CACHE COHERENCE DIRECTORIES FOR SCALABLE MULTIPROCESSORS

Richard Simoni

Technical Report: CSL-TR-92-550

October 1992

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 943054055

Abstract

Directory-based protocols have been proposed as an efficient means of implementing cache coherence in large-scale shared-memory multiprocessors. This thesis explores the trade-offs in the design of cache coherence directories by examining the organization of the directory information, the options in the design of the coherency protocol, and the implementation of the directory and protocol.

The traditional directory organization that maintains a full valid bit vector per directory entry is unsuitable for large-scale machines due to high storage overhead. This thesis proposes several alternate organizations. ***Limited pointers*** directories replace the bit vector with several ***pointers*** that indicate those caches containing the data. Although this scheme performs well across a wide range of workloads, its performance does not improve as the read/write ratio becomes very large. To address this drawback, a ***dynamic pointer allocation*** directory is proposed. This directory allocates pointers from a pool to particular memory blocks as they are needed. Since the pointers may be allocated to any block on the memory module, the probability of running short of pointers is very small. Among the set of possible organizations, dynamic pointer allocation lies at an attractive cost/performance point.

Measuring the performance impact of three coherency protocol features makes the virtues of simplicity clear. Adding a clean/exclusive state to reduce the time required to write a clean block results in only modest performance improvement. Using request forwarding to transfer a dirty block directly to another cache that has requested it yields similar results. For small cache block sizes, write hits to clean blocks can be simply treated as write misses without incurring significant extra network traffic. Protocol features designed to improve performance must be examined carefully, for they often complicate the protocol without offering substantial benefit.

Implementing directory-based coherency presents several challenges. Methods are described for preventing deadlock, maintaining a mode1 of parallel execution, handling subtle situations caused by temporary inconsistencies between cache and directory state, and tolerating out-of-order message delivery. Using these techniques, cache coherence can be added to large-scale multiprocessors in an inexpensive yet effective manner.

Key Words and Phrases; cache coherence, cache consistency, shared memory, directory-based, limited pointers, dynamic pointer allocation, protocol

Acknowledgements

I would like to thank my advisor, Mark Horowitz, for the steadfast support he has given me throughout my years at Stanford. Mark is that rare person who is not only technically brilliant, but also has the patience and desire to teach. It has been a privilege to know him and to work with him. I also thank John Hennessy and Stephen Harris for serving diligently on my reading committee, and Anoop Gupta and Greg Kovacs for serving on my oral examination committee. I greatly appreciate their efforts on my behalf. I would like to express my gratitude to Charlie Org-ish and Laura Schrager for putting in many long hours to keep our computers running smoothly. Thanks also to Margaret Rowland and Darlene Hadding, whose administrative help has proved invaluable time and again.

There are many other friends and colleagues who have had a strong impact on my journey through graduate school. Anant Agarwal must bear part of the blame for this thesis, as it was he who first piqued my interest in the topic. I thank him and the rest of the MIPS-X and DASH groups for being great folks to hang out with, both at work and play. I would also like to thank my fellow members of the Diners Club, and especially Craig Dunwoody and John Vlissides, for their friendship and for keeping me laughing through miasmas thick and thin. I have enjoyed the good company and cheer of many others as well, including Arturo Salz, Doug Pan, Steve Richardson, Bob Alverson, Steven Przybylski, Mike Chow, John Acken, and Karen Huyser.

Most importantly, my deepest gratitude and affection go to Ma and Da, my parents, and to Helen, my wife and best friend. I cannot express how much their love and support mean to me. I dedicate this work to them.

This research was funded by a NSF Graduate Fellowship Award and by DARPA contract N00039-91-C-0138.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Parallel processing has long been regarded as a promising architectural means of boosting computer performance; however, in practice this benefit has often proven elusive. In particular, the cost/performance of machines built with a modest degree of parallelism is often nearly outstripped at their introduction by easier-to-build uniprocessors that are brought to market with more advanced technology. Nevertheless, technology improvements continue to shrink the volume of space occupied by a single processing node, making it possible to build ***large-scale*** multiprocessors incorporating hundreds or thousands of processors, with the attendant potential for speeding program execution by several orders of magnitude. A continuing obstacle to realizing this potential is the difficulty of programming such machines. To combat this problem, a great deal of interest has developed in building large-scale multiprocessors in which the hardware provides a simple ***shared memory*** programming model. This model allows processors to communicate by reading and writing a single address space shared by all of them.

One of the primary challenges facing designers of these machines is building an efficient memory system. The speed of CPUs continues to improve faster than the speed of high-density memory chips, making it increasingly difficult to supply processors with sufficient memory bandwidth. Sharing data further exacerbates this memory bandwidth problem, since multiple processors may be referencing the same memory modules. Furthermore, it is impossible to physically locate all memory nearby all of the processors, so some references must incur long access latencies due to interconnect delays. Average memory latency can be reduced somewhat by distributing the global shared memory across the processing nodes. Even so, if a processor is actively sharing data, then it will reference some data that is not locally resident.

One approach to improving the characteristics of the memory system is to follow the lead of uniprocessor designers by pairing each processor with a cache. Caches improve latency and bandwidth by using small, fast memories that are tightly coupled with the CPU. In a multiprocessor, they also reduce the bandwidth strain due to data sharing by allowing each processor to cache its own copy of a shared data value. Unfortunately, allowing multiple processors to simultaneously cache a given datum leads to the cache ***consistency*** problem, also known as the cache ***coherence*** problem. If one processor writes a shared data value in its cache, the other cached copies of the data become stale. A processor that reads a stale copy of the data does not receive the most recently written value, violating the shared memory model we would like to provide.

The simplest solution to the cache coherence problem is to disallow the caching of shared data. The performance effects stemming from the longer latencies that result from caching only private data are demonstrated by Figure 1.1. [1] The vertical axis shows the fraction of uniprocessor utilization that is achieved by each processor. The horizontal axis shows the fraction of data references that are to shared data. The solid curve indicates the results if all data sharing occurs between neighboring processors, while the other curves represent sharing between randomly selected processors in different sized systems. Even for the best case of nearest-neighbor sharing, only 15% shared data references are required to drop the per-processor utilization to half that seen on a uniprocessor. Yet shared data percentages of 30% or higher are not uncommon in shared memory applications [45]. Clearly, if shared memory machines are to give reasonable performance in the presence of significant data sharing, the latency problem must be addressed.

We can reduce the latency of accesses to shared data by allowing that data to be cached, and including a mechanism to maintain consistency across the caches. The consistency mechanism comes into play when a processor writes a piece of data that resides in multiple caches. Each of the caches that now contain a stale copy of the data

[1]Full details of the performance model used to generate this data are given in Appendix A.

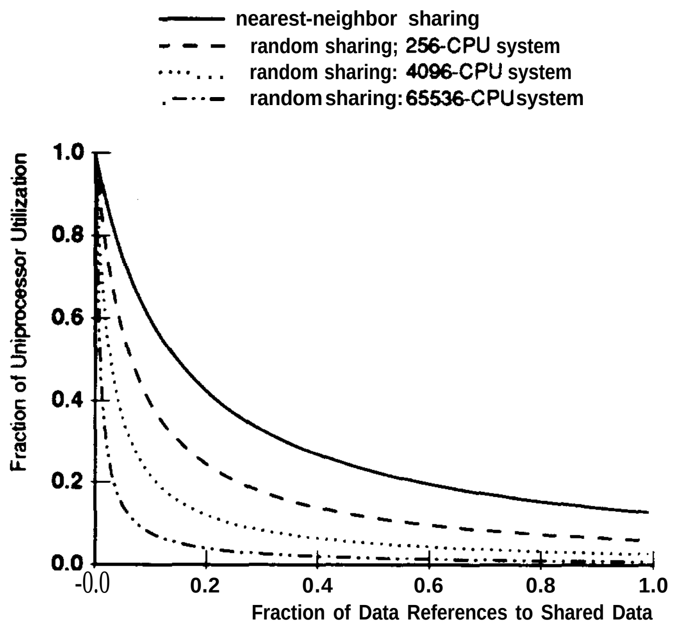

Figure 1.1: Per-processor performance without caching shared data.

is notified. In an ***invalidation*** protocol, those caches invalidate their stale copy. If a processor later accesses the data again, the reference will miss in the cache and an up-to-date copy is retrieved from main memory. In an ***update*** protocol, the newly-written value is distributed to the caches containing the data so that they may bring their stale copies up-to-date.

Figure 1.2 shows the performance of a machine that uses an invalidation-based scheme to allow shared data to be cached. As in Figure 1.1, the vertical axis indicates the per-processor utilization, as a fraction of uniprocessor utilization. However, the horizontal axis does not show the fraction of data references that are to shared data (as it did in Figure 1.1), since many shared references hit in the caches and therefore do not require main memory to be accessed. Instead, the horizontal axis shows the miss ratio due to invalidations for all data references. As before, the curves show both nearest-neighbor sharing and random sharing in different sized systems. Each case is now represented by two curves, to show a range of results depending on the fraction of data references that are to shared data. The upper curve sets this fraction to 0.2, while the lower curve uses a fraction of 0.8. Since the curves diverge primarily at low miss ratios where the fraction

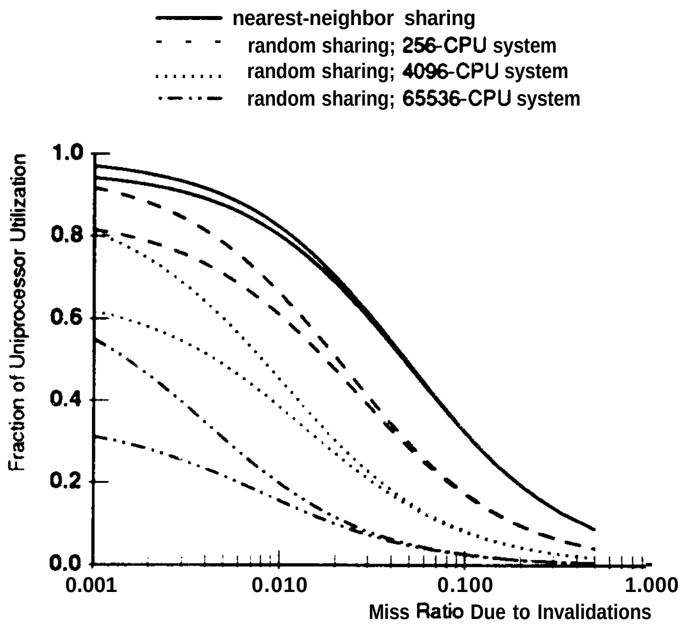

Figure 1.2: Per-processor performance with cached shared data.

of shared data references is likely to be smaller, most workloads will tend to fall closer to the upper curve than the lower.

The data in the graph indicates that caching shared data allows the processors to operate at reasonably high utilization for a wide range of program sharing characteristics. Recall from Figure 1.1 that without caching shared data, the per-processor utilization with nearest-neighbor sharing fell to 50% of uniprocessor utilization when the percentage of shared data references reached only 15%. If shared data is cached, Figure 1.2 shows that utilization does not fall this low until a relatively high 5% miss rate due to invalidations is reached (regardless of the fraction of shared references). Clearly, caching shared data is key to achieving good multiprocessor performance.

1.1 Cache Coherence Directories

As we mentioned, caching shared data requires a cache coherency mechanism to maintain the shared memory model. In small-scale multiprocessors where all of the processing nodes sit on a single bus, the coherency problem is usually solved with a *snoopy* (or *snooping)* cache protocol [23, 39, 30] in which all writes to shared data are broadcast on the bus. The caches in the system *snoop* on the bus, checking to see if they have a cached copy of the written data. If a cache does have a copy, it either updates its copy with the new value (in an update protocol) or invalidates its now-stale copy of the data (in an invalidation protocol). When a miss to a block of data is transmitted on the bus, the caches also snoop to see if they have a modified copy of the data that should be supplied to the requesting processor.

Though some have been proposed [55, 24], systems with large numbers of processors that rely on snoopy cache strategies to maintain cache consistency have severe interconnect design constraints due to their reliance on broadcast messages. From the standpoint of interconnect design, a more versatile class of protocols is *directory* protocols [27, 50, 12, 7, 3]. In these schemes a separate directory memory is maintained that identifies the caches containing each cache-line-sized block of memory. When data is written, the directory information is used to direct invalidation or update messages (depending on the type of protocol) to only those caches that must receive them. Since messages are directed at particular processing nodes, these protocols are well suited to general point-to-point interconnection networks, and are therefore an attractive option for large-scale multiprocessors.

An example of a straightforward directory scheme is an invalidation protocol described by Censier and Feautrier [12]. The directory information for each block of memory data resides with that data at main memory in some additional directory bits that are added to every memory block. Each of these directory entries contains (1) a dirty bit that is set if the block is modified in one of the caches, and (2) an n-bit vector of valid bits (where n is the number of caches in the system) identifying those caches with a copy of the data (see Figure 1.3). When a cache reads the block from memory, its valid bit in the bit vector is set. When a cache writes a shared block, it sends a message to main memory, which in turn uses the information stored in the corresponding directory entry to send invalidation messages to the other caches with copies of the data. If a cache miss results in a request for a block that is dirty in another cache, the directory information is used to retrieve the block from that cache.

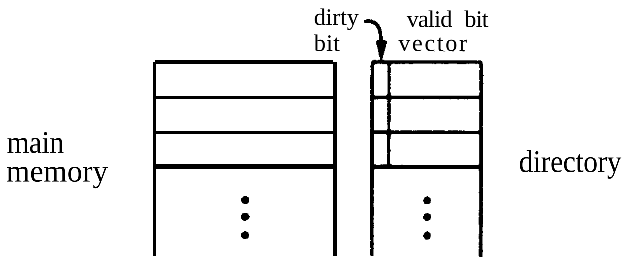

Figure 1.3: Conventional directory scheme.

1.2 Coherency Trade-offs

This thesis explores the trade-offs in the design of cache coherence directories. There are several steps involved in this design. At the highest level, the organization of the directory information must be chosen. In a large-scale machine, Censier and Feautrier's traditional organization with a valid bit per cache in each entry is not feasible due to the large amount of memory that the directory would occupy. Chapter 2 proposes and evaluates two alternate directory organizations that improve storage efficiency while maintaining high performance. Both schemes replace the costly valid bit vector with *pointers,* which are bit fields that encode the identity of a cache, but differ in how these pointers are assigned to main memory blocks.

After selecting a directory organization, the designer must then choose the features to be included in the coherency protocol. For any given feature, the performance advantages must be weighed against the additional complexity required to understand and implement the protocol. In Chapter 3 we measure the effects of several protocol options on reference latency and network traffic.

