# Cache Invalidation Patterns in Shared-Memory Multiprocessors

Anoop Gupta and Wolf-Dietrich Weber

*Abstract*—**For constructing large-scale shared-memory multiprocessors, researchers are currently exploring cache coherence protocols that do not rely on broadcast, but instead send invalidation messages to individual caches that contain stale data. The feasibility of such directory-based protocols is highly sensitive to the cache invalidation patterns exhibited by parallel programs. In this paper, we analyze the cache invalidation patterns of several parallel applications. Our results are based on multiprocessor simulations with 8, 16, and 32 processors. To provide deeper insight into the observed invalidation behavior, we link the invalidations seen in the simulations to the high-level objects causing them in the programs. To predict what the invalidation patterns would look like beyond 32 processors, we propose a classification scheme for data objects found in parallel programs. The classification scheme provides a powerful conceptual tool to reason about the invalidation patterns of parallel applications. Our results indicate that it should be possible to scale "well-written" parallel programs to a large number of processors without an explosion in invalidation traffic. At the same time, the invalidation patterns are such that directory-based schemes with just a few pointers per entry can be very effective. The paper also explores the variations in invalidation behavior with different cache line sizes. The results indicate that cache line sizes in the 32-byte range yield the lowest data and invalidation traffic.**

*Index Terms*—**Cache coherence, cache invalidation patterns, memory traffic, parallel application behavior, shared-memory multiprocessors.**

## I. Introduction

A critical issue in the design of shared-memory multiprocessors is the cache coherence strategy. Most existing multiprocessors [10], [15], [19], [25] rely on a shared bus and use a broadcast-based snoopy protocol to keep the caches coherent [12], [20], [22]. However, such multiprocessors are not scalable, since the shared bus soon becomes a bottleneck. As an alternative, researchers have again started looking at cache coherence protocols that do not rely on broadcast, a common example being directory-based protocols [2], [5], [14]. In directory-based protocols the system maintains state about which caches have a copy of each memory block. On a write, invalidation messages are sent only to those specific caches that contain the memory block. The performance of directory-based protocols depends critically on the distribution of the number of remote caches that need to be invalidated on shared writes. The invalidation distribution is also vital in determining the viability of directory schemes that provide only a limited set of pointers per directory entry [2]. In this paper we investigate the distribution of invalidations, how it relates to data objects in the application, and how it is affected by changes in the number of processors and the cache line size.

Our study is based on the invalidation traffic produced by a set of five application programs. Four of the five applications selected are "real" parallel programs, in that they solve real-world problems and significant effort has gone into obtaining good processor efficiency with them. (These four are also part of the SPLASH parallel applications suite [23].) The remaining application (Maxflow) is smaller, but it is still interesting in that it could form the kernel of larger applications. Our results are based on memory reference streams obtained from the above applications when running with 8, 16, and 32 simulated processors.

While it is valuable to know the invalidation distributions with the relatively small numbers of processors that we can simulate realistically, our ultimate goal is to build machines with hundreds or even thousands of processors [14]. Toward this goal of predicting the invalidation distributions for a much larger number of processors, we link the observed invalidation patterns to the high-level program data structures (objects) that cause them, and present a classification of such objects on the basis of their expected invalidation behavior. We find that it is far more accurate to extrapolate the behavior of each class of data object than to simply extrapolate the composite behavior of an application. For the application types we have considered, our results indicate that it is quite possible to write parallel programs for which the invalidation traffic does not explode as the number of processors is increased. Our results also indicate that directory-based schemes with just three to four pointers per entry should work quite well for executing well-designed parallel programs.

The paper also explores the variations in invalidation behavior and memory system traffic with different cache line sizes. We explore cache line sizes between 4 and 256 bytes. As cache lines are increased in length, we observe a slight shift of invalidation patterns to larger invalidations. With an increase in line size, we also find that the data traffic generally goes up, the coherence traffic comes down, and that the overall traffic is minimum (or close to minimum) when the line size is 32 bytes.

The remainder of the paper is structured as follows. The next section explains our simulation environment and assumptions.

Manuscript received September 15, 1989; revised May 5, 1991. A. Gupta and W.-D. Weber are supported by DARPA Contracts N00014-87-K-0828 and N00039-91-C-0138. In addition, A. Gupta is supported by an NSF Presidential Young Investigator Award, and W.-D. Weber is supported by an IBM graduate fellowship.

A. Gupta and W.-D. Weber are with the Computer Systems Laboratory, Stanford University, CA 94305. The electronic mail addresses are gupta@cs.stanford.edu and lupus@sambal.stanford.edu, respectively.

IEEE Log Number 9200197.

Section III introduces the five applications used in this study and gives a brief overview of their computational behavior. In Section IV we present the basic memory reference characteristics of the applications. Section V presents the proposed classification of shared data objects in parallel programs. In Section VI we provide a detailed analysis of the invalidation behavior of each application and relate the invalidation patterns to specific data objects in the applications. Section VII presents results obtained from experimenting with different cache line sizes. Finally, Section VIII summarizes the results and presents conclusions.

## II. Simulation Environment

We use a simulated multiprocessor environment to study the behavior of the applications. The simulation environment consists of two parts: 1) a functional simulator that executes the parallel applications and 2) an architectural simulator that models the memory system of the multiprocessor.

The functional simulator used for this study is the Tango multiprocessor reference generator [7]. The Tango system takes a parallel application program and interleaves the execution of its processes on a uniprocessor to simulate a multiprocessor. This is achieved by associating a virtual timer with each process of the application and by always running the process with the lowest virtual time first.

Our architecture simulator assumes shared memory partitioned among the processing nodes, infinite caches, and a directory-based cache-coherence protocol. We have made no special effort to assign a processor's data to memory that is physically close to that processor. Memory pages are simply assigned to memory modules using the lower bits of the virtual page number. Infinite caches are used in the simulator to enable us to study data-sharing effects without any distortions introduced by finite-sized caches. The cache coherence protocol used is an invalidation-based scheme similar to that used by the Stanford DASH multiprocessor [14]. Except when specifically studying the effects of varying the cache line size, the default line size used is 4 bytes. In order to keep the simulator simple and architecture independent, we further assume that all instructions execute in a single cycle.

The simulator gathers statistics on invalidation behavior and message traffic. It also keeps track of each shared write by source code file and line number. This allows us to link the invalidation behavior observed back to the high-level language objects causing it. To observe the behavior of synchronization objects, statistics on locks are maintained by address. At each unlock operation, the number of processors waiting to obtain the lock is recorded. Because of Tango, our current simulation environment is significantly more efficient than the trace-driven environment used in our previous study [27]. We are thus able to run entire programs and can capture the complete invalidation behavior of the applications.

As Torrellas *et al.* observed [26], the level of compiler optimization makes a significant difference to the ratio between shared and private memory references in an application. Consequently, for this study, all applications were compiled with optimization level 2 (-O2) using the Mips Computer Systems C compiler (version 1.31).

## III. Application Programs

In this section we describe the data structures and computational behavior of the applications. This is important background for Section VI, where we relate invalidation traffic to high-level objects. The applications were selected to represent a variety of algorithms used in an engineering computing environment. All of the applications are written in C and use the Argonne National Laboratory macro package [16], [17] for synchronization and sharing primitives. The synchronization primitives used include locks and barriers. Further details about four of five the applications can be found in the SPLASH report [23].

### A. *Maxflow*

Maxflow finds the maximum flow in a directed graph. This is a common problem in operations research and many other fields. The program is a parallel implementation of an algorithm proposed by Goldberg and Tarjan [11]. The bulk of execution time in Maxflow is spent in picking activated nodes from the graph, adjusting the flow along these nodes' incoming and outgoing edges, and then activating their successor nodes. Maxflow exploits parallelism at a fine grain.

Maxflow does not assign the nodes of the graph to processors statically. Instead, task queues are used to distribute the load. Each processor has its own local task queue and needs to go to the single global task queue only when its local queue is empty. Tasks are put on to the global queue only when processors are waiting there, and on to the local queue otherwise. Note that the task queues are made up of the nodes themselves, linked together with appropriate pointers. Locks are used to serialize access to each node element, but contention for these is fairly low as there are many more nodes than processors. In Section VI we will see that most cache invalidations are related to the global task queue and the migration of node and edge data from one processor to another. We used a graph with 400 nodes, arranged as a 20x20 grid, for our studies.

### B. *MP3D*

MP3D [18] simulates a three-dimensional wind tunnel using particle-based techniques. It is used to study the shock waves created as an object flies at high speed through the upper atmosphere. A version of MP3D that runs on the Cray-2 is being used extensively at NASA for research.

The overall computation of MP3D consists of evaluating the positions and velocities of molecules over a sequence of time steps, and gathering relevant statistics. During each time step, molecules are picked up and moved according to their velocity vectors, taking into account collisions with the boundaries and other molecules. The main data structures consist of a particle array and a space array. The *particle array* holds the molecules and records their positions, velocities, and other attributes. The *space array* corresponds to a fine grid imposed on the three-dimensional space being modeled. Attributes of the space-array cells specify the boundaries of the tunnel and the location of the physical object. The space array is also used to determine collision partners for molecules and to keep track of statistics (e.g., density and energy of molecules) about the physical space it models.

The simulator is well suited to parallelization because each molecule can be treated independently at each time step. In our program, the molecules are assigned statically to the processors. No locking is employed while accessing cells in the space array as contention is expected to be rare, and occasional errors can be tolerated due to the statistical nature of the computation. A single lock protects the global number of collisions counter. The only other synchronization used is a barrier, which is invoked between the different phases of the program. There are six barrier invocations per time step. MP3D was run for 5 time steps with 10 000 molecules and a 14x24x7 space array containing a flat plate object.

### C. Water

Water [4] performs an $N$-body molecular dynamics simulation of the forces and potentials in a system of water molecules. It is used to predict some of the physical properties of water in the liquid state.

