11
5-21-91 JS ①

UCRL-LR-106975

The Cache Group Scheme for Hardware-Controlled Cache Coherence and the General Need for Hardware Coherence Control in Large-Scale Multiprocessors

Joseph Edward Hoag
(M.S. Thesis)

March 1991

Lawrence Livermore National Laboratory

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

UCRL-LR-106975
Distribution Category UC-705

UCRL-LR--106975

DE91 012106

The Cache Group Scheme for Hardware-Controlled Cache Coherence and the General Need for Hardware Coherence Control in Large-Scale Multiprocessors

Joseph Edward Hoag
(M.S. Thesis)

Manuscript date: March 1991

LAWRENCE LIVERMORE NATIONAL LABORATORY
University of California • Livermore, California • 94551

[illegible]

The Cache Group Scheme for Hardware-controlled Cache Coherence

and

The General Need for Hardware Coherence Control in Large-scale Multiprocessors

By

JOSEPH EDWARD HOAG

B.S. (Brigham Young University) 1989

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Contents

Abstract

A scheme that employs cache grouping and incomplete directory state in order to reduce the cost of maintaining directory state is introduced. This thesis discusses the cache grouping scheme, describes the protocols necessary for its implementation, and gives the results of detailed simulations of the scheme using various application codes. The effects of changing cache group size and using sophisticated multicast and combination features in the interconnect are explored.

It is discovered that the cache grouping scheme can equal the performance of full-directory schemes, while costing much less. The system is relatively insensitive to cache group size. Advanced multicast and combination features in the network work best when used together, and have especially beneficial effect for codes that exhibit a high rate of one-to-many invalidates. The simulation of a machine employing the cache grouping scheme indicates significant performance gains over an identical machine without a shared data cache. We also discuss the time and coding required to coax efficiency out of codes run on large scale parallel machines without hardware coherent cache mechanisms.

1 Introduction

1.1 Multistage interconnection networks and memory latency

For some years now, multiprocessors composed of commercially available microprocessors have been making significant performance gains. As the problems that we wish to solve become larger and more complex, the need arises for additional computation power in the form of scalable parallel machines with large numbers of processors. There are various ways to connect the processors together on such machines [1], including but not limited to:

- The shared bus: A shared bus, shown in Figure 1, is the least expensive method of interconnection. It can be viewed as a wire over which all processors pass messages and make memory requests. The problem with the shared bus is that it can be saturated by a small number of processors. If 1000 processors need to access 1000 different memory locations, they will all have to wait their turn for control of the bus. Shared buses present a potential communication bottleneck that makes them undesirable for connecting large numbers of processors. The largest commercial shared-bus multiprocessor systems use around 30 processors, but this bound has dropped with time as processor performance has increased.

- Complete connection: In Figure 2, we show 4 processors that are completely connected to one another. If 1000 processors were connected in such a manner, then all could make requests of each other simultaneously. However, N processors need N^2 wires to connect them, and this is expensive. In addition, each processor would need N ports. It is these expenses that preclude the use of complete interconnection for large numbers of processors.

- The full crossbar: In Figure 3, we show processing elements connected to memory elements through a full crossbar. A crossbar is a set of N^2 switchboxes that will route messages from any processor to any memory element; this results in excellent performance. Like the completely connected network, the crossbar becomes far too expensive when a large number of processors need to communicate with a large number of memories. In practice, it is rare to see a crossbar that is larger than 8x8.

Multistage interconnection networks (MINs) are the present answer to the difficult problem of connecting N processing elements to N memory elements; they are a nice compro-

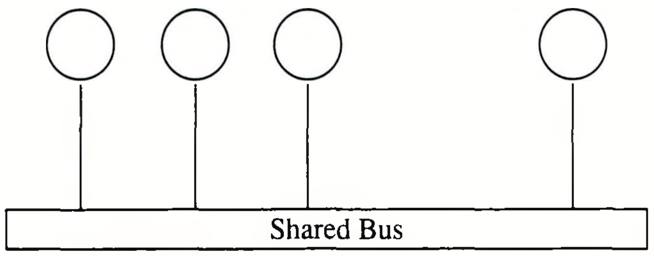

Figure 1: Shared bus architecture

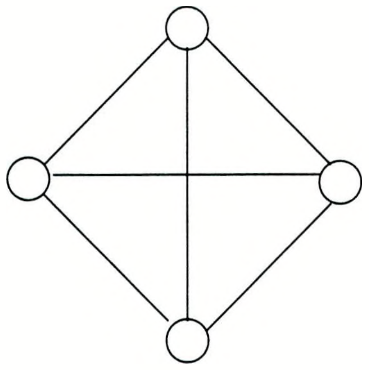

Figure 2: Completely connected network

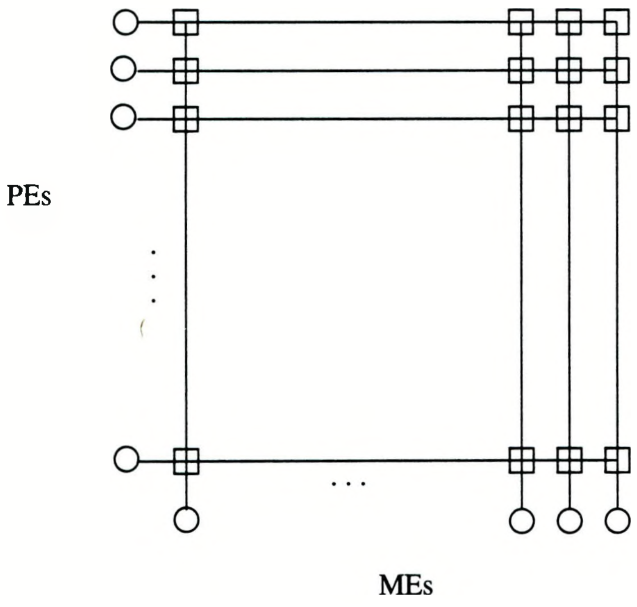

Figure 3: Full crossbar network

mise between cost and performance. An example of a multistage interconnection network is shown in Figure 4. Multistage interconnection networks have $log_k(N)$ stages, where k is the fan-out of the switchnodes, and N communication lines between stages. The number of communication wires needed in a MIN is $O(Nlog_k(N))$, as opposed to N^2 for the completely connected and crossbar networks.

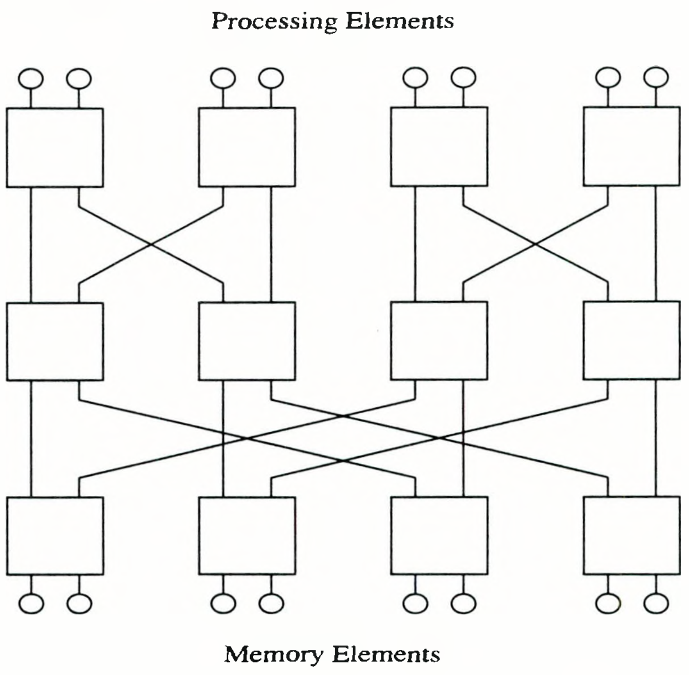

Figure 4: Multistage Interconnection Network using 2x2 Switchnodes

Unfortunately, communication between processors and memory is relatively slow in a MIN. Processors are not directly connected, so each communication must cross the network. In a typical system, shown in Figure 5, it takes on the order of $log_k(N)$ time for a memory request to be granted. If there is a large amount of traffic in the network, then the latency will be even longer. This can make shared memory references slow, compared to local memory references. This is a major drawback to multistage interconnection systems.

1.2 The cache coherence problem and its present solutions

A well-known solution to the memory latency problem is the use of cache, as shown in Figure 6, which temporarily holds copies of memory locations as they are reused. However, the problem of cache coherence is encountered. By definition [2]: "A memory scheme is coherent if the value returned on a LOAD instruction is always the value given by the latest

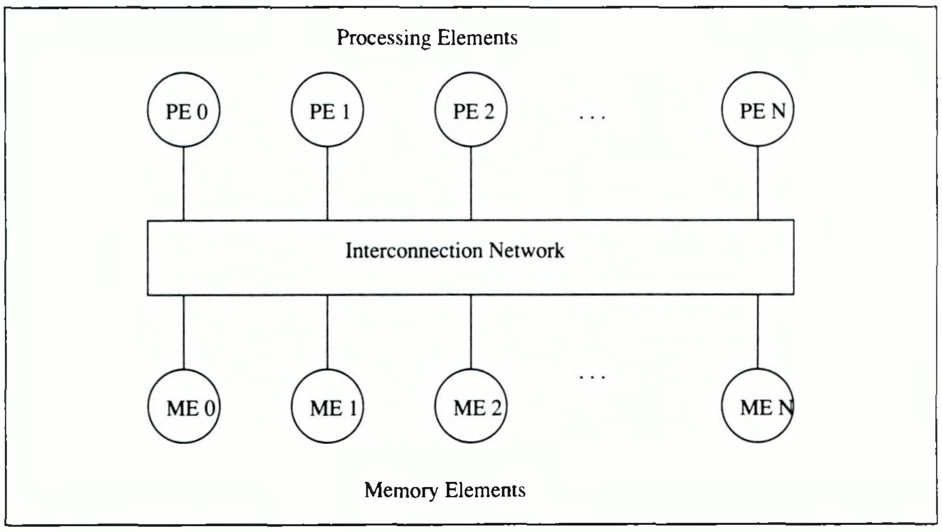

Figure 5: Typical MIN setup

STORE instruction with the same address." If no protocols or constraints are imposed on shared memory caches, then they will most likely not be coherent.

As an example, consider a simple spin-wait routine. Suppose processor A is waiting for processor B to write to shared memory location X. If A and B both have a copy of X in their shared-memory caches, and no coherence scheme is being employed, then A will spin forever on a piece of stale data. Even if processor B eventually writes to X, processor A will have no way of recognizing it. In Figure 7(a) we show the state of the two caches, A and B, before B writes to X. In Figure 7(b) we show the state of the two caches after B has written to its copy of X. This accurately illustrates the disastrous results of having shared-memory data caches without any enforcement of cache coherence. A program is almost guaranteed to run incorrectly.

Most hardware solutions to the cache coherence problem involve using memory controllers, labeled by K in Figure 6, to enforce coherence. Some sort of state information will be stored there for every cache line controlled by that particular controller. When a request comes in for a particular cache line, the memory controller will grant the request only after taking appropriate actions to insure coherence. These actions could include the invalidation of previously granted ("outstanding" or "out") copies of the cache line. A cache line can be in one of two modes in the cache: *read-only*, or *readable*, meaning that no modifications can be applied to the cache line; and *modifiable*, or *writable*, meaning that the line can be

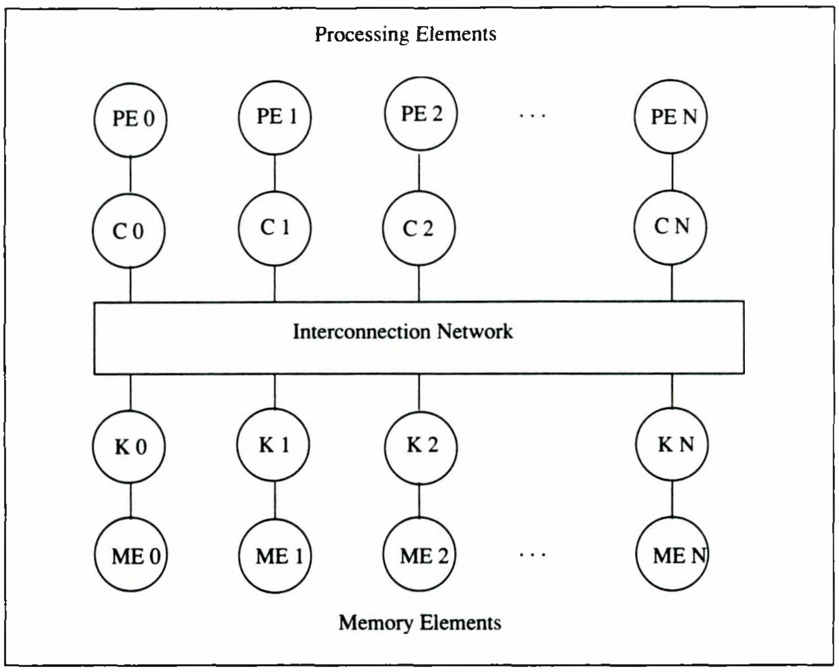

Figure 6: MIN with coherent cache system. The C's are caches and the K's are memory controllers.

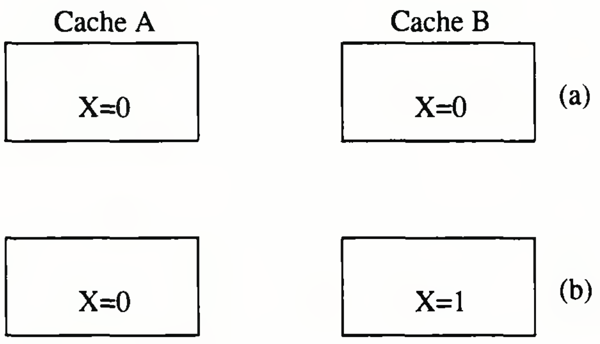

Figure 7: Shortcoming of a non-coherent shared memory cache

modified in cache. Usually, read-only copies of a cache line can be granted to any number of caches, but only one writable copy may be outstanding at a time. When a writable copy is out, no readable copies can be out.

In Figure 8, we detail the actions that a coherent system would take to insure coherence, using our spin-wait example above. After step (a), the initial state, cache B sends a request for a writable copy of X. Step (b) shows the caches after the memory controller invalidates their read-only copies of X. Step (c) shows cache B being granted line X in writable mode, and writing a 1 to X. Cache A is still spinning on X, and so sends a request for a read-only copy of X. Step (d) shows the caches after the memory controller invalidates and retrieves cache B's writable copy of X. Step (e) shows cache A finally getting the updated copy of X. Thus, independent of software, cache A can "see" the change to X.

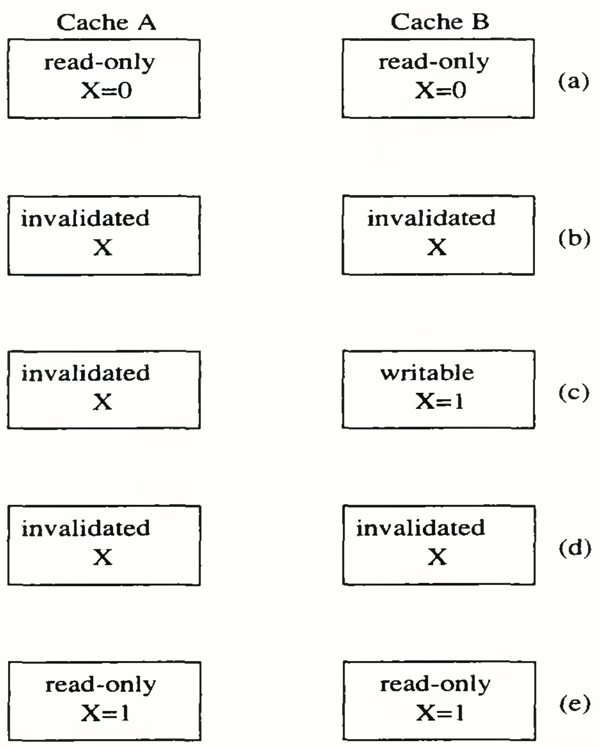

Figure 8: Typical actions taken to insure coherence

There are quite a few proposed hardware solutions[1] to the cache coherence problem, but they come in two general classes: directory schemes and broadcast schemes. These two classes are represented by the full-directory scheme [3] and the 2-bit protocol [4], re-

[1]There are also many compiler-assisted software-based coherence schemes, but these are beyond the scope of this thesis.

spectively. The full-directory scheme of Censier and Feautrier suffers in that it requires an excessive amount of controller memory. It requires $(N + 1)$ bits of controller memory per cache line, where N is the number of processors in the system, to explicitly record the location of each copy of a cache line. The 2-bit protocol of Archibald and Baer requires very little controller memory, but uses full broadcasts to implement coherence state changes. These broadcasts saturate the interconnection network, and unduly impact many of the caches.