Once the directory organization and coherency protocol decisions have been made, the designer must consider implementation issues, which we address in Chapter 4. We begin by pursuing a basic design for a processing node, emphasizing the directory and other parts of the node relevant to the directory. We then use this design to demonstrate the functionality that must be supported by any implementation, such as preventing deadlock, maintaining a model of parallel execution, and handling subtle situations caused by temporary inconsistencies between cache state and directory state. In addition, we show how the protocol can be modified to tolerate messages between nodes arriving in

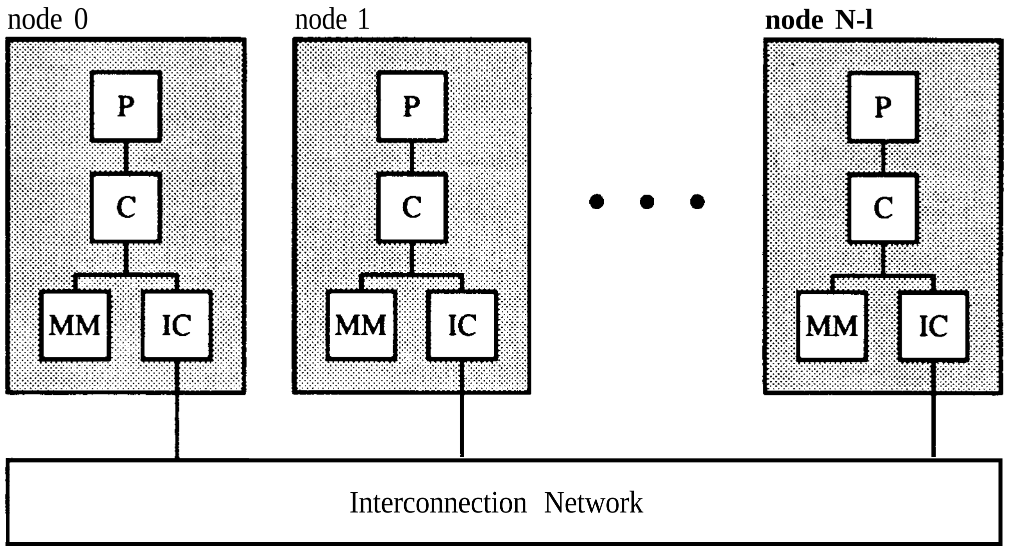

Figure 1.4: The basic system assumed by this thesis. Each node consists of a processor (P), cache (C), main memory (MM), and interconnect controller (IC).

a different order than they were sent, an additional functionality that may be needed in some designs.

1.3 System Assumptions

Throughout this thesis we make some assumptions about the machines we are considering. In general, the machine is organized as shown in Figure 1.4. The system is a shared memory multiprocessor that uses a large number N of high-performance CPUs. Each processor is paired with its own cache that may contain shared data.[2] The globally shared main memory is distributed across the nodes of the system, along with the corresponding directory entries.

Messages may be sent from any node to any other through an interconnection network. This network has only point-to-point capability, that is, there is no hardware mechanism

[2]While this thesis consistently refers to a processor's cache using the singular inflectional form for brevity, we are implicitly referring to all levels of caching on a node if a system with multi-level caching is under consideration. In this case we assume that a mechanism is implemented within a node to keep the multiple levels of caching consistent. Examples of such a mechanism include snooping on an intra-node bus and incorporating inclusion bits with cache tags to maintain the multi-level inclusion property [9].

to implement broadcasts efficiently. The interface at each node to the network is called the *interconnect controller,* which provides its node with reliable communication to and from the other nodes in the system. In general, we assume that messages sent from one node to another are delivered in the same order they were sent, though Section 4.5.4 demonstrates how this constraint may be relaxed

Chapter 2

Directory Organizations for Scalability

A primary step in designing a directory-based coherence mechanism is selecting a directory organization. The directory organization defines the storage structures that make up the directory and the nature of the information stored within. In this chapter, we introduce *limited pointers* and *dynamic pointer allocation* directories. These organizations potentially provide good performance while incurring modest memory overhead, even in large-scale machines. We examine the characteristics of both strategies, and compare them with different directory-based approaches that others have proposed.

2.1 Motivation

While directory-based schemes are attractive for maintaining cache consistency in large-scale machines (see Section 1. 1), naively organizing the directory data results in unacceptably high memory overhead. For instance, consider the Censier/Feautrier organization [12]. This traditional scheme maintains a vector of valid bits, one bit per processor, for each data block at main memory. Figure 2.1 shows the resulting memory overhead. The graph shows that this organization does not support more than about 100 CPUs; beyond this point, the memory overhead due to the directory storage becomes prohibitive. Furthermore, even if the memory overhead was reasonable in a large-scale machine, the large size of the valid bit vector would cause implementation difficulties. In particular,

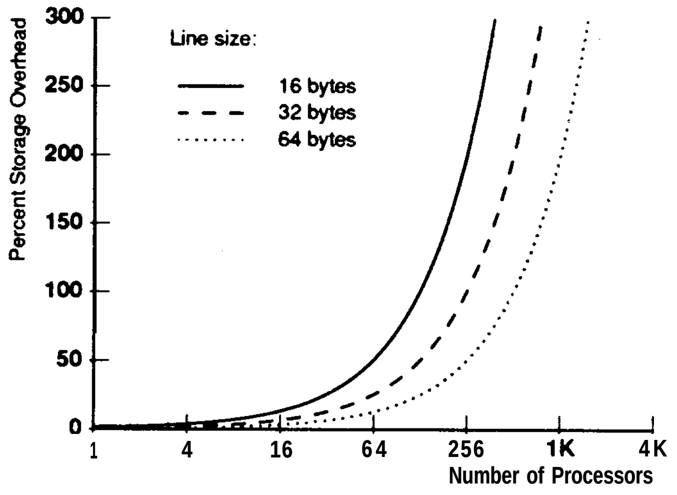

Figure 2.1: Storage overhead for the Censier/Feautrier directory organization.

the encoder/decoder unit used to translate between bit positions in the vector and the corresponding processor numbers would be difficult to build, and the wide directory memory is costly from an interconnection standpoint.

Since traditional directory organizations incur excessive memory overhead, other options must be explored. A general approach for reducing memory overhead is to develop schemes that either reduce the directory's length, that is, the number of directory entries, or the directory's width, that is, the number of bits per entry. While reducing the directory length can be worthwhile in some situations (Appendix E describes a length-reducing method), in this chapter we focus on reducing the width of the directory, since this is advantageous from an implementation standpoint in any case.

2.2 Limited Pointers Directories

The first directory organization we propose is the *limited pointers* scheme [3]. Simulations of small-scale multiprocessors have shown that usually no more than a few caches contain a given block of data when a write occurs [3, 19, 53]. Because the data is invalidated in all but one cache on a write, the valid bit vector in traditional directories rarely has more than a few of its bits set. It is therefore more efficient to replace the vector with *several pointers,* which are $\log_2 n$-bit fields that encode the unique identities of those

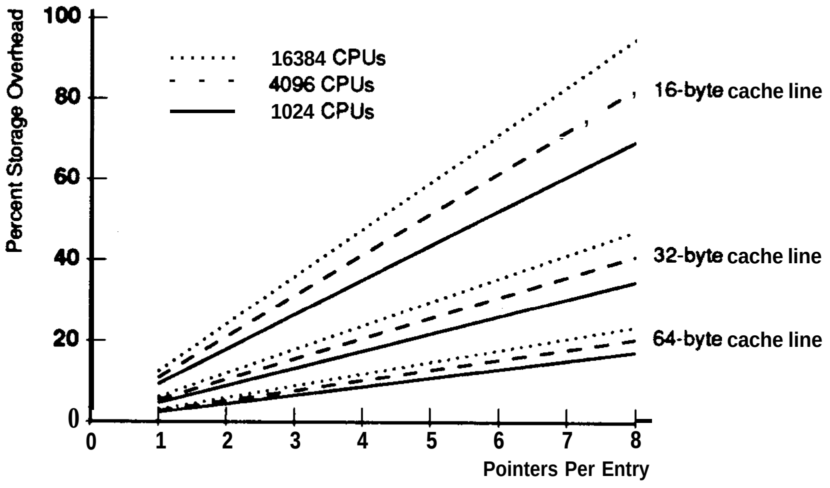

Figure 2.2: Storage overhead for limited pointers directories.

caches containing the data block. The resulting storage overhead is shown in Figure 2.2. This overhead is calculated by dividing the number of bits in a directory entry by the number of data bits represented by that entry, as follows:

$$\%\text{ overhead} = \frac{p\log_2 n + p + 1}{8b} \cdot 100 \qquad (2.1)$$

where n is the number of processors, b is the number of bytes in a cache block, and p is the number of pointers in a directory entry. The graph shows that we can provide each directory entry with at least three pointers for most large-scale systems, while keeping the memory overhead under 36%.

Of course, with only a small number of pointers in each entry, it is possible to run short in a given entry. This occurs when a request due to a read miss arrives at memory and the directory discovers no free pointers remaining in that entry to record the requesting cache. There are two basic strategies for handling this situation. In a *broadcast* scheme (denoted Dir; B [3][1]), a *broadcast bit* in the directory entry is set, indicating that the directory is no longer keeping track of the caches containing that data. When a processor eventually writes the data, the directory must resort to broadcasting an invalidation request to all caches. *In* a *no-broadcast* scheme (denoted Dir; NB), no more than i cached copies of a block of data may exist at any given time, where i is

'The subscript i refers to the number of pointers in each entry.

the number of pointers in each directory entry. When a free pointer is needed and none are available, a pointer is randomly selected and freed by invalidating the data from the cache the pointer identifies.

We have seen that limited pointers directories incur reasonable storage overhead, and we will see in Chapter 4 that its implementation is straightforward Up to this point, however, the performance potential of limited pointers directories in large-scale machines has been speculative at best. Simulation results [3, 53, 13] have come from small-scale workloads (4 to 64 processors), and the observation that only a few caches normally contain a datum has been only qualitatively explained. In the upcoming sections we introduce an analytic model that predicts these results from measurable workload characteristics. The model not only explains the behavior seen in the small-scale simulations, but also allows us to extrapolate forward to evaluate the efficiency of limited pointers directories under large-scale workloads.

2.2.1 Modeling the Number of Pointers Required

Our initial objective is to model the number of pointers required to maintain full caching information for a single block of data q. To do this, we assume the processors issue a sequence of references between writes to q as shown in Figure 2.3. Assume there are m processors that may access the block q. A processor is *selected* at random to make the next reference. If the processor has not previously accessed q in the reference sequence, it is known as a new processor; otherwise, it is an *old* processor. Regardless, the processor may access q, or may reference some other data block. Another pointer is required any time a new processor is selected that reads q. *The* sequence ends when any CPU writes the data block q, since it will be invalidated from all but one of the caches at that point.

For all $1 \leq i \leq m$ we can now calculate the probability that exactly **i** processors access q during a sequence, thereby occupying i pointers in the corresponding directory entry. While Appendix B contains the details of this calculation, we now excerpt the information from the appendix necessary to interpret the results in the upcoming sections. Given our model of reference behavior shown in Figure 2.3, the calculation requires the following parameters: m, the number of processors accessing the block q; r_n and r_o, the

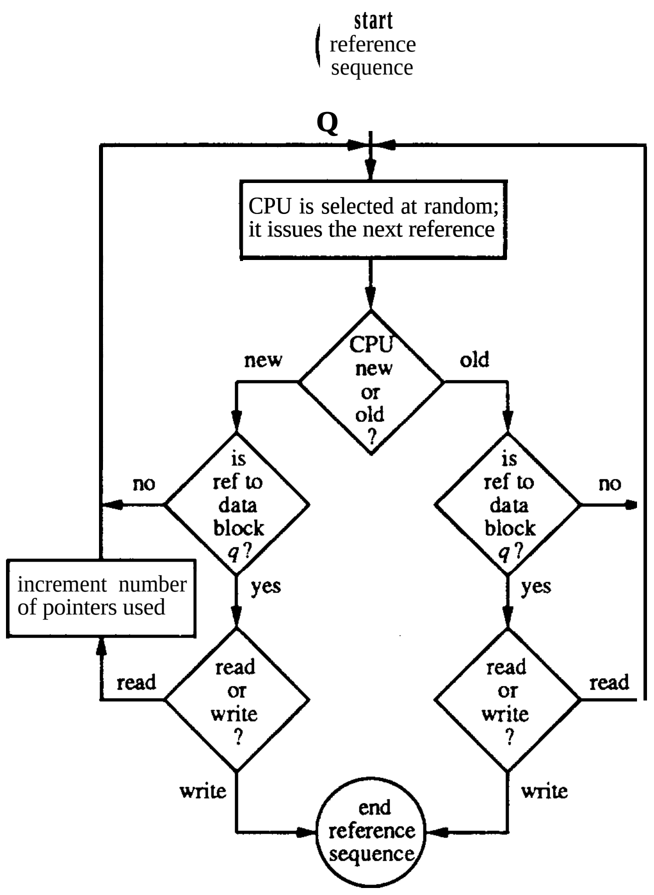

Figure 2.3: Sequence of references modeled.

read probabilities for new and old processors accessing q; and g_{n_i} and g_{o_i}, the probability with which new and old selected processors access q if **i** processors have already accessed it. The values of g_{n_i} and g_{o_i} can be derived by using a model of reference behavior exhibited by the m processors with regard to the data block. It would be simplest to assume that all processors, when selected, access q with equal probability. However, this assumption results in a poor approximation of the number of pointers used in actual multiprocess address traces.

A more realistic reference behavior assumes that one of the processors accesses the data more frequently than any of the other processors. We use a single ratio to describe this relationship: a, the probability with which *the* ***primary*** processor accesses q when selected divided by the probability with which each of the secondary processors accesses q when one is selected.

The model for the probability distribution of the number of pointers used is now complete. It takes four parameters as inputs, each with respect to the single block of data being modeled: the number of processors m accessing the data, the read probabilities r_n and r_o, and the ratio of the primary and secondary processor access probabilities a. In the upcoming sections we will verify this model against multiprocessor address traces and examine the model results using a variety of parameter values.

2.2.2 Verifying the Model

We verified our model against several multiprocessor address traces generated by scheduling multiple processes on a uniprocessor VAX in a round-robin fashion [22, 53]. These traces include references from 16 processors executing a single parallel application, and contain about 0.5 million references per CPU. No operating system references are included in the traces. The applications, all written in C, are P-Thor [47], a parallel logic simulator; LocusRoute [41], a global router for VLSI standard cells; Maxflow [11], an algorithm to find the maximum flow in a directed graph; and MP3D [36], a three-dimensional particle simulator for rarified flow. The applications allocate shared memory and synchronize parallel processes through calls to the Argonne National Laboratory macro package [35]. A detailed description of the algorithms used by each application appears elsewhere [53].

Our verification methodology is to compare the distribution of the number of pointers used as measured from an address trace with that predicted by the model. To measure the distribution from the trace, a cache event simulator steps through the references, assuming infinite-sized processor caches. When a write occurs, a histogram bin corresponding to the number of pointers used by that 16-byte block is incremented. Note that the more frequently a block is written, the greater overall impact of that block on the histogram results. We have omitted "uninteresting" data blocks, those for which the pointer count never exceeds one.

Keep in mind that our model is designed to accurately represent the references to a *single* block of data. Since the traces include references to many blocks, each with different characteristics, we must apply the model to each block individually in order to represent the program's reference behavior accurately. We ta the following steps to generate a number-of-pointers distribution from the model that is directly comparable with that measured from the traces. First, we run another cache event simulator on the address trace to extract the model input parameters for each of the blocks referenced (again, ignoring "uninteresting" data blocks and assuming infinite-sized caches). Second, we use the model with those input values to calculate the predicted distribution for each block. Finally, we take the weighted sum of the per-block distributions to generate a single aggregate distribution. Each per-block distribution is weighted by the fraction of writes to that block in the trace, i.e., the number of writes to that block divided by the total number of writes in the trace. In this way, the aggregate distribution for the model is weighted in the same manner as for the trace (see previous paragraph).

