and the dynamic scheme can handle them effortlessly and better than static GLOW. The dynamic scheme performs up to 1.29 times faster than SCI in 2 dimensions and up to 1.33 times faster in 3 dimensions. However, combining fails to provide any significant performance improvement.

APSP (Figure 4) and TC (Figure 5) show similar

		GAUSS			SPARSE			APSP			TC		
	Nodes	**C**	**D**	**S**	**C**	**D**	**S**	**C**	**D**	**S**	**C**	**D**	**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 2: Speedup using Combining (C), Dynamic GLOW (D), and Static GLOW (S) over SCI on 16 to 128 nodes, 2 and 3 dimensions.

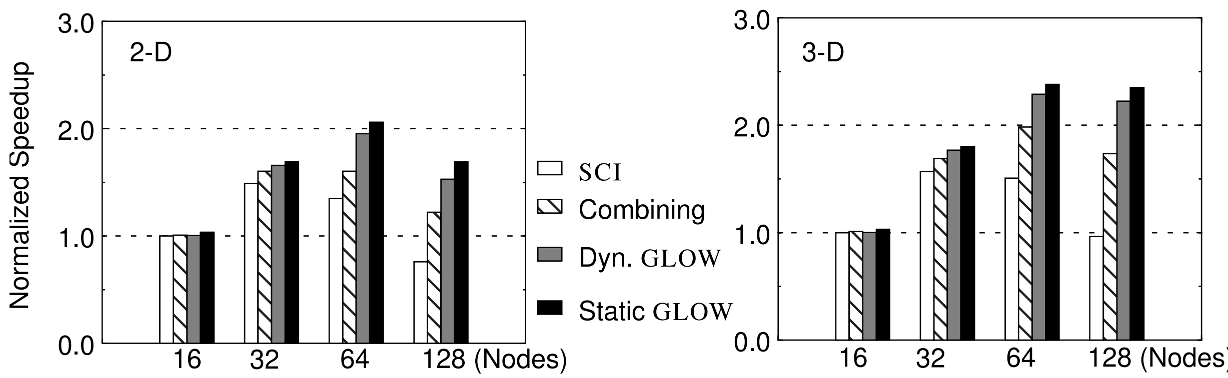

FIGURE 2. Speedup for GAUSS (with respect to SCI on 16 nodes) in 2 and 3 dimensions (16 to 128 nodes).

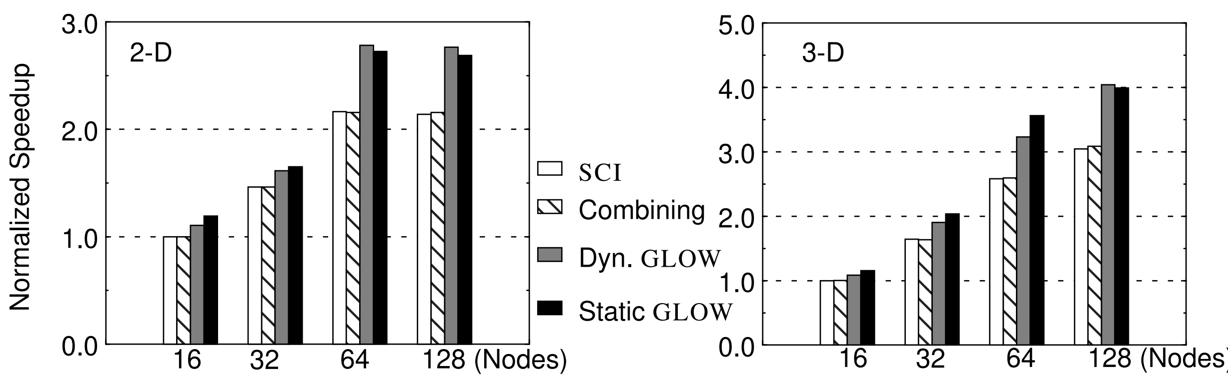

FIGURE 3. Speedup for SPARSE (with respect to SCI on 16 nodes) in 2 and 3 dimensions (16 to 128 nodes).