We present a new solution, an interpolation between the full-directory scheme and the 2-bit protocol. It attains the performance of the full-directory scheme, while the memory requirements to track cache line location can be bounded by $log_2(N)$.

1.3 Structure of Thesis

The cache grouping scheme will be presented in detail in section 2. The protocol and hardware necessary for the implementation of such a scheme will be described in section 3.

In section 4, we give the results of detailed simulations of the cache grouping scheme. These simulations are performed on an extension of the Cerberus multiprocessor simulator [5]. Many issues are resolved. The scheme can and does equal the performance of a full-directory scheme. Cache group size generally does not affect overall performance. The use of a coherent cache significantly improves the effectiveness of a system. In section 4, we also discuss the shortcomings of two alternative schemes, the one-read scheme and the broadcast scheme.

In section 5, we give examples of the cost of *not* having a coherent cache mechanism on a large-scale shared memory multiprocessor. A significant amount of time, effort and coding must go into coaxing performance from applications on such machines. A good amount of software effort could be saved through the use of hardware which supports a coherent shared memory cache.

A short discussion is presented in section 6, summarizing the thesis.

2 The Cache Group Scheme

Traditional methods of coherence enforcement run into difficulties when one tries to scale the size of the system upward. The *full-directory* scheme [3] of Censier and Feautrier and the *2-bit* protocol of Archibald and Baer [4] represent the two ends of the spectrum of these traditional methods.

The *full-directory* scheme employs an $(N+1)$-bit vector in the memory controller for each cache line (Figure 9), where N is the number of processing elements in the system. N bits are used to explicitly record the location(s) of outstanding copies of that cache line. The $i'th$ location bit being set means that processor i has a copy of the cache line. A "dirty" bit is used to record the existence of a modifiable copy. Coherence is maintained by means of point-to-point (i.e., one cache at a time) invalidations. The problem with this scheme is that the memory needed to track cache line location grows linearly with the number of processors, N, so that it does not prove to be scalable. For example, in a 1024-processor system with 16-byte cache lines, it would take 1025 bits in controller memory to handle each 128-bit cache line in main memory.

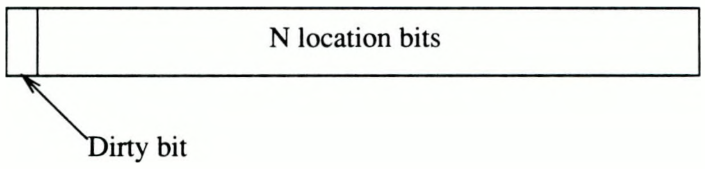

Figure 9: Memory controller record of cache line with *full-directory* scheme.

The *2-bit* protocol, or *minimal state* scheme, reduces the number of bits necessary to track each cache line to 2 (Figure 10). There is no knowledge of the location of each cache line. Instead, a cache line is in one of four states: ABSENT (no copies are outstanding), PRESENT1 (one read-only copy is outstanding), PRESENT* (many read-only copies outstanding), and PRESENTM (one modifiable copy outstanding). This scheme minimizes the number of bits needed to record location information by not remembering the positions of any outstanding copies of a cache line. Since there is no knowledge of the location(s) of outstanding copies of a line, full broadcasts are necessary for most coherence operations. These broadcasts not only degrade the performance of the interconnection network, but they also adversely affect any caches not actually holding the line in question.

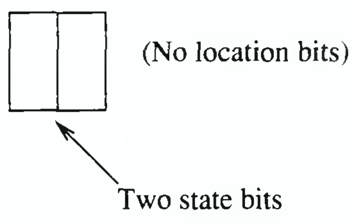

Figure 10: Memory controller record of cache line using two-bit scheme.

In this section, we discuss 3 innovations that can be used to efficiently enforce coherence without using a disproportionate amount of controller memory: *cache groups, multicasting* and *return reply combination.* We call the combination of these three concepts the *cache grouping scheme.* Our primary idea is the concept of *cache groups*; we advocate the use of a directory scheme with a bit for each cache group instead of one per cache. These cache groups can be of arbitrary size. As cache group size grows, more caches are needlessly invalidated with multicasts. Small cache group sizes necessitate more controller memory to partially track cache line location. In order to efficiently enforce the cache group scheme in the hardware, we propose the implementation of *multicasting* and *return reply combination.* Multicasting enables the interconnection network to quickly propagate invalidation requests to cache groups. Return reply combination, as we describe it, is a low-cost method of combining invalidation acknowledgements (from the caches) in the interconnection network. Unlike some other combining schemes [6], it does not require additional memory at the switchnode level.

2.1 Cache Groups

The idea of cache groups is an interpolation between the *full-directory* scheme and the *2-bit* protocol. It is an attempt to capture the performance of a full directory without its excessive memory requirements.

Baer and Girault suggest combining the *2-bit* protocol with the use of a cache index for a single outstanding copy[7]. By doing this, broadcasts can be avoided for the frequent case of one-to-one data sharing. We expand on this idea by giving these index bits something to do in the case of one-to-many sharing.

When a cache line is in the PRESENT1 or PRESENTM state. exactly one copy of it is currently out. $Log_2(N)$ location bits will be used to explicitly track the cache line, where N is the number of processors (Figure 11). When a cache line is in the PRESENT* state (many readable copies outstanding), then N/G bits will be used to partially track the locations of outstanding lines, where G is the size of a cache group (Figure 12).

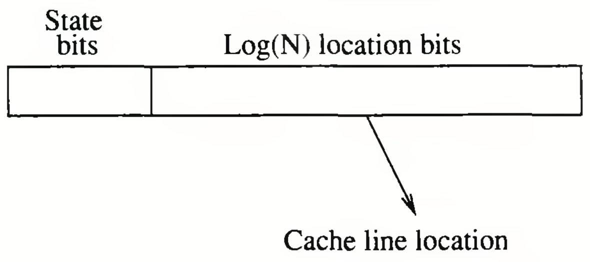

Figure 11: Cache line representation in memory controller with exactly one copy outstanding (*exact* encoding).

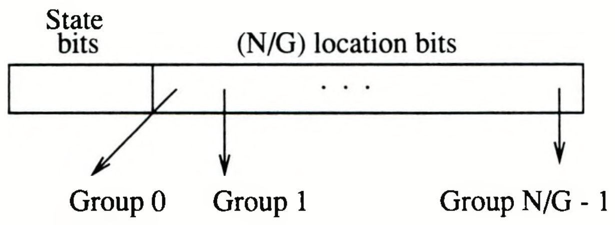

Figure 12: Cache line representation in memory controller with indeterminate number of copies outstanding (*partial* encoding).

When in PRESENT* state, where an undetermined number of copies of a cache line are distributed to the processors, a location bit being set means that "this cache group *MAY* hold up to G copies of this cache line." Some of the caches in the targeted cache group may NOT have the line in question, and will be unduly affected by the invalidations to which they are subjected. Nevertheless, the scheme allows for a substantial reduction in the number of "useless" invalidate requests, compared to the minimal state solution of Archibald and Baer.

Line State	Bits needed to record location(s) of line
ABSENT	0
PRESENT1	$log_2(N)$
PRESENT*	N/G
PRESENTM	$log_2(N)$

Table 1: Bits required to track cache line location(s) in various cache states. N is the number of processors, G is the size of each cache group.

As we show in Table 1, $max(log_2(N), N/G)$ bits are required to track the location of each cache line, compared with N bits in the classic *full-directory* scheme. If the number of cache groups is set so that $N/G \simeq log_2(N)$, then the memory requirements of this scheme will grow as $log_2(N)$, which gives it the quality of being scalable in a practical sense.

The idea of cache grouping was also proposed by Gupta, Weber and Mowry [8]; they called this mechanism *coarse vectors*. Gupta et al. used the Stanford DASH architecture [9] as a model. The DASH architecture features clusters of processors connected by a mesh. In order to evaluate their version of the scheme, they performed event-driven simulations, using Tango [10] to generate multiprocessor references.

As an example of how cache groups work, consider a 16-cpu system with 4 cache groups of 4 caches each. If cache line X has been granted to cache 12 as read-only, then the four location bits will be used to explicitly identify cache 12 (Figure 13). If cache 5 then requests X as a read-only line, the location bits will be converted to "partial" representation, where each bit represents a cache group (Figure 14).

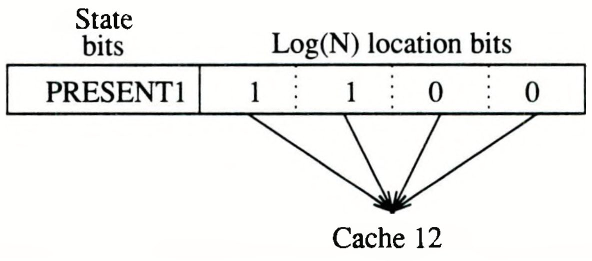

Figure 13: Setting of location bits for one copy out to cache 12.

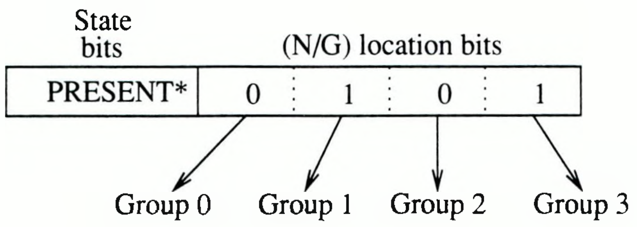

Figure 14: Setting of location bits for one copy out to cache 12, one copy out to cache 5.

2.2 A Note on Coherence and Ordering

In our scalable coherent cache system, *return receipts* and the *wait* instruction are used to restrict the ordering of memory operations when this is required by the application. These features are borrowed from an earlier version of the Cerberus multiprocessor simulator [5], which did not support caches for shared memory, but still had a problem to be solved with respect to the ordering of main memory operations.

The idea behind return receipts is that for every request through the processor-memory interconnect, a receipt is returned to the requestor which indicates that the requested action has been performed. In the case of the Cerberus multiprocessor simulator the return receipt for a read request was the returned data, but an explicit receipt was generated for write requests as well. Because data is not returned to the processor for a write, the return receipt is necessary to allow the processor to keep track of when its writes have been performed.

In the Cerberus multiprocessor simulator, each processor kept a receipt counter. Every time a memory request was *issued*, the counter was incremented. Every time a request was known to be *performed* (i.e., the processor received a return receipt) the counter was decremented. The *wait* instruction caused a processor to hold issue on any further instructions until the receipt counter was zero, and therefore all pending memory operations were complete. This seemingly innocuous mechanism provides for completely dynamic enforcement of ordering between groups of memory references when it is required. In our proposed scalable coherent cache system we use return receipts and the *wait* instruction in the same way. The processing of cache misses, to different cache lines, can be handled concurrently both within a cache and between the many caches in the system. Return receipts provide knowledge of when state changes for cache lines are complete, and the *wait* instruction causes a processor to wait for completion when needed.

An alternative approach to coherence and ordering is the notion of the global synchronizing variable [2]. Those who have some experience in parallel programming know that it is undesirable to statically declare a global synchronizing variable, or a volatile variable for that matter. A variable may be used for communication and synchronization in one instance, and then get used for normal computation later on. If the variable were treated as a global synchronizing variable all the time, performance would be adversely affected. By using the *wait* instruction, which allows the processor to keep track of when memory operations are complete, one solves the ordering problem in a completely dynamic fashion.

Through the correct use of the *wait* instruction, any memory location can *temporarily* be made into a global synchronizing variable. One simply surrounds the specific access in question with *wait* instructions. The first *wait* instruction will force all previous memory accesses by the processor to be performed before the synchronization variable is accessed. The second *wait* instruction will force access to the synchronization variable to be performed before any further memory accesses are started. This enforces weak ordering, as defined by Dubois, Scheurich and Briggs[2].

2.3 Multicasting

Broadcast schemes to enforce cache coherence are generally not looked upon favorably due to the excessive amount of network traffic that they produce. Cache grouping reduces the impact of broadcasts by limiting them to groups of caches that may hold the cache line in question. These specific broadcasts are called *multicasts.* To further reduce the impact of broadcasts, we provide efficient hardware support for multicasts in the processor-memory interconnection network.

In a packet-switching multistage interconnection network, packets are transmitted over the network a stage at a time. The route through the network depends on a packet's routing tag for each stage. Most networks currently utilize $log_2(k) * log_k(N)$ bits for the routing tag, where k is the fan-out of each switchnode and N is the number of processors in the system. At each of $log_k(N)$ stages in the interconnection network, $log_2(k)$ bits of the routing tag determine the next switchnode (or endpoint) to which the packet will be sent. Packets are thus transmitted on a one-PE-to-one-ME basis, or a one-ME-to-one-PE basis.

We suggest using $k * log_k(N)$ bits for each routing tag, with the k bits per stage enabling us to route a packet out to multiple output ports of a switchnode. For normal routing, only one bit in each routing field will be set and a single packet will be routed to its destination. If more than one routing bit is set for any of the fields, the packet will be multicast onto the appropriate output ports of the switchnode.

As an example, consider the 8-processor system composed of 2x2 switchnodes shown in Figure 15. A normal point-to-point routing tag would have 3 fields of one bit each. A multicast routing tag would have 3 fields of *two* bits each, for independent control of each output of a switchnode. Point-to-point communications can still be accomplished by setting only one routing bit per stage.

A multicast from memory 4 to the top group of 4 processors is shown in Figure 15. This would be typical of a memory controller sending invalidates to a cache group of size 4. Without the multicast mechanism in the switchnodes, four separate invalidates would have been necessary.

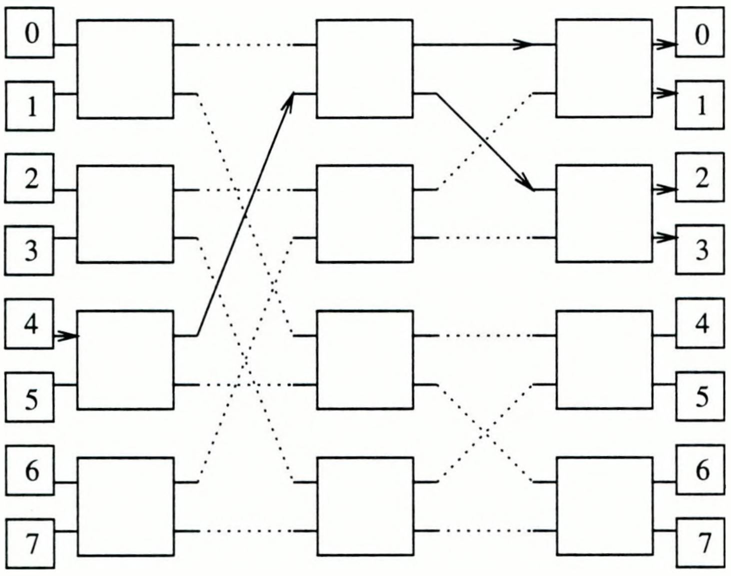

Figure 15: Path of message with routing tag (10)(11)(11) (solid lines).