The main data structure in Water is a large array of records that is used to store the state of each molecule. As in MP3D, the molecules are statically split among the processors. During each time step, the processors calculate the interaction of the atoms within each molecule, and of the molecules with each other. For each molecule, the owning processor calculates the interactions with only half of the molecules ahead of it in the array. Since the forces between the molecules are symmetric, each pairwise interaction between molecules is thus considered only once. The state associated with the molecules is then updated. We note that while some portions of the molecule state are modified at each interaction, others are changed only between time steps. There are also several variables holding global properties that are updated continuously. Water was run for 2 time steps with 288 molecules.

### D. PTHOR

PTHOR [24] is a parallel logic simulator developed at Stanford University. It uses a conservative distributed-time simulation algorithm which is a modified version of the Chandy–Misra algorithm [6].

The primary data structures associated with the simulator are the logic elements (e.g., AND-gates, flip-flops), the nets (the wires linking the elements), and the task queues which contain activated elements. Each element has a preferred task queue to increase data locality. PTHOR alternates between two distinct phases: element evaluation and deadlock resolution. During element evaluation, each processor executes the following loop. It removes an activated element from its task queue (activation list) and determines the changes on the element's outputs. It then looks up the net data structure to determine elements that are affected by the output change and potentially schedules those elements on to other processors' task queues. When a processor's task queue is empty, it steals elements from other processors' task queues. When all activation lists are empty, a simulation deadlock has been reached and is resolved in a separate phase. During this deadlock resolution phase, more elements are activated. PTHOR was run for a simple RISC processor circuit with 5060 logic elements for 10 clock cycles.

### E. LocusRoute

LocusRoute [21] is a global router for VLSI standard cells. It has been used to design real integrated circuits, and offers a high-quality routing.

The LocusRoute program exploits parallelism by routing multiple wires in a circuit concurrently. Each processor executes the following loop: it picks a wire to route from the task queue; it then explores alternative routes; and finally it chooses the best route and places the wire there. The central data structure used in LocusRoute is a grid of cells called the *cost array*. Each row of the cost array corresponds to a routing channel for standard cells. LocusRoute uses the cost array to record the presence of wires at each point, and the congestion of a route is used as a cost function for guiding the placement of new wires. No locking is needed in the cost array, which is accessed and updated simultaneously by several processors, because the effect of occasional contention is tolerable. Each routing task is of a fairly large grain size, which prevents the task queue from becoming a bottleneck. For this study we used the Primary1 circuit consisting of 1266 wires and a 481x18 cell cost array.

## IV. Program Characteristics

Table I gives an overview of the characteristics of the five applications when run with 32 processors. For each application, we give the number of data references and the breakdown in terms of reads and writes. We also show the number of shared reads and shared writes. In addition to absolute numbers, the columns also list the number of references in each category as a fraction of all data references. The last two columns give the average number of invalidations caused by each invalidating write, and the number of invalidating writes per 1000 data references. Invalidating writes correspond to write hits to clean data and write misses.

In our study, private and shared references are distinguished as follows. Each application shares data between processes by placing it in a special shared data space. We define *shared blocks* to be those that are in the shared data space. We define *shared references* to be reads and writes to shared blocks. We note that depending on the task distribution strategy used and the dynamics of a particular run, it is possible that some shared blocks are referenced by only one process during the entire run.

From Table I, we see that the proportion of reads and writes is similar to what one might expect in uniprocessor programs—the fraction of reads varies from 62% in MP3D to 83% in LocusRoute. The statistics for shared references, however, vary considerably from application to application. For example, the ratio between shared reads and writes varies from about 1.5:1 for MP3D to about 9:1 for LocusRoute. Overall, considering all applications, shared reads are greatly favored

TABLE I
GENERAL APPLICATION CHARACTERISTICS

| Application | num of CPUs | data refs mill | reads mill | reads % | writes mill | writes % | shared reads mill | shared reads % | shared writes mill | shared writes % | avg. invals per inv-write | inv-writes per 1000 refs |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Maxflow | 32 | 25.5 | 18.5 | 72 | 7.0 | 28 | 12.5 | 49 | 2.2 | 8 | 1.4 | 58 |
| MP3D | 32 | 2.3 | 1.4 | 62 | 0.9 | 38 | 1.0 | 46 | 0.7 | 30 | 1.0 | 221 |
| Water | 32 | 48.0 | 34.1 | 71 | 13.9 | 29 | 7.1 | 15 | 1.0 | 2 | 1.2 | 11 |
| PTHOR | 32 | 16.6 | 13.5 | 81 | 3.1 | 19 | 7.2 | 43 | 0.9 | 5 | 1.5 | 32 |
| LocusRoute | 32 | 18.5 | 15.3 | 83 | 3.2 | 17 | 8.7 | 47 | 1.0 | 5 | 1.6 | 27 |

over shared writes as compared to uniprocessor programs. As another example, the fraction of shared references varies from about 17% for Water to about 76% for MP3D.[1] In Water, each interaction between molecules requires a fair amount of local calculation. Thus updates to the states of the molecules are relatively infrequent, and the fraction of shared references is very low. In MP3D, on the other hand, most of the data manipulation occurs directly on the shared data, and hence the proportion of shared references is large. While these variations are not unexpected, since they depend closely on the nature of the application and the way in which it is parallelized, they are indicative of the variety in the applications being evaluated.

The second to last column in Table I gives the average number of invalidations per invalidating write. This number is an important metric for directory-based cache coherence schemes because a large value indicates the need for many pointers per directory entry. As we can see, this number is less than two for all applications, even though all runs are with 32 processors. The last column of Table I gives the average number of invalidating writes per 1000 data references. The product of the entries in the last two columns is a good indicator of the amount of invalidation traffic that an application is expected to generate per unit time. We only give average numbers here, and these look quite favorable. However, averages are limited in the information they provide. Consequently, we provide detailed invalidation distributions and their analysis in Section VI.

## V. CLASSIFICATION OF DATA OBJECTS

In this section we present our classification of data objects based on invalidation behavior. The classification allows us to explain a given application's invalidation distribution in terms of the underlying high-level data structures of that application. More importantly, it represents a model that enables us to predict the application's invalidation behavior for much larger number of processes than is feasible for us to simulate. We propose to distinguish the following classes of objects:

1) Code and read-only data objects.
2) Migratory objects.
3) Mostly-read objects.
4) Frequently read/written objects.
5) Synchronization objects.
   - low-contention synchronization objects
   - high-contention synchronization objects

*Code and read-only data objects:* These objects do not generally pose a problem to directory schemes because they are written only once at the time when the relevant page is first brought into memory, or when the data are initialized. Invalidations are hence *very* infrequent. A fixed database is a good example of read-only data.

*Migratory data objects:* These objects are manipulated by only one processor at any given time. Shared objects protected by locks often exhibit this property. While such an object is being manipulated by a processor, the object's data resides in the associated cache. When the object is later manipulated by some other processor, the corresponding cache entries in the previous processor are invalidated. Migratory objects occur frequently in parallel programs. The nodes in Maxflow are a good example of migratory data. Each node is looked at by several processors over the complete run, but there is only one processor manipulating each node at any one time. Migratory data usually cause a high proportion of *single* invalidations, irrespective of the number of processors working on the problem.

*Mostly-read data objects:* These objects are read most of the time, and written only every now and then. An example is the cost array of LocusRoute. It is read frequently, but written only when the best route for a wire is decided. It is a candidate for many invalidations per write, because many reads by different processors occur before each write. However, since only the writes cause invalidations and writes are infrequent, the overall number of invalidations is expected to be quite small.

*Frequently read/written objects:* These objects are *both* read and written frequently. Although each write causes only a small number of invalidations, writes occur frequently, and so the total number of invalidations can be quite large. An example of a frequently read/written object is the variable that holds the number of processors waiting on the global task queue in Maxflow. It is continually checked by all processes, and is updated whenever a process starts or stops waiting on the global task queue.

*Synchronization objects:* These objects correspond to the synchronization primitives used in parallel programs, the most frequent examples being locks and barriers. We further divide synchronization objects into two categories, low-contention and high-contention objects, since these two exhibit differ-

[1] The fraction of references that are to shared data is somewhat larger than that reported by Eggers [8]. This is most likely due to the fact that we are compiling with the -O2 flag, which tends to reduce local references through register allocation.

ent invalidation behavior. Low-contention synchronization objects, such as distributed locks that protect access to a collection of shared data objects, usually have very few or no processes waiting on them. As a result, most often they cause zero or a very small number of invalidations. Low-contention locks are thus easy to implement and optimize for in directory-based multiprocessors. High-contention synchronization objects, on the other hand, usually cause frequent invalidations, and the invalidations may be large if there are many contending processes. A lock protecting a highly contended task queue would be an example of such an object. If high-contention locks are treated like regular data objects in limited-pointer directories, their invalidation behavior can have a severe impact on machine performance. Some combination of software techniques (e.g., synchronization trees [28]) and hardware techniques (e.g., queueing lock primitives [13]) are probably required to efficiently support high-contention synchronization objects.

Bennett *et al.* [3] expand the classification of data objects proposed in our earlier paper [27]. They use their classification to perform adaptive software cache management on distributed shared memory machines. The reason for a finer division of objects is that some differences in object behavior are important to a software cache coherence protocol, while they make no difference in invalidation behavior. For example, the invalidation behavior of Bennett's *producer/consumer* and *read-mostly* types will be indistinguishable for the case of multiple consumers.

## VI. Application Case Studies

In this section we present the results of the detailed analysis of the invalidation traces produced by the applications. For each application, we discuss the overall invalidation patterns, the high-level objects causing the invalidations, the synchronization behavior, and the predicted invalidation behavior beyond 32 processors.

The overall invalidation behavior is presented in terms of a series of graphs as shown in Fig. 1. Parts (a), (c) and (d) are the invalidation distribution graphs for 8, 16, and 32 processors, respectively. These graphs show what proportion of invalidating writes cause 0, 1, 2, or more invalidations.

