International Journal of Engineering Sciences Paradigms and Researches (Volume 47, Issue: Special Issue of January 2018)
ISSN (Online): 2319-6564 and Website: www.ijesonline.com

Lowering Memory and Traffic Needs

*Mr.Gandhi Rath[1] *, Ms Smruti Mishra[2]*
*[1]*Assistant Professor,Dept. Of Computer Science and Engineering, NIT , BBSR*
[2]Assistant Professor,Dept. Of Computer Science and Engineering, NIT , BBSR
*gandhirath@thenalanda.com *, smrutimishra@thenalanda.com*

Abstract

There is a resurgence of interest in directory-based cache coherence techniques as multiprocessor systems rise beyond single bus architectures. To keep track of all processors that are caching a memory block, these systems rely on a directory. To keep the caches coherent after a write to that block, point-to-point invalidation signals are delivered. Using a bit vector per memory block with one bit for each processor is an easy approach to keep track of the identities of processors that are caching a memory block. The overall amount of the directory memory unfortunately increases as the square of the number of processors, which is problematic for large machines because the main memory grows linearly with the number of processors. A variety of strategies that employ a constrained number of In this article, we provide two straightforward methods that significantly lower directory memory requirements and invalidation traffic. As a new method of storing directory state information, we first introduce the coarse vector. While using less memory than previous limited pointer approaches, this one generates a lot less in-validation traffic. Second, we suggest sparse directories, which drastically reduce the amount of memory needed for directories by associating each entry with many memory blocks. The Stanford DASH multiprocessor architecture is used to evaluate the proposed methodologies in the study. Results show that sparse directories combined with coarse vectors can reduce storage requirements by one to two orders of magnitude with almost any performance loss.

1 Introduction

A critical design issue for shared-memory multiprocessors is the cache coherence scheme. In contrast to snoopy schemes [2], directory-based schemes provide an attractive alternative for scalable high-performance multiprocessors. In these schemes a directory keeps track of which processors have cached a given memory block. When a processor wishes to write into that block, the directory sends point-to-point messages to processors with a copy, thus invalidating all cached copies. As the number of processors is increased, the amount of state kept in the directory increases accordingly. With a large number of processors, the memory requirements for keeping a full record of all processors caching each memory block become prohibitive. Earlier studies [15] suggest that most memory blocks are shared by only a few processors at any given time, and that the number of blocks shared by a large number of processors is very small. These observations point towards directory organizations that are optimized to keep a small number of pointers per directory entry, but are also able to accommodate a few blocks with very many pointers.

We propose two methods for lowering invalidation traffic and directory memory requirements. The first is the *coarse vector* directory scheme. In the most common case of a block being shared between a small number of processors, the directory is kept in the form of several pointers. Each points to a processor which has a cached copy. When the number of processors sharing a block exceeds the number of pointers available, the directory switches to a different representation. The same memory that was used to store the pointers is now treated as a coarse bit vector, where each bit of the state indicates a group of processors. We term this new directory scheme Dir CV , where *i* is the number of pointers and *r* is the size of the region that each bit in the coarse vector represents. With all bits set, the equivalent of a broadcast is achieved. While using the same amount of memory, the proposed scheme is at least as good as the limited pointer scheme with broadcast—presented as Dir B in [1].

The second method we propose reduces directory memory requirements by organizing the directory as a cache, instead of having one directory entry per memory block. Since the total size of main memory in machines is much larger than that of all cache memory, at any given time most memory blocks are not cached by any processor and the corresponding directory entries are empty. The idea of a *sparse directory* that only contains the active entries is thus appealing. Furthermore, there is no need to have a backing store for the directory cache. The state of a block can safely be discarded after invalidation messages have been sent to all processor caches with a copy of that block. Our scheme of sparse directories brings down the storage requirements of main-memory-based directories close to that of cache-based linked list directory schemes such as the SCI scheme [8]. However, we avoid the longer latencies and more complicated protocol associated with cache-based directories.

Note that our two proposals are orthogonal. Sparse directories apply equally well to other directory entry formats as to the coarse vector scheme.

In this paper we compare the full bit vector scheme and existing limited pointer schemes with our coarse vector scheme. We also evaluate the performance of sparse directories. The performance results were obtained using multiprocessor simulations of four parallel applications. The multiprocessor simulator is based on the Stanford DASH architecture [11]. Our results show that the coarse vector scheme always does at least as well as all other limited-pointer schemes and is much more robust in response to different applications. While some applications cause one or the other directory scheme to degrade badly, coarse vector performance is always close to that of the full bit vector scheme. Using sparse directories adds less than 17% to the traffic while reducing directory memory overhead by one to two orders of magnitude.

The next section briefly introduces the DASH multiprocessor architecture currently being developed at Stanford. It will be used as a base architecture for our studies throughout the paper. The DASH architecture section is followed by background information

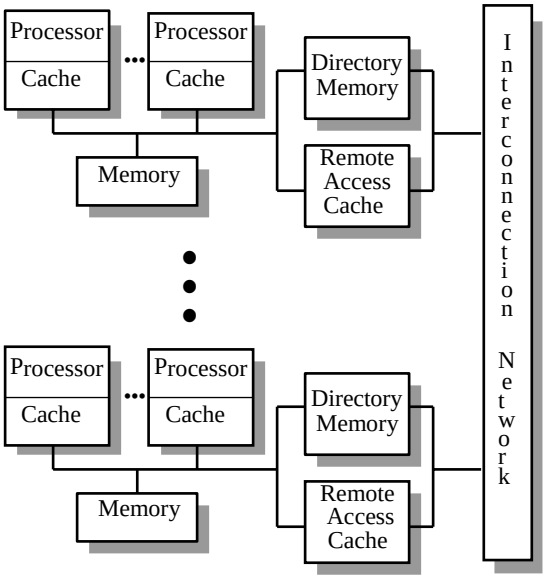

Figure 1: DASH architecture.

on directory-based cache coherence schemes, with emphasis on the memory requirements of each scheme. Section 4 introduces the directory schemes proposed in this paper. Section 5 describes the experimental environment and the parallel applications used for our performance evaluation studies. Section 6 presents the results of these studies. Sections 7 and 8 contain a discussion of the results, future work, and conclusions.

2 The DASH Architecture

The performance analysis of the different directory schemes depends on the implementation details of a given multiprocessor architecture. In this paper we have made our schemes concrete by evaluating them in the context of the DASH multiprocessor currently being built at Stanford. This section gives a brief overview of DASH [11].