In order to examine the benefits of hardware multicast support, a metric *packets processed*, P, needs to be defined. P is a count of the number of packets that need to be processed across the interconnect in order to effect an invalidation. It is a good indicator of the load that is put on the interconnection network by invalidations.

A significant reduction in network traffic can be realized through multicasting as the system gets large. If all communications were done on a *point-to-point* basis, then an invalidate to a cache group of size G would involve

$$P = G * log_k(N) \tag{1}$$

total packets being processed[2] (where k is the fan-out of each switchnode); the $log_k(N)$ stages of the network must be traversed by G independent invalidate requests. If multicasting is utilized, then an invalidate to a cache group of size G would involve

$$P = (log_k(N) - log_k(G)) + \frac{k(G-1)}{k-1} \tag{2}$$

total packets being processed.

The ramifications of the above equations, as the number of processors gets large, are shown in Table 2. The numbers in the table are the total number of packets processed for an invalidate to a cache group of size G in an N-processor system constructed of 2x2 switchnodes. G is chosen in each case such that the number of cache groups is approximately $log_2(N)$. As is evident, the use of multicasting relieves much of the strain on the network in terms of invalidation traffic.

	P	
Configuration	Point-to-point method	Multicast method
N=8, G=4	12	7
N=16, G=4	16	8
N=32, G=8	40	16
N=128, G=16	112	33
N=1024, G=128	1280	257

Table 2: Packets processed using 2x2 switchnodes, Point-to-point vs. Multicast

Note that the topology of the interconnection network is crucial for the implementation of multicasting. In Figure 15, the memory controllers must be on the left and the caches on the right in order for the caches to be efficiently partitioned into cache groups. This topology causes invalidate packets to travel "down a tree," which naturally facilitates broadcasting.

[2] A packet can be processed up to $log_k(N)$ times as it traverses the interconnect. A multicast from a switchnode is counted as 1 packet processed.

There are several memory controller strategies that could be utilized to implement multicasting. For example, the memory controller could be smart enough to recognize that neighboring cache groups have a copy of a line, and so could combine the resultant multicasts to these groups. In our simulations, a very simple strategy is employed, which is to look at every cache group individually as invalidation multicasts are sent out. It requires no special logic to implement this simple method.

2.4 Return Reply Combination

Multicasts are used to invalidate copies of a cache line which are present in one or more groups of caches. In our scheme, return receipts are required from each cache to which an invalidation request has been sent. These return receipts are counted by the memory controller. Once the receipts are all accounted for, the memory controller grants the writable copy. Without any special handling of return receipts they would have to be counted individually by the memory controller, leading to a bottleneck if invalidates of multiple readable copies are frequent. We provide for combination of return receipts in the switchnodes so that this problem can be avoided.

Unlike the relatively simple modification for multicast support, return reply combination requires a more sophisticated modification to the switchnodes of the Cerberus multiprocessor simulator. The Cerberus processor-memory interconnection network is composed of k-input, k-output switchnodes, each with k^2 buffers. The original structure of a buffer is shown in Figure 16. The packet input (*in*), packet output (*out*), buffer full (*bf*), output inhibit (*oi*), output busy (*ob*), and packet selector signals (s, p) are used in an identical fashion as they were in [11].

Return receipt packets for multicast invalidate requests are special packets which have both a unique identifier for the request and a counter field which records how many return receipts the packet represents. The buffers which feed a given output port in the switchnode have their *match* lines connected to each other, and their sum out (*so*) lines feed a k-input adder which feeds the sum in (*si*) lines of the k buffers with its result (Figures 17 and 18). The handling of conflicts for the output port is done in the same way as was done in [11], with the following additional treatment of multicast return receipts. If the buffer which wins the conflict for the output port contains a return receipt in a suitable position, the identifier of this packet is written on the *match* lines and the counter of this packet is

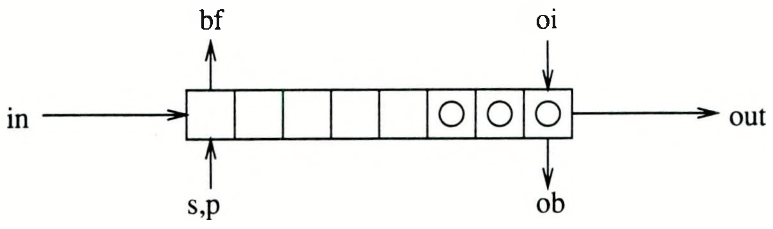

Figure 16: Original switchnode buffer.

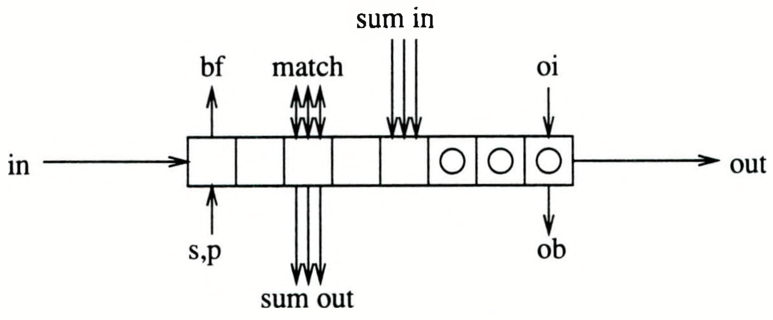

Figure 17: Modified switchnode buffer.

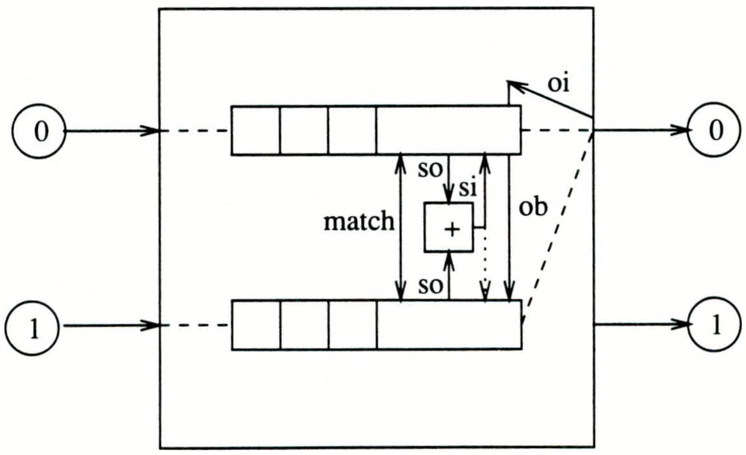

Figure 18: Modified 2x2 switchnode.

written on the *so* lines. The buffers which lose the conflict for the output port read the *match* lines and check their lead slot, or more deeply if possible[3], for a matching packet. If they find a matching return receipt they write the sum field of that packet on their *so* lines and drop the packet on the floor. The buffers write zero on their *so* lines if no matching packet is found. The k-input adder adds the partial sums together and the buffer which issued the match request replaces its partial sum with the output of the adder (obtained from the *si* lines).

By combining return receipts for multicast invalidate requests, the potential bottleneck at the memory controller can be avoided. The combining function requires a more complicated buffer, and k k-input adders to be associated with each switchnode. These adders would have to be wide enough to accommodate the maximum node count for the system.

It is important to note that our method of return reply combination requires no additional memory at the switchnode, as do some other combination schemes [6]. Also, note that the topology of the interconnection network is once again crucial to the efficient implementation of reply combination. In Figure 15, the caches must be on the right and the memory controllers on the left in order for invalidation replies to combine as early as possible. This causes invalidation replies to go "up a tree," which naturally facilitates combining.

In order to efficiently implement both multicasting and return reply combination, therefore, two processor-memory interconnection networks have to be used. However, only one has to implement multicasting and only one need implement return reply combination.

[3]In sections 4.3 and 4.5, we discuss the merits of "deep" combining vs. top-level combination. In general, top-level combination is sufficient.

3 Coherence Protocol and Timing

3.1 Protocol States and Actions

For each cache line in main memory, the associated memory controller needs a method to track the way in which that line is shared among all caches in the system. This is done by associating a *state* with each line. The following five states have their basis in [4]:

- ABSENT: No copies of the cache line are currently out.
- PRESENT1: One read-only copy of the cache line is currently out.
- PRESENT*: An indeterminate number of read-only copies of the cache line are currently out.
- PRESENTM: One modifiable copy of the cache line is currently out.
- LIMBO: The state of the cache line is in transition, and no access to it is allowed until the transition is complete.

We added the LIMBO state because certain state transitions do not take place instantaneously, and LIMBO was necessary to insure atomic state transitions. For example, if a line is in state PRESENTM and a request arrived for a read-only copy of that line, then an invalidate would have to be sent and acknowledged before the read-only line could be granted. During the time that write-invalidation is performed, no access should be given to the line in question. It is during these invalidation periods that the LIMBO state is employed. All requests that arrive for a cache line in main memory while it is in a LIMBO state are deferred to the wait list (described later in this section).

Communications between cache and memory controller take place through *actions* which dispatch messages through the network. These actions control the manner in which cache lines are tracked and shared.

Cache to Memory Controller Actions:

- REQ_R_LINE(cpu, line) (1 clock). Request a cache line in read-only mode.
- REQ_W_LINE(cpu, line) (1 clock). Request a cache line in writable mode.
- REPORT_R_SPILL(line) (1 clock). Notify the memory controller that a read-only line has been spilled from cache.

- REPORT_W_SPILL(line) (**Len** clocks). Notify the memory controller that a writable line has been spilled from cache, and write the line back to memory.
- ACK_R_INV(line) (1 clock). Respond to a read-invalidate request from the memory controller.
- ACK_W_INV(line) (**Len** clocks). Respond to a write-invalidate request from the memory controller, sending back the modified cache line.

Memory Controller to Cache Actions:

- GRANT_R_LINE(cpu, line) (**Len** clocks). Send a line to a cache in read-only mode.
- GRANT_W_LINE(cpu, line) (**Len** clocks). Send a writable copy of a line to a cache.
- INV_R_LINE(cpu | cache group, line) (1 clock). Request a cache (or cache group) to invalidate its read-only copy(s) of a cache line.
- INV_W_LINE(cpu, line) (1 clock). Request a cache to invalidate its writable copy of a cache line, and to send back the modified copy.

Timing is given for every action described, in terms of the number of clock ticks that elapse. Most actions take one clock. Some, however, involve buffering cache lines onto the processor-memory interconnect, and thus their timings are dependent upon cache line size. **Len** (for cache line length) will be the symbol for "some amount of time proportional to cache line size." In our simulations, **Len** is 1 clock for every 8 bytes of cache line.

3.2 Cache Architecture and Protocols

We modified the Cerberus multiprocessor simulator to model quite a few cache configurations. The size, in bytes, of a cache line can be any power of 2, minimum 8 bytes. The number of lines in each cache can be any power of two. Each cache can be direct-mapped, 2-way associative or 4-way associative.

A line in the cache has the following components, as shown in Figure 19:

- Dirty bit: Set if the cache line is modifiable.
- Valid bit: Set if the cache line is valid. Cache lines are invalidated by resetting this bit.

- High order address bits: The low $log_2(L)$ bits of the main memory address of a cache line, where L is the number of lines in the cache[4], determine the placement of the line within the cache. It is necessary to store the remaining high-order bits in order to be able to reproduce the full shared memory address of the cache line.
- Data bytes: The actual data that has been granted from shared memory. The number of these bytes is synonymous with the cache line size, W.

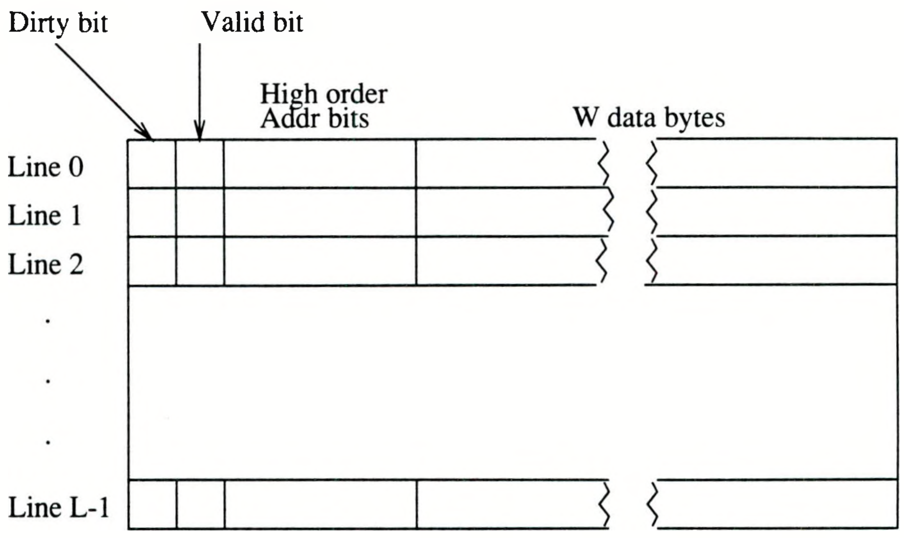

Figure 19: Direct-mapped Cache Configuration

Unlike current microprocessor designs, the Cerberus processor does not stall on the first miss for a line. The cache has 5 *request slots* where *request records* can be stored. A request record is generated when a cache miss occurs. It contains the register to be loaded from memory (or, for write misses, the value to be written to memory), the size of the memory request (byte, short, word or double), the exact address to be read/written, and the type of the request (READ or WRITE) (see Figure 20). If all of these slots are occupied, then the processor will stall on a cache miss.

There may be multiple request records awaiting the same cache line being granted from main memory. When the requested cache line arrives, it services all request records that are waiting on it.

[4]The *total* number of lines in the cache is L multiplied by the associativity level.

Reg/Value	Address	Size	Type

Figure 20: Request slots

When multiple steps are required for the cache to respond to an event, then these steps will be numbered below by roman numerals. When conditional courses of action are described, these possible courses of action will be assigned the same roman numeral followed by different lower-case letters. A step 0 is sometimes included for error detection.

The cache must respond to the following events:

- Read Hit(ADDR,REG): The contents of ADDR are loaded into register REG. A latency of four clocks expires to load a register, but independent loads issue and are completed at a rate of one per clock.

- Read Miss(ADDR,REG):

  0. If the desired *address* is already being waited upon by the cache, then stall the cpu until it arrives.

  Ia. If the desired *cache line* is currently being waited on by a different request record, then simply create a new request record. This takes one clock.

  Ib. If the desired cache line is **not** already being awaited, then a REQ_R_LINE(cpu, ADDR) is sent to the appropriate memory module. This takes one clock, plus one clock for creating the request record.

- Write Hit(ADDR,REG): The contents of register REG are copied to ADDR. The associated STORE instruction is a one-clock pipelined instruction.

- Write Miss(ADDR,REG):

0. If the desired *address* is already being waited upon by the cache, then stall the cpu until it arrives.

Ia. If the desired *cache line* is present in read-only form, then invalidate it and send a REPORT_R_SPILL(line). This takes one clock. Then, send a REQ_W_LINE(cpu, ADDR) to the appropriate memory module. This involves one clock for creating the new request record and one clock for sending the request.

Ib. If the desired cache line is currently being waited on by a different request record, then simply create a new request record. This takes one clock.

Ic. If the desired cache line is **not** already being awaited, then a REQ_W_LINE(cpu, ADDR) is sent to the appropriate memory module. This takes one clock, plus one clock for creating the request record.

- GRANT_R_LINE(LINE):

  I. If there is room for LINE in the cache, then skip to IV.

  II. Invalidate the appropriate line. If cache is associative, invalidate the least recently used line in a slot.

  IIIa. If the line being invalidated is read-only, send a REPORT_R_SPILL(spilledLINE) to the appropriate memory controller. This takes one clock.

  IIIb. If the line being invalidated is writable, send a REPORT_W_SPILL(spilledLINE) to the appropriate memory controller. This takes **Len** clocks.

  IV. Insert LINE into the cache. Service all appropriate request records. Every register that is loaded will be available in 2 clocks.