We distinguish between *large* invalidations and *frequent* invalidations. A large invalidation is caused by a write to a line that is cached by many processors. Frequent invalidations are caused by frequent writes and need not necessarily be large invalidations. Ideally, the invalidation distribution graphs should contain a large proportion of small invalidations, as these can be handled efficiently by directory-based cache schemes. By comparing the invalidation distributions for 8, 16, and 32 processor runs, we can begin to get a feeling for how the invalidations scale with a larger number of processors.

For the 32-processor run we give additional information. Part (e) gives the proportion of reads, writes, shared reads, and shared writes. Part (f) breaks invalidating writes and invalidations down by important data objects found in the application. Part (g) shows the composition of the invalidation distribution of part (d). Each bar of (d) is normalized to 100 and broken down into its data object components. We are thus able to tell, for example, that invalidating writes causing 31 invalidations in Maxflow (0.1% of all invalidating writes) are made up of 80% writes to global values and 20% writes to edge elements of the graph being manipulated [Fig. 1 (g)].

Finally, part (b) presents the synchronization behavior for the 32-processor run. The graph shows the distribution of waiters at all *unlock* operations. For example, low-contention locks should show a very small number of waiters at each unlock operation. Note that the distribution of waiters in these graphs is shown only for locks, since the behavior of waiting processes at barriers depends strongly on the particular barrier implementation chosen (for example, tree-structured versus flat releases). We indicate the number of unlock operations and the number of barriers encountered in text on the graphs.

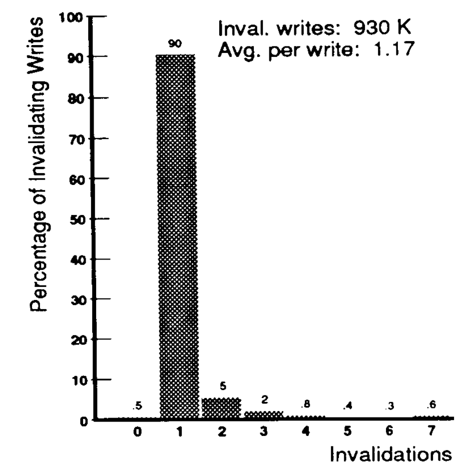
### A. Maxflow

From Figs. 1(a), (c), and (d) we see that a large fraction of the invalidations in Maxflow are single invalidations. These are mostly caused by the manipulation of node and edge data structures, portions of which are good examples of migratory data. What happens is as follows. One processor picks up an active node and pushes flow through it. Later, when the node is reactivated, some other processor picks it up and starts processing it, thus causing a single invalidation. Sometimes, however, the node gets picked up by the same processor as before, in which case we do not see any invalidating writes, because the node data is most likely still in the processor's cache. The likelihood of the same processor picking up a node, however, decreases as more processors are added, and this results in more invalidating writes. The trend is quite clear as one moves from part (a) to (c) to (d) of Fig. 1. The invalidating writes go from 0.93 M to 1.17 M to 1.48 M.

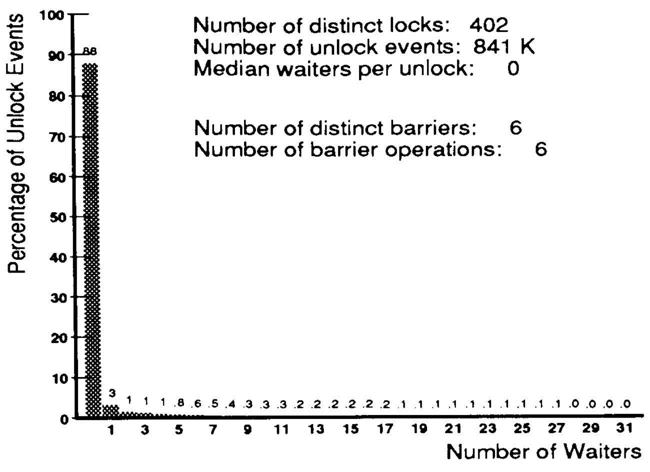
As the number of processors is increased, we also observe that the invalidation distribution slowly shifts to larger invalidations. While with 8 processors only about 9% of shared writes cause more than one invalidation, this figure moves up to 15% with 32 processors. There are three types of data objects causing this shift: 1) data associated with the global task queue, 2) node labels, and 3) edge link pointers. All of these fall into the frequently read/written category. We now consider each of these in turn.

The count of the number of processors waiting for the global task queue to become nonempty is checked frequently by all processors. It is also written frequently, namely whenever a process starts waiting on the global task queue. For the 32-processor run, it has an average of 6 invalidations per invalidating write and the highest number of shared writes to any single data object. The global task queue pointer is a close second. The above two categories are combined in Figs. 1(f) and (g) under *global values*. Here we see, for example, that close to 80% of the writes causing invalidations in all 31 other processors can be attributed to these two data objects.

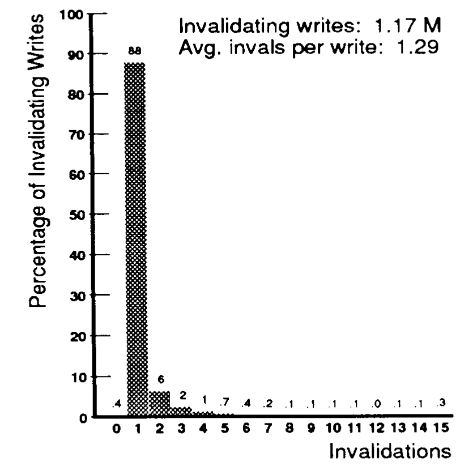
Node labels are constantly read as processors push flow through nodes, and are also frequently modified. Edge link pointers are traversed whenever flow across an edge is examined, and they are modified whenever an edge becomes



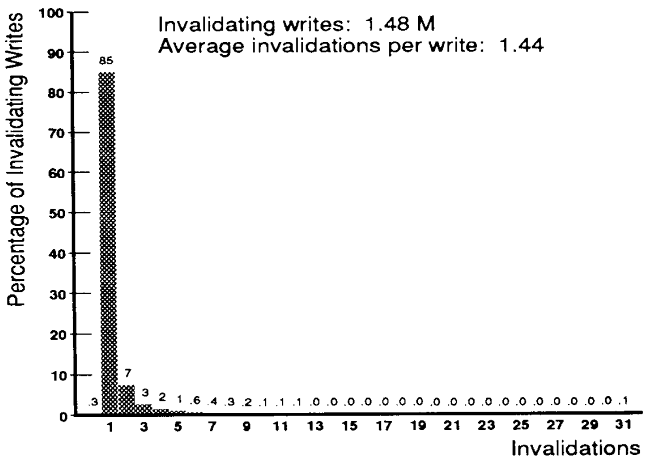
(a)



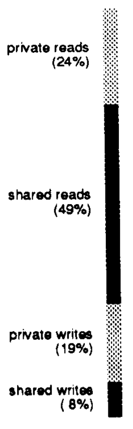
(b)

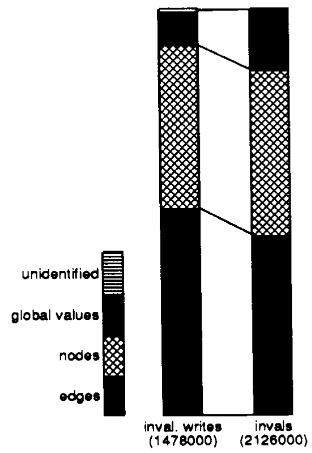


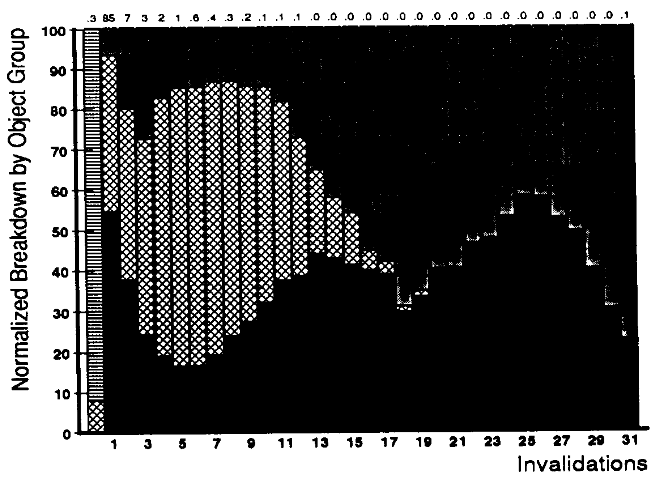
(c)



(d)







(e) (f) (g)

Fig. 1. Behavior of Maxflow.

active or inactive. Since edges connect different nodes, and different nodes are examined by different processors, there are always many different processors traversing the edge lists and modifying the pointers. The node labels and edge link pointers account for most of the increase in small invalidations as we increase the number of processors.

Fig. 1(b) shows the synchronization behavior of Maxflow. There are 402 locks total. Of these, 400 are *distributed* locks, one for each of the 400 graph nodes. These are accessed frequently, but there is very little contention for them. These distributed locks are an excellent example of low-contention locks. The remaining two locks are used to protect access to the global task queue and the variable that counts the number of processors blocked on the global task queue. These two locks incur significant contention and are responsible for the "tail" of large number of waiters in Fig. 1(b). The average number of waiters for these two locks is about 5. We note that there are also six barriers that are exercised once each during the run of the program.

We now use the object classification to see how the invalidation distributions are expected to change as the number of processors is scaled. We expect little change in the invalidations produced by the migratory portions of the graph node and edge structures. They should continue to produce the single invalidations typical of migratory data. The global task queue pointer and block count, on the other hand, are frequently read/written data and are expected to have an increasing average number of invalidations per write. This is also true for the node labels and edge pointers. In addition, we expect to see more waiters at the global queue locks as contention for them increases. If the program is to scale well as number of processors is increased, we must reduce contention for the global task queue and we must partition the graph so that the number of processors sharing the frequently read/written node label and edge pointer objects is small.