We extract the model parameters for a given data block as follows. The values for r_n and r_o are simply calculated by dividing the number of new and old reads (respectively) to the block by the total number of references to that block. To determine m and a, we first split the trace of references to that block into sequences of references that end in a write. In this way, we split the trace into the same units assumed by our model (recall Figure 2.3). We set m to be the maximum number of CPUs that reference a block during any of its sequences. Calculating a is more involved. To compute this value for a block, we first average across all sequences the fraction of references by the CPU accessing the block most frequently (within that sequence), second most frequently, etc. We then set a to be the ratio of the fraction of references by the most frequent CPU to the average of the fraction of references by the other CPUs.

Recall that we began with a model that assumes all processors, when selected, access the data block with the same probability (see Section 2.2.1). In this case, $g_{n_i} = g_{o_i}$, and the dependence of the number of pointers used on g_{n_i} and g_{o_i} is removed (see Appendix B). Figure 2.4 shows our attempt to verify this model against the traces using the methodology described above. Each graph represents a different application program. The solid lines show the distribution of the number of pointers used in the directory entry at the instant writes occurred, as measured from the address traces. The dashed lines show the same distribution as predicted by the model. The accuracy of this

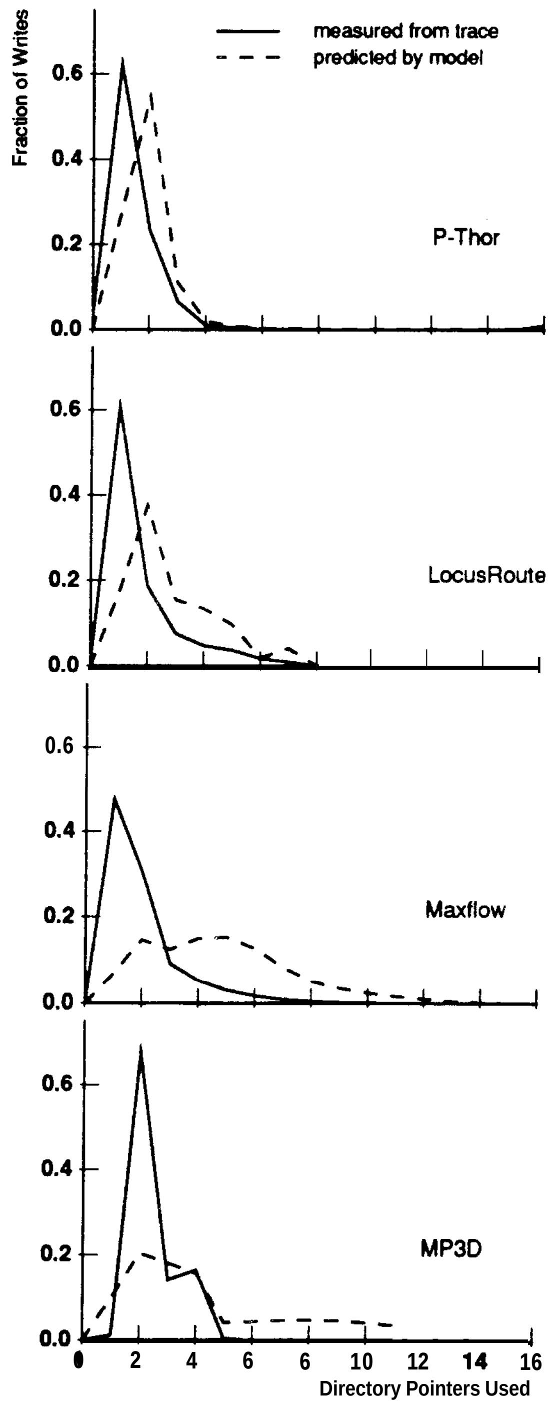

Figure 2.4: Model results assuming each CPU accesses data with equal probability.

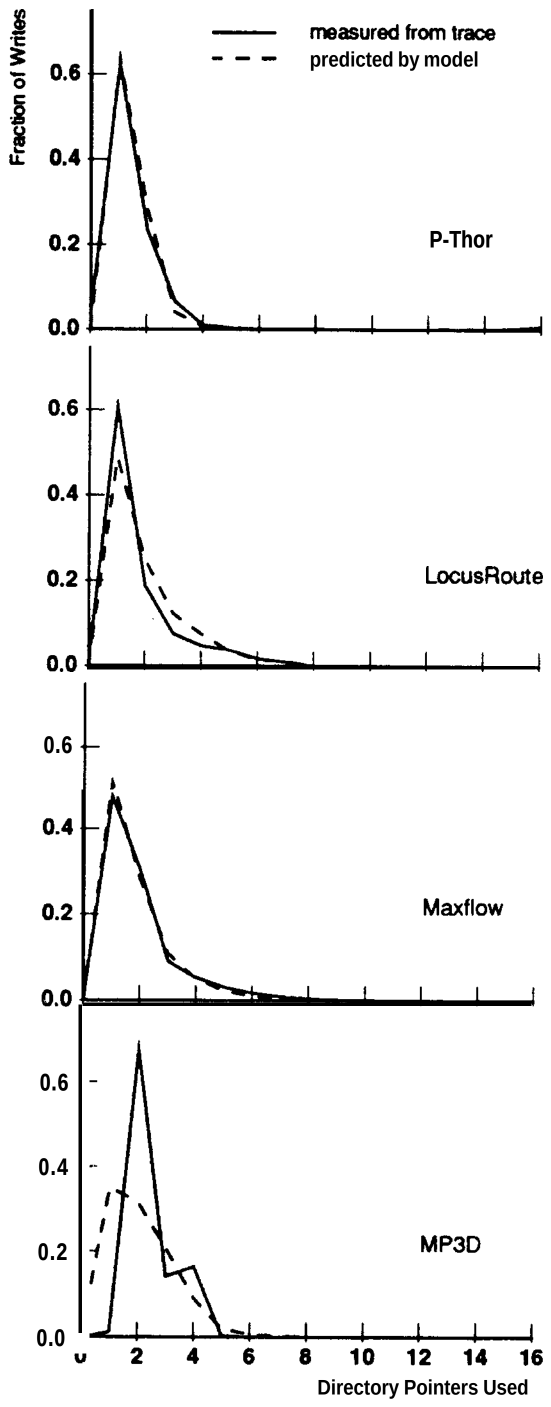

Figure 2.5: Model results assuming one primary and multiple secondary CPUs.

model is poor in several important respects. The "peak" of the distribution curves is not predicted correctly for any of the four benchmarks, and the number of pointers at which the curves "tail-off" to zero is severely over-estimated for Maxflow and MP3D. We are primarily concerned with this latter effect, since an important application of the model is predicting the number of pointers needed to avoid running short. At least the model is consistently pessimistic in this respect, but a model that yields a tighter upper bound would be preferable.

We therefore proceed to the model that incorporates the more complex notion of reference behavior in which a data block is accessed by a single primary processor with . greater probability than by any of several secondary processors. The results using this model are shown in Figure 2.5. The solid lines show the same distributions measured from the address traces as in Figure 2.4. The dashed lines show the distributions as predicted by the model. In general, the model does a good job of predicting the directory behavior of the applications. The only exception is for MP3D; an examination of the reference stream reveals a single, repetitive reference pattern throughout the trace as the cause of the model's failure. Since the model operates probabilistically, relying on average reference behavior, a single, repetitive sequence of reads and writes can easily yield results that do not match a distribution that takes into account all reference patterns producing the same model input parameters.

2.2.3 Model Results Under Large-Scale Workloads

In the previous section, we verified our model against simulation results showing the typical number of caches containing a datum at a write to be small. The model is useful as an explanation of those results, because they can now be related to familiar workload parameters such as read/write ratios. But more exciting is the prospect of using the model as a tool to predict the feasibility of limited pointers directories in shared-memory machines with hundreds or thousands of processors. In this section we first use our multiprocessor address traces to help select a range of input parameters to drive our model. We then examine the number of pointers used if an unlimited number are available, and evaluate the increase in network traffic and cache misses when pointers

Application	r_n	r_o	
P-Thor	0.84	0.75	138.;
LocusRoute	0.96	0.76	120.0
Maxflow	0.98	0.73	15.2
MP3D	0.99	0.64	11.3

Table 2.1: Average parameter values for each application.

are limited. Lastly, we assess two potential worst-case scenarios, increasing the number of processors accessing the data and increasing the read/write ratio.

Choosing Input Parameters

The results of our model, as with any model, depend on the values of the input parameters. Of course, different programs exhibit different referencing behaviors, and even within a single program, the properties of the data blocks may differ significantly. Accordingly, it is important to identify a reasonable *range* of parameter values that make sense, and evaluate the model across that range. Although we measure parameters directly from the application programs discussed in Section 2.2.2, we will evaluate the model across a much broader range of parameter values in order to extend the scope of the results to include a wide variety of parallel workloads.

Table 2.1 shows average model parameters for each of the applications, weighted across the blocks by the fraction of writes to each block. First consider the read ratio parameters: r_n, the probability that a new processor accessing the data issues a read, and r_o, the analogous probability for an old processor. Interestingly, the values for r_n are significantly higher than for r_o. Since we have no reason to believe these values will differ in programs for large-scale multiprocessors, we will model blocks using r_n values ranging from 0.9 to 1.0 and r_o values from 0.75 to 1.0. Table 2.1 also shows measured values for a, the ratio of frequencies with which the primary and secondary processors access the block. The a ratios in our small-scale benchmarks range from 1 (typically seen in blocks for which m=2) to over 300,000. We will use our model with a values spanning from 10 to 500, since most blocks fall into that range.

The only remaining model parameter is m, the number of processors actively accessing a given data block. It is likely that temporal locality and the nature of scalable parallel algorithms will serve to keep m relatively small (in the ones or tens) in a large-scale machine. We therefore primarily consider smaller values of m, and often use m=64 to be conservative. Since this hypothesis is still open to question, however, we also consider values of m that approach the size of large-scale machines, up to 4096 processors.

A caveat regarding the scope of our results is in order. In choosing model parameters, we consider only shared, writable blocks since, in general, read-only code and data remain consistent across caches without the need for coherency mechanisms such as directories. Read-only data may be marked as such (in the directory or TLB entries, for instance), thereby preventing coherency transactions. While this marking may have implications on the operating system and/or the programmer (or compiler), these considerations are beyond the scope of this thesis.

Number of Pointers Used

With reasonable values of the input parameters in hand, we can now look at some modeling results. We begin by examining the number of pointers used in a directory entry if an unlimited number of pointers are available. Figure 2.6 shows the distribution of the number of pointers used when a write occurs, assuming a relatively small number of processors accessing the data (m=16), and typical read/write ratios as indicated by our benchmarks (r_n=.9, r_o=.75). Each curve on the graph represents a different value of a, the ratio of the primary and secondary processor access probabilities. The most striking aspect of the distributions is the decrease in the number of pointers used as a increases. This dichotomy of reference frequency between the primary and secondary processors, along with the fact that many blocks exhibit very small values of m, explains the high peaks we see at a small number of pointers in the benchmark distributions.

Another important conclusion we can draw from Figure 2.6 is that there are plausible workloads for which it is not unusual to run short of pointers in a machine with two or three pointers per entry. While exhausting the available pointers is not frequent, neither is it very rare. This implies that designers must consider the performance of the directory in this situation, considering both the impact on system issues such as network traffic

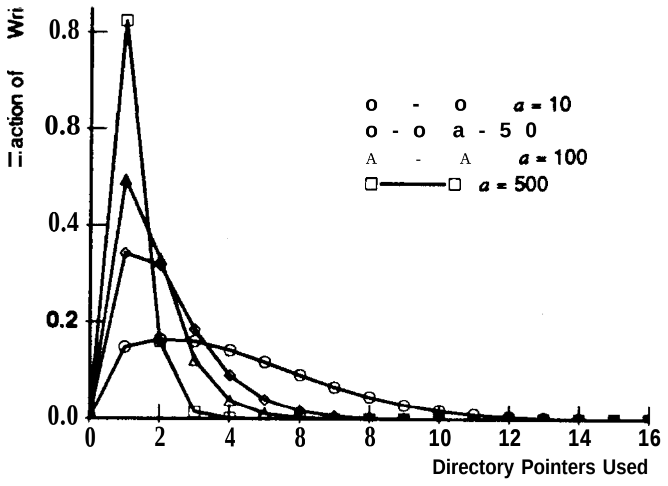

Figure 2.6: Number of pointers used with r_n=.9, r_o=.75, and m=16.

and cache miss rate as well as the latency through the directory itself.

The primary characteristics of the distribution curves for some other workloads are shown in Table 2.2.[2] Notice that for a given number of CPUs m accessing the data, the number of pointers used is relatively insensitive to the read parameters r_n and r_o. It takes a substantial increase in the fraction of reads (boosting r_n to 1.0 and r_o to .9) and a low value of a (10) before significantly more pointers are used. Furthermore, increasing the number of processors accessing the data m causes more pointers to be used, but the growth is much slower than linear. Even with m=4096, writes are frequent enough that significantly fewer than m CPUs access the data before it is written. And while the number of pointers used is still high, the next section shows that the impact on system performance of using fewer pointers is fairly small.

Impact of Fewer Pointers

The model results we have presented can be used to determine the number of pointers required to maintain full caching information most of the time. But we are also interested in the impact of using fewer pointers per entry, since it may be possible to achieve nearly the same performance at lower cost. To evaluate this impact, we can adjust our model to

[2]The distribution curves themselves are not shown due to space constraints. However, all of the graphs are similar to Figure 2.6 in shape.

		a:	10		50		100		500	
		percentile:	50	95	50	95	50	95	50	95
m	r_n	r_o								
16	.9	.75	4	9	2	5	2	4	1	2
16	1.0	.75	5	11	2	5	2	4	1	2
64	.9	.75	6	18	4	11	3	8	1	3
64	1.0	.75	14	28	6	15	3	10	2	4
64	.9	.9	6	22	5	16	4	12	2	5
64	1.0	.9	21	41	10	26	6	18	2	6
4096	.9	.75	7	29	7	29	7	28	7	24
4096	1.0	.75	148	304	128	275	101	238	31	98
			number of pointers used							

Table 2.2: Median (50th percentile) and 95th percentile of the distribution of the number of pointers used.

predict the number of “extra” invalidations that occur when the number of pointers per entry is limited. By “extra” invalidations we mean those that would not have occurred if the directory had a full valid bit vector per entry.

Of course, the number of extra invalidations depends on the mechanism used by the directory when it needs a free pointer but none remain. Let us first consider the broadcast strategy Dir; B with a broadcast bit in each directory entry. If this bit is set when a write occurs, then an invalidation must be sent to every cache in the system. In a directory with full caching information, invalidations would have been sent only to those caches containing the data. Under Dir; B, the extra invalidations are those sent to the other caches. These invalidations increase the amount of traffic that the interconnection network must bear. Notice that the number of extra invalidations depends not only on the number of processors actively accessing the data, but also on the number of caches in the entire system.