The DASH architecture consists of several processing nodes (referred to as *clusters*), interconnected by a mesh network (see Figure 1). Each processing node contains several processors with their caches, a portion of the global memory and the corresponding directory memory and controller. Caches within the clusters are kept consistent using a bus-based snoopy scheme [13]. Inter-cluster consistency is assured with a directory-based cache coherence scheme [10]. The DASH prototype currently being built will have a total of 64 processors, arranged in 16 clusters of 4. The prototype implementation uses a full bit vector for each directory entry. With one state bit per cluster and a single dirty bit, the corresponding directory memory overhead is 17 bits per 16 byte main memory block, i.e., 13.3%.

What follows is a brief description of the protocol messages sent for typical read and write operations. This information is useful for understanding the message traffic results presented in Section 6. For a read, the cluster from which the read is initiated (local cluster) sends a message to the cluster which contains the portion of main memory that holds the block (home cluster). If the directory determines the block to be clean or shared, it sends the response to the local cluster. If the block is dirty, the request is sent to the owning cluster, which replies directly to the original requestor. For a write, the local cluster again sends a message to the home cluster. A directory look-up occurs and the appropriate invalidations are sent to clusters having cached copies (remote clusters). At the same time, an ownership reply is returned to the local cluster. This reply also contains the count of invalidations sent out, which equals the number of acknowledgement messages to expect. As each of the invalidations reaches its destination, invalidation acknowledgement messages are sent to the local cluster. When all acknowledgements are received by the local cluster, the write is complete.

3 Directory Schemes for Cache Coherence

Existing cache coherent multiprocessors are built using bus-based snoopy coherence protocols [12, 7]. Snoopy cache coherence schemes rely on the bus as a broadcast medium and the caches snoop on the bus to keep themselves coherent. Unfortunately, the bus can only accommodate a small number of processors and such machines are not scalable. For scalable multiprocessors we require a general interconnection network with scalable bandwidth, which makes snooping impossible. *Directory-based* cache coherence schemes [4, 14] offer an attractive alternative. In these schemes, a directory keeps track of the processors caching each memory block in the system. This information is then used to selectively send invalidations/updates when a memory block is written.

For directory schemes to be successful for scalable multiprocessors, they must satisfy two requirements. The first is that the bandwidth to access directory information must scale linearly with the number of processors. This can be achieved by distributing the physical memory and the corresponding directory memory among the processing nodes and by using a scalable interconnection network [11]. The second requirement is that the hardware overhead of using a directory scheme must scale linearly with the number of processors. The critical component of the hardware overhead is the amount of memory needed to store the directory information. It is this second aspect of directory schemes that we focus on in this paper.

Various directory schemes that have been proposed fall into the following three broad classes: (i) the full bit vector scheme; (ii) limited pointer schemes; and (iii) cache-based linked-list schemes. We now examine directory schemes in each of these three classes and qualitatively discuss their scalability and performance advantages and disadvantages. Quantitative comparison results are presented in Section 6.

Full Bit Vector Scheme (Dir)

This scheme associates a complete bit vector, one bit per processor, with each block of main memory. The directory also contains a dirty-bit for each memory block to indicate if some processor has been given exclusive access to modify that block in its cache. Each bit indicates whether that memory block is being cached by the corresponding processor, and thus the directory has full knowledge of the processors caching a given block. When a block has to be invalidated, messages are sent to all processors whose caches have a copy. In terms of message traffic needed to keep the caches coherent, this is the best that an invalidation-based directory scheme can do.

Unfortunately, for a multiprocessor with processors, bytes of main memory per processor and a block size of bytes, the directory memory requirements are ^{2} bits, which grows as the square of the number of processors. This fact makes full bit vector schemes unacceptable for machines with a very large

number of processors.

Although the asymptotic memory requirements look formidable, full bit vector directories can be quite attractive for machines with a moderate number of processors. For example, the prototype of the Stanford DASH multiprocessor [11] will consist of 64 processors organized as 16 clusters of 4 processors each. While a snoopy scheme is used for intra-cluster cache coherence, a full bit vector directory scheme is used for inter-cluster cache coherence. The block size is 16 bytes and we need a 16-bit vector per block to keep track of all the clusters. Thus the overhead of directory memory as a fraction of the total main memory is 13.3%, which is quite tolerable for the DASH multiprocessor.

We observe that one way of reducing the overhead of directory memory is to increase the cache block size. Beyond a certain point, this is not a very practical approach because increasing the cache block size can have other undesirable side effects. For example, increasing the block size increases the chances of false-sharing [6] and may significantly increase the coherence traffic and degrade the performance of the machine.

Limited Pointer Schemes

Our study of parallel applications has shown that for most kinds of data objects the corresponding memory locations are cached by only a *small* number of processors at any given time [15]. One can exploit this knowledge to reduce directory memory overhead by restricting each directory entry to a small fixed number of pointers, each pointing to a processor caching that memory block. An important implication of limited pointer schemes is that there must exist some mechanism to handle blocks that are cached by more processors than the number of pointers in the directory entry. Several alternatives exist to deal with this *pointer overflow*, and we will discuss three of them below. Depending on the alternative chosen, the coherence and data traffic generated may vary greatly.

In the limited pointer schemes we need $\log_2$ bits per pointer, while only one bit sufficed to point to a processor in the full bit vector scheme. Thus the full bit vector scheme makes more effective use of each of the bits. If we ignore the single dirty bit, the directory memory required for a limited pointer scheme with pointers is $\log_2$, which grows as $\log_2$ with the number of processors.

Limited Pointers with Broadcast Scheme (Dir B)

The Dir B scheme [1] solves the pointer overflow problem by adding a broadcast bit to the state information for each block. When pointer overflow occurs, the broadcast bit is set. A subsequent write to this block will cause invalidations to be broadcast to *all* caches. Some of these invalidation messages will go to processors that do not have a copy of the block and thus reduce overall performance by delaying the completion of writes and by wasting communication bandwidth.

The Dir B scheme is expected to do poorly if the typical number of processors sharing a block is just larger than the number of pointers *i*. In that case numerous invalidation broadcasts will result, with most invalidations going to caches that do not have a copy of the block.

Limited Pointers without Broadcast Scheme(Dir NB)

One way to avoid broadcasts is to disallow pointer overflows altogether. In the Dir NB scheme [1], we make room for an additional requestor by invalidating one of the caches already sharing the block. In this manner a block can never be present in more than *i* caches at any one time, and thus a write can never cause more than *i* invalidations.

The most serious degradation in performance with this scheme occurs when the application has read-only or mostly-read data objects that are actively shared by a large number of processors. Even if the data is read-only, a continuous stream of invalidations will result as the objects are shuttled from one cache to another in an attempt to share them between more than *i* caches. Without special provisions to handle such widely shared data, performance can be severely degraded (Section 6 presents an example).