## B. MP3D

Figs. 2(a), (c), and (d) show the invalidation distributions for the MP3D program, the 3-D particle-based simulator. Here again the distributions are dominated by single invalidations. However, as we increase the number of processors, the invalidation distributions remain essentially the same.

Most accesses to shared data by MP3D consist of updating the properties of a given particle or space array entry. This results in a sequence of reads closely followed by writes to the same locations. Depending on whether the data object was previously accessed by the same processor or not, either a single invalidation or no invalidations result. These data behave in a migratory fashion, with each interval of active use being very short.

From part (g) of Fig. 2 we see that most of the larger invalidations are due to a variable that keeps track of the average probability of collision for each cell in the three-dimensional space array. This variable is read by different processors during a time step as they decide whether or not a collision occurred. It is updated only between time steps. There are also a few global properties that are read by every processor but again are updated only between time steps. Both of these object groups fall into the mostly-read category.

Part (b) of Fig. 2 shows the synchronization behavior of MP3D. Work is distributed statically in MP3D, and there is very little synchronization overhead. The distribution of waiting processes in Fig. 2(b) is entirely due to one lock that protects the access to a set of global counters. After every time-step each processor updates the global count with its own local count.

The effect of the mostly-read data found in MP3D is minor because of the low frequency of writes to this data. The remaining data behave strictly migratory and we thus expect little change in the invalidation distribution of MP3D as it is scaled to a larger number of processors.

## C. Water

Figs. 3(a), (c), and (d) show the invalidation distributions for the Water code. The distributions are made up almost entirely of single invalidations. There is only a slight increase in the number of invalidating writes and in the average invalidations per write as the number of processors is increased.

The main data structure in the Water code is a large array of records, one for each water molecule. Most of the time is spent calculating the pairwise interaction of molecules. At the end of each interaction, a portion of the state of the two molecules is updated. This portion of the molecule data is migratory, and causes only single invalidations. There is another portion of the molecule record that is also read while computing the pairwise interactions, but it is updated only between time steps. Since the molecules allocated to a processor interact with only half of all the other molecules (see Section III-C), at the end of each time step half of the processors have cached this mostly-read data. Consequently the update causes invalidations to half the total number of processors. Part (g) of Fig. 3 illustrates this clearly.

There are also a small number of variables that hold global properties of the water molecule system. These again fall into the mostly-read category. They are read by all processors throughout, and updated between time steps. At each update, invalidations are sent to all processors.

There is very little synchronization in Water, since the work is partitioned statically. There is a set of distributed locks, one for each molecule, and a small number of individual locks to protect the updates of global values. There is very little contention for the distributed locks. While there is some contention for the update of the global values, contention is low enough that it is not a factor in the overall lock waiter distribution [see Fig. 3(b)].

We do not expect the invalidation distribution of the Water code to change significantly as the number of processors is increased, because most of the data is migratory. The mostly-read structures are written very infrequently and thus cause only minor increases in the average number of invalidations per invalidating write.

## D. PTHOR

Figs. 4(a), (c), and (d) show the invalidation distributions

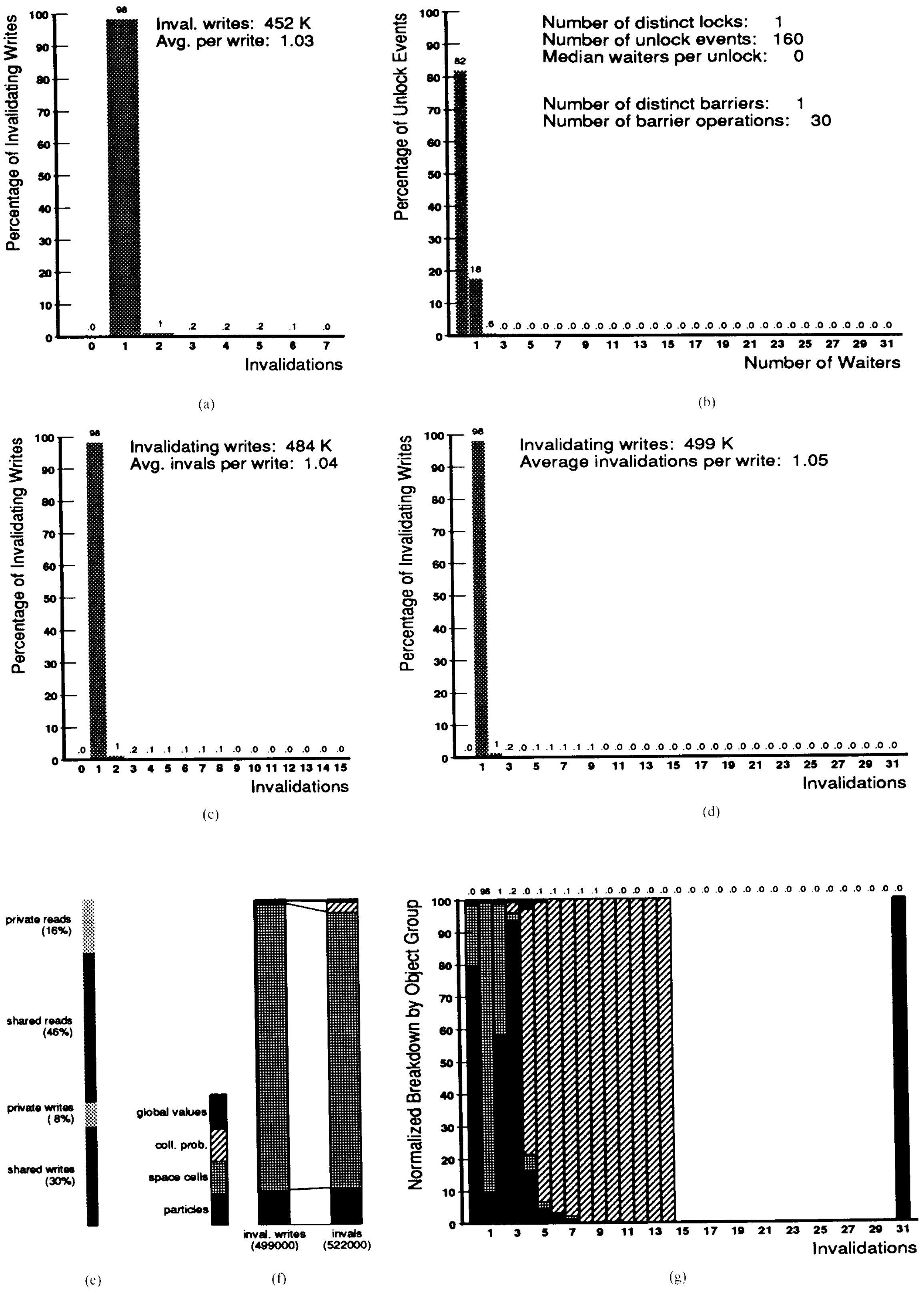


Fig. 2. Behavior of MP3D.

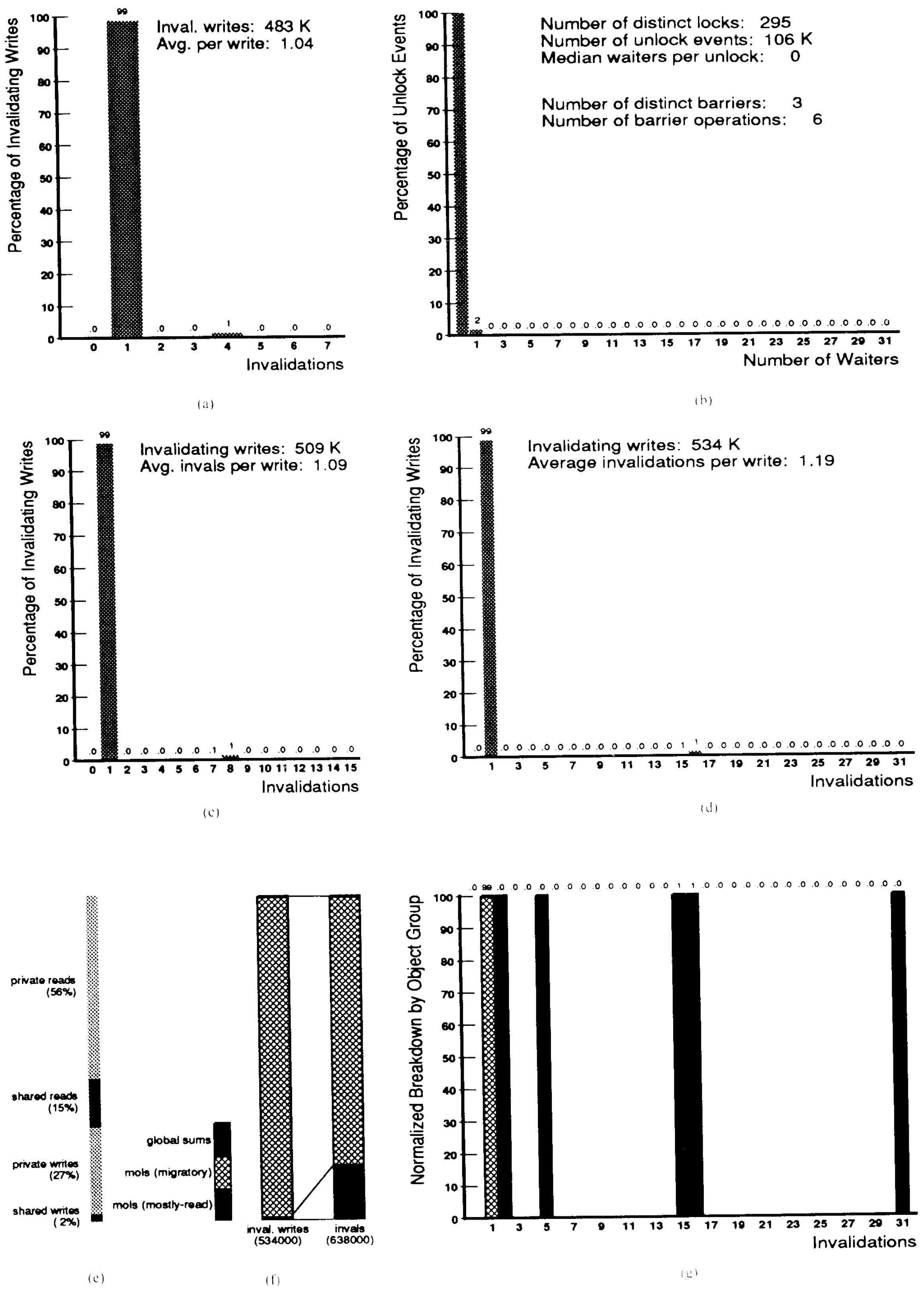
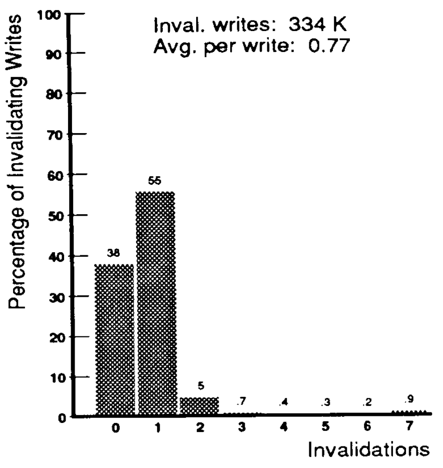
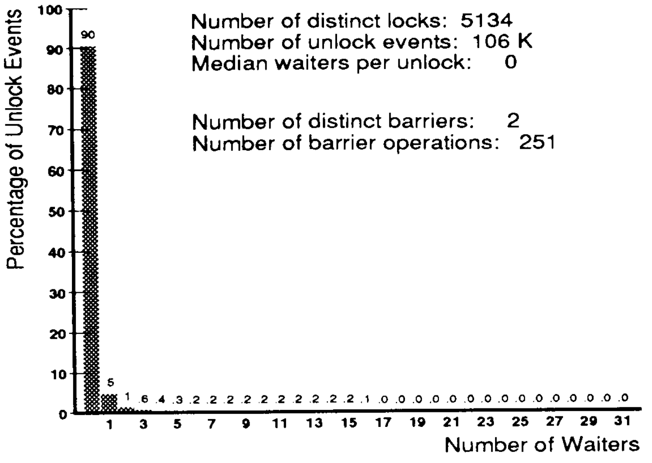