- GRANT_W_LINE(LINE):

  I. If there is room for LINE in the cache, then skip to IV.

  II. Invalidate the appropriate line. If cache is associative, invalidate the least recently used line in a slot.

  IIIa. If the line being invalidated is read-only, send a REPORT_R_SPILL(spilledLINE) to the appropriate memory controller. This takes one clock.

  IIIb. If the line being invalidated is writable, send a REPORT_W_SPILL(spilledLINE) to the appropriate memory controller. This takes **Len** clocks.

IV. Insert LINE into cache. Service all appropriate request records.

- INV_R_LINE(LINE):

  0. If LINE is present and modifiable, give an error message.

  I. If LINE is present as read-only in the cache, then invalidate it. This decision takes 2 clocks.

  II. Regardless of whether the line was in cache in step I, send back an ACK_R_INV(LINE). This takes one clock.

- INV_W_LINE(LINE):

  0. If LINE is present in read-only mode, give an error message.

  I. If LINE is present and writable in the cache, then invalidate it. This decision takes 2 clocks.

  IIa. If LINE was present and writable in step I, then send an ACK_W_INV(LINE), which will take **Len** clocks.

  IIb. If LINE was **not** present and writable in step I, then simply ignore the invalidation request. This is necessary if a writable line is spilled just before the invalidation request comes in.

3.3 Memory Controller Architecture and Protocol

Memory is logically interleaved throughout the system in intervals of one cache line, as shown in Figure 21. Each memory controller maintains a record of each cache line that it controls.

The memory controller tracks the state of each line and the location of each copy of each line under its control. There is also a *wait list* associated with each memory controller (see Figure 22). Each element of the wait list is a memory request that cannot be granted until a state transition is effected.

Each element in the wait list has a wait counter associated with it to track acknowledged invalidations. When a request is enqueued onto the end of the wait list, its wait counter is set to the number of invalidations expected before that request can be granted. As invalidation acknowledgements arrive for that request, the wait counter is decremented. When the wait

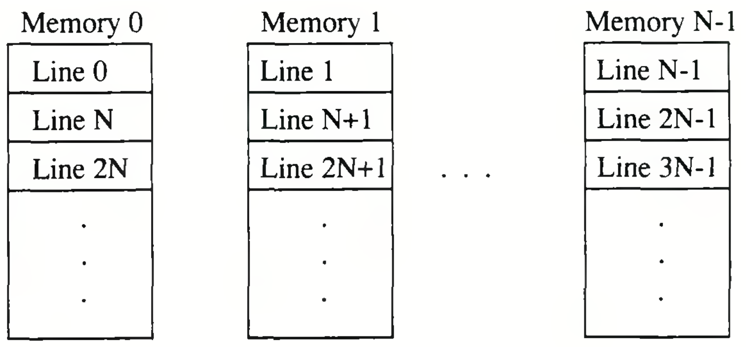

Figure 21: Interleaving of main memory

counter for a memory request in the list is decremented to 0, then that memory request can be granted.

The memory controller is a state machine that grants a cache line, or effects state transitions, according to the present state of the line. For the remainder of this section, we describe the protocol that determines the behavior the memory controller.

The terms *exact* encoding and *partial* encoding are used frequently in this section. An exact encoding of a cpu (CPU) into the location bits for a cache line implies that $log_2(N)$ bits are used to explicitly identify CPU. A partial encoding means that the cache group bit (within the location bits of the line) associated with CPU is turned on.

When multiple steps are required for the memory controller to respond to an event, then these steps will be numbered below by roman numerals. When conditional courses of action are described, these possible courses of action will be assigned the same roman numeral followed by different lower-case letters.

The memory controller must respond to the following events:

- REQ_R_LINE(CPU, LINE):

  Ia. If LINE is in state ABSENT, then update its state to PRESENT1. Exactly encode CPU into the location bits for LINE. Send a GRANT_R_LINE(CPU, LINE) (**Len** clocks).

  Ib. If LINE is in state PRESENT1, then update its state to PRESENT*. Convert

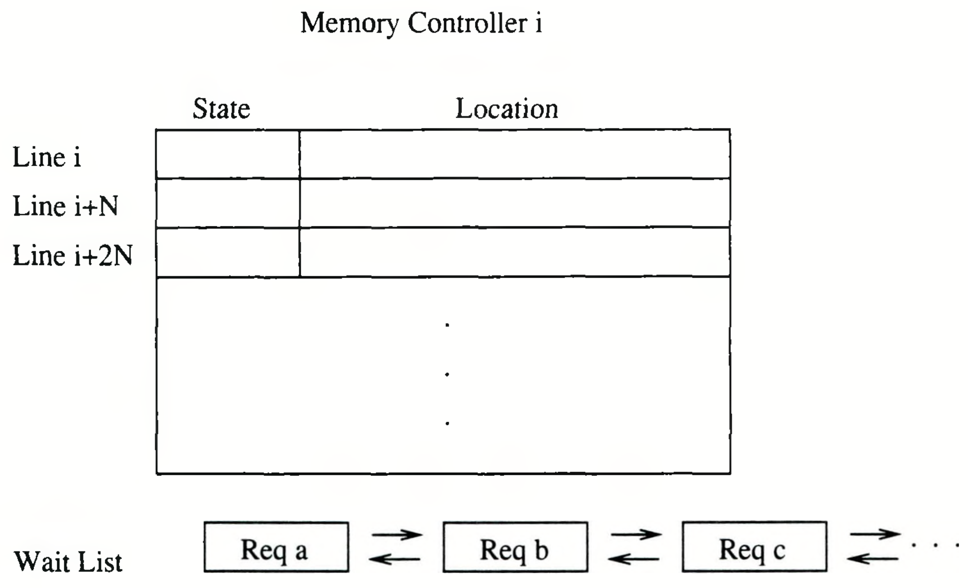

Figure 22: Logical Structure of Memory Controller

the location bits to partial encoding, and partially encode CPU. Send a GRANT_R_LINE(CPU, LINE) (**Len** clocks).

Ic. If LINE is in state PRESENT*, then partially encode CPU into the location bits for LINE. Send a GRANT_R_LINE(CPU, LINE) (**Len** clocks).

Id. If LINE is in state PRESENTM, then put the request on the back end of the wait list (1 clock) and update the state of LINE to LIMBO. Send an INV_W_LINE(XCPU, LINE), where XCPU is the cpu that presently holds LINE in a writable mode (1 clock).

Ie. If LINE is in state LIMBO, then put the request on the back end of the wait list (1 clock).

- REQ_W_LINE(CPU, LINE):

Ia. If LINE is in state ABSENT, then update its state to PRESENTM. Exactly encode CPU into the location bits for LINE. Send a GRANT_W_LINE(CPU, LINE) (**Len** clocks).

Ib. If LINE is in state PRESENT1, then put the request on the tail of the wait list and update its state to LIMBO (1 clock). Send an INV_R_LINE(XCPU, LINE), where XCPU is the cpu that presently holds LINE in read-only mode (1 clock).

Ic. If LINE is in state PRESENT*, then put the request on the tail of the wait list and update the state of LINE to LIMBO (1 clock). Multicast an INV_R_LINE(GROUP, LINE) to each cache group holding a copy of LINE (1 clock per multicast).

Id. If LINE is in state PRESENTM, then put the request on the tail of the wait list and update the state of LINE to LIMBO (1 clock). Send an INV_W_LINE(XCPU, LINE), where XCPU is the cpu that presently holds LINE in a writable mode (1 clock).

Ie. If LINE is in state LIMBO, then put the request on the tail of the wait list (1 clock).

- ACK_R_INV(LINE):

  I. Traverse the wait list until the request for LINE is found. Each element traversed takes 1 clock.

  II. Decrement the request's wait counter. If the counter is greater than zero, then goto VII.

  III. (All invalidations have been performed for LINE). Dequeue the request from the wait list. Update the state of LINE to PRESENTM.

  IV. Issue a GRANT_W_LINE(XCPU, LINE), where XCPU is the cpu that issued the waiting request (**Len** clocks). Exactly encode XCPU into the location bits for LINE.

  V. Traverse the wait list in search of another request for LINE (1 clock per element traversal). If none are found, goto VII.

  VI. (Another request for LINE is waiting). Send an INV_W_LINE(XCPU, LINE) to the cpu that just received LINE as writable (1 clock). Update the state of LINE to LIMBO.

  VII. Done.

- ACK_W_INV(LINE):

  I. Traverse the wait list until the request for LINE is found. Each element traversed takes 1 clock.

II. Dequeue the request (REQ) from the wait list. If REQ is a request for a writable line, then goto step III of ACK_R_INV(LINE).

III. (REQ is for a read-only copy of LINE). Update the state of LINE to PRESENT1. Issue a GRANT_R_LINE(XCPU, LINE), where XCPU is the cpu that issued REQ (**Len** clocks). Exactly encode XCPU into the location bits of LINE.

IV. Traverse the wait list (1 clock per element) in search of any other REQ_R_LINE request for LINE. If none are found, goto VII.

V. (Another read request, OTHERREQ, has been found in the wait list). Issue a GRANT_R_LINE(XCPU, LINE), where XCPU is the cpu that issued OTHERREQ (**Len** clocks). Update the state of LINE to PRESENT*. Convert the locations bits of LINE to partial encoding if necessary. Partially encode XCPU into the location bits of LINE.

VI. Goto IV.

VII. Traverse the wait list (1 clock per element) in search of any REQ_W_LINE request for LINE. If none found, goto X.

VIII. (A write request for LINE, WREQ, has been found in the wait list). If LINE is in state PRESENT1, then issue an INV_R_LINE(XCPU, LINE) to the cpu holding line in read-only mode (1 clock). If LINE is in state PRESENT*, multicast an INV_R_LINE(GROUP, LINE) to every cache group holding LINE (1 clock per multicast).

IX. Update the state of LINE to LIMBO.

X. Done.

- REPORT_R_SPILL(LINE):

  Ia. If LINE is in state PRESENT1, then update its state to ABSENT.

  Ib. If LINE is in state PRESENT* or LIMBO, ignore this.

  Ic. If LINE is in any other state, give an error message.

- REPORT_W_SPILL(LINE):

  Ia. If LINE is in state PRESENTM, then copy LINE back to main memory and update its state to ABSENT.

Ib. If LINE is in state LIMBO, then treat this as an ACK_W_INV.

Ic. If LINE is in any other state, give an error message.

4 Simulation Results

In this section, we give the results obtained from detailed simulations of the cache grouping scheme using the Cerberus multiprocessor simulator. A short description of Cerberus is presented, followed by an explanation of the application codes used to test the scheme.

Several issues are addressed in this section. First, we wanted to test the performance of cache grouping relative to a full-directory scheme. We also wanted to test a system with cache grouping relative to one with no cache at all, to ascertain the advantages (if any) of cache coherence in general. In addition, we wanted to test the effects of cache group size. Finally, we did some less detailed tests showing the shortcomings of two alternative schemes, the one-read scheme and the broadcast scheme.

4.1 The Cerberus Multiprocessor Simulator

Cerberus was originally developed by Brooks, Darmohray and Axelrod [5]. It is a scalable, general-purpose shared memory multiprocessor simulator on which to develop and benchmark parallel algorithms. The Cerberus machine is composed of autonomous RISC processors connected to a shared memory through a packet-switched interconnection network. The functional units of each CPU are fully pipelined, including accesses to shared memory.

The Cerberus package contains complete compiler, assembler, loader and library support for the *virtual computer* called the Cerberus machine. The resulting software package and utilities model the UNIX programmer interface as faithfully as possible.

The processor instruction set for each Cerberus CPU was derived from that of the *Ridge 32*, a RISC architecture computer manufactured by the now defunct *Ridge Computers Inc.* of Santa Clara, California. A number of important constraints had to be satisfied by the instruction set, including but not limited to:

- Suitability of the instruction set for a *fully pipelined* processor timing model.
- Load/store operations that were cleanly separated from the computation operations, required to give an optimizing compiler the ability to schedule memory and computation operations to mask memory latency.
- A minimum of unused processor state that must be updated as each instruction is

executed. By unused state, we refer to the condition codes of a processor that are typically updated by each instruction but only used for conditional branches.

- Fixed instruction formats. This reduced the instruction decode in the simulator to a single **C** switch statement.
- An absolute minimum number of instructions. The more ways there are to do a particular operation with the instruction set, the greater the support that needs to be built into the compiler.

Cerberus is very valuable in that detailed execution statistics can be obtained without artificially perturbing execution. For example, timing statistics can be obtained without taking any simulated time. Also, Cerberus enables us to explore a pipelined architecture with numbers of processors not otherwise available.

A number of modifications were made to original Cerberus in order to implement our cache coherence scheme:

- Simulation codes were written for the cache and memory controller. The protocols described in the last section were faithfully modeled in these codes.
- All memory requests were directed to the cache, instead of the processor-memory interconnect. Likewise, the memory controller was made to serve as a buffer between the interconnect and the memory.
- The interconnection network was modified to handle multicasting. Routing tags were altered to support this feature.
- Wormhole cut-through routing was introduced to the interconnection network. Previously, each packet could be moved in one clock. Now, with large cache line sizes, a packet could take several clocks to transfer from one switchnode to another.
- System calls were now handled through a block of private memory in each processing node. The instructions LOADBUF, LOADBUFB, STOREBUF, and STOREBUFB were added to manipulate this block of private memory.
- The **bstats()** and **estats()** system calls were added to control the gathering of cache and memory statistics.

In order to run large simulations, we needed to be able to run the simulator in parallel on the BBN TC2000 multiprocessor. Since the TC2000 has no coherent shared memory cache support in hardware, caching of shared memory had to be done explicitly in software. This necessitated significant restructuring of the simulator code so as to be able to decouple the simulator to run efficiently in parallel on the TC2000. However, the functionality of the simulator remained constant throughout the structure modifications.

The following are some of the parameters associated with the Cerberus machine equipped with a coherent cache model:

- N is the total number of processing elements in the system.
- n is the order of the system, or the number of levels in the interconnect. Since 2x2 switchnodes were used exclusively in the simulations presented herein, $N = 2^n$ and $n = log_2(N)$.
- W stands for the "width" of a cache line, defined as the number of data bytes per cache line. W Can be any power of 2, with a minimum of 8.
- A is the associativity level of the cache. A can be 1, 2 or 4; an associativity level of 1 implies a direct-mapped cache.
- L is the "length" of the cache, measured in cache lines. The total number of lines in any cache is $A * L$. L can be any power of 2.
- G is the size of each cache group[5]. G must be a power of 2, since 2x2 switchnodes are being used. Also, G must be less than or equal to N.

4.2 The simulated codes: *gauss*, *psim*, *relax* and *flag*.

Gauss is a linear system solver that uses Gaussian elimination to solve a linear system of equations, a 3x3 example of which is given below:

[5]Setting G to 1 is virtually the same as using a full-directory scheme. The only difference is that in the cache group scheme the memory controller will *not* reset the location bit for a cache upon receiving a line-spill notification from that cache. This makes very little difference in terms of performance.

$$\begin{aligned} a^1_{11}x_1 + a^1_{12}x_2 + a^1_{13}x_3 &= b^1_1 \\ a^1_{21}x_1 + a^1_{22}x_2 + a^1_{23}x_3 &= b^1_2 \\ a^1_{31}x_1 + a^1_{32}x_2 + a^1_{33}x_3 &= b^1_3 \end{aligned}$$

The *reduction* phase of the code reduces the matrix to upper triangular form:

$$\begin{aligned} a^1_{11}x_1 + a^1_{12}x_2 + a^1_{13}x_3 &= b^1_1 \\ a^2_{22}x_2 + a^2_{23}x_3 &= b^2_2 \\ a^3_{33}x_3 &= b^3_3 \end{aligned}$$

The ***back substitution*** phase of the code then obtains a solution, element by element. In the first step of the back substitution, the last element of **x** is solved using the last equation and the equations above it are simplified by substitution. This exposes another element of **x** to direct solution, followed by another substitution of its value into the equations above it. This process continues until all **x** elements have been solved. For a more detailed description of the algorithm, see [12].