It is a straightforward matter to compute the number of extra invalidations under various workloads from the data presented in the previous section. By then calculating to first order the base number of misses and hits in a given reference sequence, we can

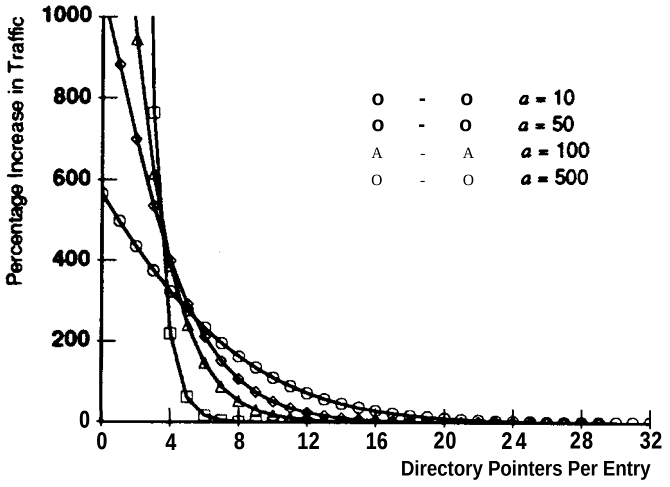

Figure 2.7: Extra invalidations for Dir; B with r_n=.9, r_o=.75, and m=64 in a 64-CPU system.

derive the percentage increase in network traffic due to the broadcast strategy.[3] This increase is shown in Figure 2.7 for a workload with 64 processors accessing the block. This data assumes the entire system is comprised of those 64 CPUs. We see that the increase in traffic can be very large; for instance, with three pointers per entry the traffic for this workload grows by at least a factor of 4 due to broadcasts. This poor behavior is not surprising, since all caches must receive an invalidation for each reference sequence in which the available pointers are exhausted. More distressing is the fact that the number of extra invalidations can be significant even if the directory entry runs short of pointers infrequently. For instance, with r_n=.9, r_o=.75, and a=100, Table 2.2 indicates that roughly eight pointers per entry are sufficient to handle most sequences of references. Yet Figure 2.7 shows that under these conditions the traffic is still increased by over 50%.

This negative characteristic of Dir_i B schemes is exacerbated by adding more processors to the system. Consider the workload used in Figure 2.7, but in a larger system, say with 4096 CPUs. The number of extra invalidations is now roughly two orders of magnitude higher than those seen in Figure 2.7. Continuing the example from the previous paragraph, a machine with eight pointers per entry must send about 200 extra invalidations each time the data is written if a=100. Even though the directory does not

[3]To simplify the analysis, we assume the base level of traffic consists only of cache misses and invalidations.

run out of pointers very often, it incurs a very high cost when it does run short, especially in a large-scale system such as this. The conclusion we can draw about Dir; B schemes is that running short of pointers has disastrous consequences on their performance. If they are to be successful, enough pointers must be provided to ensure that they are exhausted in only very exceptional cases. Since this is a difficult condition to satisfy, we believe broadcast limited pointers directories are not a viable option for large-scale systems.

We now turn our attention to nobroadcast (Dir; NB) directory schemes, in which no more than i caches may have copies of a block of data, where i is the number of pointers per directory entry. If i caches contain a datum and another cache then makes a request, the datum must be invalidated from one of the caches so the request can be satisfied. The advantage of this mechanism is that it is never necessary to broadcast an invalidation message to all caches in the system. The disadvantage is that read misses may cause an invalidation that would not have been necessary had full caching information been available in the directory. Not only does the invalidation boost interconnection network traffic, but the processor whose cache receives the invalidation may read the data again before it is next written. Had the invalidation not occurred that read would have probably hit in the cache; as it is, a miss will be incurred. Therefore, no-broadcast schemes have a negative impact on both network traffic and miss rate. We can use our model to evaluate the magnitude of that impact.

Under no-broadcast schemes, the extra invalidations are those sent to caches whose corresponding processors read the data again before the data is next written. That is, extra invalidations include only those that cause a cache miss that would not have occurred with a full valid bit vector per directory entry. Of course, other invalidations may be sent before the write occurs, but these are not extra since they would have been sent anyway when the data was eventually written. The number of extra invalidations may be calculated by extending the model (see Appendix B).

Figures 2.8 through 2.11 show the impact of using a no-broadcast limited pointers scheme, assuming 64 CPUs are actively accessing the data. The horizontal axes indicate the number of pointers per entry, while the vertical axes denote the percentage **increase** in the number of misses.[4] This miss rate increase is relative to the *inherent* miss rate that

[4]**Notice this is not the number of percentage points by which the miss rate is increased.**

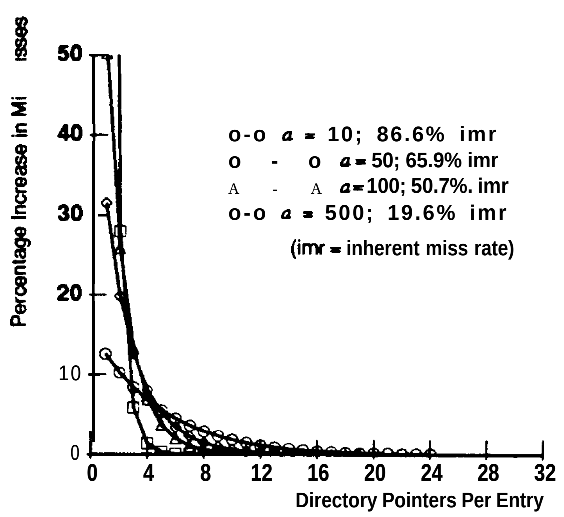

Figure 2.8: Percentage increase in cache misses for Dir$_i$ NB with r_n=.9, r_o=.75, and m=64.

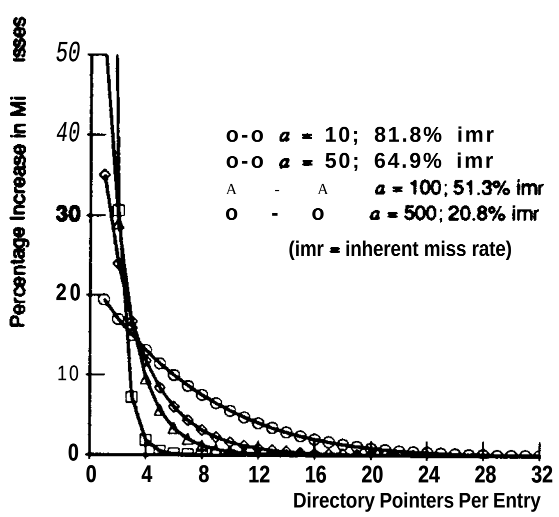

Figure 2.9: percentage increase in cache misses for Dir; NB with r_n=1.0, r_o=.75, and m=64.

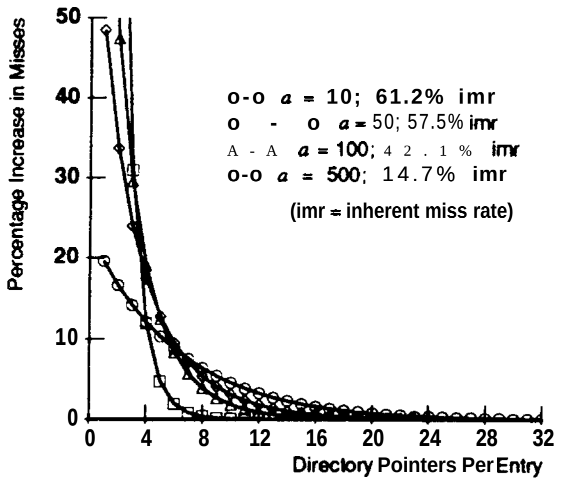

Figure 2.10: Percentage increase in cache misses for Dir$_i$ NB with r_n=.9, r_o=.9, and m=64.

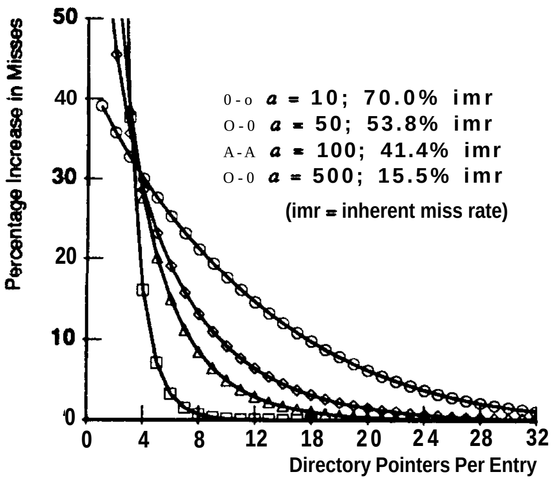

Figure 2.11: Percentage increase in cache misses for Dir; NB with r_n=1.0, r_o=.9, and m=64.

would occur if the number of pointers were unlimited; this value is indicated in the key of each graph. The inherent miss rates are interesting in themselves, since we see that invalidation-based consistency protocols result in miss rates for actively shared blocks that are generally high compared to uniprocessor miss rates.

Another interesting observation is that the impact on the number of misses is often much lower than we might conclude from the distribution of the number of pointers used if an infinite number are available. For instance, with m=64, r_n=r_o=.9, and a=50, Table 2.2 indicates that about sixteen pointers are sufficient to handle most reference sequences. Yet we see in Figure 2.10 that only six pointers per entry are required to limit the increase in the number of misses to 10% or so. This effect is explained by the only moderate probability that an invalidated cache loads the data again before the sequence of references ends with a write. There are many workloads with m=64 for which only three to five pointers per entry are sufficient to limit the increase in the number of misses to less than 15%.

As we mentioned, the no-broadcast limited pointers scheme has an impact on the network traffic as well. Not surprisingly, the expression for the increase in the number of network messages is very similar to that for the increase in the number of misses. In practice, the difference is negligible; therefore, the graphs of Figures 2.8 through 2.11 represent not only the increase in the number of cache misses, but also the increase in the amount of network traffic.

Increasing the Number of Processors

Since limited pointers schemes were introduced [3], one of the concerns has been that they lack scalability. While we believe the number of processors actively accessing a datum will remain low even in large-scale machines, our model makes it easy to predict the results if m does in fact grow large. Figure 2.12 shows the results for one set of parameters, assuming the directory entry has four pointers. The horizontal axis shows m, the number of processors actively accessing the block of data. As before, the vertical axis shows the percentage increase in the number of misses using the no-broadcast strategy. The impact on the miss rate peaks and then begins to fall as m is increased. This is because more processors accessing the data decreases the probability that a processor

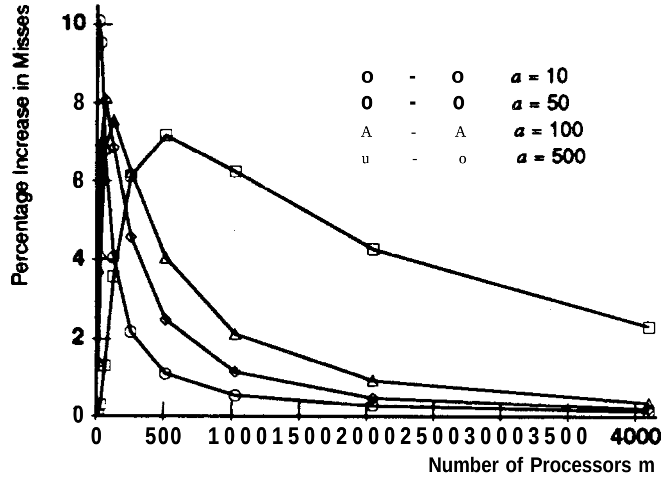

Figure 2.12: Percentage increase in cache misses with 4 pointers per entry, r_n=.9, and r_o=.75.

whose cached copy has been invalidated will read the data again before it is written. In other words, there is usually no negative impact of invalidating the block from one of the caches since it is unlikely the corresponding processor will read the data again before it is written. Another way to look at it is that data exhibiting high values of m is rarely read more than once by the same processor before the data is written; the inherent miss rate for such blocks is therefore extremely high, and limiting the number of pointers cannot make the number of misses much higher.

The bottom line is that blocks that are actively accessed by large numbers of CPUs at a given time will exhibit very high miss rates. While limited pointers directories obviously do nothing to solve this problem, neither do they make the problem much worse. If blocks like this do occur in large-scale programs, then the no-broadcast limited pointers strategy performs almost as well as a full bit vector of valid bits.

Increasing the Read/Write Ratio

It turns out that the relative performance of no-broadcast limited pointers schemes suffers more from a high fraction of reads than from a large number of processors. To clearly demonstrate the cause of this effect, Figure 2.13 shows absolute miss ratios rather than the increase in miss ratio (the metric used in previous graphs). We have set r_n=1.0; the

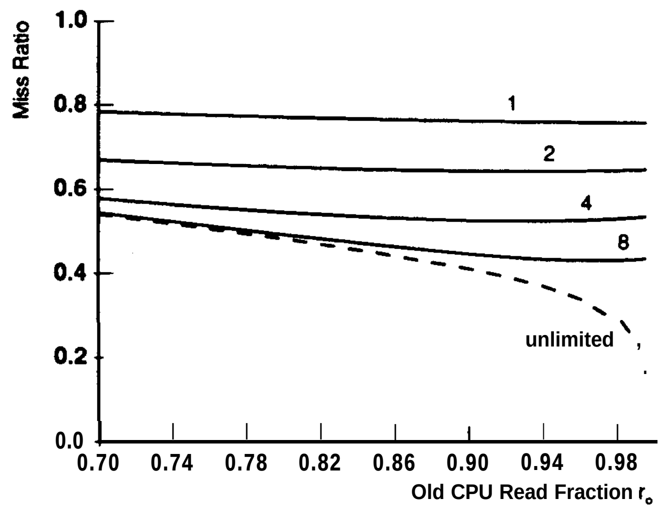

Figure 2.13: Cache miss ratio with r_n=1.0, a=100, and m=64. The curve labels indicate number of pointers per entry.

horizontal axis represents r_o, and the vertical axis shows the resulting miss ratios. Interestingly, the miss ratio for the limited pointers schemes does not increase as r_o approaches 1.0. This is because the additional misses caused by invalidations to clean blocks are offset by a reduction in invalidation misses due to writes. Hence, the performance of a limited pointers directory for a given block exhibiting a high read/write ratio is similar to an unlimited pointers directory for a block with a more moderate read/write ratio.

As the read/write ratio increases, the performance gap between limited and unlimited pointers widens due to a faster decrease in the miss ratio for the latter. Furthermore, while high read/write ratios are considered unusual in uniprocessor programs, they do occur with some frequency in some of our small-scale multiprocessor benchmarks. For instance, 12% of the references to shared data in LocusRoute are made to blocks exhibiting r_n=1.0 and r_o>0.95. On the other hand, practically none of the references in Maxflow and MP3D fall into this category. We can judge the overall effect of each program's behavior on the performance of limited pointers using trace-driven simulation. A limited pointers directory with three pointers per entry incurs 28%, 29%, and 22% more cache misses than a directory with unlimited pointers for LocusRoute, Maxflow, and MP3D, respectively. The cache miss overhead due to limited pointers is more substantial for P-Thor: 249% with three pointers per entry. Most of this overhead is due to only two blocks. In general, we find in our limited set of programs that some have a few variables, often synchronization variables, that are read widely but rarely written. This is similar to the findings of Chaiken et al [13]. Although it may be possible to specially handle those few blocks that cause performance degradation due to limited pointers, these blocks can be difficult to identify *a priori.* Another option is to develop a more robust directory organization. Towards this end we have developed the dynamic pointer allocation directory, which we describe next.