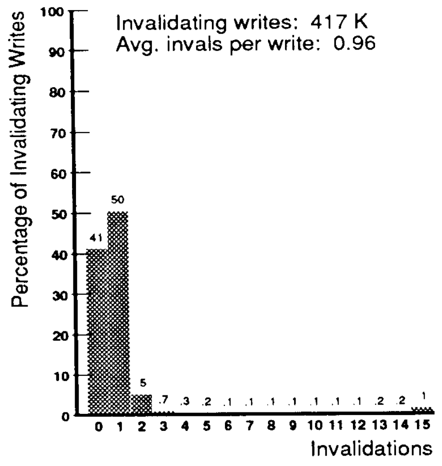
Fig. 3. Behavior of Water.



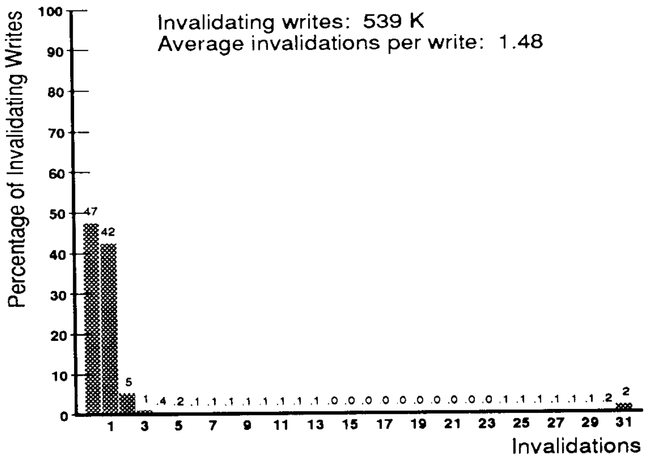
(a)



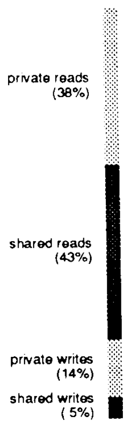
(b)



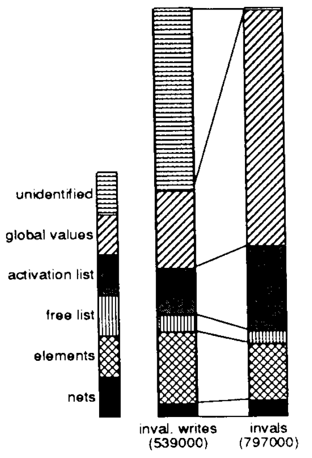
(c)



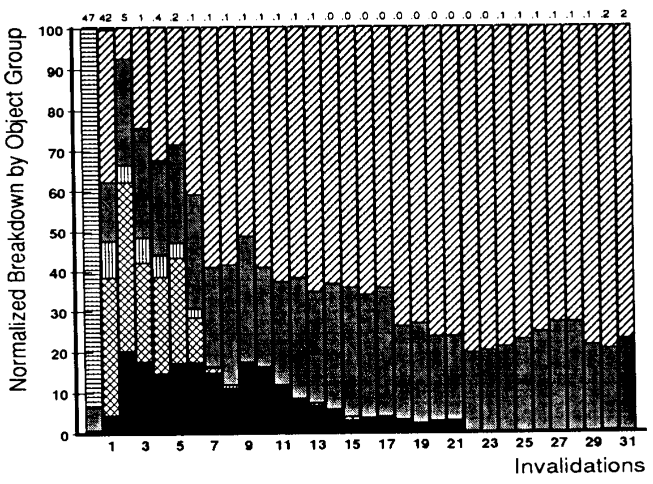
(d)



(e)



(f)



(g)

Fig. 4. Behavior of PTHOR.

for PTHOR. We again find that very few of the shared writes cause large invalidations. The basic data objects of PTHOR are the logic elements and net data structures.

In PTHOR, sharing of net data is determined by the connectivity of the circuit. Some nets, such as the clock net, are attached to many elements. They are thus cached by many processors and cause large invalidations when written. Typically though, most nets connect only a few elements and writes to them cause a small number of invalidations.

During the program run, the logic elements behave like migratory objects and we mostly see single invalidations. Some portions of the element data structure, however, are not modified by every processor that references them. These "longer-lived" values, such as the minimum valid time of an element, fit into the mostly-read category and result in larger invalidations when they are updated.

The head pointers of the free lists for data structures are usually migratory. However, the head pointer is checked before taking an item off a given free list. If the list is empty, many processors could cache the head pointer and it becomes mostly-read for a short phase.

The large invalidations in PTHOR are due to mostly-read global data objects. Common examples are the heads of the activation lists (task queues), and the count of number of processors waiting for the deadlock phase. These are checked frequently by most processors, but are changed relatively infrequently. Link pointers in the activation list also fall into the mostly-read category.

Most of the zero invalidations are caused when the element and net data structures are initialized, in parallel, at the beginning of the run.

The synchronization behavior shown in Fig. 4(b) is dominated by element locks. These distributed locks show very little contention. Most of the time there are no waiters when an unlock occurs. The larger number of waiters at unlock operations are almost all due to a single lock that is used to protect the count of processors that have reached the deadlock phase of the Chandy–Misra simulation algorithm.

As the number of processors is scaled, we expect that the invalidations produced by the element data structures would not increase, since they act as migratory objects. The invalidation patterns due to the net data structures should also not change (beyond a point) as the connectivity of the circuit remains the same. We expect larger average invalidations per invalidating write for the mostly-read global objects and the activation list link pointers. Overall, we expect a steady shift of the invalidation distribution toward larger invalidations, unless new locality-enhancing heuristics are added.

### E. LocusRoute

Parts (a), (c), and (d) of Fig. 5 show the invalidation distributions for LocusRoute. The largest source of invalidations in LocusRoute is the global cost array. The cost array is a good example of mostly-read data. It is frequently read while testing different routes for a wire, but is written only when the wire route is decided. The average number of invalidations per shared write of the cost array is about 2 with 32 processors, but some writes can cause up to 17 invalidations. Small invalidations are much more common, because in LocusRoute there is enough locality to keep the number of processors actively sharing a region of the cost array small.

Another large source of invalidating writes is a collection of variables, labeled *misc data* in Figs. 5(f) and (g), that are migratory. The most frequently used ones of this set are the RouteRecords, which are used by the processors as they route a wire. They are reused by other processors for routing other wires, and cause only zero and single invalidations. The data structures related to the wire tasks (labeled *tasks*) are also migratory, while the flag that signals whether the task queue is empty or not (labeled *empty flag*) is mostly-read. Neither one of these last two data structures, however, accounts for a very large fraction of the total invalidations. The group labeled *global values* represents a small number of global variables. These fall into one of two categories: 1) global counts that are updated using read-modify-write operations and act as migratory objects, and 2) global flags that are read by many processors, but modified infrequently and act as mostly-read objects.

Part (b) of Fig. 5 shows the synchronization behavior of LocusRoute. There are a total of 51 locks; 46 of these are distributed locks with very little contention. Of the remaining five, only a mutual exclusion lock used for printing and the lock that controls the task queue from which processors obtain their wire tasks have any noticeable contention. However, they are used infrequently, and thus do not cause problems. The single barrier is used only once to synchronize the start of the slave processes.

As more processors are added, the average number of invalidations per shared write is expected to increase slightly, because more processors are likely to have cached a given portion of the cost array. However, since the cost array is a mostly-read object with infrequent writes, the absolute number of invalidations per data reference is expected to remain small. By exploiting geographic locality, that is by partitioning the cost array into regions and assigning wires in a region to the corresponding processor, it might be possible to further limit the growth in the number of invalidations per shared write.