Superset Scheme (Dir X)

Yet another way of dealing with pointer overflow is the *superset* or Dir X scheme (our terminology) suggested in [1]. In this scheme, two pointers are kept per entry. Once the pointers are exhausted, the same memory is used to keep a single composite pointer. Each bit of this composite pointer can assume three states: 0, 1, and X—where X denotes *both*. When an entry is to be added, its bit pattern is compared with that of the existing pointer. For each bit that the patterns disagree, the pointer bit is flipped to the X state.

When a write occurs and invalidations have to be sent out, each X in the composite pointer is expanded to both the 0 and 1 states. A set of pointers to processor caches result, which is a superset of the caches which actually have copies of the block. Unfortunately the composite pointer representation produces a lot of extraneous invalidations. In Section 4.1 we will show that the superset scheme is only marginally better than the broadcast scheme at accurately capturing the identities of processors caching copies of the block.

Cache-Based Linked List Schemes

A different way of addressing the scalability problem of full vector directory schemes is to keep the list of pointers in the processors caches instead of a directory next to memory [9, 16]. One such scheme is currently being formalized as the Scalable Coherent Interface [8]. Each directory entry is made up of a doubly-linked list. The head and tail pointer to the list are kept in memory. Each cache with a copy of the block is one item of the list with a forward and back pointer to the remainder of the list. When a cache wants to read a shared item, it simply adds itself to the head of the linked list. Should a write to a shared block occur, the list is unraveled one by one as all the copies in the caches are invalidated one after another.

The advantage of this scheme is that it scales naturally with the number of processors. As more processors are added, the total cache space increases and so does the space in which to keep the directory information. Unfortunately, there are several disadvantages. For one thing, the protocol required to maintain a linked list for each directory entry is more complicated than the protocol for a memory-based directory scheme, because directory updates cannot be performed atomically. Secondly, each write produces a serial string of invalidations in the linked list scheme, caused by having to walk through the list, cache-by-cache. In contrast, the memory-based directory scheme can send invalidation messages as fast as the network can accept them. Thirdly, while a memory-based directory can operate at main memory speeds and can thus be made of cheap and dense DRAM, the linked list needs to be maintained in expensive high-speed cache memory. The exploration of tradeoffs between memory-based and cache-based directories is currently an active area of research. In this paper, however, we only focus on memory-based directories as used in DASH-like architectures.

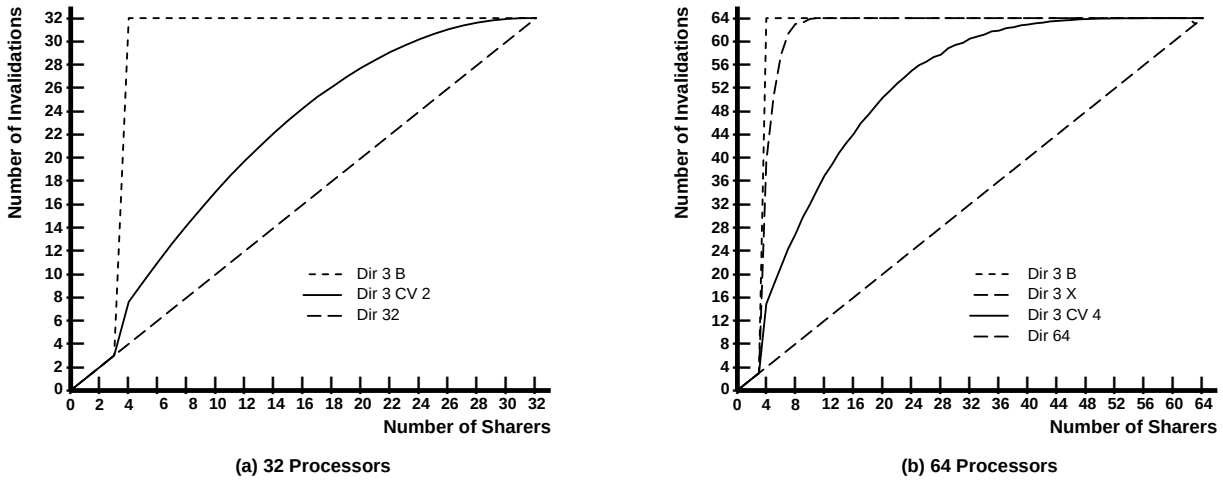

Figure 2: Average invalidation messages sent as a function of the number of sharers.

4 New Proposals

We propose two techniques to reduce memory requirements of directory schemes without significantly compromising performance and communication requirements. The first is the *coarse vector* scheme, which combines the best features of the limited pointer and full bit vector schemes. The second technique is the *sparse directory*, which uses a cache without a backing store.

Coarse Vector Scheme (Dir CV)

To overcome the disadvantages of the limited pointer scheme, without losing the advantage of reduced memory requirements, we propose the coarse vector scheme (Dir CV). In this notation, *i* is the number of pointers and *r* is the size of the region that each bit in the coarse vector represents. Dir CV is identical to the other limited pointer schemes when there are no more than *i* processors sharing a block. Each of the *i* pointers stores the identity of a processor that is caching a copy of the block. However, when pointer overflow occurs, the semantics are switched, so that the memory used for storing the pointers is now used to store a coarse bit vector. Each bit of this bit vector stands for a group of *r* processors. The region size *r* is determined by the number of directory memory bits available. While some accuracy is lost over the full bit vector representation, we are neither forced to throw out entries (as in Dir NB) nor to go to broadcast immediately (as in Dir B).

Figure 2 makes the different behaviour of the broadcast and coarse vector schemes apparent. In the graph, we assume that the limited pointer schemes each have three pointers. The graph shows the average number of invalidations sent out on a write to a shared block as the number of processors sharing that block is varied. For each invalidation event, the sharers were randomly chosen and the number of invalidations required was recorded. After a very large number of events, these invalidation figures were averaged and plotted.

In the ideal case of the full bit vector (stipple line) the number of invalidations is identical to the number of sharers. For the other schemes, we do not have full knowledge of who the sharers are, and *extraneous* invalidations need to be sent. The areas between the stipple line of the full bit vector scheme and the lines of the other schemes represent the number of extraneous invalidations for that scheme. For the Dir_3 B scheme, we go to broadcast as soon as the three pointers are exhausted. This results in many extraneous invalidations. The Dir_3 X scheme uses a composite pointer once pointer overflow occurs, and the graph shows that its behaviour is almost as bad as that of the broadcast scheme. The composite vector soon contains mostly Xs and is thus close to a broadcast bit. The coarse vector scheme, on the other hand, retains a rough idea of which processors have cached copies. It is thus able to send invalidations to the *regions* of processors containing cached copies, without having to resort to broadcast. Hence the number of extraneous invalidations is much smaller.