Psim is the network simulator upon which the Cerberus multiprocessor simulator is based. It is capable of modeling a vast variety of network sizes and topologies. *Psim* will have each of its processors fetch a number of memory words from consecutive memory locations, starting with some random location. The parameters of a *psim* run are:

- The *base* and *order* of a system, symbolized by k and n, respectively. The base of a system is the fan-out of its switchnodes. The order is the number of stages in the interconnect. There are k^n processors in a system.
- The vector length (v) is the number of words fetched from memory by each processor.
- The stride (s) is the stride of the memory accesses.
- The buffer length (b) is the number of slots in each of k^2 buffers in a switchnode.

Relax is an iterative relaxation code. The problem space is represented as a set of discrete elements, and at each iteration each element is recalculated as a function of itself and its nearest neighbors. For our particular code, we averaged each element with its eight nearest neighbors, as illustrated in Figure 23. This is called a *nine-point stencil.* Iterative relaxation is used for many algorithms, among them the calculation of capacitance[13] and ocean circulation modeling.

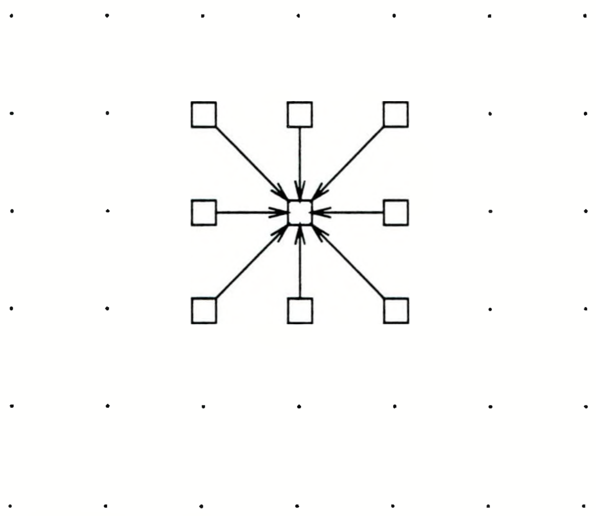

Figure 23: Nine point stencil used for iterative relaxation.

Relax requires N, the number of processors, to be a perfect square; we used 4 and 16 when we tested. The processors are tiled over the domain in such a way that each gets to compute an equal number of elements. We also took the cache group scheme into account when we decomposed the domain space as shown in Figure 24; we tried to get members of a cache group to share data with each other to enhance invalidation efficiency[6]. Ten iterations are performed by each run of *relax.*

We used another test code called *flag* to test the effectiveness of our scheme when many one-to-many or one-to-all invalidates are issued. In *flag*, an array of shared integers is accessed by all processors; processor 0 will set the 0^{th} element of the array while the other processors spin on it, then processor 1 will set the 1^{st} element of the array while the rest

[6]In fact, the method of tiling proved to make very little difference in the timing or traffic of *relax.* An arbitrary tiling gave very similar results.

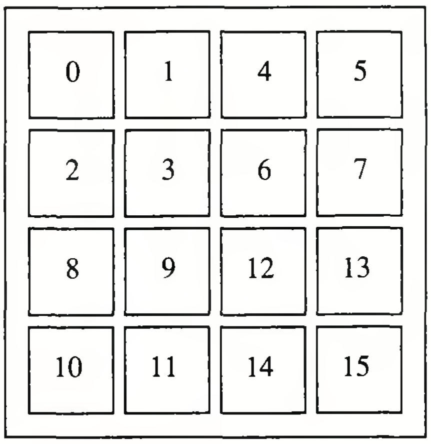

Figure 24: "Smart" domain decomposition used for 16-cpu *relax* runs.

of the processors spin on it, and so on. This results in a relatively high rate (40% - 50%) of one-to-many invalidates being issued. The *flag* code is actually the *gauss* code with the calculation portions stripped from it; it is pure synchronization code.

Gauss is a good example of a code in which there is a lot of locality, and which naturally decouples for nice parallelization. *Psim* also decouples to a certain extent, but there is enough data sharing occurring to prevent the high cache hit rates achieved by *gauss*. *Relax* is a code that exhibits a certain pattern of memory accesses, and is representative of many applications. The one-to-one invalidation rate of *flag* is much lower than that of the other codes. This makes *flag* useful for testing the effect of changing cache group size, and for ascertaining the effectiveness of multicasting and return reply combination.

4.3 Results of Cache Grouping vs. Full-directory scheme

One of the first things that we wanted to test was the efficiency of the cache grouping scheme versus that of a full-directory scheme. Certainly cache grouping uses less memory to track cache line location; we wanted to ascertain whether the inability of the cache group scheme to explicitly track every outstanding copy of a cache line would hurt the performance of the system (relative to a full-directory scheme). Also, we wanted to test the individual effectiveness of both multicasting and return reply combination.

In the *psim* and *gauss* tests run for this section, a group size of 8 (with a 32-CPU system) was used to represent the cache group scheme. Eight was chosen since it would divide the 32 caches into 4 groups, which would mean that 4 bits would be used to partially encode each cache line while in PRESENT* state. Since 5 bits are necessary to exactly encode a cache while in PRESENT1 or PRESENTM states, these 4 bits used for partial encoding are "free." The memory needed to track cache line location is therefore bounded by $log_2(N)$. A group size of 1 was used to represent the full-directory scheme. Neither multicasting nor return reply combination were used with the full-directory scheme.

Invalidation traffic[7] is measured by the number of invalidation messages that actually reach the CPUs; the way in which they are routed there (i.e. multicasting or some other method) is not taken into consideration.

The first test was on a *gauss* run solving a 128x128 matrix. Each Cerberus CPU was given a 256-Kbyte cache (W=16, L=8192, A=2). In Table 3, we show the results of the *gauss* code simulated on a Cerberus machine with 32 processors.

System configuration	normalized time	normalized inv traffic
group size = 1	1.000	1.000
group size = 8	1.001	1.001
group size = 8, multicasting	1.000	1.019
group size = 8, return reply comb	1.000	1.000
group size = 8, multicasting, return reply comb	0.996	1.018

Table 3: Effects of innovations on *gauss*-128 over 32 PEs, 256K cache

For this particular example, run-time and invalidation traffic are not significantly affected by switching from the full-directory scheme to the cache group scheme. The cache hit rate was about 94% in each run. We also tried the same suite of tests over Cerberus with a 64K cache (W=16, L=2048, A=2). The results of these simulations are shown in Table 4.

The decreased cache hit rate for the 64K cache (about 69%) had a relatively large effect on the amount of invalidation traffic, but the run-time of the cache group scheme still

[7]This traffic measure monitors messages from the memory controller to the cache. Return reply combination cuts down on the return traffic from the cache to the memory controller. Hence, invalidation traffic as we've defined it here will not be directly affected by the use of return reply combination.

System configuration	normalized time	normalized inv traffic
group size = 1	1.000	1.000
group size = 8	1.005	1.038
group size = 8, multicasting	0.999	1.049
group size = 8, return reply comb	1.000	1.041
group size = 8, multicasting, return reply comb	0.994	1.045

Table 4: Effects of innovations on *gauss*-128 over 32 PEs, 64K cache

did not suffer as compared to the full-directory scheme. This is due to the fact that the Cerberus interconnection network, with its k^2 buffers per switchnode, is able to provide a high amount of bandwidth and can easily handle the added traffic without adversely affecting the performance of the CPUs.

We ran the same sorts of simulations using *psim*. We simulated *psim -nkv* 6 2 64 (a 64 processor network, fetching a vector of length 64) on a 32-processor Cerberus machine with a 256K cache (W=16, L=8192, A=2). In Table 5, we show the results of these simulations.

System configuration	normalized time	normalized inv traffic
group size = 1	1.000	1.000
group size = 8	1.024	1.686
group size = 8, multicasting	1.016	1.686
group size = 8, return reply comb	1.021	1.687
group size = 8, multicasting, return reply comb	1.005	1.684

Table 5: Effects of innovations on *psim -nkv* 6 2 64 over 32 PEs, 256K cache

The advantages of multicasting and return reply combination became more apparent with the *psim* data. The low cache hit rate of the simulated *psim* run (34%) reflected an increase in coherence traffic generated relative to the gauss runs. Raising the group size from 1 to 8 caused some spurious read invalidations, and further increased the amount of invalidation traffic.

Once again, the network was able to absorb the extra traffic and still maintain efficiency.

When multicasting and return reply combination were turned on, the run time was only one half of one percent slower than that observed with a full-directory scheme.

The same battery of tests was run for *relax*, using 16 PEs to iterate over a 256x256 matrix of elements. The results are shown in Table 6. A cache group size of 4 was used to represent our scheme, for the same reason that a cache group size of 8 was chosen for the above 2 tests. Once again, there is virtually no difference in performance between the full-directory scheme and our scheme.

System configuration	normalized time	normalized inv traffic
group size = 1	1.000	1.000
group size = 4	1.000	1.549
group size = 4, multicasting	1.000	1.549
group size = 4, return reply comb	1.000	1.548
group size = 4, multicasting, return reply comb	1.000	1.548

Table 6: Effects of innovations on *relax* 256x256 over 16 PEs, 256K cache

We show in tables 7 and 8 that multicasting and return reply combination can be highly effective for codes that exhibit a high rate of one-to-many invalidates, such as *flag*. For this particular example, the performance of the cache grouping scheme not only equals but *betters* that of a full-directory scheme. The reason for this is that the one-to-many invalidates so typical of the *flag* code are handled much more efficiently by multicasting than by point-to-point invalidation; there are very few "useless" invalidates. Note that the effects are more pronounced in the larger (128-PE) system.

In order to measure the effectiveness of return reply combination, we use the two metrics *reply hits* and *reply misses*. Any time two return replies combine in the network, it is counted as one reply hit. Reply hits are a good measure of the effectiveness of return reply combination. Any time that two return replies are in the same switchnode, but fail to combine because they do not reach their respective buffer heads at the same time, it is counted as one reply miss. Reply misses give us a good idea of the performance improvement that would result from the implementation of multi-level combination in the switchnode. In the simulator we have developed, return reply combination is implemented with only top-level combination.

Note from the *flag* results that multicasting and return reply combination work best when they are used together. Multicasting causes invalidate requests to arrive at their respective caches at roughly the same time, which means that the resulting return replies are much more likely to combine on the way back to the memory controllers. In the 128-PE run, return reply combination by itself resulted in 37587 reply hits and 9485 reply misses. When return reply combination was aided by multicasting, it resulted in 85175 reply hits and 686 reply misses. When multicasting is employed, one-deep reply combination appears to be quite sufficient.

System configuration	normalized time	normalized inv traffic
group size = 1	1.000	1.000
group size = 4	1.012	1.022
group size = 4, multicasting	0.942	1.026
group size = 4, return reply comb	0.976	1.017
group size = 4, multicasting, return reply comb	0.904	1.034

Table 7: Effects of innovations on *flag* over 16 PEs.

System configuration	normalized time	normalized inv traffic
group size = 1	1.000	1.000
group size = 32	1.010	1.102
group size = 32, multicasting	0.989	1.142
group size = 32, return reply comb	0.940	1.106
group size = 32, multicasting, return reply comb	0.874	1.130

Table 8: Effects of innovations on *flag* over 128 PEs.

The results of the tests shown in this section indicate that, within the bounds of our simulations, the cache group scheme can equal or better the performance of a full-directory scheme without incurring its undesirable memory expenses. When a high degree of one-to-many data sharing is exhibited, as was the case in the *flag* code, then return reply combination and multicasting are very effective in boosting the performance of the system.

One expects the positive impact of return reply combination and multicasting to improve as the size of the system grows.

4.4 Results of Cache Grouping vs. No Cache

The implementation of hardware support for cache coherence requires a certain amount of expense. This section addresses the question of whether the added performance warrants the expense of such support. Does a coherent cache greatly improve performance, or does the coherence traffic bring a machine to a standstill? Also, we are interested in checking whether our coherence scheme continues to enhance performance as the number of processors grows (i.e. whether it is *scalable*).

We first ran some simulations of the *gauss* code performing a 128x128 linear system solution. The simulations were run on original Cerberus (hereafter referred to as uncached Cerberus) and on Cerberus equipped with a coherent shared memory cache (or cached Cerberus). The cache configuration for these runs is W=16, A=2, L=8192. Group size was selected as the smallest group size such that the number of groups was less than or equal to $log_2(N)$.

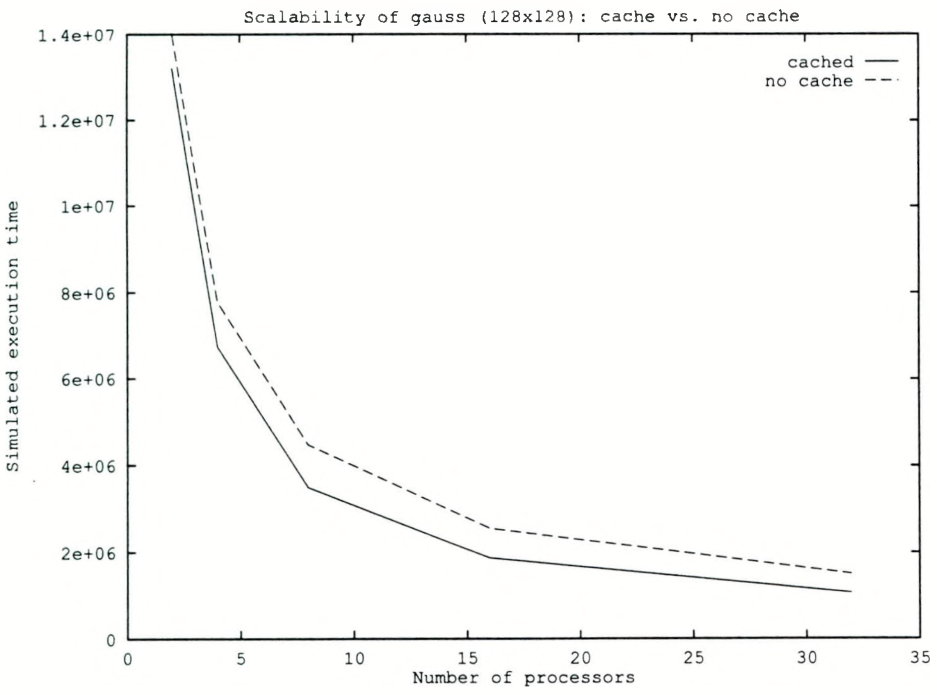

Figure 25: *Gauss* timings for uncached Cerberus vs. cached Cerberus.

In Figure 25, we show the simulated run-time of 128x128 *gauss*, comparing cached Cerberus to uncached Cerberus. The FLOP rates for the two machines are shown in Figure 26. As the number of processors increases, so does the advantage of a coherent shared memory cache. The primary reason for this is that as the system gets larger, the processor-memory interconnect gets deeper. Uncached Cerberus begins to take a long time to ship data back and forth across the interconnect. The relatively high cache hit rate (93% - 98%) of this code allows the cache in the cached version to save much of the expense of shared memory accesses. As the interconnect gets deeper, this savings increases.