2.3 Dynamic Pointer Allocation Directories

The second directory organization we propose is *the dynamic pointer allocation* scheme [43], which takes advantage of the fact that most data blocks are shared at any given time by only a few caches, while a few blocks may be widely shared. Like the limited pointers directories described in Section 2.2, this directory scheme uses pointers that contain the unique identities of those caches containing the data. However, the number of pointers available in an entry is not fixed, but is rather allocated on-the-fly from a pool of available pointers. When a pointer is no longer needed, it is returned to the pool.

2.3.1 Directory Organization

The organization of the dynamic pointer allocation directory is shown in Figure 2.14. The primary structure is a memory containing a number of pointers, each paired with a *link*.[5] These pairs form the basic data structure that allows us to construct linked lists of cache pointers. In addition, each block of main memory has an associated dirty bit, a link to a list of pointers allocated to the block, and *an empty bit* that indicates whether or not the block's list of pointers is empty. When the list is not empty, the last pointer links back to the main memory block, forming a circular list. At system start-up, a single linked list consisting of all the pointers is built. A special register known as the free *list link* is set to contain the address of the first pointer/link pair; this register is a link to the head of the *free fist*. On a cache miss, a pointer is removed from the free list

[5] Although the links operate as pointers to other locations in the memory, we will be careful to always refer to them as "links," to avoid confusing them with the cache pointers.

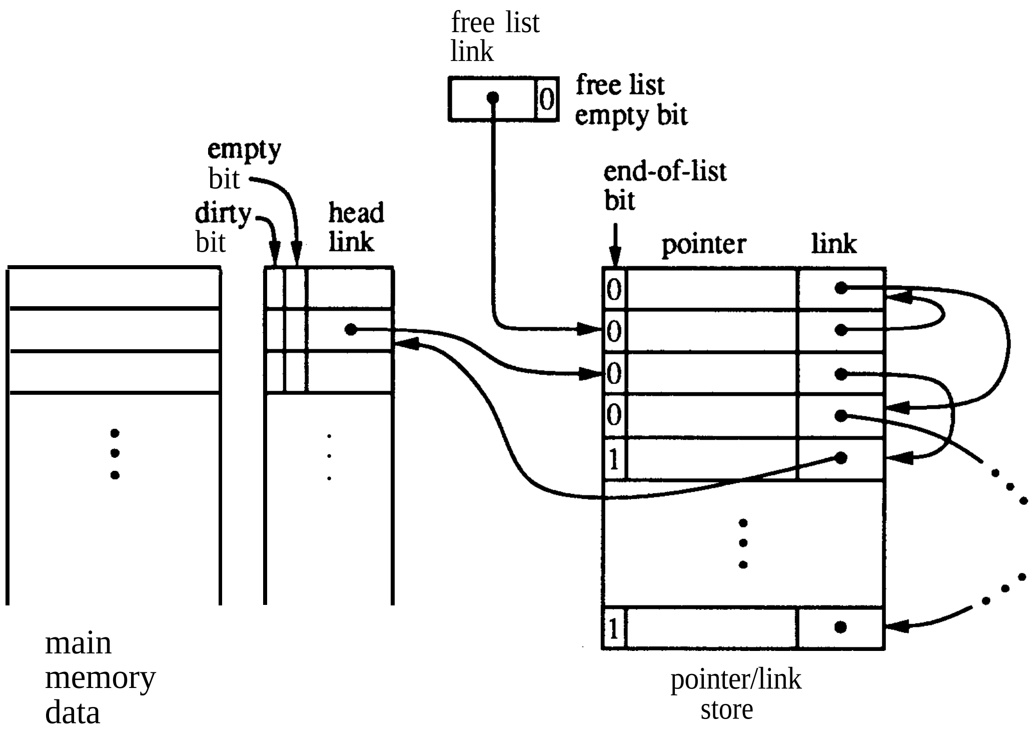

Figure 2.14: Basic dynamic pointer allocation scheme.

and added to the head of the list for the desired memory block. When permission to write a block is requested by a cache, the directory steps through the block's pointer list, sending invalidations to each cache on the list and returning the pointers to the head of the free list. The end of the list is reached when the *end-of-list bit* associated with each pointer/link pair is found to be set.

The dynamic pointer allocation directory adds less storage to each data block than a standard limited pointers directory with several pointers per entry. Table 2.3 shows the percentage memory overhead that must be added to main memory to store the dirty bits, empty bits, and head links. Since caches are growing larger with each new generation of systems, the length of the pointer/link store must grow as well. Therefore, the directory overhead will rise over time, due to increases in the width of the pointer list head link field. Nonetheless, Table 2.3 shows that even increasing the size of the pointer/link store by a factor of 16 results in only a modest overhead increase for a given cache line size. The dynamic pointer allocation scheme therefore results in good storage efficiency even

Line Size	*Number of pointer/link pairs*		
	32K	128K	512K
16 bytes	13.3%	14.8%	16.4%
32 bytes	6.6%	7.4%	8.2%

Table 2.3: Dynamic pointer allocation main memory storage overhead.

in the presence of large caches.

We expect the pointer/link store to scale gracefully over time as well. As we mentioned, the number of pointers required on a node will depend on the size of the caches. We will see that a large number of pointers can be easily built using the same sized parts from which caches are typically constructed. The number of pointers that can be provided will therefore scale at the same rate as the size of cache memories.

2.3.2 Directory Protocol

The protocol for the dynamic pointer allocation directory is very similar to that of a straightforward Dir_i NB scheme (see Chapter 4). For instance, as in a limited pointers directory, it is possible to run short of free pointers. This occurs if a pointer is to be allocated from the free list but the free list is found to be empty. The correct action to take in this situation is to use some means to select a pointer (or multiple pointers) and then free it by sending an invalidation to the cache identified by the pointer. It is probably reasonable to select the pointer on a pseudo-random basis using a free-cycling hardware register. Because the address of the block is needed to send an invalidation, the list beginning at the pointer indicated by the cycling register must be traversed until the last pointer on the list is found. At this point, because the list is circular, the link contains the address of the data block. This address and the pointer can be used to send an invalidation, thereby freeing the pointer/link pair.

An interesting effect occurs if caches are allowed to replace clean blocks without notifying the directory. *Stale* pointers indicating caches that no longer contain the data may accumulate in the pointer/link store. If they are not returned to the free list, these stale pointers could potentially occupy most of the pointers that would otherwise be free, perhaps causing the directory to run short of free pointers frequently. This is undesirable since processing a read miss (the most frequent directory operation) is considerably slower at the directory if no free pointers remain. Therefore, while not required for correctness, we recommend enhancing the protocol by having caches send *replacement notifications* to the appropriate directory when a clean block is replaced.[6]

Of course, replacement notifications consume network bandwidth, and removing stale pointers consumes directory bandwidth. We first estimate the increase in traffic due to replacement notifications. Consider a workload with an amount of sharing such that the network can bear no additional traffic without substantially reducing processor throughput. This is a worst-case workload for our purposes, since there is no excess network bandwidth available to carry replacement notifications. To estimate the increase in traffic, we must calculate both the base level of traffic and the additional traffic due to replacement notifications. The base level of traffic is primarily due to invalidation misses (i.e., misses due to invalidations) and *uniprocessor* misses [20] (i.e., misses due to cache interference). Though the rate of invalidation misses for our worst-case workload will vary across machines, aggressive implementations will likely support an invalidation miss at least every 50 data references. We assign each of these misses a cost of 4 messages (an invalidation/acknowledge pair and a subsequent miss request/reply). We assume the uniprocessor miss rate for data references is below 2%, which is typical for direct-mapped caches of at least 64KB [40]. Each of these misses costs 2 messages (a request and reply). The additional replacement notification traffic is incurred on those uniprocessor misses that cause clean data to be replaced. Smith [46] finds that roughly half of the data misses in his uniprocessor address traces cause a clean line to be replaced. Each of these misses costs 1 message to send the replacement notification. To compute the maximum percentage increase in traffic due to replacement notifications, we weight the invalidation and uniprocessor miss rates by the corresponding cost per miss:

$$incr_{\mathrm{max}} = \frac{\left(0.02\,\frac{\text{uni. misses}}{\text{reference}}\right)\left(0.5\,\frac{\text{notifications}}{\text{uni. miss}}\right)(1\ \text{msg.})}{\left(\frac{1}{50}\,\frac{\text{inv. misses}}{\text{reference}}\right)(4\ \text{msgs.}) + \left(0.02\,\frac{\text{uni. misses}}{\text{reference}}\right)(2\ \text{msgs.})} \cdot 100 = 8.3\%$$

[6]Separate notifications are not required when dirty blocks are replaced, since they must be written back to memory anyway.

This means that a network that could otherwise support an invalidation miss every 50 data references must be improved to handle a modest 8.3% more traffic if it is to deliver identical performance once replacement notifications are added to the protocol. Furthermore, our calculation of the traffic increase is conservative for several reasons. First, we have not counted invalidation messages that do not result in an invalidation miss, nor have we counted messages required to retrieve a dirty block from a cache in order to service a miss. Second, we assumed all replacement notifications require a separate message, when in fact some may be "piggybacked" on the cache miss request replacing the clean block. Third, we have not considered the effect that replacement notifications sometimes actually reduce traffic: without notifications, the directory will send needless invalidations to caches that have already replaced the block. These invalidations cost 2 messages (the acknowledgement and the reply), while the replacement notification that obviates them costs only one message. Finally, we have not accounted for the differences in message size; since replacement notifications contain no data, they consume less bandwidth than other messages such as cache miss replies.

We can also estimate the effect of consuming additional directory bandwidth to remove stale pointers. Let us assume the cache miss rate is 2%. We also assume 30% of the instructions issue a data reference, and the processors are 70% utilized. Finally, assume it takes 20 processor cycles on average at the directory to remove a stale pointer when a replacement notification arrives. We can now calculate the fraction of the directory bandwidth consumed by processing replacement notifications as follows:

0.7	instructions per cycle
x 0.3	data references per instruction
x 0.02	data reference misses per data reference
x 0.5	replacement notifications per data reference miss
x 20	cycles per replacement notification
0.042	

While 4.2% is not insignificant, neither is it enough additional directory utilization to dramatically degrade machine performance. For this reason, and because the network traffic is not substantially increased, we believe replacement notifications to free stale pointers are a useful addition to the dynamic pointer allocation protocol.

One final concern about the dynamic pointer allocation scheme is the speed of the directory operations. After all, unlike limited pointers organizations, the pointers cannot be accessed concurrently; instead, the linked list must be serially traversed to reach all the pointers. By examining each of the commonly occurring cases at the directory, we see that there is no inherent slowdown due to list traversal. Read misses (the most frequent directory operation) require accessing only the first pointer on the list. Write misses and write hits to clean blocks require traversing the list to send invalidation messages, but these messages would have to be queued for delivery serially anyway, so no time is lost. Write-backs imply that the block is dirty, i.e., the list contains only one pointer. Replacement notifications require the list to be searched to reclaim the pointer, but unlike many operations, the processor that sent the message is not stalled, so memory latency is not directly affected. Furthermore, most lists are short, resulting in low search times. We therefore conclude that there is no inherent speed limitation in the dynamic pointer allocation protocol caused by accessing multiple pointer/link pairs in a directory operation.

2.3.3 Implementation Issues

Perhaps the simplest approach for implementing the dynamic pointer allocation scheme is shown in Figure 2.15. The resources associated with the directory are placed on a single bus; a simple state machine (not shown) synchronously arranges the bus accesses required to perform a desired directory operation. The resources are as follows: the memory storing the head links associated with each data block, the store of pointer/link pairs and its address register, the register containing the free list link, and two temporary registers. These temporary registers are needed to traverse a list while sending invalidations and to move a single pointer/link pair from the middle of a list to the free list. One of them is also sometimes used to address the head links storage if the directory runs short of pointers. The detailed steps that must be performed for most common directory operations and their associated delays are given in Appendix D. We find that directory operations that access a single pointer may be performed quite rapidly. For instance, consider a read miss in a cache. The time required to add the cache to the block’s list of pointers

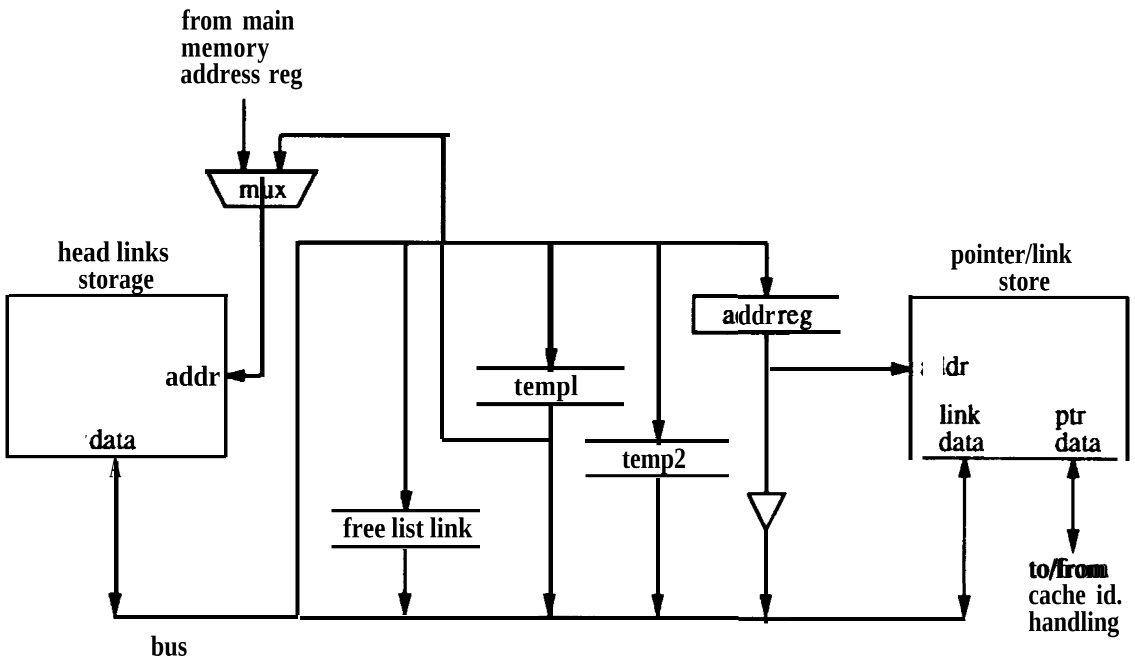

Figure 2.15: Dynamic pointer allocation implementation.

is dominated by memory access time: a read-modify-write on the head links storage and a write to the pointer/link store. If the pointer/link store is built with static RAM, then the entire directory operation takes only slightly longer than the dynamic RAM access required to supply the cache line data. If this data DRAM is not as wide as a cache line (i.e., multiple accesses are required to supply the cache line data), then the directory operation will likely be as fast as the data access. More complex directory operations involving multiple pointers take longer to complete, but as we explained in Section 2.3.2, this delay would be incurred in any case to queue outgoing messages.