The coarse vector scheme also has advantages in multiprogramming environments, where a large machine might be divided between several users. Each user will have a set of processor regions assigned to his application. Writes in one user's processor space will never cause invalidation messages to be sent to caches of other users. Even in single application environments we can take advantage of data locality by placing processors that share a given data set into the same processor region.

Sparse Directories

Typically the total amount of cache memory in a multiprocessor is much less than the total amount of main memory. If the directory state is kept in its entirety, we have one entry for each memory block. Most blocks will not be cached anywhere and the corresponding directory entries will thus be empty. To reduce such a waste of memory, we propose the *sparse directory*. This is a directory *cache*, but it needs no back-up store because we can safely replace an entry of the sparse directory after invalidating all processor caches which that entry points to.

As an example, if a given machine has 16 MBytes of main memory per processor and 256 KBytes of cache memory per processor, no more than 1/64 or about 1.5% of all directory entries will be used at any one time. By using a directory cache of suitable size, we are able to drastically reduce the directory memory. Thus either the machine cost is lowered, or the designer can choose to spend the saved memory by making each entry wider. For example, if the Dir CV scheme were used with a sparse directory, more pointers *i* and smaller regions *r* would result. The directory cache size should be chosen to be at least as large as the total number of cache blocks. An additional factor of 2 or 4 will reduce the probability of contention over sparse directory entries if memory access patterns are skewed to load one directory more heavily than the others. This contention occurs when several memory blocks mapping to the same directory entry exist in processor caches and thus

Table 1: Sample machine configurations.

number of clusters	number of processors	total main memory space (MBytes)	total processor cache space (MBytes)	block size (Bytes)	directory scheme	directory overhead
16	64	1024	16	16	Dir_{16}	13.3%
64	256	4096	64	16	sparse Dir_{64}	13.1%
256	1024	16384	256	16	sparse Dir_8CV_4	13.3%

keep knocking each other out of the sparse directory. Similar reasoning also provides a motivation for making the sparse directory set-associative. Since sparse directories contain a large fraction of main memory blocks, tags need only be a few bits wide. Sparse directories are expected to do particularly well with a DASH-style architecture. In DASH, no directory entries are used if data from a given memory module is cached only by processors in that cluster. Since we expect processes to allocate their non-shared data from memory on the same cluster, no directory entries will be used for such data. Furthermore with increasing locality in programs, fewer data items will be remotely allocated and thus fewer directory entries will be needed.

The ratio of main memory blocks to directory entries is called the *sparsity* of the directory. Thus if the directory only contains 1/16 as many entries as there are main memory blocks, it has sparsity 16. Table 1 shows some possible directory configurations for machines of different sizes. For these machines, 16 MBytes of main memory and 256 KBytes of cache were allocated per processor. A directory memory overhead of around 13% has been allowed throughout. Processors have been clustered into processing nodes of 4—similar to DASH. The first line of the table is close to the DASH prototype configuration. There are 64 processors arranged as 16 clusters of 4 processors. For this machine, the full bit vector scheme Dir_{16} is easily feasible. As the machine is scaled to 256 processors, we keep the directory memory overhead at the same level by switching to sparse directories. The sparse directories contain entries for 1/4 of the main memory blocks (sparsity 4). As we shall see in Section 6, even much sparser directories still perform very well. For the 1024 processor machine, the directory memory overhead is kept constant and the entry size is kept manageable by using a coarse vector scheme (Dir_8CV_4) in addition to using a directory with sparsity 4. Note that this is achieved without having to resort to a larger cache block size.

5 Evaluation Methodology

We evaluated the directory schemes discussed in the previous sections using an event-driven simulator of the Stanford DASH architecture. Besides studying overall execution time of various applications, we also looked at the amount and type of message traffic produced by the different directory schemes.

Our simulations utilized Tango [5] to generate multiprocessor references. Tango allows a parallel application to be executed on a uniprocessor while keeping the correct global event interleaving intact. Global events are references to shared data and synchronization events such as lock and unlock requests. Tango can be used to generate multiprocessor reference traces, or it can be coupled with a memory system simulator to yield accurate multiprocessor simulations. In the latter case the memory system simulator returns timing information to the reference generator, thus preserving a valid interleaving of references. We used this second method for our simulations.

Our study uses four benchmark applications derived from four different application domains. LU comes from the numerical domain and computes the L-U factorization of a matrix. DWF is from the medical domain and is a string matching program used to search gene databases. MP3D comes from aeronautics. It is a 3-dimensional particle simulator used to study airflow in the upper atmosphere. Finally, LocusRoute is a commercial quality standard cell routing tool from the VLSI-CAD domain.

Table 2: General application characteristics.

Application	shared refs (mill)	shared reads (mill)	shared writes (mill)	sync ops (thou)	shared space (MBytes)
LU	8.9	6.0	2.9	13	0.65
DWF	17.5	16.2	1.0	277	3.89
MP3D	13.5	8.8	4.7	1	3.46
LocusRoute	21.3	20.2	1.1	24	0.72

Table 2 presents some general data about the applications. It shows the total number of shared references in the application run and the breakdown into reads and writes. Shared references are defined as references to the globally shared data sections in the applications. The number of shared references varied slightly from run to run for the non-deterministic applications (LocusRoute and MP3D). We show the values for the full cache, non-sparse, full bit vector runs. The table also gives the amount of shared data touched during execution, which is an estimate of the data set size of the program.

All runs were done with 32 processors and a cache block size of 16 bytes. We did not use more processors because currently few of our applications achieve good speedup beyond 32 processors. For our evaluation studies, we assumed that a directory memory overhead around 13% was tolerable, which allowed us about 17 bits of directory memory per entry. This restricts the limited pointer schemes to three pointers and the coarse vector scheme to regions of size two. The schemes examined in this study are thus Dir_3CV_2, Dir_3B and Dir_3NB. We also used Dir_{32}, the full bit vector scheme, for comparison purposes. Once sparse directories are introduced, the overhead naturally drops dramatically—by one to two orders of magnitude, depending on sparsity. For example, a full bit vector directory with sparsity 64 requires 32 bits to keep track of the processor caches, 1 dirty bit, and 6 bits of tag. Instead of 33 bits per 16-byte block we now have 39 bits for every 64 blocks, a savings factor of 54.