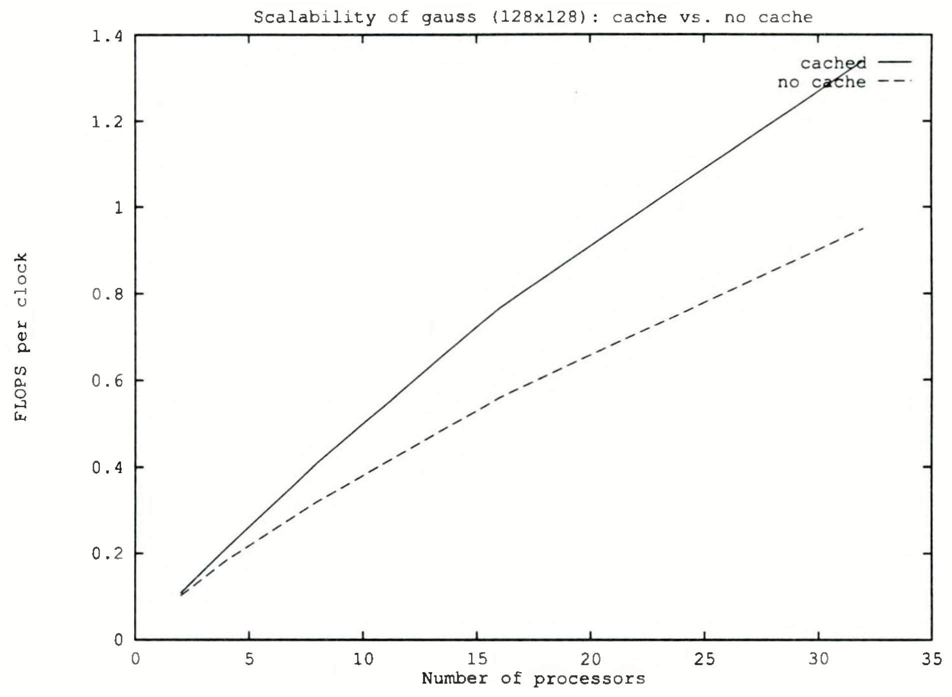

Figure 26: *Gauss* FLOP rate for uncached Cerberus vs. cached Cerberus.

Psim also scaled favorably. A *psim -nkv* 7 2 64 run was simulated over 2, 4, 8, 16, and 32 processors. In Figure 27, we show the simulated run-times that resulted, and the time improvement of these runs ($1 - \frac{cached\ time}{uncached\ time}$) is shown in Table 9. Once again, our cache group scheme scales nicely. Time improvement relative to the uncached machine tapers off only as the asymptotic concurrency limit of the benchmark is reached.

The improvements shown by cached Cerberus over uncached Cerberus for the *psim* runs occur for the same reasons as for the *gauss* runs. As the network gets deeper, the cache saves more and more memory latency time.

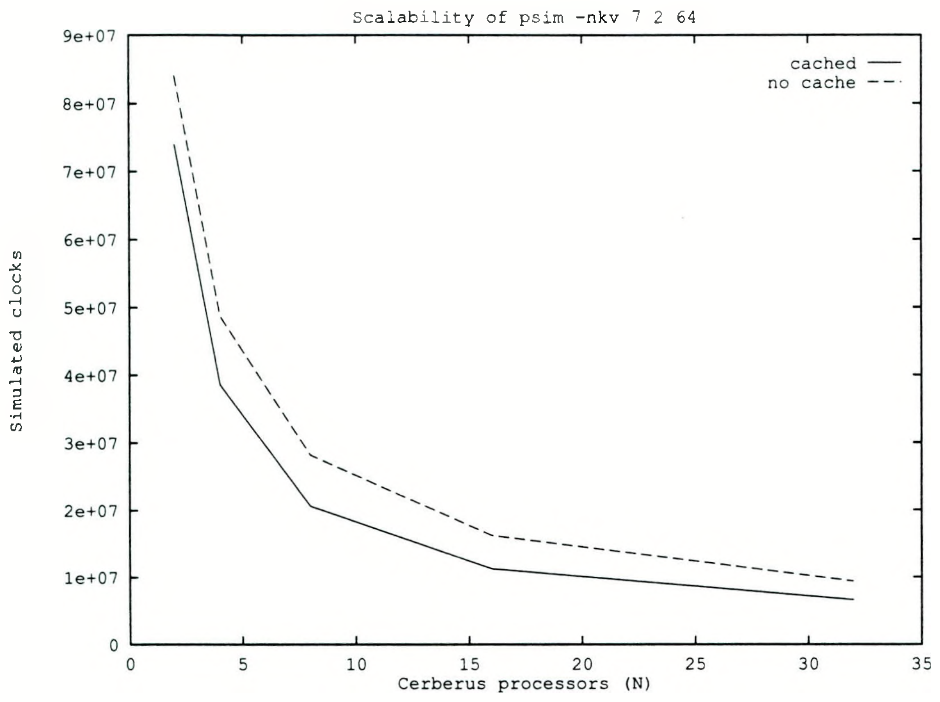

Figure 27: *Psim* timings for uncached Cerberus vs. cached Cerberus.

Simulated Processors (N)	uncached clocks	cached clocks	improvement
2	84080451	74020496	12%
4	48695298	38611226	21%
8	28148302	20589724	27%
16	16188350	11277313	30%
32	9384505	6597257	30%

Table 9: Effects of scaling on *psim -nkv* 7 2 64, 256K cache

The iterative relaxation code (*relax*) was tested in the same manner. The results are shown in Table 10. Once again, the code adapts better to scaling on the cached machine than on the uncached machine.

The test results in this section have shown that a machine equipped with a coherent cache will indeed outperform a machine that is not so equipped. More importantly, the performance gap widens as the number of processors grows. This leads us to believe that a coherent shared memory cache would be very beneficial on any "massively" parallel machine. In this case, large coherent caches are effectively used as automatic local memories.

N	uncached clocks	cached clocks	improvement
4	25970530	24607972	5%
16	7042661	6234715	11%

Table 10: Effects of scaling on 256x256 relaxation.

4.5 Effects of Changing Cache Group Size

One of the advantages of the cache group scheme is that the memory required to track cache line location can grow as $log_2(N)$, where N is the number of processors, *if the cache group size is sufficiently large.* However, increasing the cache group size also increases the number of caches that will be hit by "useless" invalidates during multicasts. Does increasing the cache group size have a significant effect on the performance of a program?

One important aspect of the cache grouping scheme is that one-to-one cache line sharing is supported in an efficient manner. That is to say, when a cache line is granted in a PRESENTM or PRESENT1 state, the memory controller has the ability to exactly track which cache holds the line. No multicasts are needed to invalidate a line in either of the above states, and invalidation traffic is kept to a minimum. If a significant portion of the cache lines are granted on a one-to-one basis, then the multicasts necessitated by lines granted in PRESENT* mode will have a relatively small impact on performance and traffic.

Gauss and *psim* were both tested as to how they were affected by increasing cache group size. As usual, each of 32 simulated processors was equipped with a 256-Kbyte cache (W=16, L=8192, A=2). Multicasting and return reply combination were enabled. Cache group size was varied from the minimum 1 to the maximum 32.

In Table 11, we show the results from the *gauss* runs. Run-time was not significantly affected by increasing cache group size; invalidation traffic gets worse as some "useless" invalidates are issued as group size increases.

group size	normalized time	normalized inv traffic
1	1.000	1.000
2	0.997	1.049
4	0.997	1.107
8	0.997	1.206
16	0.997	1.347
32	0.997	1.691

Table 11: Effects of changing cache group size on 128x128 *gauss* over 32 PEs, 256K cache.

The cache hit rate for *gauss* was about 94%. In all cases, over 96% of the invalidations were performed on a one-to-one basis. This extremely high amount of one-to-one invalidations minimizes the impact of multicasts generated by one-to-many invalidations.

The results for similar runs of *psim*[8] are shown in Table 12. Around 91.7% of the invalidates were one-to-one in all runs. The cache hit rate was around 33.8% in all runs. The invalidate traffic went fairly high as group size went up, but the simulated time only went up 2%. The traffic did not come close to flooding the network, so performance was not severely impeded.

Neither *psim* nor *gauss* were significantly affected by changing the cache group size. Both had one-to-one invalidate rates somewhere above 90%. *Relax*, on the other hand, exhibited a one-to-one invalidate rate of about 80%. It had a much better chance of showing some timing fluctuations due to changing cache group size.

We tested *relax* of a 256x256 element matrix over 16 simulated processors, trying cache group sizes of 1, 2, 4, 8 and 16. Eight-byte cache lines were used, in order to cut down on false sharing; false sharing involves two caches accessing different bytes within the same

[8]It should be noted that cache group sizes of 16 and 32, for a 32-cpu system, would not be used in practice. A group size of 8 results in 4 groups, which can be partially encoded into 4 bits. These 4 bits are "free" in a 32-bit system, since 5 bits are necessary for exact encoding. Any larger granularity *may* adversely affect performance but *will not* lower cost.

group size	normalized time	normalized inv traffic
1	1.000	1.000
2	1.001	1.138
4	1.002	1.341
8	1.005	1.684
16	1.010	2.270
32	1.021	3.128

Table 12: Effects of changing cache group size on *psim -nkv* 6 2 64 over 32 PEs, 256K cache

cache line. We wanted the one-to-one invalidate rate to be as low as possible, and false sharing raises that rate. The number of lines in each cache, L, was raised to 16384, in order to maintain the standard total cache size of 256 Kbytes.

The results of the iterative relaxation cache group size tests are shown in Table 13. Even at the relatively low 80% one-to-one invalidation rate, varying the cache group size made very little difference to the overall performance of the program.

group size	normalized time	normalized inv traffic
1	1.000	1.000
2	1.000	1.192
4	1.000	1.549
8	1.000	2.139
16	1.002	3.287

Table 13: Effects of changing cache group size on 256x256 relaxation over 16 PEs, 8-byte cache line.

We show in Table 14 that increasing the cache group size does not adversely affect the performance of the *flag* code. In fact, a beneficial effect is observed. When the lion's share of the invalidates are either one-to-one or one-to-all, then very little is lost in the way of system performance when larger cache groups are employed. In fact, large cache groups allow the one-to-all invalidates to be performed in a much more efficient manner. We included reply

group size	normalized time	normalized inv traffic	reply hits	reply misses
1	1.000	1.000	30788	8073
2	0.948	1.019	43826	1778
4	0.922	1.056	53003	230
8	0.942	1.066	57745	456
16	0.937	1.081	67103	295
32	0.936	1.129	85175	686
64	0.941	1.216	116765	1352

Table 14: Effects of changing cache group size on *flag* over 128 PEs.

hit and reply miss measures in Table 14 to make the following points:

- There is a good amount of combination occurring in the reply network; top-level return reply combination is effective.
- It appears that there is little to be gained through the implementation of multi-level return reply combination.

The results from this section lead us to believe that for most real codes, cache group size could be set to whatever is convenient to the hardware of the system. In codes where the one-to-one invalidation rate is high, only a small amount of multicasts occur, and so system performance is not damaged. In codes such as *flag* where there is a high rate of one-to-many sharing, multicasts to large groups tend to enhance the overall performance of the system.

4.6 Notes on other cache coherence schemes

4.6.1 The *one-read* scheme

Given the results above, one may question the necessity of ever having multiple readable copies outstanding. With the 80-95% one-to-one invalidation rate that seems to be prevalent in most codes, could we not greatly simplify the system by allowing only one readable copy to be out at a time? For lack of a better name, we call this the "one-read" scheme. It would assuredly cut down on the intelligence needed for both the memory controllers and the switchnodes. The memory controllers would not have to handle cache groups or multicasts; the switchnodes would not have to handle multicasting or return reply combination. Could such a system be implemented without significant loss of performance?

In Figure 28, we graphically show that the answer is no; the one-read scheme severely impedes performance. The one-read scheme is not only significantly worse than the cache group scheme, it does much worse than *no cache at all.* In Table 15, we show some particular performance measurements that illustrate the undesirability of the one-read scheme. Note that traffic is measured in millions for the one-read scheme, and thousands for the cache group scheme. The effective critical region produced by the one-read scheme kills the performance of the system.

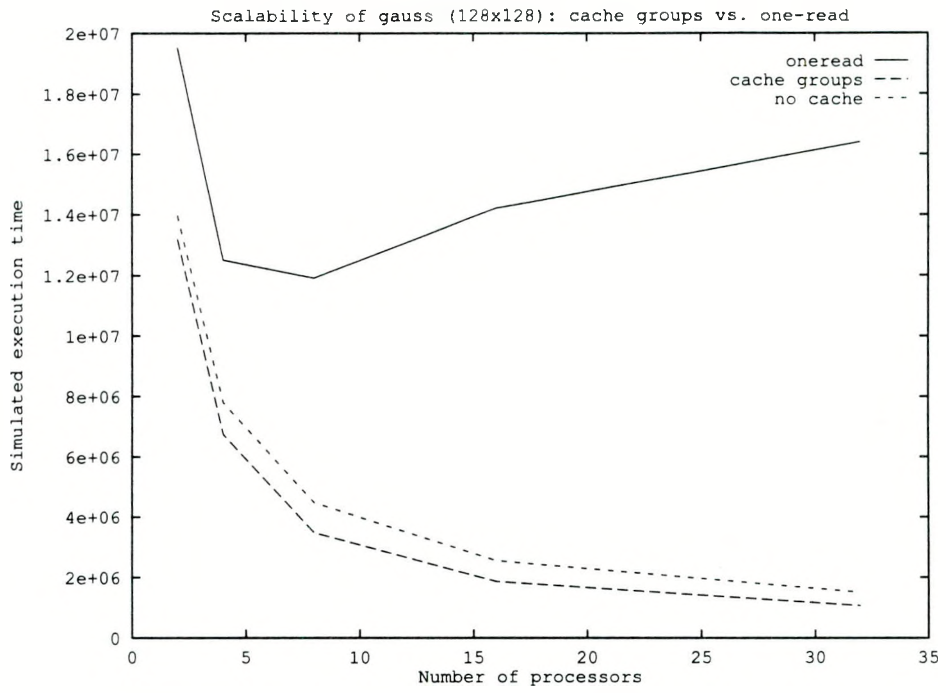

Figure 28: Cache groups vs. one-read for 128x128 *gauss*.

What would cause such an increase in network traffic? There are three categories of data use that cause the one-read scheme to fail, examples of which can be found in Figure 29 [12], in which we show a code fragment from the *gauss* benchmark:

- **Write-once read-many data**: The shared variable *dims* (lines 1, 6 and 11) is written once at the beginning of the program, and read many times thereafter, by every processor. After each processor does an iteration of the k, i or j loop, the loop

	cache group scheme			one-read scheme		
		cache	total		cache	total
N	clocks	hit rate	invalidation traffic	clocks	hit rate	invalidation traffic
2	13.2M	0.98	42K	19.5M	0.45	12.3M
4	6.7M	0.98	48K	12.6M	0.36	15.1M
8	3.4M	0.97	53K	12.1M	0.33	16.1M
16	1.9M	0.96	59K	14.2M	0.29	17.7M
32	1.1M	0.93	70K	16.4M	0.30	18.8M

Table 15: Performance of 128x128 *gauss* code : cache group scheme vs. one-read scheme.

variable must be compared to *dims*. This causes an enormous amount of invalidate traffic to be generated, since each processor must wait to get its own copy of *dims*. If multiple readable copies of such variables are allowed, they can reside permanently in the cache.

- **Synchronization data**: Line 2 causes a processor to wait until the next pivot row has been stabilized. Once again, this will generate a large amount of traffic; while the pivot row completes its operations the network will be flooded by read requests for *flag[k]*. The cache group scheme allows a processor to loop on such a variable in cache memory; when it is finally modified the caches will be updated. The one-read scheme forces all such spin waiting to be done over the interconnect.

- **Other widely shared data**: The j-loop (line 11) has all processors referencing the pivot row for their calculations. With only one copy of any of the elements out at a time, performance is once again severely wounded. The situation is even worse than no cache at all, since multiple readable copies of a piece of data can exist in the cacheless system.

In order to get reasonable performance out of the one-read scheme, a significant amount of software modification would have to be performed on any code. This defeats the whole purpose of a hardware coherent cache mechanism, which is to reduce software cost by supporting implicit use of data locality. We are convinced that the one-read scheme is not a viable hardware option. *The capability to grant multiple read-only copies of a cache line is absolutely essential in any parallel machine with a coherent shared memory cache.*

```
for(k=0; k<dims; k++) {
  while(flags[k]==0); /* wait for the pivot row to be stable. */
  /* Custom forall loop which makes sure that the same processor
  handles the SAXPY on a given row. */
  for(i = k + 1 + (_TINDEX + _TSIZE - (k % _TSIZE)) % _TSIZE;
      i < dims; i += _TSIZE) {
    double temp = A[i][k];
    if(temp == 0.0) continue;
    A[i][k] = 0.0;
    temp /= A[k][k];
    for(j=k+1; j<dims; j++) A[i][j] -= A[k][j] * temp;
    B[i] -= B[k] * temp;
    if(i == k+1) flags[i] = 1;
  }
}
```

Figure 29: Reduction loop of *gauss* code

4.6.2 The *minimal state* scheme.

The two-bit protocol, or minimal state scheme, of Archibald and Baer [4] requires no memory to track cache line location. Instead, it does full broadcasts every time a coherence action needs to be performed. In terms of memory expense. the minimal state scheme is indeed very scalable. Would the frequent broadcasts hurt the performance of a large scale multiprocessor?