## VII. Effect of Cache Line Size

We now investigate the effect of different cache line sizes on invalidation patterns and traffic. While larger cache lines are desirable from the point of hardware efficiency and the prefetching they provide in multiprocessors, they can also cause significant increases in message traffic. There are several reasons for this. First, a larger cache line size increases the minimum communication granularity between processes. For example, even if a process wants to communicate a single word of information to another process, the minimum data that is sent across the network is still the whole cache line, thus increasing traffic volume. (This assumes an invalidation-based cache coherence protocol.) Second, parallel programs usually exhibit less spatial locality than sequential programs. For example, if a cache line is large and contains multiple data items (with each data item corresponding to a different

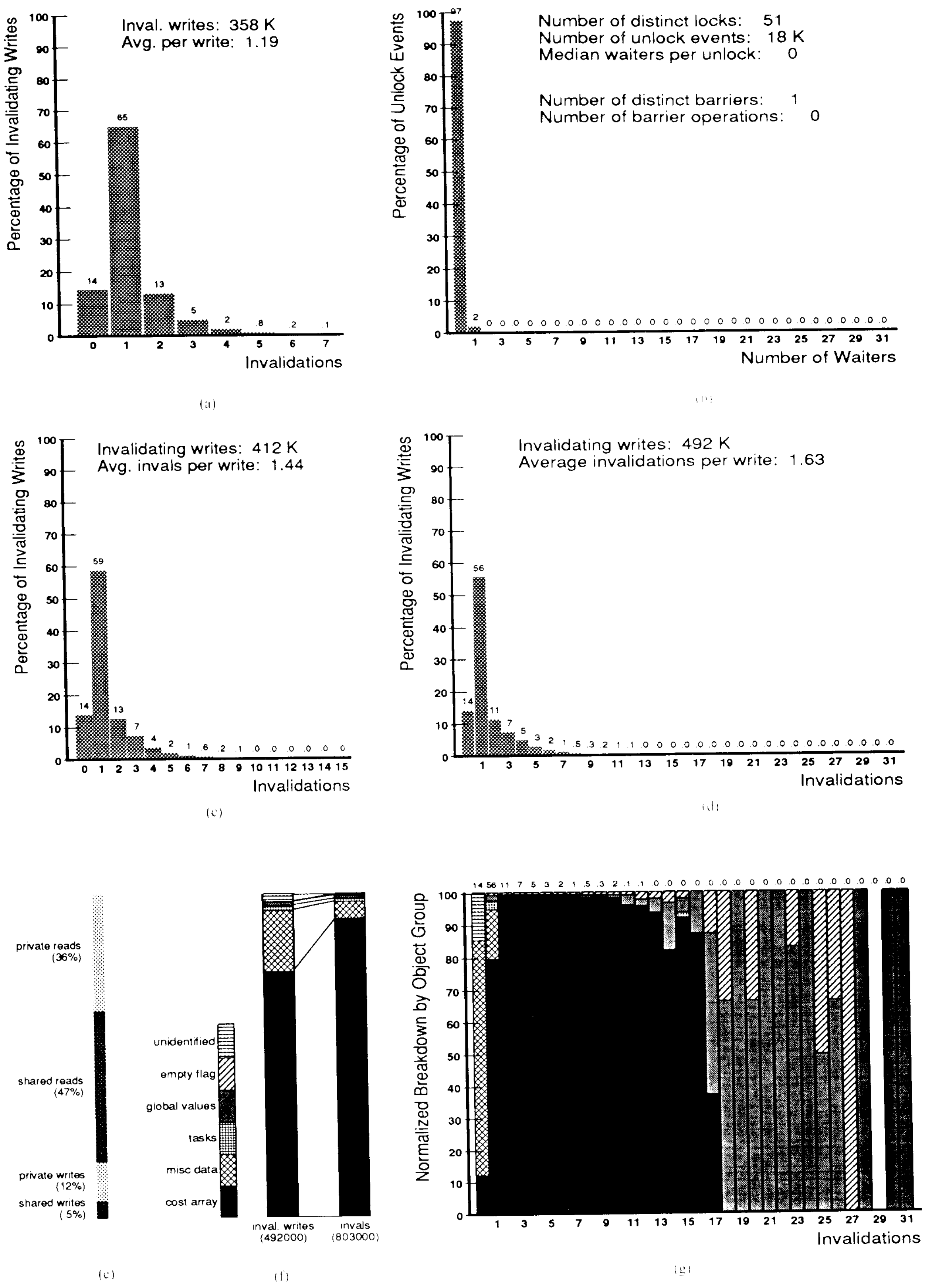


Fig. 5. Behavior of LocusRoute.

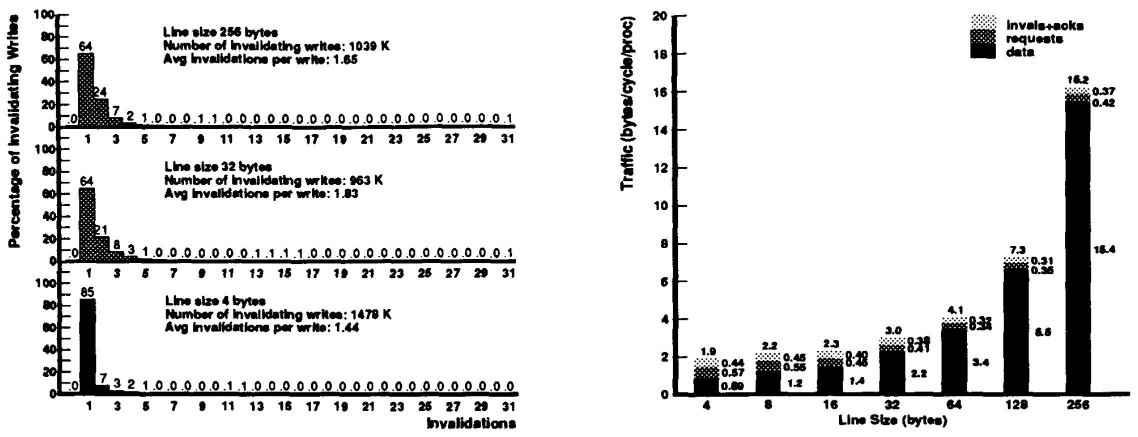


Fig. 6. Maxflow behavior with increasing line size.

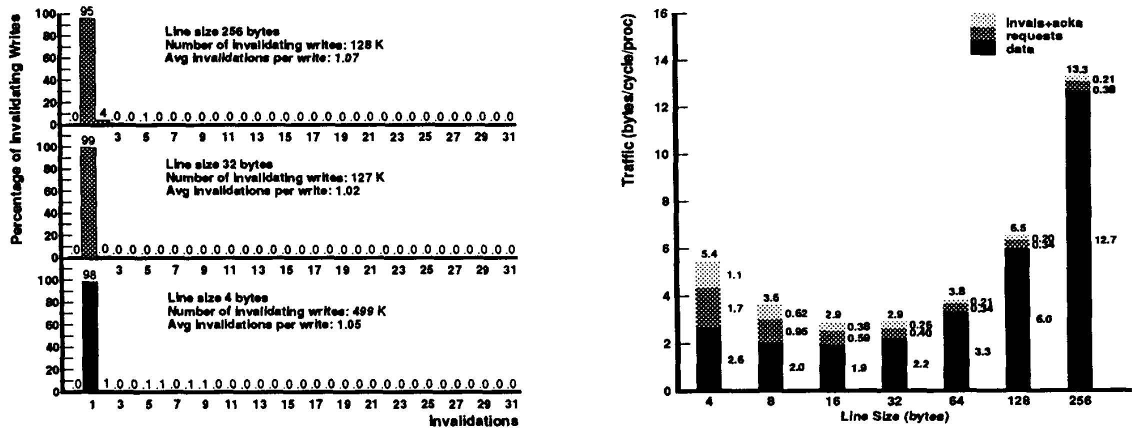


Fig. 7. MP3D behavior with increasing line size.

subtask), in a parallel program those subtasks may be evaluated on different processors. The spatial locality in this case is less than in a sequential version of the program, where all subtasks are processed one after the other by the same processor. Third, the number of invalidating writes and the message traffic may also increase due to *false sharing* [26]. Using the previous example again, if each subtask performs multiple modifications to the corresponding data item, then the cache line will bounce back and forth between the multiple processors, although no real communication is involved.

In the data presented so far we have used a cache line size of 4 bytes. This line size eliminates all false sharing. We now present results for cache line sizes between 4 and 256 bytes. In Figs. 6–10, we show two graphs for each application. The left graph shows the changes in the invalidation distribution as the line size is increased. The right graph shows the amount of message traffic generated and its breakdown into components for different line sizes. The messages correspond to those that are required by the DASH cache-coherence protocol [14]. To compute traffic, we count three types of messages: invalidations and acknowledgments (7 bytes each), requests (7 bytes), and data messages ($7 + linesize$ bytes). The size of the messages was obtained by assuming 2 bytes for routing, 1 byte for control, and 4 bytes for address. We take the total traffic in bytes and divide it by the product of the total cycles for the run and the number of processors to arrive at the traffic rate in bytes per cycle per processor. Since our simulation runs were done assuming an ideal memory system, where each instruction execution and memory access takes a single cycle, the traffic rate in fact indicates bytes per instruction executed (rather than bytes per clock cycle).