The DASH simulator is configured with parameters that correspond to those of the DASH prototype hardware. The processors have 64 KByte primary and 256 KByte secondary caches. Local bus requests take on the order of 23 processor cycles. Remote requests involving two clusters take about 60 cycles and remote

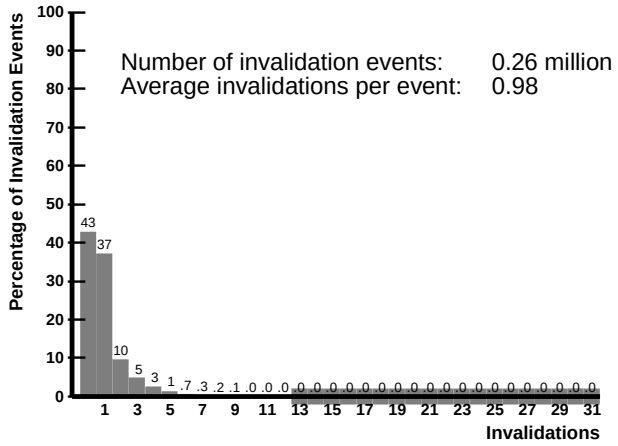

Figure 3: Invalidation distribution, LocusRoute, Dir $_{32}$.

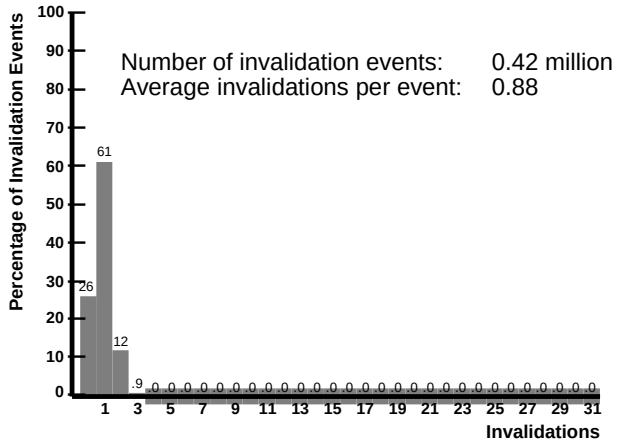

Figure 4: Invalidation distribution, LocusRoute, Dir $_3$NB.

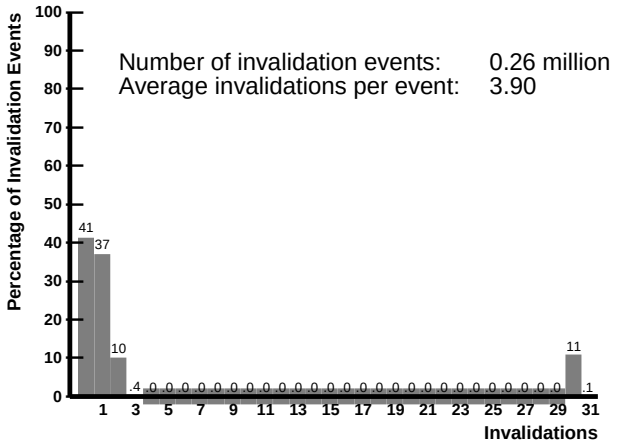

Figure 5: Invalidation distribution, LocusRoute, Dir $_3$B.

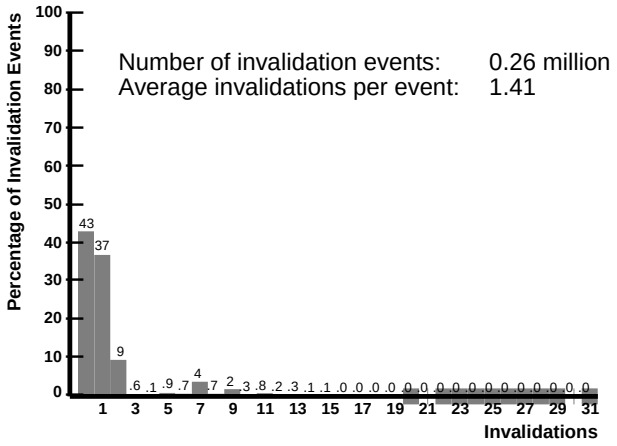

Figure 6: Invalidation distribution, LocusRoute, Dir $_3$CV$_2$.

requests with three clusters have a latency of about 80 processor cycles. In the simulator, main memory is evenly distributed across all clusters and allocated to the clusters using a round-robin scheme.

The following messages classes are used by the simulator:

Request messages are sent by the caches to request data or ownership.

Reply messages are sent by the directories to grant ownership and/or send data.

Invalidation messages are sent by the directories to invalidate a block.

Acknowledgement messages are sent by caches in response to invalidations.

The simulator also collects statistics on the distribution of the number of invalidations that have to be sent for each write request. The invalidation distribution helps explain the behaviour of the different directory schemes.

6 Simulation Results

The results presented in this section are subdivided as follows. The first subsection gives invalidation distributions for the different directory schemes. These impart an intuitive feel for how the different schemes behave and discusses their advantages and disadvantages. The next two subsections present the results of our main study. The first one contrasts the performance of our coarse vector scheme with that of other limited-pointer schemes. The second subsection presents results regarding the effectiveness of sparse directories.

Invalidation Distributions

Figures 3-6 give the invalidation distributions of shared data for the LocusRoute application. We do not present results for other applications for space reasons. Also, the LocusRoute distributions illustrate the trends of the different schemes well. In Figure 3 we see the distribution for the full bit vector scheme (Dir$_{32}$) which is the *intrinsic* invalidation distribution and is the best that can be achieved. In the case of the Dir$_{32}$ scheme, only writes that miss or hit a clean block are invalidation events. We note that most writes cause very few invalidations, but that there are also some writes that cause a large number of invalidations. The number of invalidation events is 0.26 million and each event on average causes 0.98 invalidations for a total of 0.25 million invalidations.

Figure 4 shows the invalidation distribution for Dir$_3$ NB. Since no broadcasts are allowed, no more than three caches can share a

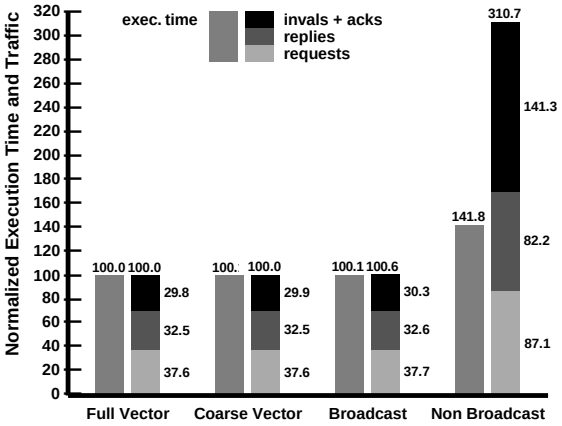

Figure 7: Performance for LU.