There are various options for implementing the directory resources and bus shown in Figure 2.15 and the associated controller. Obviously, a board-level implementation is possible, using discrete integrated circuits or a register file part for the registers and a programmable logic array to build the controller. More compact options include conventional gate array or programmable gate array parts. A single part could easily accommodate the registers, the portion of the bus running between them, and the controller to generate the bus read/write signals for each resource. The I/O of the part would consist of the internal bus and the address bits for the pointer/link memory, as well as control signals for interfacing the directory with the rest of the processing node. The storage for the head links would of course be implemented using the same technology as main memory (dynamic RAM in most designs), while the pointer/link store would probably be built using conventional static RAM for speed

Let us consider the pointer/link store in the context of the amount of memory and the parts count requited to implement a given number of pointers. Although our goal is to provide cache coherency in large-scale machines of the future, we will consider typical cache sixes and RAM parts of today. As previously mentioned, we expect the ratio of . the number of pointers to the cache size achievable with a certain number of parts to remain constant as circuit density increases, since caches and pointers are both built from the same storage technology. We will use the common 32K-by-8-bit static RAM chip as our basic building block, and then calculate the ratio of the number of pointers to the number of blocks in the cache. We will assume we want to use the same number of building block parts in the pointer/link store as in the cache. Since the memory for a cache typically occupies little board space, this is not an unreasonable &sign option.

The width of each link field depends on the number of addressable memory blocks on a node, since the field must contain the return links at the end of the lists. Assuming each node has 64MB of main memory and the cache line size is 16 bytes, the link field must be 22 bits wide. In addition, let us assume a very large-scale machine, say with 4096 processors. Each pointer field is therefore 12 bits. This results in a total pointer/link width of 34 bits, so the minimum number of eight-bit-wide parts we can use to implement the pointer/link store is five parts. For this cost we get 32,768 pointers. The same number of parts yields a 128KB cache, moderately large by today's standards. Since each cache is therefore comprised of 8 192 lines, the ratio of pointers to cache lines is 4: 1. Of course, with perfectly distributed node addresses for the data cached in the system, a 1:1 ratio would be sufficient. So using the same number of parts to implement the pointer/link store and the cache yields an "uneven distribution margin ratio" of 4. We should also note that the 4: 1 ratio becomes an 8: 1 ratio in a machine with a 32-byte line size.

Our results indicate that a 4:1 ratio of pointers to cache blocks results in little or no degradation of overall system performance. To demonstrate this, we examine two extreme cases. If a large number of processors are accessing data from a given memory module, then each processor must be generating very little traffic to the module; otherwise, the module will be overloaded and the effect of the directory is unimportant. With each CPU sending so few requests to the module, the directory cannot have much impact on the performance of the processors. Now consider the other extreme, a small number of processors accessing data from the module. Here the ratio of pointers to cache blocks can be significant: obviously, if this number of CPUs is less than or equal to this ratio, then it is impossible to run short of pointers. Even if there are a few more processors, the directory is still unlikely to run short of pointers, since data from that module would have to occupy a substantial fraction of each processor's cache. Furthermore, programs will become less likely to exhibit this type of behavior as more sophisticated memory allocation techniques become commonplace. However, if the directory does run short of pointers, we can compute the expected worst-case increase in the number of cache misses (see Appendix C for complete details). Our results confirm that running short of pointers in a module has minimal impact if a large number of processors are accessing data from that module. If only a small number of processors are accessing the data, and processors emit references to their caches at five times the rate the directory can handle requests, then we find a 4:1 ratio of pointers to cache blocks causes no more than about 3 percentage points to be added to the cache miss rate when the directory runs short of pointers. An 8:1 ratio reduces this miss rate increase to 1.7 percentage points.

2.4 Trade-offs Between Directory Schemes

We have seen that dynamic pointer allocation is more robust than the basic no-broadcast limited pointers directory, in that it can handle blocks that are shared by many processors without performance degradation. Others have proposed alternative directory mechanisms with the same goal in mind. In this section we describe some of these schemes and compare them with the basic limited pointers and dynamic pointer allocation directories.

There are several variations on the basic limited pointers strategy. Gupta et al. have suggested a *coarse vector* technique that combines a broadcast strategy with a bit vector used to limit the scope of the broadcast [26]. When a directory entry runs short of pointers, the bits of the entry, which were formerly organized as pointers, are reorganized as a bit vector. In this mode, each bit is turned on if at least one of the corresponding n/b caches contains the data, where n is the number of caches in the system and b is the number of bits in each directory entry.[7]

Another proposed limited pointers scheme is the LimitLESS directory under development at MIT [14]. When a free pointer is needed in an entry and none remain, the local processor takes an interrupt. The trap handling software then performs the appropriate actions. The advantage of this scheme is that the software control provides great flexibility in handling infrequent situations while eliminating the need for extra hardware to take care of them.

We can qualitatively compare these schemes based on the results from Section 2.2.3. The data blocks fall roughly into three categories. First, there are blocks for which the available pointers are rarely exhausted. All of the limited pointers schemes (with the possible exception of broadcast schemes) perform well for these blocks. Second, there are blocks with only moderately high read/write ratios that use all of their pointers with some frequency. Finally, there are blocks with very high read/write ratios. We saw in Section 2.2.3 that standard no-broadcast schemes have only a minor negative impact for blocks with moderately high read/write ratios, but do not exploit the opportunity to significantly reduce miss rates for blocks with high read/write ratios. On the other hand, the coarse vector and LimitLESS schemes will perform well for blocks with high read/write ratios since writes are infrequent and miss rates are low. However, their performance suffers for blocks with moderately high read/write ratios. For these blocks, the coarse vector strategy will cause substantial traffic due to extra invalidations unless the number of caches represented by each bit (n/b) can be kept small. The LimitLESS scheme incurs considerable overhead for these blocks since misses occur relatively frequently and a significant fraction of them will require software traps.

As we see, none of these schemes are ideal for all types of data. Since in each case the directory has limited resources, there will always be a sequence of references that causes the directory to perform poorly. The goal is to design a directory in such a way

[7]In [26], the scheme is presented in the context of the DASH architecture. Using this architecture, each bit corresponds to $n/(4b)$ caches, since 4-cache clusters are kept self-consistent using a standard snoopy cache mechanism.

that this sequence lies outside the range of referencing behavior patterns exhibited by most programs. All of the limited pointers schemes incur some performance penalty on any block for which the pointers are exhausted. By sharing all of the pointers available to a memory module across all of the blocks in that module, dynamic pointer allocation drastically reduces the probability of running short of pointers. This feature ultimately makes dynamic pointer allocation more robust to different types of reference behavior, allowing it to perform equally well for each of the three categories of data blocks described above.

All of the schemes we have discussed maintain all information pertaining to a given data block in a directory at main memory. Another option is to decentralize this information. One such organization maintains a distributed linked list of pointers indicating the caches with copies of a given memory block [28, 51, 54]. The pointers on the list are stored with the data at the caches, while a field indicating the first node on the list is kept in a standard directory structure at main memory. The advantage of this organization is that sufficient pointers are always available for all cached data in the system. However, because the directory is decentralized, operations on lists of pointers are not atomic, resulting in substantial additional protocol complexity. Furthermore, these operations require more network messages and incur greater latency than their centralized directory counterparts. In short, decentralized directories avoid the infrequent situations that may not be handled well by centralized schemes, but in doing so incur traffic and/or latency penalties on most directory operations.

2.5 Summary and Conclusion

We have introduced the *limited pointers* directory as a means of achieving acceptable storage overhead for cache coherence directories in large-scale multiprocessors. This directory associates each data block with several *pointers* that can identify a small number of caches containing the data. By using an analytic model verified by trace-driven simulation, we have demonstrated that the performance degradation relative to a full-size directory is small across a wide range of workload characteristics, including a large number of processors actively accessing the data. The exception is blocks for which the read/write ratio is very high. As the read/write ratio increases, the performance of a full valid bit vector improves greatly while the performance of limited pointers remains roughly constant. Therefore, limited pointers directories do not take advantage of blocks with high read/write ratios.

While limited pointers directories may be enhanced through hybrid mechanisms that handle high read/write ratio situations in a special manner, a more robust solution is the *dynamic pointer allocation* directory. Rather than supplying a fixed number of pointers for every block in the system, pointers are dynamically allocated to blocks as they are needed. This scheme places no upper limit on the number of pointers that can be used by a given block. Since the pool of available pointers is shared by all blocks on a memory module, the probability of running short of pointers is greatly reduced compared to the limited pointers strategies. Furthermore, the directory can be made to process most individual operations as fast as the limited pointers directory. The dynamic pointer allocation strategy is therefore an attractive means of organizing the information in cache coherence directories.

Chapter 3

Coherency Protocol Design Options

In the previous chapter, we removed an obstacle to building large-scale, cache-coherent machines by introducing several directory organizations that provide storage efficiency without sacrificing performance. Another primary ingredient in the recipe for these machines is the cache coherence *protocol.* This protocol specifies the actions taken by the caches and directories in response to incoming requests.

Designing a coherence protocol amounts to a classic trade-off between performance and implementation complexity. Given the cost and performance goals for the machine, the decision to include or omit a feature in the protocol is straightforward if the "pros and cons," that is, the additional performance and complexity, are known quantities.

In this chapter, we take a simple coherency protocol and evaluate the performance improvement gained by adding each of three features. The first is the addition of a *clean/exclusive* state in both the caches and the directory. This allows many write hits to clean blocks to be satisfied immediately by the local cache, assuming the block resides in no other caches. The second feature we study is treating write hits to clean blocks as if they were write misses, which simplifies the protocol at the expense of higher memory-to-cache bandwidth utilization. The final enhancement we evaluate is *forwarding dirty* blocks directly from one cache to another cache whose processor has accessed the data. This optimization saves an extra message through the directory.

There is a substantial body of previous work evaluating cache coherency protocols. We will review these results in more detail in later sections when we consider each protocol feature. In general, the most important difference between our work and previous research is that we use cost metrics such as message counts that are appropriate for directory-based coherency in a large-scale system. Many previous protocol studies assumed small-scale, single-bus architectures. Because these protocols often take advantage of the inexpensive broadcast mechanism inherent in the bus (e.g., snooping cache protocols), the degree of performance improvement gained by protocol enhancements will be different than the equivalent enhancement in large-scale architectures.

The next section describes the base coherency protocol to which the enhancements are added. Section 3.2 then presents the simulation methodology we have used, followed by three sections detailing each of the proposed protocol features and their accompanying simulation results.

3.1 Base Protocol

We use the Censier/Feautrier protocol as a base [12]. This protocol makes the assumption that the memory module state (i.e., the directory information) includes the location of each cached copy of each memory block, as well as an indication of whether each block is dirty (see Section 1.1). Briefly, the protocol operates by executing one of the following sequences for each reference, depending on the type of reference and cache state:

- *Write hit to a dirty block* or *read hit.* These references are satisfied immediately by the processor cache, requiring no further action in the memory system.

- *Read miss.* In this situation, the cache sends a read miss request to the appropriate memory module. If the block is not dirty in a cache, the directory sends the data in a reply message. If the block is dirty, the directory sends a request to that cache to return the data and mark it clean. When the data arrives, the directory sends it to the cache that made the original request.

- *Write hit to a clean block.* In this case, the cache sends a message to the appropriate memory module to get permission to proceed with the write. The directory sends invalidation requests to the other caches containing the block. Each of these caches invalidates its copy and returns an acknowledgement reply to the memory module. If the programming model is to be sequentially consistent [31], then the directory must not return write permission until the directory has received all of these acknowledgements. If a weaker model of execution is allowed (e.g., weak ordering [42] or release consistency [21]), then the directory can return write permission immediately, before sending invalidations. Later, the directory sends another reply indicating that all invalidations have completed.

- *Write miss.* In this situation, the cache sends a write miss request to the appropriate memory module. If the block is not dirty in any cache, then the directory proceeds as in the case of a write hit to a clean block, except that the data block is returned along with write permission. If the block is dirty in a cache, then the protocol proceeds as in the case of a read miss to a dirty block, except that the cache supplying the block must invalidate it locally rather than mark it clean.

In addition to the actions described above, cache misses may result in block replacements. If the replaced block is dirty, then it is written back to main memory in a separate message.

We chose the Censier/Feautrier protocol as the base protocol for several reasons. First, it is well-known. Second, others have frequently used it as a base protocol [56, 18, 5, 38, 37]. Third, it is a simple protocol, yet not naive. It assumes only basic directory information (i.e., the caches containing the data and whether the data is dirty) is available, and does not include any enhancements that are obviously optional. The Censier/Feautrier protocol is therefore well-suited for our purpose of evaluating protocol enhancements.

3.2 Evaluation Methodology

We evaluate each of the protocol features using trace-driven simulation. The address traces are the 16-processor traces described in Section 2.2.2. The simulator maintains the state of the caches and the directory, modifying the state appropriately as each reference in the input trace is processed. All coherency actions related to a given reference are performed before the simulator proceeds to the next reference. Since we are interested in a class of machines rather than a particular machine, we avoid system-dependent performance metrics such as bandwidth and cycle counts. Instead, we count the number of messages that must be sent between caches and main memory. To measure memory latency, the simulator records the number of network messages required to satisfy each reference. To measure network traffic, the simulator counts the total number of messages sent.

To properly interpret our results, it is important to bear in mind the relationship between memory latency and overall performance. A given percentage reduction in latency does not translate to an equivalent reduction in execution time, since not all processor cycles are spent waiting on main memory to respond. For example, if we assume a processor is 50% utilized, then lowering memory latency by 10% results in only a 5% reduction in execution time.[1]

Of course, the simulation results are affected by the model of multiprocessor execution assumed by the simulation model. Each of the simulations is run once under assumptions of sequential consistency, and again assuming weak ordering. In our simulator, the only difference between these two models is whether or not the latency of invalidations is hidden. Under weak ordering, on a write to a clean block, we assume the memory module returns write permission before the invalidations are sent, thereby hiding the entire latency of those invalidations. With sequential consistency, the write permission is returned after the invalidations have been sent and acknowledgement has been received. In this case we make the simplifying assumption that multiple invalidations are sent (and acknowledgements are collected) in parallel; therefore, a latency of one invalidation/acknowledgement round trip (two network messages) is incurred. While this does not take into account the serialization of messages through the network port, we believe little error is introduced since the serialization time is probably small compared to the message transit time. In addition, the majority of these references in our applications cause no more than one invalidation anyway.

Note also that the assumptions made by our simulator with respect to weak ordering cannot accurately reflect the full set of weakly-ordered machines that could be built.

[1]This makes this simplifying assumption that all processor idle time is due to memory latency.

Our simulator assumes that only invalidation latency is hidden; however, weak ordering allows many other latency-hiding techniques to be used. For instance, write buffers help hide the latency of writes. In the upcoming sections, we will comment on how the results for weakly-ordered systems may be affected by latency-hiding schemes we have not simulated.

3.3 Clean/Exclusive State

The first protocol enhancement we evaluate is adding a *clean/exclusive* state to each cache and directory block. On a read miss, if the requested block resides in no caches, then the cache line is left in the clean/exclusive state after the miss is satisfied. In this way, if the block is eventually written, the write can proceed immediately in the cache without first obtaining write permission from the directory. While this potentially reduces both latency and traffic, there is a source of increased latency and traffic as well. If a miss now occurs in a different cache on the same block, then the directory shows the block clean/exclusive in the original cache, and must query that cache since the block may actually be dirty. Our goal in this section is to determine the degree by which the benefits of the clean/exclusive state outweigh the drawbacks, if indeed they do.