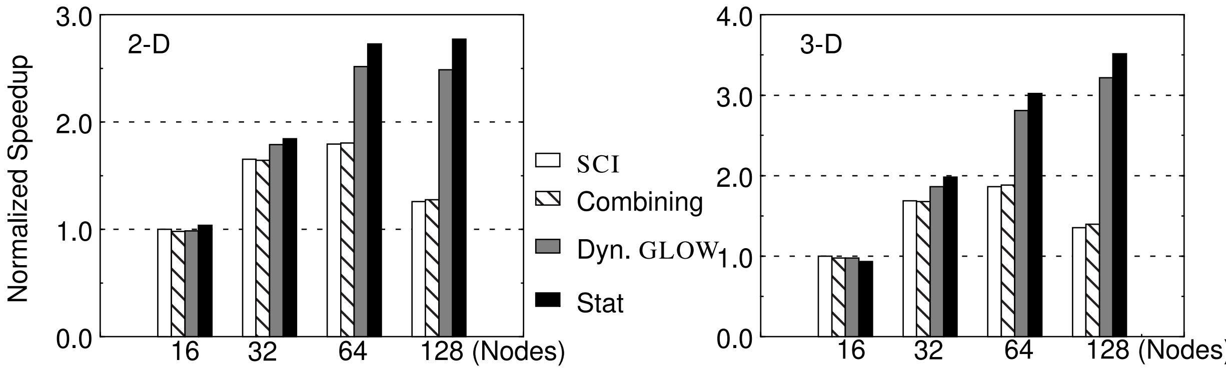

FIGURE 4. Speedup for APSP (with respect to SCI on 16 nodes) in 2 and 3 dimensions (16 to 128 nodes).

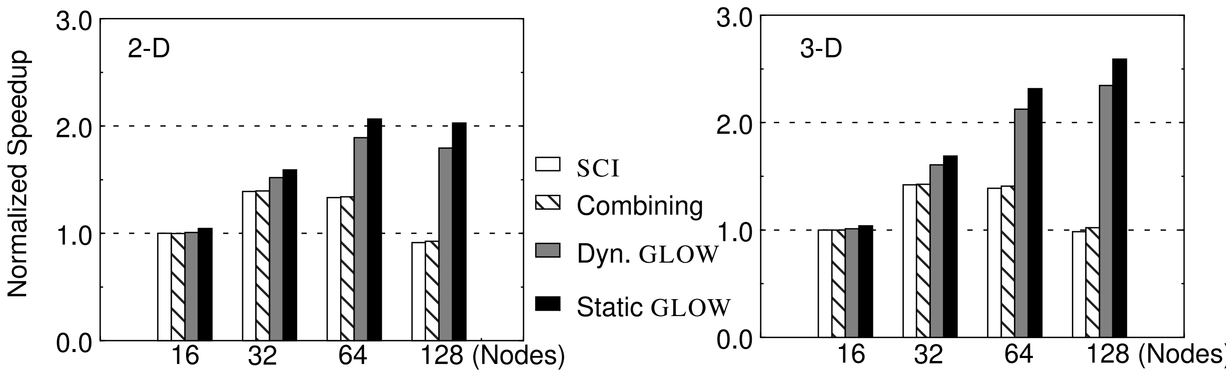

FIGURE 5. Speedup for TC (with respect to SCI on 16 nodes) in 2 and 3 dimensions (16 to 128 nodes).

behavior. With SCI APSP does not scale beyond 64 nodes and TC does not scale beyond 32 nodes. The static GLOW is the best option. For APSP, static GLOW is 2.20 times faster than SCI in 2 dimensions and 2.59 times faster in 3 dimensions (see Table 2). Similarly, for TC static GLOW is 2.22 and 2.64 times faster than SCI for 2 and 3 dimensions respectively (Table 2). For both programs combining again fails to show any performance improvement.

To summarize the results: dynamic GLOW consistently tracks the performance of static GLOW while combining only works for one program (GAUSS). These 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 or switch latency) while the behavior of dynamic GLOW with regard to the number of intercepted requests remains largely unaffected (see Section 6.2).

6.1 Sensitivity to window size

One interesting result we present in this paper is that the sliding window scheme is largely insensitive to the size of the recent-addresses queue especially for the larger systems (at least for the four programs we examined).