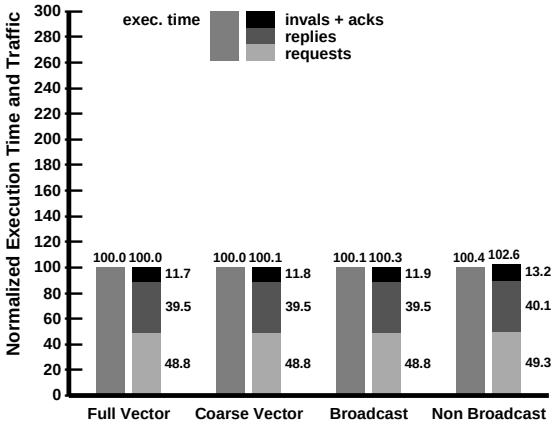

Figure 9: Performance for MP3D.

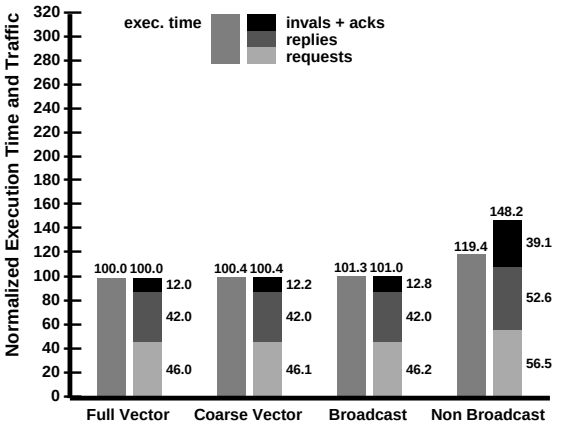

Figure 8: Performance for DWF.

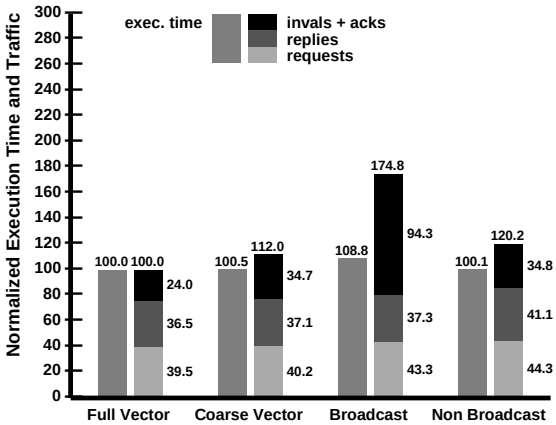

Figure 10: Performance for LocusRoute.

given block at any one time. This also means that we never see more than three invalidations per write. Unfortunately, there are also many new single invalidations, caused by replacements when a block wants to be shared by more than three caches. For Dir_3NB it is possible for reads to cause invalidations, and this is why the number of invalidation events is so much larger. Although the average number of invalidations per event has decreased to 0.88, the total number of invalidations has increased to 0.37 million.

The distribution for Dir_3B is shown in Figure 5. We see that the number of smaller invalidations goes back to the level seen for the full vector scheme. However, any writes that caused more than three invalidations in the full vector scheme now have to broadcast invalidations. For most broadcasts, 30 clusters have to be invalidated, since the home cluster and the new owning cluster do not require an invalidation. This serves to drive the average invalidations per event up to 3.9 and the total to 1.01 million invalidations.

In the Dir_3CV_2 scheme, shown in Figure 6, we are able to respond to the larger invalidations without resorting to broadcast. The peaks at odd numbers of invalidations are caused by the granularity of the bit vector. Also note the absence of the large peak of invalidations at the right edge that was present for the broadcast scheme. There are an average of 1.41 invalidations per event and 0.36 million total invalidations.

In conclusion, we see that the both the broadcast and non-broadcast schemes can cause invalidation traffic to increase. In the case of the broadcast scheme this increase is due to the broadcast invalidations, which can be relatively frequent if there are only a small number of pointers. For the non-broadcast scheme, the extra invalidations are caused by replacing entries when more caches are sharing a block than there are pointers available. The coarse vector scheme strikes a good balance by avoiding both of these drawbacks and is thus able to achieve performance closer to the full bit vector scheme.

Performance of Different Directory Schemes

Figures 7-10 show the performance achieved and data/coherence messages produced by the different directory schemes for each of the four applications. All runs use 32 processors, 64 KByte primary and 256 KByte secondary caches, and a cache block size of 16 bytes. The total number of messages is broken down into requests (which include writebacks), replies, and invalidation+acknowledgement messages.

Observe that the number of request and reply messages is about the same for the first three schemes (Dir , Dir CV and Dir B) for a given application. This is expected since all three schemes have similar request and reply behaviour. Dir CV and Dir B oc-

casionally send out extraneous invalidations, but that is the only difference compared to the full bit vector scheme. For Dir NB, on the other hand, invalidations sometimes have to be sent even for *read* requests, when pointer overflow occurs. These invalidations can later cause additional read misses with the associated increase in request and reply messages.

Let us now look at each of the applications individually and discuss the results. LU exhibits the problem discussed in the previous paragraph. In Figure 7, we see a greatly increased number of request and reply messages as well as a very large number of invalidation and acknowledgement messages for the Dir NB scheme. In LU each matrix column is read by all processors just after the pivot step. This data is actively shared between many processors and Dir NB does very poorly.

Read-shared data is also the cause of the poorer performance of Dir NB for DWF. The pattern and library arrays are constantly read by all the processes during the run. The other schemes are virtually indistinguishable.

In MP3D (Figure 9) most of the data is shared between just one or two processors at any given time. This sharing pattern causes an invalidation distribution that all schemes can handle well. The coarse vector and broadcast schemes show almost no increase in execution time or message traffic, and even the non-broadcast scheme takes only 0.4% longer to run.

LocusRoute (Figure 10) is interesting in that it is the only application in which the Dir NB scheme outperforms Dir B. The central data structure of LocusRoute is shared amongst several processors working on the same geographical region. Whenever the number of sharers exceeds the number of pointers in Dir B, a broadcast results on a write. The Dir NB scheme does better with this kind of object, because the invalidations due to pointer overflow often do not cause re-reads.

Throughout this section the message traffic numbers diverge more than the execution times for the various schemes. Since we simulate a 32 cluster multiprocessor with 32 processors, there is only one processor per cluster. The local cluster bus is thus underutilized. In a real DASH system, with four processors to a cluster, the cluster bus will be much busier. We consequently expect the performance degradation due to an increased number of messages to be larger than shown here.

Comparing the performance of the different schemes for the various applications, we see that the Dir NB does much worse than the other schemes for most applications. Only in LocusRoute does it perform better than one of the other schemes. Secondly, while we expect the Dir CV scheme to always perform as well as the broadcast scheme, we see that it can do significantly better for some applications. Finally, we note that the coarse bit vector scheme sends very few extraneous messages. For the worst case application (LocusRoute) Dir CV only sends about 12% more messages than the ideal full bit vector scheme.