3.3.1 Background

The clean/exclusive state for directory-based cache coherency was first proposed by Yen and Fu [56]. Their paper briefly develops a simple analytic performance model, concluding that the clean/exclusive state "typically" reduces the average memory access latency by 5% to 15%, compared to the Censier/Feautrier protocol. However, it is not stated what range of model input parameters are considered "typical." Furthermore, the analysis is flawed: it ignores the increased miss latency described above due to having to check cached clean/exclusive blocks before supplying them to other caches.

In his thesis, Archibald points out these problems and performs a simulation study of the clean/exclusive state based on a synthetic workload [7]. Unfortunately, only small levels of sharing are considered, with no more than 5% of the data references to shared blocks (our address traces show 10% to 42%). The results are significant nonetheless. With almost no sharing, Archibald finds the clean/exclusive protocol outperforms the Censier/Feautrier protocol, but never by more than about 5%. In addition, this advantage decreases as the degree of sharing rises, since shared data may incur the cost of checking cached clean/exclusive blocks that other caches need to load. For these synthetic workloads, it would appear that the clean/exclusive state is not worth the cost. However, Archibald does not evaluate an aggressive variant of the clean/exclusive protocol that can be used in a system with a weaker coherency model. We will describe and evaluate this variant in Sections 3.3.2 and 3.3.3.

The Illinois protocol was the first bus-based, snooping cache protocol to incorporate the clean/exclusive state [39]. While that paper and others (e.g., [6]) present performance results from analytic models or synthetic workloads, the results are not relevant for large-scale machines. This is because the bus snooping mechanism affects the cost of a miss to a block that is clean/exclusive in another cache. This type of miss incurs significant additional latency and traffic in a large-scale machine to contact the cache containing the block, while a machine with snooping caches can check the cache during its usual snooping cycle at no additional cost.

3.3.2 Modifications to Base Protocol

This section describes how we modify the base Censier/Feautrier protocol to include the clean/exclusive feature. A bit is added in each cache tag and each directory entry to allow representation of the clean/exclusive state. Note, however, that a processor has permission to write a block that is clean/exclusive in its cache at any time. Therefore, if the directory shows a block to be clean/exclusive, it may be clean/exclusive (since it was supplied to the cache as such by the directory) or it may be dirty (because the processor may have since written the block).

To support the clean/exclusive state, we modify the actions defined by the Censier/Feautrier protocol (described in Section 3.1) as follows:

- *Read miss.* In this situation, the cache sends a read miss request to the appropriate memory module. If the directory shows the block is not clean/exclusive, then the actions described in Section 3.1 apply. If the directory indicates the block is clean/exclusive, then the actions depend on whether the protocol is *non-aggressive* or *aggressive.* We will present results for both variants in Section 3.3.3. In the non-aggressive protocol, the directory sends a special message to the cache containing the block. On receipt, the cache marks its copy of the block clean, and returns either an acknowledgement if the block was clean/exclusive in the cache, or the contents of the block if it was dirty. Since memory now has an up-to-date copy of the data, the original request is satisfied forthwith. However, the read miss incurs a latency of two full round trips in the network. The aggressive scheme eliminates one of these round trips from the latency by immediately returning the data in main memory to satisfy the original request. Afterwards, the message is sent to the cache with the block, just as in the non-aggressive case. If the reply indicates the cache's copy was dirty, then the directory must send an invalidation to the cache that issued the original request. This invalidation serves to remove the stale data that was initially returned to the cache.

- *Write hit to a clean block.* In this case, if the cache tag indicates the block is not clean/exclusive, then the protocol proceeds as described in Section 3.1. If the block is clean/exclusive, then the write is allowed to proceed in the cache immediately. In the aggressive protocol, the cache must also send a message to the directory indicating that the data is now dirty. The directory modifies its state accordingly, and returns an acknowledgement. These messages are required to enforce the memory consistency model (e.g., weak ordering).

- *Write miss.* In this case, the cache sends a write miss request to the appropriate memory module. If the block is not marked clean/exclusive in the directory, then the actions described in Section 3.1 take place. If the directory instead shows the block clean/exclusive, then the protocol proceeds just as in the non-aggressive read miss case described above, except that the cache already containing the block invalidates it rather than leaving it in a clean state.

Note that the aggressive protocol cannot be used if sequential consistency is to be maintained. We therefore evaluate three systems: a sequentially consistent system with a non-aggressive protocol, and weakly ordered systems with non-aggressive and aggressive protocols.

3.3.3 Results

The first step in evaluating the clean/exclusive state is measuring the frequency of each *event,* that is, each type of reference, in the address traces. For example, using the simulator described in Section 3.2, we determine the frequency of read misses to unmodified blocks that are not marked clean/exclusive at the directory. The next step is to assign costs to each type of event, for each protocol variant we are evaluating. Since we are using the number of network messages as the metric for both latency and traffic, the unit of cost is the number of network messages incurred. For instance, read misses to unmodified blocks not marked clean/exclusive cost two messages (one round trip to the directory) in both latency and traffic. For most events, the reference is not satisfied until all associated network messages are complete; for these events, the latency cost is the same as the traffic cost. However, for some types of events the reference can be satisfied before all of the associated traffic is necessarily completed (e.g., under weak ordering the data can be supplied to satisfy a write miss to a clean block before the invalidations have completed). In these cases, the latency cost is lower than the traffic cost. The event frequencies for each of the applications and the latency and traffic costs we have assigned to each event are given in Appendix F.

We can now multiply the event frequencies by their associated latency and traffic costs to obtain the aggregate latency and traffic for each protocol. The results are shown in Figures 3.1 and 3.2. Each graph in the figures represents a different application. The horizontal axes indicate the size of each processor cache, assuming a direct-mapped organization. The vertical axes show the total number of messages averaged over all data references in the address trace.

Let us consider the latency graphs of Figure 3.1. The most striking result is that the clean/exclusive state buys very little performance if a non-aggressive protocol is used. Under both sequential consistency and weak ordering, adding the clean/exclusive state never reduces the latency by more than 10.0% for any of the applications. And the

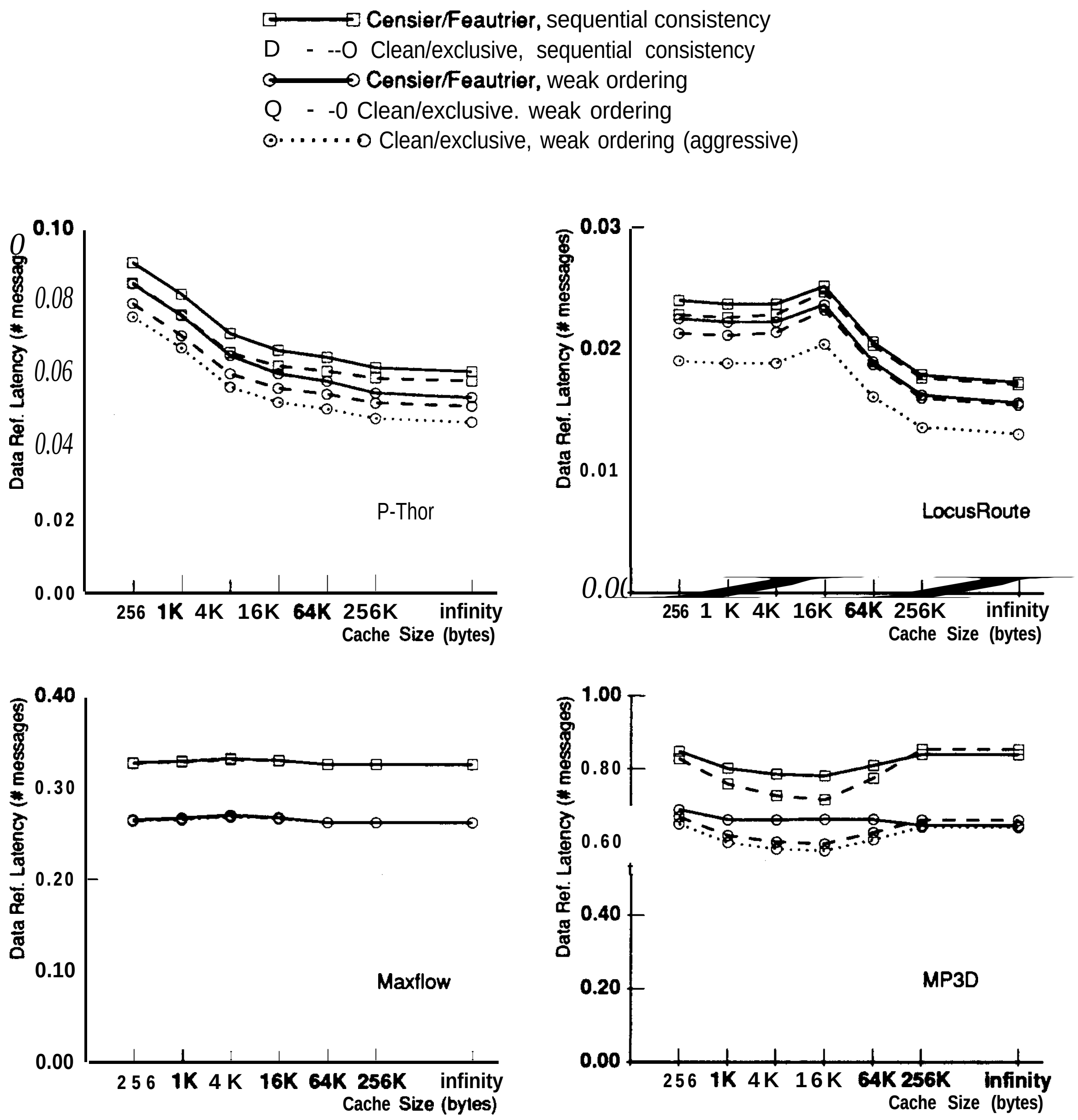

Figure 3.1: Average latency per data reference, expressed as the number of network messages required before a reference can be satisfied by the local cache. Note that the scales of the vertical axes differ.

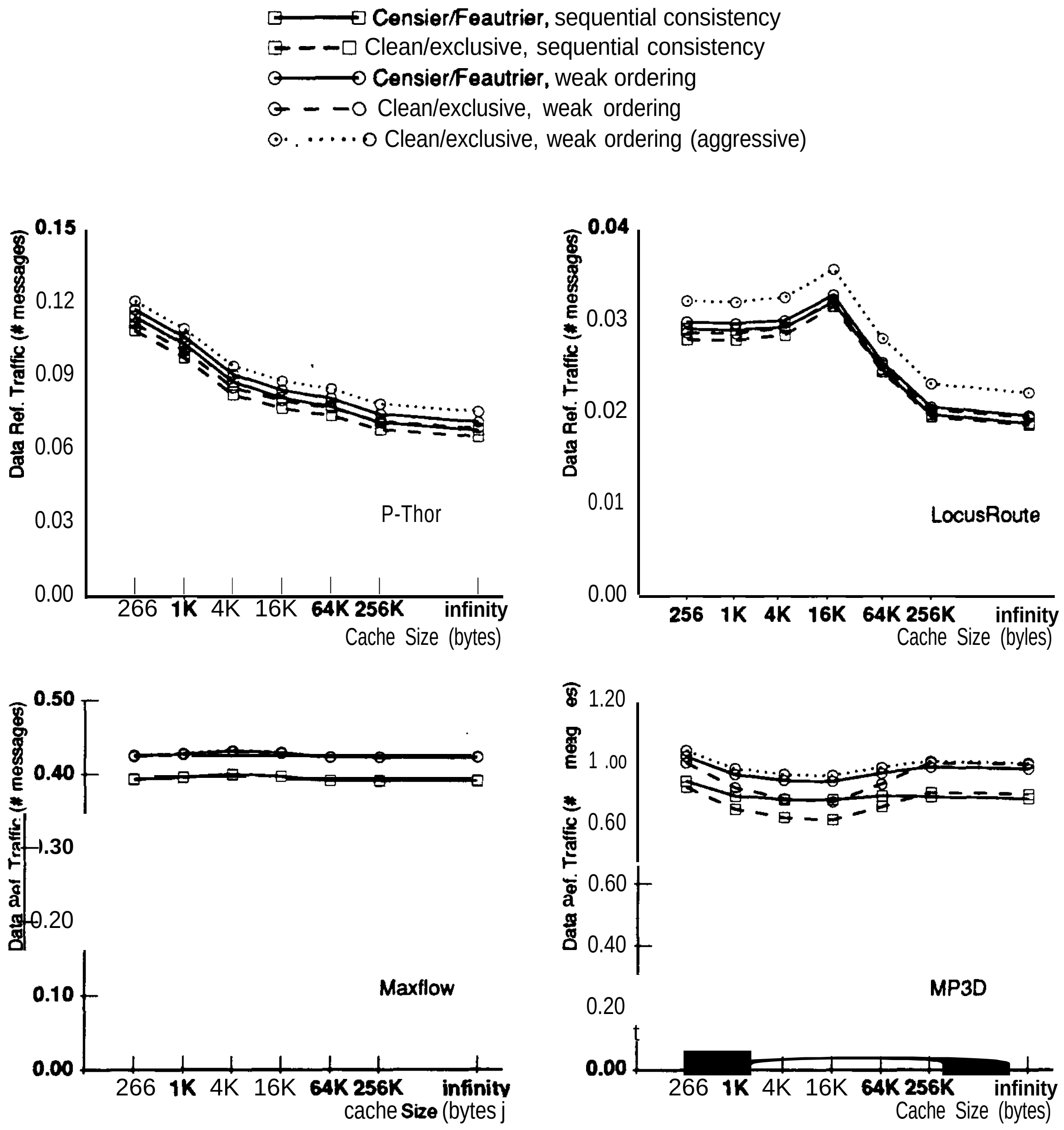

Figure 3.2: Average traffic per data reference, expressed as the number of network messages required. Note that the scales of the vertical axes differ.

greatest improvement is seen in small caches; for all of the applications, the percentage latency reduction peaks at caches of 16KB or less. With small caches, blocks are replaced more often, increasing the number of writes to clean/exclusive blocks, resulting in greater latency reduction than for larger caches.

Let us now turn our attention to larger caches. In the limit, that is, for infinite-sized caches, the clean/exclusive state reduces the latency by only 4.4% or less, and actually increases the latency slightly for Maxflow and MP3D. This increase is an extreme demonstration of the effect that causes the latency improvement to be small in general. Although the clean/exclusive state removes some messages that would otherwise be needed to gain write permission, it also increases the number of messages required to process misses to blocks marked clean/exclusive at the directory. Unfortunately, the increased latency of the latter effect cancels most of the latency reduction gained by the former, resulting in small improvements at best and slight degradation at worst.

The aggressive protocol performs somewhat better than the non-aggressive protocol, because a read miss to a block marked clean/exclusive in the directory need not wait for the round trip message to the cache already containing the block. Compared to the base Censier/Feautrier protocol under weak ordering, the aggressive protocol with the clean/exclusive state reduces latency by up to 18.5%. Even so, the latency reduction varies significantly across workloads. No improvement is seen in Maxflow, which exhibits very few write hits to clean/exclusive blocks. Even though these events are more frequent in MP3D, the improvement is only 1.0% for infinite-sized caches, because the event frequency is still small relative to other events incurring latency.