## A. Invalidation Patterns

We first examine the left graphs in Figs. 6–10. For line sizes of 4, 32, and 256 bytes the graphs show the invalidation

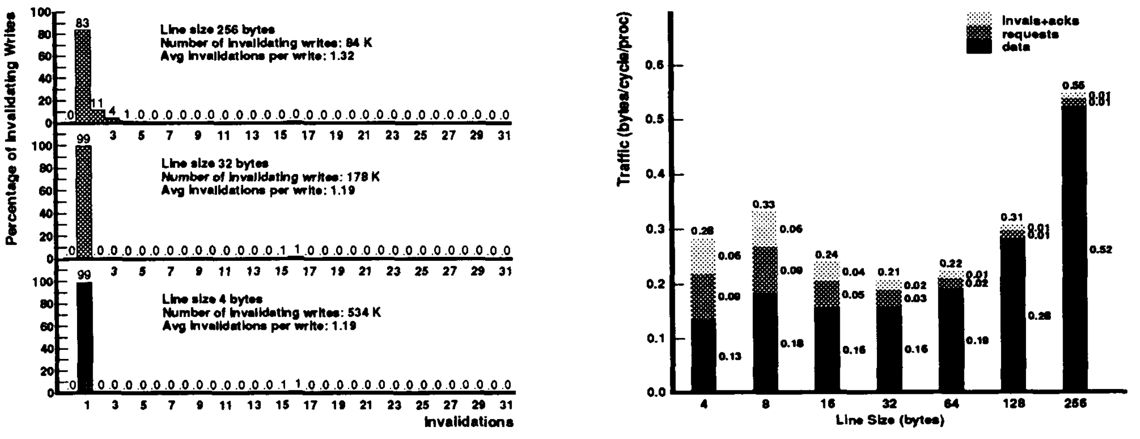


Fig. 8. Water behavior with increasing line size.

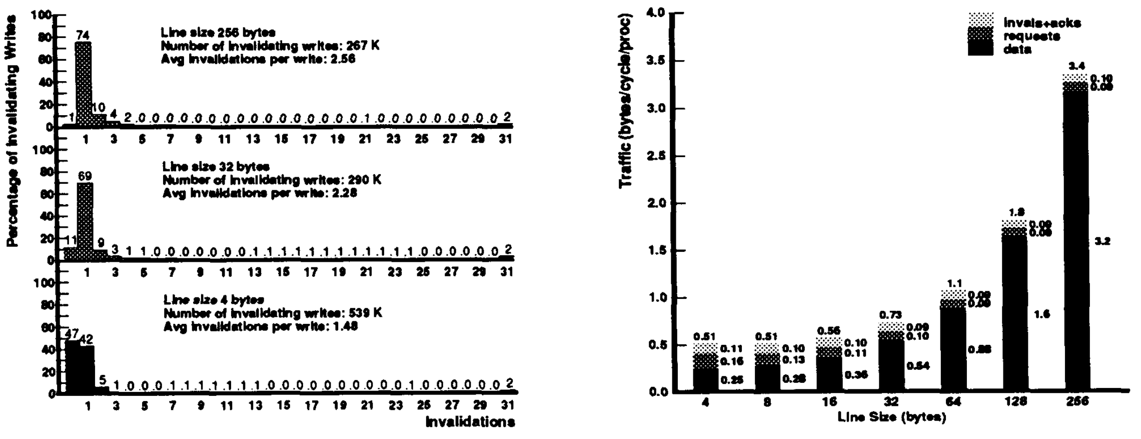


Fig. 9. PTHOR behavior with increasing line size.

distributions, and they list the total number of invalidating writes and the average size of invalidations.

Focusing on the effect of line size on the number of invalidating writes, we observe that the outcome depends on the relative sizes of the data objects in the program and the cache lines. If the typical data objects are larger than the line size, we will need to update several cache lines every time a complete object is written. Consequently, as the line size is increased, fewer lines will be modified, and we should see a decrease in the number of invalidating writes. This effect can be seen in all five applications when the line size is increased from 4 bytes to 32 bytes. For example, for the Water code (Fig. 8), the number of invalidating writes decreases from 534 K with 4-byte lines to 178 K with 32-byte lines. On the other hand, when the line size gets larger than the typical objects, several objects will fit into each cache line, and additional invalidating writes will be generated due to false sharing. Maxflow (Fig. 6) exhibits this trend when going from 32 to 256 byte lines, with the invalidating writes increasing from 963 K to 1039 K. The other applications do not exhibit this trend. For some of these other applications the typical data objects are larger than 256 bytes. For others the apparent object size is increased by reference patterns in which a given processor accesses neighboring objects consecutively.

Considering the effect of line size on the average size of invalidations, there are again several distinct effects that come into play. First, a larger line size is expected to increase the number of processors sharing a cache line (due to false sharing), thus increasing the size of invalidations. Second, depending on the spatial locality exhibited by different classes of objects (e.g., migratory versus mostly-read objects) in the program, an increased line size may reduce the number of invalidating writes causing a single invalidation more than those causing several invalidations, or vice versa. In programs where writes that cause smaller invalidations are reduced by a greater amount, the average size of invalidations will

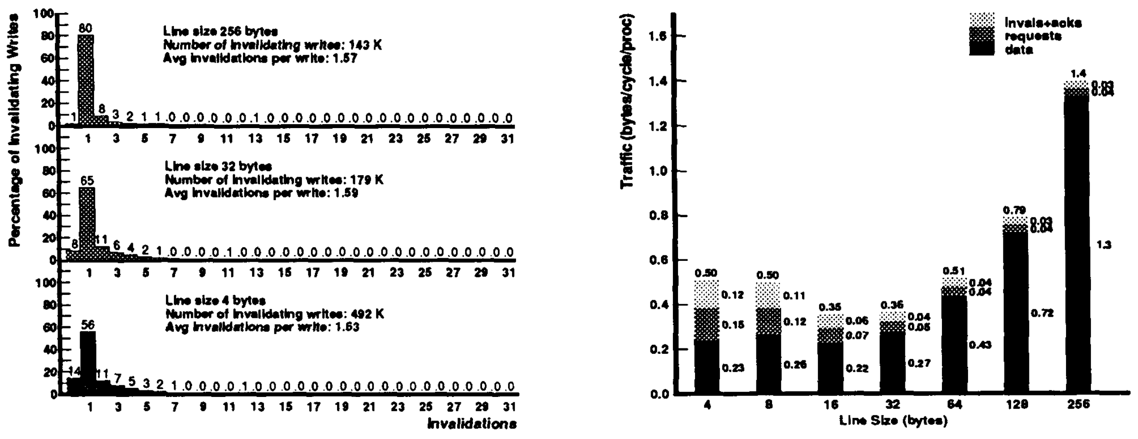


Fig. 10. LocusRoute behavior with increasing line size.

go up. For example, in PTHOR, a large fraction of zero invalidations (with 4-byte lines) are caused during the parallel initialization phase. These initialization accesses show very good locality, and as the line size is increased the number of writes causing zero invalidations diminishes dramatically. The writes causing larger invalidations do not decrease in the same proportion, thus increasing the average number of invalidations per invalidating write. In LocusRoute, on the other hand, the cost array references show good locality. These data are mostly-read and cause many medium-sized invalidations. When the line size is increased the number of invalidating writes due to the cost array decreases, bringing down the average number of invalidations per invalidating write. We see that the effect of line size on invalidation distribution is complex and not easily predicted. In general, though, there is a slight trend toward larger invalidations as the cache line size is increased.

## B. Message Traffic

Let us now look at the right graphs in Figs. 6–10. We show the message traffic in bytes per cycle per processor for different line sizes. As the cache line is increased, there are typically fewer messages of each type. As a result, the traffic due to request and invalidation messages decreases, since the message size remains constant regardless of the line size. For data messages, however, the message size increases with line size. This effect tends to offset the reduced number of data messages. Depending on the program, one or the other effect may dominate for a given line size. However, for very large line sizes, the data message size always dominates, and data traffic is largest for all applications. The minimum data traffic is achieved with line sizes as small as 4 bytes for applications with little data locality (such as Maxflow), or as large as 16 bytes for applications with good data locality (such as LocusRoute). When we look at *total* traffic, the minimum is shifted further toward larger cache line sizes, because of the continually decreasing trend of the request and invalidation message traffic. Overall, traffic is minimum, or close to minimum, for line sizes around 32 bytes.

Several researchers have studied the effect of cache line size on message traffic. In general their data favors cache line sizes smaller than the ones we find best in our study. We now briefly discuss reasons for this apparent discrepancy. Agarwal and Gupta [1] present results for several snoopy coherence schemes using traces obtained from a four processor system. Their traffic data favor smaller block sizes primarily because they simulate bus-based snoopy protocols in contrast to the directory-based protocols studied in this paper. In a bus-based snoopy protocol, the amount of invalidation traffic is quite small, since each invalidating write causes only a single bus transaction. In contrast, in a directory-based protocol the invalidation traffic can be quite large, since separate invalidations must be sent to each processor caching the data and acknowledgments must be received back. Since the benefits of the reduced invalidation traffic (with larger line size) are not so significant for snoopy protocols (as a fraction of total traffic), they favor smaller line sizes. The Agarwal and Gupta study also does not model request traffic (messages to request memory lines from remote processing nodes), and this again favors smaller line sizes. Similar comments apply to the work of Eggers *et al.* [9], who also simulate a bus-based system. Their results are, however, closer to ours. Torrellas *et al.* [26] compute traffic by simply multiplying the number of shared misses by the cache line size, where shared misses include read and write misses as well as write hits to clean data. While this method may provide an accurate traffic estimate for bus-based systems, our model with different message sizes and a fixed message size overhead is more accurate for general interconnects.

## VIII. Generalizations and Conclusions

We have proposed several classes of data objects that can be distinguished by their use in parallel programs and by their invalidation traffic patterns. By merging the invalidation behavior of the individual data objects used in an application, we can gain insight into the overall invalidation behavior of the application. We can also predict the invalidation behavior beyond the 32 processor limit of our current simulation studies.

The code and read-only data objects are, in general, easy to handle. Since they are written very infrequently, they cause very few invalidations. Some directory schemes, however, do not allow a memory location to be present in more caches than there are pointers in the directory entry (for example $\mathrm{Dir}_i\mathrm{NB}$ schemes in [2]). We would normally expect such schemes to recognize code and handle it differently, thus alleviating part of the problem. However, read-only data is much harder to detect, especially since it is written at least once at initialization time, and it may cause problems for such schemes.