In Figure 6 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 two benchmarks exhibit similar behavior. It is evident in Figure 6 that the window size does not seriously affect the performance of the dynamic GLOW. 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.

6.2 Sensitivity to switch latency

That dynamic GLOW works well for the four benchmarks while combining partially works for only one of them suggests the latter is sensitive to the congestion

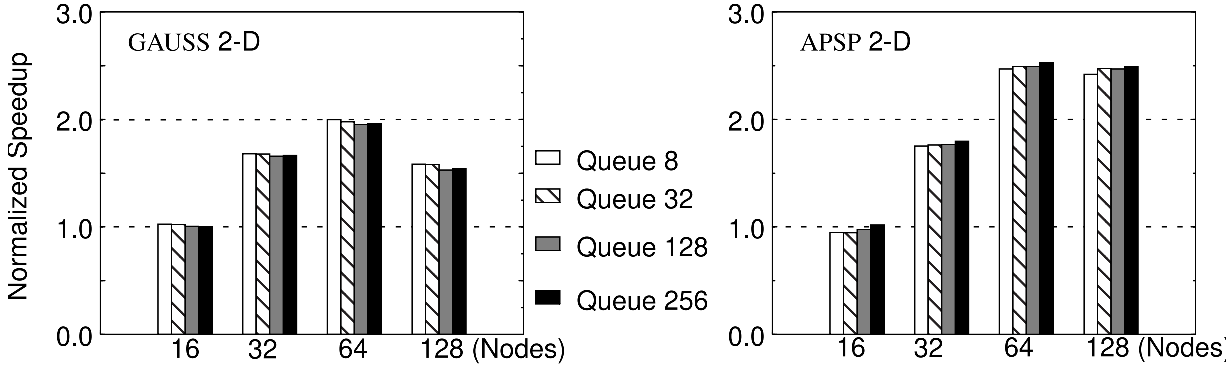

FIGURE 6. Sensitivity analysis for the size of the recent-addresses queue (speedup of the dynamic GLOW with respect to SCI on 16 nodes).