Performance of Sparse Directories

The method used for evaluating sparse directories was very similar to that used to evaluate the different directory schemes. There were two key differences. Firstly, the simulator was configured to use a sparse directory instead of keeping a complete directory. Secondly, we used *scaled* processor caches to achieve a more realistic size relationship of the sparse directories and processor caches. The slow speed of the simulator limited us to relatively small application data sets. As a result, if we had used the regular 256 KBytes of cache per processor, the whole data set would have fit into the caches. In such a case we would have been unable to experiment with sparse directories larger than the processor caches but smaller than the total memory blocks in the system. Instead, the caches were scaled to keep the ratio of data set size to cache size of our runs similar to that of data set size to cache size for a full blown application problem on a real DASH multiprocessor. For example, for DWF a full blown problem on a 64-processor DASH would occupy all of the 1 Gbyte of main memory (see Table 1). This is 64 times the total cache space. In our simulation, the data set size was 3.9 MBytes. So to preserve the data set to cache ratio, the total cache space for our 32-processor simulation was reduced to 64 KBytes, which is 2 KBytes per processor. We experimented with sparse directories that have entries from one to four times the total number of cache lines in the system (shown as size factor 1 to 4 in the graphs).

When an entry needs to be allocated in the sparse directory, we first look to see whether the slot it maps to is empty. If so, it is filled. Otherwise we have to replace an existing entry. Invalidations are sent out and the now empty slot is filled. Empty slots are also created when a processor cache replaces and writes back a dirty line.

Effect of Sparsity

Figures 11-12 show the effect of directory sparsity on performance. We chose to present results for LU and DWF only. The results for MP3D were very similar to those of DWF, so for lack of space we omit them here. For LocusRoute, even for full-scale runs the data set is expected to be small enough that sparse directories will perform as well as non-sparse directories. So again we omit the results in this subsection.

In Figures 11 and 12 we show execution times for LU and DWF as the directory sparsity is varied. We consider the cases where the number of directory entries in the system is a factor of 1, 2, or 4 times the total number of cache blocks in the system. For these runs we used sparse directories of associativity 4 and use a random replacement policy (see below). The results suggest that even directories with the same size as the processor caches perform well. The worst case application (LU) shows only a 10.4% increase in execution time when going from a non-sparse, full bit vector directory to a sparse directory equal in size to the processor caches. When the directory size is increased to 2 or 4 times the cache size, the performance degradation of sparse directories is very small.

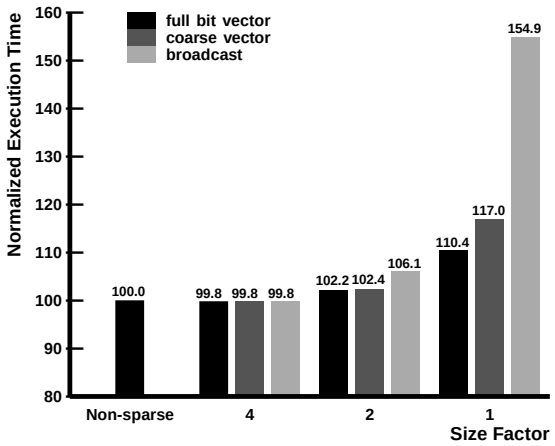

Figure 11: Sparse directory performance for LU.

For the size factor 1 directory in LU we see a large performance difference between the coarse vector and the broadcast schemes.

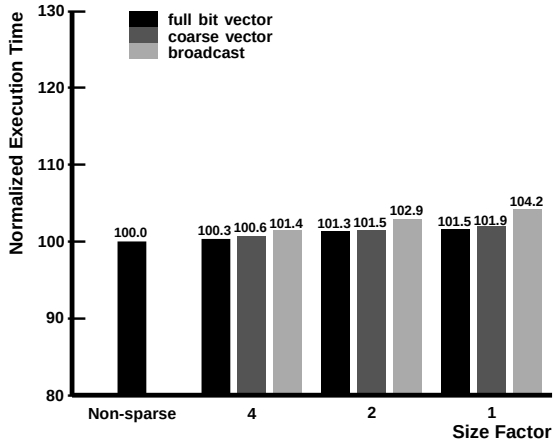

Figure 12: Sparse directory performance for DWF.

In LU, the pivot column is shared between all processors. When directory replacements are more frequent, as is the case for very sparse directories, only some of the processes may get a chance to access this data between replacements. When the replacement does occur, enough sharers exist to cause a broadcast for the Dir B scheme while the Dir CV only needs to send a few invalidations.

For DWF the performance is fairly flat across schemes and size factors. The performance does not vary much from scheme to scheme because the invalidation behaviour of DWF is handled equally well by all schemes. The performance is flat across size factors because DWF is a wave-front algorithm that has a relatively small working set at any moment in time. This ensures that even very sparse directories do not suffer from excessive replacements.

Effect of Associativity and Replacement Policy

Since a sparse directory has fewer entries than main memory has blocks, it is possible for several active blocks to map to the same directory entry. While a set-associative sparse directory can handle this situation, entries in a direct mapped sparse directory would keep bumping each other out, leading to poor directory performance.

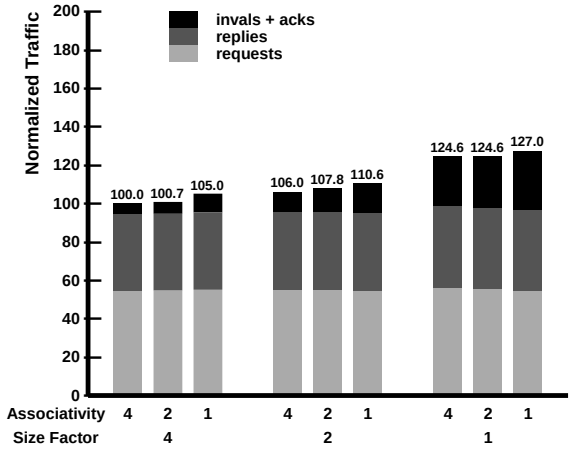

Figure 13: Effect of associativity in sparse directory (LU).

We used LU as a sample application to study the effect of sparse directory associativity and replacement policy. The full bit vector scheme was used in these studies. Figure 13 shows message traffic numbers for associativities of 1, 2 and 4 with directory size factors 1, 2 and 4. We show traffic numbers because they show the trends better than the execution time results. For each of the size factors, associativity 4 is equal to or slightly better than associativity 2, which in turn is better than direct-mapped by a larger margin. The benefits from set-associativity seem to be small, but we do expect associativity to make sparse directories more robust to different application behaviours.