Migratory data objects move from one processor to another as execution progresses, but they are never manipulated by more than one processor at any one time. Migration of the data object causes at most single invalidations, because each processor writes the object before relinquishing control of it. Single invalidations are expected, even as the number of processors is increased. We note that a large number of these invalidations could be avoided if the processors (or the software) were to flush the data items out of their cache when the data were no longer needed.

Mostly-read data have potential for causing a large number of invalidations, since each write is preceded by several reads from multiple processors. The average number of invalidations caused by each write is thus high. Fortunately, writes to this kind of data are infrequent and hence the total invalidation traffic is not very large. With more processors, we expect an increase in the average number of invalidations per shared write, because it is likely that more processors will have touched the data object before a write to it takes place. This effect may be partially mitigated by taking advantage of locality, that is, by partitioning the data set and tasks such that each data portion is referenced by only a small subset of the processors.

Frequently read and written data present a big problem in terms of invalidations. Not only does each write cause several invalidations, but writes are also frequent. Frequently read/written data are expected to show increased invalidations as more processors are used, because more reads and more writes to the data items will take place. If possible, this type of data object should be avoided for parallel applications with a large number of processes. However, as in the case of high-contention synchronization objects, some hardware support such as fetch&op primitives [14], can reduce invalidation traffic.

Synchronization objects are found in all parallel applications. In well-designed applications contention for the critical sections protected by the locks is minimal and thus the invalidation traffic caused by the locks is small. As multiprocessors are scaled, it may not always be possible to avoid high-contention synchronization objects. Invalidation traffic can then be reduced by means of various hardware/software support features. For example, high-contention locks with many processes waiting can be implemented using queueing locks [13]. These locks release waiting processes one by one without causing large invalidations. Similarly, if the directory has only a few pointers per entry, the compiler may construct fan-in and fan-out trees for implementing barriers, thus reducing both the latency and the frequency with which the pointer overflow mechanism is triggered [28].

Experiments with various cache line sizes indicate that best invalidation behavior is achieved when the cache line matches the size of the data objects being shared. Both line sizes that are too small and line sizes that are too large can drive up the number of invalidations. When the line sizes are too small, each migration of an object causes several invalidations. When they are too large, false sharing may lead to additional invalidations. In terms of overall traffic, we find that the number of messages typically decreases as the line size is increased. However, since data message get larger, there is an intermediate line size that yields minimum overall traffic. Our data show that a line size of 32 bytes is quite reasonable for large-scale multiprocessors. This line size allows efficient transfer of data across a relatively high-latency network, and it is also likely to increase performance due to prefetch. The negative effects of a large line size, namely slightly larger invalidations as well as increased traffic, are still tolerable at this cache line size.

In summary, in this paper we have presented data about the invalidation patterns of five applications using 8, 16, and 32 processor runs. We have introduced a classification of data objects by invalidation behavior. This serves as a conceptual aid for reasoning about and predicting the invalidation behavior of an application. The classification is also useful for predicting the invalidation behavior beyond the number of processors currently simulated. Such extrapolations suggest that the average number of invalidations per invalidating write will remain small for well-designed applications, thus supporting the use of directory-based cache-coherence for large-scale multiprocessors. Effort has to be put into limiting contention over synchronization objects, exploiting locality, and reducing frequently read/written data objects. Finally, line size studies have shown that the overall message traffic is minimum (or close to minimum) when a cache line size of 32 bytes is used.

## Acknowledgment

We wish to thank F. Carrasco, J. McDonald, J. P. Singh, L. Soule, and J. Rose for use of their applications, and for explaining the details of the data structures used by them. We are grateful for the feedback given by members of the DASH multiprocessor project at Stanford.

## References

[1] A. Agarwal and A. Gupta, "Memory reference characteristics of multiprocessor applications under MACH," in *Proc. SIGMETRICS*, May 1988, pp. 215–225.

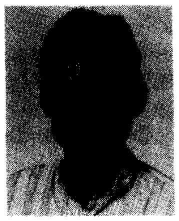
[2] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *Proc. 15th Int. Symp. Comput. Architecture*, June 1988, pp. 280–289.

[3] J. Bennett, J. Carter, and W. Zwaenepoel, "Adaptive software cache management for distributed shared memory architectures," in *Proc. 17th Int. Symp. Comput. Architecture*, May 1990, pp. 125–134.

[4] M. Berry *et al.*, "The Perfect Club benchmarks: Effective performance evaluation of supercomputers," Tech. Rep. 827, Center for Supercomput. Res. Develop., May 1989.
[5] M. Censier and P. Feautier, "A new solution to coherence problems in multicache systems," *IEEE Trans. Comput.*, vol. C-27, no. 12, pp. 1112–1118, Dec. 1978.
[6] K.M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Commun. ACM*, pp. 198–206, Apr. 1981.
[7] H. Davis and S. Goldschmidt, "Tango: A multiprocessor simulation and tracing system," Tech. Rep. CSL-TR-90-439, Stanford Univ., July 1990.
[8] S. Eggers and R. Katz, "A characterization of sharing in parallel programs and its application to coherency protocol evaluation," in *Proc. 15th Int. Symp. Comput. Architecture*, June 1988, pp. 373–382.
[9] ——, "The effect of sharing on the cache and bus performance of parallel programs," in *Proc. 3rd Int. Conf. Architectural Support for Programming Languages Oper. Syst.*, Apr. 1989, pp. 257–270.
[10] Encore Computer Corp., *Multimax Technical Summary*, 1986.
[11] A. Goldberg and R. Tarjan, "A new approach to the maximum flow problem," in *Proc. 18th ACM Symp. Theory Comput.*, 1986, pp. 136–146.
[12] J.R. Goodman, "Using cache memory to reduce processor-memory traffic," in *Proc. 10th Int. Symp. Comput. Architecture*, June 1983, pp. 124–131.
[13] J.R. Goodman, M.K. Vernon, and P.J. Woest, "Efficient synchronization primitives for large-scale cache-coherent multiprocessors," in *Proc. 3rd Int. Conf. Architectural Support for Programming Languages Oper. Syst.*, Apr. 1989, pp. 64–75.
[14] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the DASH multiprocessor," in *Proc. 17th Int. Symp. Comput. Architecture*, May 1990, pp. 148–159.
[15] T. Lovett and S. Thakkar, "The Symmetry multiprocessor system," in *Proc. Int. Conf. Parallel Processing*, vol. I, Aug. 1988, pp. 303–310.
[16] E. Lusk, R. Overbeek, *et al.*, *Portable Programs for Parallel Processors*. New York: Holt, Rinehart, and Winston, 1987.
[17] E. Lusk, R. Stevens, and R. Overbeek. *A Tutorial on the Use of Monitors in C: Writing Portable Code for Multiprocessors*, Argonne National Laboratory, Argonne, IL 60439, 1986.
[18] J. McDonald and D. Baganoff, "Vectorization of a particle simulation method for hypersonic rarified flow," in *Proc. AIAA Thermodynamics, Plasmadynamics and Lasers Conf.*, June 1988.
[19] L. Monier and P. Sindhu, "The architecture of the Dragon," in *Proc. 30th IEEE Int. Conf.*, IEEE, Feb. 1985, pp. 118–121.
[20] R. Katz, S. Eggers, D. Wood, C. Perkins, and R. Sheldon, "Implementing a cache consistency protocol," in *Proc. 12th Int. Symp. Comput. Architecture*, June 1985, pp. 276–283.
[21] J. Rose. "LocusRoute: A parallel global router for standard cells," in *Proc. Design Automat. Conf.*, June 1988, pp. 189–195.
[22] L. Rudolph and Z. Segall, "Dynamic decentralized cache consistency schemes for MIMD parallel processors," in *Proc. 12th Int. Symp. Comput. Architecture*, pp. 355–362, June 1985. Also SIGARCH Newsletter, vol. 13, issue 3, 1985.
[23] J.P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford parallel applications for shared-memory," Tech. Rep. CSL-TR-91-469, Stanford Univ., Apr. 1991.
[24] L. Soule and A. Gupta, "Characterization of parallelism and deadlocks in distributed logic simulation," in *Proc. 26th Design Automat. Conf.*, June 1989, pp. 81–86.
[25] C. Thacker and L. Stewart. "Firefly: A multiprocessor workstation," in *Proc. 2nd Int. Conf. Architectural Support for Programming Languages Oper. Syst.*, Oct. 1987, pp. 164–172.
[26] J. Torrellas, M. Lam, and J. Hennessy, "Measurement, analysis, and improvement of the cache behavior of shared data in cache-coherent multiprocessors," Tech. Rep. CSL-TR-90-412, Stanford Univ., Feb. 1990.
[27] W.-D. Weber and A. Gupta, "Analysis of cache invalidation patterns in multiprocessors," in *Proc. 3rd Int. Conf. Architectural Support for Programming Languages Oper. Syst.*, Apr. 1989, pp. 243–256.
[28] P.C. Yew, N.F. Tzeng, and D.H. Lawrie, "Distributing hot-spot addressing in large scale multiprocessors," *IEEE Trans. Comput.*, vol. C-36, no. 4, pp. 388–395, Apr. 1987.

**Anoop Gupta** is an Assistant Professor of Computer Science at Stanford University. Prior to joining Stanford, he was on the research faculty of Carnegie Mellon University, where he received the Ph.D. in 1986. His primary interests are in the design of hardware and software for large scale multiprocessors. He is currently leading the design and construction of the DASH scalable shared-memory multiprocessor at Stanford. He has also worked extensively in the area of parallel applications.

Dr. Gupta was the recipient of a DEC faculty development award from 1987 to 1989, he received the NSF Presidential Young Investigator Award in 1990, and he currently holds the Robert Noyce faculty scholar chair in the School of Engineering at Stanford.



**Wolf-Dietrich Weber** received the B.A. and B.E. degrees from Dartmouth College in 1986 and the M.S. degree in electrical engineering from Stanford University, Stanford, CA, in 1987.

He is a Ph.D. candidate in the Computer Systems Laboratory at Stanford University, where he is part of the DASH multiprocessor project. His research interests focus on directory-based cache coherence for large shared-memory multiprocessors.