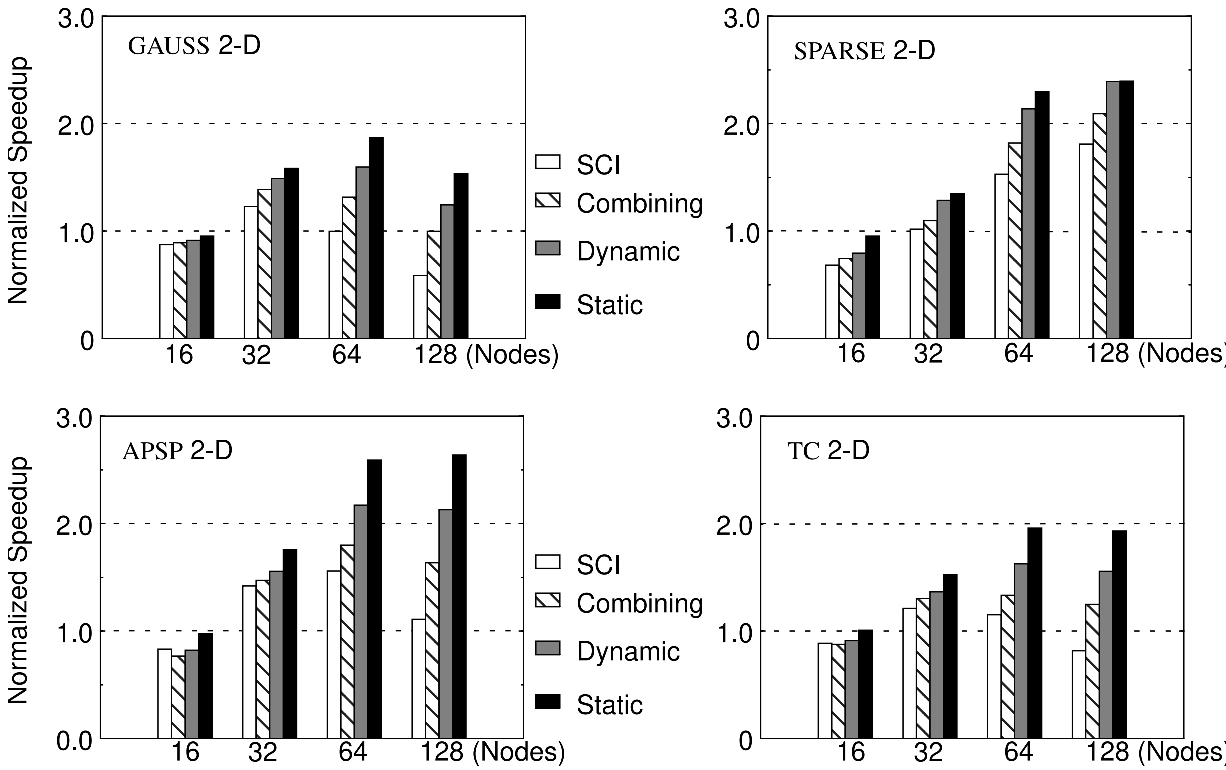

FIGURE 7. Speedup results with slower switches. Contention in the switch nodes makes ordinary combining competitive but it also slows down the whole system. Speedups for SCI, combining, dynamic and static GLOW shown with respect to SCI on 16 nodes with fast switches.

characteristics of applications. In this section, to confirm that combining is also sensitive to network parameters (while the recent-addresses queue scheme is not) we perform a sensitivity study on network parameters. In particular we examine what happens when we slow down the switches by increasing the latency required 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. The reason is that increasing link latency and end-point latency tends to space messages farther apart. In contrast, increasing the switch latency creates more congestion.

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

	GAUSS (FAST/SLOW)				SPARSE (FAST/SLOW)			
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
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
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
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
	APSP (FAST/SLOW)				TC (FAST/SLOW)			
N.	SCI	C	D	S	SCI	C	D	S
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	1.65/ 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	1.79/ 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	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 3: Speedup using fast and slow switches for SCI (SCI), Combining (C), Dynamic GLOW (D), and Static GLOW (S). The base case is SCI on 16 nodes with fast switches.

Figure 7 shows results for the four benchmarks for a system with slow switches. We use the same base case as before to assess the effect of the slow switches on performance. Thus we derive speedups by dividing the execution time of SCI on 16 nodes with fast switches by the execution time of SCI, combining, dynamic and static GLOW with slow switches (for 16 to 128 nodes).

Because of the slow switches SCI exhibits lower speedups than before (see Table 2). However, static and dynamic GLOW are affected less than SCI. This is because the GLOW extensions reduce considerably the number of ring crossings —the switches are used less to transport messages across rings. In Table 3 we summarize 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 the 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.

The performance of combining relative to dynamic 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 easily at least half the performance benefit of dynamic GLOW.

The performance of dynamic GLOW is still higher than combining but it 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.

7 Conclusions

In this paper we have introduced a generalization on combining that we use to dynamically detect widely shared data. When we detect such data we invoke the GLOW extensions to cache coherence protocols to efficiently handle them. 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. In previous work the GLOW extensions had to be invoked by generating special requests for the widely shared data. This meant that the user (the programmer or possibly the compiler) identified the widely shared data and with a variety of mechanisms informed the hardware of the nature of the data. Unfortunately, user involvement and implementation difficulties make such an approach less appealing. In this paper we show that we can dynamically discover the widely shared data and still obtain satisfactory performance (compared to the static methods).

The method we propose 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 left the switch. The interesting characteristic of our scheme is that in large systems even a small window performs very well. Dynamic GLOW 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. Finally, we found that combining is sensitive to application and network characteristics making it less effective than our method.

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. Grisham, 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 Shard-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] S. 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. (ftp.cs.wisc.edu)

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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

[24] 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.

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

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

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

[28] A. J. Bennett, P. H. J. Kelly, J. G. Refstrup, S. A. M. Talbot. "Using Proxies to Reduce Controller Contension 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.

[29] Sarah A. M. Talbot, Paul H. J. Kelly. "Reducing Controller Contension in Shared-Memory Multiprocessors Using Combining and Two-Phase Routing". Tech. Report Department of Computing, Imperial College of Science, Technology and Medicine, University of London, 1997.