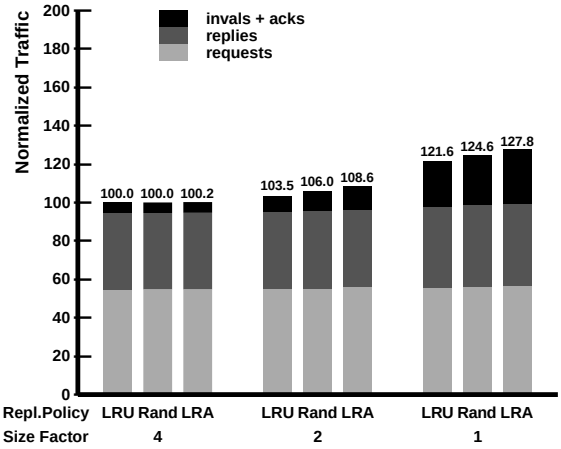

Figure 14: Effect of replacement policies in sparse directory (LU).

For set-associative directories, there is a choice of replacement policies. We explored random, least-recently-used (LRU) and least-recently-allocated (LRA) schemes. LRU keeps the different sets in each entry ordered by time of access and replaces the least recently used one. LRA only keeps track of the allocation time of each set in the entry and replaces the one that was allocated first. The results for an LU run using a sparse directory with set-associativity 4 and a full bit vector scheme are shown in Figure 14. LRU is the most difficult to implement, and also performs the best. Even though random is the easiest to implement in hardware, it actually does better than LRA. With LRA the possibility of replacing entries that were allocated early, yet are used frequently exists. This soon leads to more replacements when the frequently used entries are accessed again.

7 Discussion

The question arises whether our proposals introduce additional complexities into the architecture. The answer is very few. The coarse vector scheme does not require any modification to the protocol used for the full bit vector scheme. It merely ends up sending some extraneous invalidations. For sparse directories, on the other hand, some protocol modification is required. When an entry is being replaced in the sparse directory, and is thus effectively removed from the system, we have to invalidate all copies of the corresponding memory block cached in processor caches. Some entity has to keep track of when all the acknowledgements for these invalidations have been received. Such an entity must already exist in systems that implement weak consistency, in order to keep track of outstanding invalidations. In DASH, we have the Remote Access Cache (RAC). When a block is to be replaced in the sparse directory, the RAC allocates an entry for that block and invalidations are sent out to all cached copies. The RAC receives

the acknowledgement messages sent in response to these invalidations. The operation is complete when all acknowledgements have been received.

Another hardware issue concerns synchronization. In DASH, the directory bit vectors are also used to keep track of processors queued for a lock. In the case of the full bit vector we have enough space to keep track of all nodes. Consequently, when a lock is released, it is granted to exactly one of the waiting nodes. Once we switch to a coarse vector scheme, that is no longer the case. We are only able to keep track of which processor *regions* are queued for a lock. When the lock is released, and we wish to grant it to another node, we have to release all processors in that region and let them try to regain the lock. While this mechanism is slightly less efficient, it still avoids having to release *all* waiting processors and causing a hot spot when they all try to obtain the lock.

There are many other techniques that can be used to reduce the memory requirements of directory-based cache coherence schemes. For example, as suggested in [3], we can associate small directory entries with each memory block and allow these to overflow into a small cache of much wider entries. Similarly, we can make multiple memory blocks share one wide entry. We plan to evaluate some of these alternative schemes in the future.

8 Conclusions

We have presented two techniques for reducing the memory overhead and data/coherence traffic of directory cache coherence schemes—the coarse vector scheme and sparse directory scheme. The performance of the new schemes was analysed and compared to existing directory schemes. Our results show that the savings achieved in memory overhead and the traffic reduction are significant. Depending on the application, the coarse vector scheme produces up to 8% less memory message traffic than the next best limited pointer scheme and several factors less than the worst limited pointer scheme. The coarse vector scheme is also more robust than the other limited pointer schemes—its performance is always closest to the full bit vector scheme. While sparse directories add up to 17% to the memory coherence traffic, they can significantly reduce the directory memory overhead—by one to two orders of magnitude, depending on sparsity. We believe that a combination of the two techniques presented will allow machines to be scaled to hundreds of processors while keeping the directory memory overhead reasonable.

9 Acknowledgements

We would like to thank Helen Davis and Steven Goldschmidt for creating and supporting Tango. Dan Lenoski, Jim Laudon and Kourosh Gharachorloo provided insightful discussions. Dan Lenoski also provided Figure 1. Thanks to all the people who patiently stood by while we brought their machines to their knees with our runs. Lastly we would like to thank Henk Goosen, Jim Laudon, Dan Lenoski, Margaret Martonosi, Ed Rothberg and Mike Smith for reviewing an early version of this paper. Anoop Gupta and Todd Mowry are supported by DARPA contract N00014-87-K-0828. Wolf-Dietrich Weber is supported by an IBM Graduate Fellowship.

References

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *15th International Symposium on Computer Architecture*, 1988.

[2] James Archibald and Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

[3] James K. Archibald. *The Cache Coherence Problem in Shared-Memory Multiprocessors*. PhD thesis, Department of Computer Science, University of Washington, February 1987.

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

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

[6] S. Eggers and R. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, May 1989.

[7] Encore Computer Corporation. *Multimax Technical Summary*, 1986.

[8] P1596 Working Group. P1596/Part IIIA - SCI Cache Coherence Overview. Technical Report Revision 0.33, IEEE Computer Society, November 1989.

[9] Tom Knight, March 1987. Talk at Stanford Computer Systems Laboratory.

[10] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of 17th International Symposium on Computer Architecture*, 1990.

[11] Dan Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. Design of Scalable Shared-Memory Multiprocessors: The DASH Approach. In *Proceedings of COMPCON'90*, pages 62–67, 1990.

[12] Tom Lovett and Shreekant Thakkar. The Symmetry Multiprocessor System. In *Proceedings of the International Conference on Supercomputing*, pages 303–310, 1988.

[13] M. Papamarcos and J. Patel. A low Overhead Coherence Solution for Multiprocessors with private Cache Memories. In *Proceedings of 11th International Symposium on Computer Architecture*, pages 348–354, 1984.

[14] C. K. Tang. Cache Design in the Tightly Coupled Multiprocessor System. In *AFIPS Conference Proceedings, National Computer Conference, NY, NY*, pages 749–753, June 1976.

[15] Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.

[16] John Willis. Cache Coherence in Systems with Parallel Communication Channels & Many Processors. Technical Report TR-88-013, Philips Laboratories – Briarcliff, March 1988.