We did no direct testing of the minimal state scheme. However, the group size variation tests that we ran might provide some insight into the viability of frequent broadcasts. Recall Table 16 from the *psim* group size variation test. A group size of 32 implies a full broadcast whenever any type of one-to-many sharing is encountered. Approximately 92% of all invalidations were one-to-one; this means that 8% of the invalidations were handled by full broadcasts. If the minimal state scheme were implemented, then 100% of the invalidations would be handled by full broadcasts. One would therefore intuitively expect the performance hit relative to the full-directory scheme to be about 12 times worse than it was with the cache-group scheme. This would mean a 25% performance lag from the full-directory scheme, which is significant.

group size	normalized time	normalized inv traffic
1	1.000	1.000
2	1.001	1.138
4	1.002	1.341
8	1.005	1.684
16	1.010	2.270
32	1.021	3.128

Table 16: Effects of changing cache group size on *psim -nkv* 6 2 64 over 32 PEs.

There may be other ill effects from the minimal state scheme, and they would only get worse as one grew the number of PEs in the system. Surely a "massively" parallel machine would suffer from frequent full broadcasts. *It is essential for any coherent shared memory system to be able to explicitly track cache lines in the case of one-to-one data sharing.*

4.7 Summary of Simulation Results

The cache grouping scheme, while much less expensive than a full-directory scheme, can equal or better a full-directory scheme in terms of system performance. The ability to explicitly record the location of a single cache line in the case of one-to-one sharing is necessary, as is the ability to grant multiple readable copies of a cache line.

A system that employs cache grouping outperforms a similar system with no cache; the invalidation traffic is not so great that it bogs down the interconnect. The system seems to be insensitive to the size of cache groups. For codes that exhibit a high rate of one-to-many data invalidates, system performance is aided significantly through the use of multicast and combining features in the interconnect. Multicasting and return reply combination are most effective when used together. Top-level return reply combination works very effectively for such codes; there appears to be little to gain through the implementation of multi-level return reply combination.

5 Software Costs of Coherence Enforcement

A large percentage of funds spent on computing go towards software development and maintenance. At Lawrence Livermore National Laboratory, hundreds of millions of dollars are spent on computing each year, and it is estimated that 80% of that figure goes toward software. In this section, we focus on the amount of time, effort and coding required to attain high performance for codes that are run on a large-scale parallel machine without a hardware-enforced cache coherence mechanism. Many of these expenses could be avoided, for *all* codes, if a coherent shared memory cache were present.

Recently, there have been efforts to run *psim* and *gauss* on a 63-cpu BBN TC2000, which is not equipped with a coherent shared memory cache. Rather, the user must explicitly manage the caches and insure data coherence. This section deals with the effort it took to get these codes to run efficiently, in parallel, on the TC2000.

In order to make valid observations about speedup and efficiency, two symbols here need to be defined:

- T_S - "Serial Time" - The time it takes to run a code on one processor, using only private, copy-back cached memory. This is a good indicator of what the single-processor performance is for the machine.
- T_n - "n-way parallel time" - The time to run in parallel over n processors, all memory references going to shared memory unless explicitly routed otherwise by the programmer. Ideally, $T_n = \frac{T_S}{n}$ and $T_1 = T_S$.

5.1 The *psim* network simulator on the TC2000

Picano, Brooks and Hoag [14] did an in-depth study describing their efforts to run *psim* efficiently on the TC2000. A number of modifications were made to the code to place data in local memory. These modifications were done in phases, and performance results were gathered after each phase.

The Phase 0 parallel code was written for a shared memory multiprocessor with a coherent shared data cache; all simulation work was done in shared interleaved memory. 50 lines were added to or modified from the serial code to produce the Phase 0 parallel code. This version of the code had been run with very good results on the Sequent and Alliant multiprocessors; T_1 was very close to T_S on these machines for the Phase 0 code. The Phase

0 code performed very badly on the TC2000. however; T_1 was about $8 * T_S$. The Phase 0 version of the code was 1501 lines long. The reason for this poor performance was the lack of cache support for shared memory.

In the Phase 1 parallel code, we streamlined the structures that held the simulated switchnodes, cpu ports and memory ports, so that these structures could be safely cached by a single PE for the duration of a run. All portions of the switchnode structure. for instance, that had to be accessed by other switchnodes (for communications purposes) were put into separate shared data arrays. For example, see Figure 30 for the structure of a buffer; buffers are used in switchnodes, cpu ports and memory ports to handle packet collisions. The *flag* field was stripped out in Phase 1 and put into a separate array, since it might need to be accessed by more than one physical cpu.

```
struct BUFFER {
        PACKET *head;   /* Pointer to the head packet in the buffer. */
        PACKET *tail;   /* Pointer to the tail packet in the buffer. */
        int count;      /* The number of packets in the buffer. */
        BOOLEAN flag;   /* We move a packet if flag is TRUE. */
        lock_t access;  /* Lock for buffer access. */
};
```

Figure 30: BUFFER Data structure.

This enabled us to cache entire switchnode, cpu port, and memory port data structures for the duration of a run. We did not have to worry about flushing these structures out of the cache, since only the cache in which a structure resided would ever need to access that structure.

Since most of the work of the network simulator is done in the interconnection network (the switchnodes), this modification resulted in significant improvement in execution time. The Phase 1 code was 1814 lines long, and T_1 was reduced to about $2 * T_S$ for this version of the code on the TC2000.

In the Phase 2 code, we removed *extraneous sharing* wherever possible. *Necessary sharing* represents shared information flow between 2 or more **different** processing elements (PEs), which means that data must reside in shared memory in order to insure coherence.

Extraneous sharing represents information that is placed in shared memory. but accessed by only one PE. For an example network shown in Figure 31. necessary sharing is performed between stages 1 and 2, since packets are moved between two different TC2000 processors. There is extraneous sharing between stages 0 and 1. since all packet "movements" between these stages actually go from one TC2000 processor to itself[9]. The Phase 2 code was modified to detect this, and used private cached memory to handle information flow where the code would normally suffer from extraneous sharing.

This modification put T_1 very close to T_S on the TC2000. The Phase 2 code was 2224 lines long.

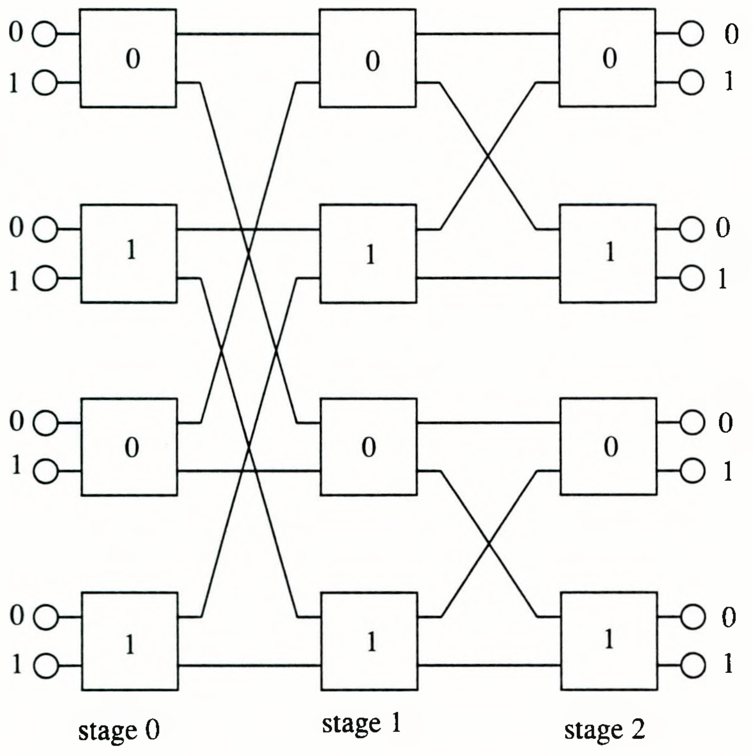

Figure 31: Mapping an 8-cpu *psim* run onto 2 TC2000 processors. The numbers represent the PE onto which each individual structure is mapped. The squares represent switchnode structures; the circles represent memory ports on one side and cpu ports on the other side.

[9]If the number of PEs is changed, such data might actually be shared between PEs. Extraneous sharing must be detected "on the fly."

In the Phase 3 parallel code, we replaced two shared memory accesses through the switch by one shared memory access through the switch and one local memory access when possible. For example, packet transfers in the network are simulated by having the simulated cpu, memory or switchnode that contains the packet write a pointer to it out to a shared memory location. The receiver of the packet then reads the packet pointer out of that shared memory location, on the same simulated clock, thus simulating a transfer. Phase 3 made the shared memory "communication" location local to the reader. Each packet movement now required approximately 2439 nsecs[15] (1889 remote write + 550 local read) instead of 3802 nsecs (1889 remote write + 1913 remote read).

This modification resulted in only a modest speedup on the TC2000, since Phase 2 already had T_1 very close to T_S. The Phase 3 code was 2327 lines long.

The effort to efficiently run *psim* on the TC2000 was a summer project for one student, with 2 other people closely collaborating and many more offering help. Phase 1, data structure streamlining, would be unnecessary with a coherent shared memory cache, since the cache-able portion of a structure would automatically be cached in such a system. Phase 2, extraneous sharing detection, would also not be necessary since a coherent shared memory cache would do all possible transactions in cache automatically. Therefore, Phase 2 (2224 lines for our *psim* example) performance would be attained by Phase 0 code (1501 lines) on a machine equipped with a coherent shared memory cache (as had been the experience on the Sequent multiprocessors possessing coherent caches). Such a machine would thereby place T_S very near T_1 for the 1500-line Phase 0 version of the code.

5.2 The *Gauss* linear system solver on the TC2000

The original serial *gauss* code was a relatively trivial program. The 22-line baseline parallel code (Figure 32), written with the aid of the Parallel C Preprocessor (PCP)[16], involved the addition or modification of 7 lines to the serial code. The modifications are shown in boldface in Figure 32. All of the *gauss* codes mentioned in this section are listed in Appendix A.

The performance of the serial code on the TC2000 is shown in Table 17. The serial code is the proper point of reference for parallel speedup measurement. The performance of the baseline parallel code on the TC2000 is also displayed in Table 17. It is evident that this version of the parallel code is seriously degraded by the cost of references to shared memory,

```
void dgauss(double **a, double *b, int dim) {
    for(int k = 0; k < dim; k += 1) { /* reduction outer loop */
        forall(int i = k+1; i < dim; i+= 1) {
            double temp = a[i][k];
            if(temp == 0.0) continue;
            a[i][k] = 0.0;
            temp /= a[k][k];
            for(int j = k+1; j < dim; j += 1) {
                a[i][j] -= a[k][j] * temp;
            }
            b[i] -= b[k] * temp;
        }
        barrier;
    }
    for(int i = dim - 1; i >= 0; i -= 1) { /* back substitution outer loop */
        master{
            b[i] /= a[i][i];
        }
        barrier;
        forall(int k = i - 1; k >= 0; k -= 1) {
            b[k] -= a[k][i] * b[i];
        }
    }
    barrier;
}
```

Figure 32: Baseline parallel *gauss* code.

since $T_1 \approx 9 * T_S$. An intolerable 11% efficiency is achieved for a 700x700 solution over 10 processors.

Dimension	$MFLOPS_S$	$MFLOPS_1$	$MFLOPS_{10}$	Eff_{10}
100	1.30	0.16	1.01	0.08
200	1.36	0.16	1.26	0.09
300	1.38	0.16	1.32	0.10
400	1.40	0.16	1.42	0.10
500	1.41	0.16	1.47	0.10
600	1.37	0.16	1.44	0.11
700	1.28	0.16	1.47	0.11

Table 17: Serial and baseline parallel code performance.

The code was further modified to tile the processors properly when running on a machine with a coherent shared memory cache. This version of the code insured that given rows were accessed by the same processor on every iteration, so that those rows could stay in cache. This new code was 44 lines long, and performed very well on the Alliant FX/8 and Sequent Symmetry parallel machines. It was written in such a way that it would do very well on any multiprocessor equipped with a coherent shared memory cache.

However, the 44-line code performed the same as the 22-line code on the TC2000. Since shared data is not automatically cached on the TC2000, the modifications did nothing to improve performance.

In order to achieve high performance on the TC2000, the *gauss* code was completely re-written. Shared data caching and explicit localization were explicitly handled in this version of the code, which was 106 lines long. Also, the code needed tuning to get rid of some memory hotspots that plagued its performance. Finally, the code performed as displayed in Table 18. (Recall that the speedup is measured against the serial performance shown in Table 17). A 69% efficiency rate is achieved for a 700x700 solution over 48 TC2000 processors, compared to 11% efficiency for similar runs with the 22-line and 44-line codes. An even higher 79% efficiency rate is achieved for the 1000x1000 solution over 48 PEs.

These relatively high efficiencies could be (*and have been*) attained by the 44-line code on machines equipped with a coherent shared memory cache, such as the Sequent Symmetry.

Dimension	$MFLOPS_{48}$	$Speedup_{48}$	Eff_{48}
100	3.40	2.6	0.05
200	10.47	7.7	0.16
300	18.85	13.7	0.29
400	27.65	19.8	0.41
500	31.34	22.2	0.46
600	38.32	28.0	0.58
700	42.25	33.0	0.69
800	44.74	35.0	0.73
900	46.96	36.7	0.76
1000	48.39	38.1	0.79

Table 18: Performance of 106-line code over 48 processors.

5.3 Discussion

It is clear from our experience with the *psim* and *gauss* codes that one pays a large software penalty for the lack of a coherent shared memory cache on a scalable multiprocessor. This penalty is paid in terms of the software effort that is necessary to achieve efficient parallel performance on such a machine. It is not simply a matter of typing in some extra code. It is a matter of gradually tuning a program to use local memory as much as possible, and making frequent checks to insure that the program still runs correctly. Efficient parallelization on a machine without a coherent shared memory cache is a tedious and onerous task.

Parallel code version	Line count	Efficiency
Baseline (pure shared memory)	22	11%
Coherent cache	44	11%
Explicit localization	106	69%

Table 19: 700x700 *gauss* solution performance over 48 TC2000 PEs.

The costs of hand-coded coherence are shown in Tables 19 and 20. The 44-line *gauss* code will run with the efficiency of the 106-line *gauss* code on any machine with a coherent cache. Likewise, the 1501-line Phase 0 *psim* code would run with the performance of the 2224-line Phase 2 code on such a machine. That represents a savings of 723 lines and many

Parallel code version	Line count	Efficiency
Phase 0	1501	$T_1 \approx 8 * T_S$
Phase 1	1814	$T_1 \approx 2 * T_S$
Phase 2	2224	$T_1 \approx T_S$
Phase 3	2327	$T_1 \approx T_S$

Table 20: *Psim* parallel code performance.

man-months of programmer effort.

A coherent shared memory cache would be a one-time expense for a scalable multiprocessor. Running without a coherent cache means that a significant amount of time and effort will be expended for *every* code that is ported to the machine. Given the tremendous expense that goes toward software, a coherent shared memory cache would end up saving a considerable amount of time and money during the useful lifetime of a given multiprocessor.

6 Discussion

We have shown through simulation that the cache grouping scheme is comparable in performance to the full-directory scheme, though the cache grouping scheme is much less costly. The memory required to track cache line location can be bounded by $O(log_2(N))$ for cache grouping, compared to $O(N)$ for a full-directory scheme. Detailed simulation showed that system performance was not sensitive to cache group size.

Multicasting and return reply combination were highly effective for codes that exhibited a relatively high rate of one-to-many invalidates. Also, they work better when they are used in concert. Top-level return reply combination appears to be effective, and it does not look as though there would be a lot of profit from the implementation of multi-level return reply combination. This is in sharp contrast to the results of NYU-Ultracomputer-style combination methods [6]. Our top-level reply combination scheme works well due to the simultaneous nature of multicasts; return replies leave their respective caches at about the same time, and so are very likely to meet in top-level combination in the network.

The implementation of a coherent cache, using the cache grouping scheme, showed a significant improvement over a similar system with no cache. More importantly, the improvement increased with the number of processors simulated, leading us to believe that the performance of "massively parallel" machines of 1000 or more processors would be greatly improved by the addition of a coherent shared memory cache.

We also looked into some alternative coherence schemes, namely the one-read scheme and the broadcast scheme. The one-read scheme performed very poorly due to the amount of traffic it generated. The broadcast scheme failed for the same reason. We find that two capabilities are essential for any scalable coherence scheme: the ability to exactly track single cache line location for one-to-one data sharing, and the ability to grant multiple readable copies of a cache line.

We have shown moderate increases of performance of up to 30% with the addition of a coherent cache. It should be noted that our simulations were limited to what would be fairly small runs on a "real" machine; time would not permit larger simulations. We did, however, show that the improvement due to cache improved as the number of processors was increased. Trends also indicated improved performance with *larger problem size.* Therefore, we could expect a greater improvement from the cache running a relatively large "real" problem on a large number of processors.

Increased system performance, however, is not the only advantage offered by a coherent shared memory cache. The Lawrence Livermore National Laboratory spends hundreds of millions of dollars every year on software development. Our experience with scalable shared memory multiprocessors without coherent shared memory caches is that a significant amount of software effort is required to shape codes to run efficiently in parallel [14]. A significant amount of explicit localization of memory and decoupling of data structures is typically required. A coherent shared memory cache, supported in the hardware, would greatly cut down on the software costs of efficiently parallelizing existing code. The added hardware to support such a cache need not be overly expensive. The added performance of a coherent shared memory cache, as well as the decreased software development time on such a machine, would be well worth the hardware costs associated with it.

References

[1] Siegel, H.J., *Interconnection Networks for Large-Scale Parallel Processing.* Lexington Books, D.C. Heath and Company, Lexington, Massachusetts, 1985.

[2] Dubois, Michel, Christoph Scheurich, and Faye Briggs, "Memory Access Buffering in Multiprocessors," *Proceedings of the 13th Annual Symposium on Computer Architecture*, Vol. 14, Num. 2, ACM, pp. 434-442, June 1986.

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

[4] Archibald, James and Jean-Loup Baer, "An Economical Solution to the Cache Coherence Problem", *Proceedings of the 11th International Symposium on Computer Architecture*, SIGARCH Newsletter, Vol. 12, Num. 3, IEEE, pp. 355-362, June 1984.

[5] Brooks, E.D. III, T.S. Axelrod and G.A. Darmohray, "The Cerberus Multiprocessor Simulator." In G. Rodrigue, editor, *Parallel Processing for Scientific Computing*, pp.384-390, SIAM, 1989.

[6] Almasi, George S. and Allan Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings Publishing Co., 1989, pp. 434-441.

[7] Baer, J.-L. and C. Girault, "A Petri Net Model for a Solution to the Cache Coherence Problem", *Proceedings of the First International Conference on Supercomputing Systems, IEEE*, pp. 680-689.

[8] Gupta, Anoop, Wolf-Dietrich Weber and Todd Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," *Proceedings of the 1990 International Conference on Parallel Processing*, Vol. I, Penn State University Press, pp. 312-321, 1990.

[9] Lenoski, D., J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, M. Horowitz and M. Lam. "Design of Scalable Shared-Memory Multiprocessors: The DASH Approach." In *Proceedings of COMPCON '90*, pp. 62-67, 1990.

[10] Davis, H., S. Goldschmidt and J. Hennessy. "Tango: A Multiprocessor Simulation and Tracing System." Stanford Technical Report – in preparation, 1989.

[11] Brooks, E.D. III, "A Butterfly Processor-Memory Interconnection for a Vector Processing Environment". *Parallel Computing.* Volume 4, 1987.

[12] Darmohray, Gregory A. and Eugene D. Brooks III, "Gaussian Techniques on Shared Memory Multiprocessor Computers," Lawrence Livermore National Laboratory Tech. Rep. UCRL-97939, Livermore, CA, January 1988.

[13] Fox, Geoffrey C. and Steve W. Otto, "Algorithms for concurrent processors," *Physics Today*, pp. 50-59, May 1984.

[14] Picano, Silvio, Eugene D. Brooks III, and Joseph E. Hoag, "MIMD Implementations of a Network Simulator on a Large Scale, Shared Memory Multiprocessor," Lawrence Livermore National Laboratory Tech. Rep. UCRL-JC-105468, Livermore, CA, November 1990.

[15] BBN Advanced Computers Inc., *Inside the TC2000*, Cambridge, MA 1989.

[16] Brooks, Eugene D. III, "PCP: A Parallel Extension of C that is 99% Fat Free," Lawrence Livermore National Laboratory Tech. Rep. UCRL-99673, Rev. 1, Livermore, CA, September 1989.

A Gaussian elimination code listings

This appendix details the various gauss codes that were mentioned in section 5.

A.1 Baseline version of *gauss* code

The following is the baseline parallel code; the lines in bold face were the additions/modifications necessary to produce this code from the serial code:

```
void dgauss(double **a, double *b, int dim)
{
    /* reduction outer loop */
    for(int k = 0; k < dim; k += 1) {
        forall (int i = k+1; i < dim; i +=1) {
            double temp = a[i][k];
            if(temp == 0.0) continue;
            a[i][k] = 0.0;
            temp /= a[k][k];
            for(int j = k + 1; j < dim; j +=1) {
                    a[i][j] − = a[k][j] ∗ temp;
            }
            b[i] − = b[k] ∗ temp;
        }
        barrier;
    }
    /* backsolve outer loop */
    for(int i = dim − 1; i >= 0; i − = 1) {
        master {
            b[i] /= a[i][i];
        }
        barrier;
        forall (int k = i−1; k >= 0; k − = 1) {
            b[k] − = a[k][i] ∗ b[i];
        }
    }
    barrier;
}
```

A.2 Coherent cache version of *gauss* code

The following code is the "coherent cache" version that forces a cpu to operate on the same row(s) of the matrix for the duration of the operation. It performed very well on multiprocessors with coherent shared memory caches, but not very well on the TC2000.

```
void dgauss(double **a, double *b, int dim)
{
    /* Flags are initialized to zero: */
    static int flags[1024];
    master {
        flags[0] = 1;
    }
    /* reduction outer loop */
    for(int k = 0; k < dim; k += 1) {
        /* Wait for the pivot row to be stable: */
        while (flags[k] == 0);
        for (int i = k + 1 + (_tindex + _tsize -
                     (k%_tsize))%_tsize;
                           i< dim; i+= _tsize) {
            double temp = a[i][k];
            if(temp == 0.0) continue;
            a[i][k] = 0.0;
            temp /= a[k][k];
            for( int j= k + 1; j< dim; j+= 1) {
                a[i][j] -= a[k][j] * temp;
            }
            b[i] -= b[k] * temp;
            if (i == k + 1) flags[i] = 1;
        }
    }
    barrier;
    /* Now we perform dim back substitutions.
        Note that the meaning of flag == 0
        now means that the data is ready
        whereas before it meant not ready.
    First solve for the last x: */
    master {
        b[dim-1] /= a[dim-1][dim-1];
        /* Indicate x[dim-1] is solved. */
        flags[dim-1] = 0;
    }
    /* backsolve outer loop */
    for(int i = dim - 1; i >= 1; i - = 1) {
        if (_tindex == ((i-1) % _tsize)) {
            while (flags[i] == 1]);
            b[i-1] - = a[i-1][i] * b[i];
            b[i-1] / = a[i-1][i-1];
            /* Indicate x[i-1] is solved. */
            flags[i-1] = 0;
        }
        else {
            /* Wait for x[i] */
            while (flags[i] == 1);
        }
        for (int k= _tindex; k< i-1; k+= _tsize) {
            b[k] - = a[k][i] * b[i];
        }
    }
    barrier;
```

```
}
```

A.3 TC2000 version of *gauss* code

The following code is the re-written version that ran efficiently in parallel on the TC2000:

```
/* This parallel routine solves the linear system A.X = B
using Gauss elimination and local memory.
The matrix rows are stripped out to the processors and
the pivot row copied into each processor for the SAXPY operations.
The routine mungs a, leaving the results of the reduction in it, and
puts the solution X in the array B.
    */

#include <stdio.h>
#include <pcp.h>

#define MAXDIM 1024
static int flags[MAXDIM];


void dgauss(a, b, dimension)
double **a;
double *b;
int dimension;
{
    register dim = dimension;
    register int i, j, k;
    register int lc;
    int nrows;
        private static int not_alloc = 1; /* local memory allocation flag */
    private static double **pa;       /*  local memory a matrix rows  */
private static double *pb;        /*  local memory b values        */
double pivot_b;
double pivot[MAXDIM];  /*  local pivot row                    */




MASTER {
    flags[0] = 1;
}
    /*  first time allocate local memory    */
if (not_alloc) {
    /*  calculate no. of rows for each processor  */
    nrows = dim/_TSIZE;
    if (nrows*_TSIZE < dim) nrows++;
    if((pa =
      (double **)pratalloc(nrows, dim, sizeof(**pa))) == NULL) {
        fprintf(stderr, "pratalloc for pa failed\n");
        exit(1);
    }
    if((pb =
     (double *)prmalloc((unsigned)(nrows * sizeof(*pb)))) == NULL){
        fprintf(stderr, "prmalloc for pb failed\n");
        exit(1);
    }
    not_alloc = 0;
}
    /* copy a, b rows to local memory, record actual row number:  */
lc = 0;
for(i = _TINDEX%_TSIZE; i < dim; i += _TSIZE) {
    for(j = 0; j < dim; j += 1) {
        pa[lc][j] = a[i][j];
    }
```

```
        pb[lc]  = b[i];
        lc++;
    }

    /* We first do dim reduction steps.
        */
    for(k = 0; k < dim; k += 1) {

        register double aSubK;
        register double bSubK;

        while(flags[k] == 0) ;
        /*  copy pivot row to local memory   */
        for (j = k; j < dim; j++) {
            pivot[j] = a[k][j];
        }
        pivot_b = b[k];
        aSubK   = pivot[k];
        bSubK   = pivot_b ;

        if(aSubK == 0.0) {    /* Check for 0 in the diagonal. */
            static lock faultLock = UNLOCKED;
            LOCK(&faultLock);
            fprintf(stderr, "gauss: a[%d][%d] = 0\n", k, k);
            exit(1);
        }

        lc = 0;
        while(_TINDEX + lc*_TSIZE < k+1) lc++;
        while(_TINDEX + lc*_TSIZE < dim) {
            register double xtemp;
            double dv;
            int dd;


            xtemp = pa[lc][k];
            if(xtemp == 0.0) {
                continue;
            }
            pa[lc][k] = 0.0;
            xtemp /= aSubK;
             dv = -xtemp;
             dd = dim - (k + 1);
             if(dim > 0) {
                 daxpy1_(&(pa[lc][k+1]),&(pi vot[k+1]),
                                 &dv, &dd);
      }
            pb[lc] -= bSubK * xtemp;
            if (_TINDEX + lc*_TSIZE == k+1) {    /* if(i == k+1) */
                /*  copy back out:  */
                    for (j = k+1; j < dim; j++) {
                    a[k+1][j] = pa[lc][j];
                }
                b[k+1] = pb[lc];
                flags[k+1] = 1;
            }
            lc++;
        }    /*  while(_TINDEX   */
    }   /* for(k   */

    BARRIER;



    /* Now we perform dim back substitutions.
        */
    MASTER {
        b[dim - 1] /= a[dim - 1][dim - 1];    /* Solve the last x  */
        flags[dim - 1] = 0;
    }
    for(i = dim - 1; i >= 1; i -= 1) {
        /* Wait for the b[i] element to be up to date. */
        while(flags[i] == 1) ;
        pivot_b = b[i];
        if (_TINDEX == ((i-1)%_TSIZE)) {
            lc = (i-1)/ _TSIZE;
            pb[lc] -= pa[lc][i] * pivot_b;
            pb[lc] /= pa[lc][i - 1];
            b[i-1]  = pb[lc];
            flags[i-1] = 0;
        }
        lc = 0;
        for (k = _TINDEX; k < i-1; k += _TSIZE) {
            pb[lc] -= pa[lc][i] * pivot_b;
            lc++;
        }
     }   /*  for (i   */
    BARRIER;
}
```

B Glossary of Terms and Variables

B.1 Terms

ABSENT: See **State**.

Interconnect: See **Interconnection Network**.

Interconnection Network: The matrix of switchnodes and wires that connects processors to memories in a multiprocessor system. The terms **interconnect** and **network** are synonymous with interconnection network.

Invalidate: A message sent to a cache, from a memory controller, instructing the cache to flush a certain cache line for the purpose of maintaining coherency.

Invalidate Acknowledgement: See **Return Reply**.

LIMBO: See **State**.

Modifiable: See **Writable**.

Multicast: A restricted broadcast. A multicast involves a memory controller sending some message, usually an invalidate, to a subset of all processors.

Network: See **Interconnection Network**.

Out: See **Outstanding**.

Outstanding: When a copy of a cache line is granted to a cache, then that copy is *outstanding*. A cache line in state ABSENT has no copies outstanding; a cache line in state PRESENT* can have many copies outstanding.

Point-to-Point Invalidation: When a memory controller sends an invalidate to exactly one processor, it is a point-to-point invalidation. When a memory controller broadcasts an invalidate to a group of processors, it is a multicast.

PRESENT*: See **State**.

PRESENT1: See **State**.

PRESENTM: See **State**.

Read-only mode: If a cache line resides in a cache in read-only mode, then the line cannot be modified in cache. The term **readable** is sometimes used synonymously with read-only.

Return reply: A message sent from a cache to a memory controller informing the memory controller that an invalidation has been performed. The term **invalidate acknowledgement** is synonymous with return reply.

State, or **cache line state**: The way in which a cache line is currently being shared. There are 5 possible states:

- ABSENT: The line is not present in any cache.
- PRESENT1: The line is present in exactly one cache, in read-only mode.
- PRESENT*: The line is present in an indeterminate number of caches, in read-only mode.
- PRESENTM: The line is present in exactly one cache, in writable mode.
- LIMBO: The line is undergoing a state transition and waiting for invalidate acknowledgements to come from the caches. A line cannot be granted in any form while in this state.

Top-level reply combination: Our scheme causes return replies to be combined in the interconnection network; the actual combination takes place at the switchnode level. The term "top-level" means that only packets at the heads of buffers are candidates for combination.

Writable mode: If a cache line resides in a cache in writable mode, then the line can be modified in cache. Of course, it can also be read. The term **modifiable** is synonymous with writable.

B.2 Variables

n: The *order* of a system; the number of stages in the interconnect.

k: The *base* of a system; the fan-out of each switchnode.

N: The total number of processing nodes in a given system. $N = k^n$.

v: The vector length of a *psim* run.

G: The size of each cache group.

L: The total number of cache lines in the cache.

A: The associativity level of the cache. An A of 1 implies a direct-mapped cache.

W: The number of data bytes per cache line. The total number of bytes per cache is $L * A * W$.

Len: **Len** (for cache line length) is the symbol for "some amount of time proportional to cache line size." In our simulations, **Len** is 1 clock for every 8 bytes of cache line. In other words, 8 bytes can be buffered into the interconnect on every clock.