We should note that private (i.e., non-shared) data accounts for some of the latency improvement using the clean/exclusive state. Our simulations subject private data as well as shared data to the cache coherence protocol. If private data can be cached incoherently, perhaps by setting a special bit to this effect in the corresponding directory entries, then the latency and traffic of the base Censier/Feautrier protocol will be reduced. This is because write hits to clean, private data blocks can proceed immediately in the cache. However, for our benchmarks, we find private data to have only a small effect on the results. Caching private data incoherently never reduces the latency of the base protocol by more than 2%. This makes sense, because as long as a private data block is not replaced, only the first write to the block accrues any savings in latency, since the block would already be dirty on subsequent writes.

Let us now consider the network traffic, as displayed in Figure 3.2. The basic result is that adding the clean/exclusive state does not significantly lower the level of traffic. For all of the applications, the traffic is never reduced by more than 7.5%. And again, for large caches, the decreases are smaller. For infinite-sized caches, the reduction in traffic is never greater than 3.8%. Because of the messages sent on write hits to clean/exclusive blocks, the aggressive protocol increases the network traffic, by up to 12.7%. This additional traffic may offset the latency improvement of the aggressive protocol somewhat due to increased network contention.

We also considered a potential modification called *2-to-1 notify* to improve protocol performance beyond the levels shown in Figures 3.1 and 3.2. In the basic protocol with clean/exclusive support, we assume that a cache obtains a block in clean/exclusive state only when the block is not cached elsewhere. In other words, a block becomes clean/exclusive on the transition from zero caches containing the block to one cache containing the block. With 2-to-1 notify, the directory sends a message to the cache with the sole remaining copy on a transition from two caches to one cache containing the block (due to replacement in a cache). When the cache receives this message it modifies the state of the block to clean/exclusive. Our simulations show virtually no difference in latency and traffic between protocols with and without 2-to-1 notify. This holds true across all but one of our applications, at all cache sizes from 256 bytes up to infinite-sized, and for both the aggressive and non-aggressive protocols. Latency and traffic actually increases for the other application (LocusRoute), though never by more than 5%. This is because cache interference between two blocks in the working set makes it likely that a block replaced from a cache will be re-loaded in the near future. 2-to-1 notify is usually a detriment to performance in this case because the block resides in one cache for only a short duration. Because of this effect, and because we find the 2-to-1 transition simply does not happen very often, we do not recommend 2-to-1 notify.

While it is tempting to also use the graphs of Figure 3.1 to compare the performance of sequential consistency and weak ordering, we must keep in mind the limitations imposed by our assumptions on the scope of our results. In particular, care is required to properly interpret the results for weak ordering. From a performance perspective, that a system is weakly ordered merely says that it is permissible to hide latency by allowing processors to have multiple outstanding references. Nothing is implied about which latency-hiding mechanisms (e.g., write buffers, prefetching, multiple-context processors [25]) are actually used The only latency that is hidden using the base protocol under weak ordering in Figure 3.1 is the invalidation message latency. Hiding this latency is easily accomplished in the directory hardware. Weak ordering offers the flexibility to incorporate other latency-hiding techniques; their potential advantage is not reflected in the latency graphs.

At this point, we can draw several conclusions about the clean/exclusive state. In a strongly ordered system, it is probably not worth the additional complexity, since the reduction in latency is very small (never more than 8.4%, or 4.1% for infinite caches), and is sometimes negative. In a weakly ordered system, only the aggressive protocol fares significantly better. However, the potential improvement is still modest: none of our benchmarks showed more than a 20% reduction in latency. Furthermore, most of this latency that is reduced can be hidden with other, more general techniques such as write buffering. Since most high-performance systems already include write buffers, adding the clean/exclusive state will not be worthwhile. We conclude that the clean/exclusive state is an “optimization” that is better left out of most designs in favor of simplicity.

3.4 Removing Write Hit Requests

The second protocol enhancement we examine is actually a protocol simplification: treating write hits to clean blocks as if they were write misses. That is, when a write hit to a clean block occurs, the cache sends a write miss request to main memory, and receives a reply granting permission to write the block and containing the block’s data. The drawback of this policy is that while the processor needed the permission to write the block, the accompanying data did not have to be sent from main memory since the block was already valid in the cache. The advantage of this scheme is a simpler protocol that correctly handles a tricky scenario in the base protocol: if two caches issue a write hit request to the directory at about the same time, the block will be invalidated from the second cache to reach the directory with its request before receiving its reply.* Removing write hit requests obviates detecting and handling this special case separately in the protocol.

3.4.1 Methodology

As before, we use the Censier/Feautrier protocol described in Section 3.1 as the base protocol. To remove the 'write hit request, we modify the base protocol by simply performing the actions of a write miss whenever a write hit to a clean block occurs.

We assume that the base and enhanced protocols both use two message sizes, one for messages that do not contain a block of data (*short* messages), and another for those that do (*long* messages). The effect of removing the write hit request from the protocol is that the short message reply from main memory on a write hit to a clean block becomes a long message reply.

To determine the overall increase in network traffic, we count the number of short and long messages required to service all of the data references in an address trace. We do this by multiplying the frequency of a type of event by the number of short (and then long) messages required by each event of that type. Summing across all event types yields the total number of short and long messages. After following this procedure for both the 'base and enhanced protocols, we are in a position to evaluate the increase in traffic.

Of course, removing write hit requests does not change the number of messages, but rather only their size. The increase in traffic will therefore depend on the relative size of short and long messages. Because this hinges on a number of factors, such as the cache block size, the number of bits in an address, and the width of the interconnection network, we present the traffic increase as a function of the ratio of the long message size to the short message size.

[2] We will examine this situation in greater derail in Section 4.5.2.

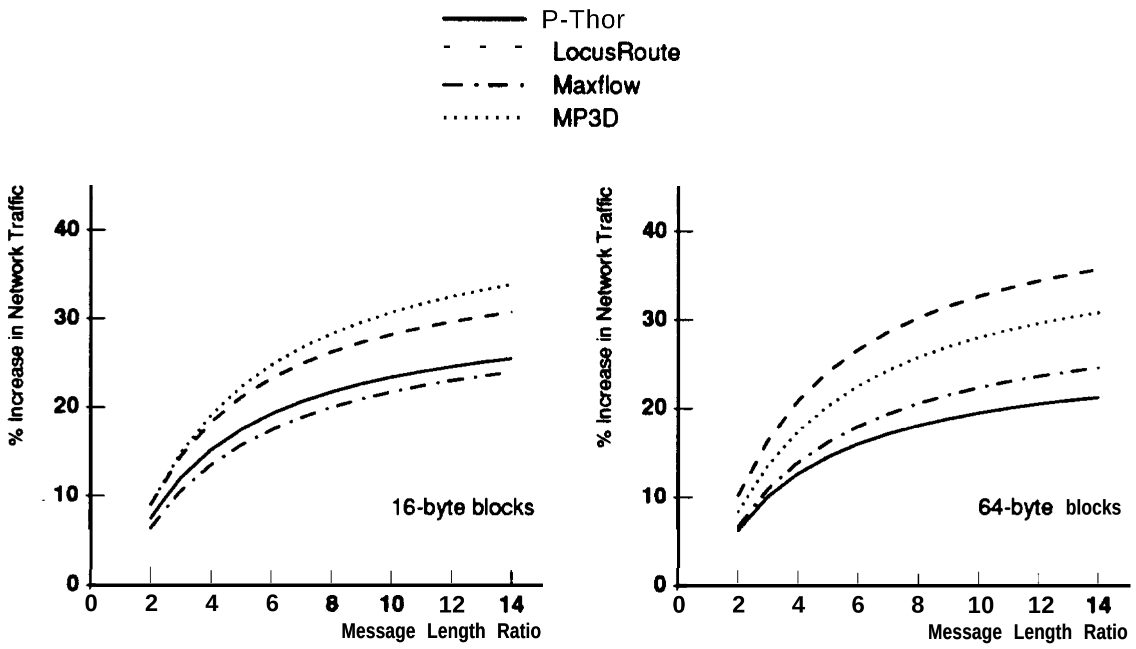

Figure 3.3: Increase in traffic due to removing write hit requests. Message *length ratio* is the length of long messages (i.e., messages containing a data block) divided by the length of short messages (i.e., messages not containing a data block).

3.4.2 Results

Figure 3.3 shows the increase in traffic due to removing write hit requests, as a function of the ratio between the size of long messages and short messages. Separate graphs are required for each block size, since varying the block size affects the stream of memory requests generated by each cache. For brevity, only the results for infinite-sized caches are shown; for all of our traces, these results reflect the worst-case increase in traffic across all cache sizes.

By using the results in Figure 3.3, we can determine the increase in network traffic caused by removing write hit requests for different systems. For instance, consider a machine with 32-bit addresses and 32-bit words. Assume the cache block size is 16 bytes, i.e., four words. If short messages are two words long (a header word and an address) and long messages are six words long (a header word, an address, and a block of data), then the ratio of message lengths is 3. We can see from Figure 3.3 that the resulting increase in consumed bandwidth is between 10% and **15%,** depending on the application and cache size. Of course, this may or may not be acceptable, depending on other factors such as the cost and performance of the network relative to other parts of the system. Keep in mind that the block size and the message length ratio are strongly correlated, since for most systems, the length of long messages will be equal to the length of short messages plus the block size. In the example above, if the cache block size were 64 bytes instead of 16 bytes, then the resulting message length ratio is 9. Figure 3.3 indicates that removing write hit requests in this system would increase traffic by 18% to 31%. Again, this increase may be acceptable depending on the circumstances. However, with the increase in traffic approaching one-third for this configuration, many designers would probably include the additional complexity of handling write hit requests.

We conclude that designers should consider removing write hit requests from their coherency protocol. In many cases, and especially for modest cache block sizes (16 bytes and smaller), the resulting increase in traffic is less than 15% for many system configurations. However, removing the write hit requests incurs a higher penalty for machines with large block sizes (64 bytes and larger). For these systems, designers should carefully examine the resulting loss in performance (or increase in cost to maintain equal performance) before purging write hit requests from the protocol.

3.5 Request Forwarding

The final protocol enhancement we evaluate *is request forwarding.* This optimization reduces the latency of a cache miss to a block dirty in another cache. it does this by having the cache containing the data forward it directly to the cache requesting the data, rather than first sending it to the directory. The drawback of request forwarding is the additional hardware complexity required. In this section we determine the latency improvement for our benchmarks, allowing us to comment on the usefulness of request forwarding vis-a-vis its hardware cost.

3.51 Background

Request forwarding has been included in many snooping cache coherency protocols for single-bus multiprocessors, and is generally considered to be a good idea for these machines. This is because the number of bus cycles is cut in half for misses to blocks dirty in a cache, and the additional implementation cost is low if the bus and cache controller already supports cache-to-cache data transfers. In a large-scale multiprocessor, the potential performance gain of forwarding is smaller and the implementation complexity is larger than for a single-bus machine, making the design decision more difficult.

Request forwarding for large-scale machines was proposed by Lenoski et al. [32] and implemented in the DASH multiprocessor. In his thesis, Lenoski also reports the improvement in reference latency due to forwarding for the DASH architecture, as measured from the prototype by a hardware monitor [34]. In Section 3.5.3 we will compare these measured values to our simulation results.

3.5.2 Modifications to Base Protocol

We again use the Censier/Feautrier protocol (see Section 3.1) as a base. To modify the base protocol for request forwarding, we alter the actions that are taken for read and write misses, if the directory discovers the block is dirty in a cache. Recall that under the base protocol, the directory sends a message to the cache containing the data, instructing the cache to return the data to the directory. When the data arrives, the directory sends it to the cache that requested it. With request forwarding, however, the cache containing the data sends it to the cache that requested it rather than the directory. In the case of a read miss, the data is also sent to the directory so it can be written back to main memory; this message is called a *sharing writeback.* For a write miss, an acknowledgement is sent to the directory.

Possible implementations of request forwarding vary widely in their protocol complexity. In a directory that delays satisfying requests for blocks with outstanding transactions, there are two basic options. First, the sharing writeback (for a read miss) or acknowledgement (for a write miss) can be sent to the directory by the cache receiving the data. This is very simple from a protocol perspective, since the directory may not process new requests for the block until the dirty data is no longer in transit. Second, the sharing writeback or acknowledgement may be sent by the cache supplying the data. The directory is unblocked sooner, but it may later send requests for the data to a cache that has not yet received it.

Now consider a directory such as the one used in DASH that does not maintain state indicating whether a block has outstanding transactions. Here, we find the forwarding protocol is more complex. Lenoski et al. relate a subtle situation that demonstrates this complexity. On a write miss the directory does not update its contents until it receives the acknowledgement message from the cache forwarding the data. This message must be received before the block is forwarded again (perhaps by an old message that has been delayed in the network); otherwise, a race may occur between multiple acknowledgement messages headed towards the directory. If the wrong message wins the race, the directory will be updated incorrectly. To solve this problem, the directory sends its own acknowledgement to a cache receiving forwarded data; the cache is not allowed to give up this data until this message is received.

Another subtle situation may occur in a machine with write hit requests, issued when a write hit to a clean block occurs.[3] If a read miss occurs on a block dirty in another cache, that cache forwards the data and sends a sharing writeback to the directory. Once the data has arrived at the cache, satisfying the read miss, the processor could later write the block. This would cause a write hit request to be sent to the directory. If the request is serviced before the sharing writeback, then there will be two dirty copies of the block in the system: one in the sharing writeback message, and one in the cache that issued the write hit request If the cache now decides to write back the dirty data, the protocol must ensure that this data is not eventually destroyed in main memory when the. now-stale data in the sharing writeback message arrives.

3.5.3 Results

To determine the improvement in latency due to request forwarding, we assign a latency cost to each type of event, just as we did for evaluating the clean/exclusive state in

[3]Lenoski et al. do not cover this situation because the DASH machine does not include write hit requests.

Section 3.3.3. The costs for request forwarding are identical to the Censier/Feautrier costs except for read and write misses to dirty blocks, which incur a latency of three network messages instead of four.

The average latencies for both the base protocol and the request forwarding protocol are shown in Figure 3.4. In the limit, if every reference causing traffic was a miss to a dirty block, request forwarding would decrease the average latency from four network messages to three, for a 25% improvement in latency. Figure 3.4 shows that for our applications, request forwarding achieves far less than that maximum theoretical improvement, with latency reductions ranging from 5.7% to 14.3% under weak ordering and 5.2% to 12.1% under sequential consistency. As before when we looked at the clean/exclusive state (see Section 3.3.3), our results for weak ordering assume that only the latency of invalidations and their replies are hid&n. If the latency for some events other than misses to dirty blocks were hidden, then the percentage reduction in latency due to request forwarding would increase, perhaps making the scheme somewhat more attractive. For instance, if all write latency is hidden under weak ordering, we find the latency reduction due to forwarding ranges from 6.5% to 20.5%. The latter reduction is seen in MP3D, which incurs a very high invalidation miss rate (about 25%) and therefore sees larger relative benefits of forwarding. In his thesis, Lenoski measures the improvement in read miss latency to be from 0.2% to 11.0% for five programs running on the DASH prototype.

We conclude that adding request forwarding to the coherency protocol results in only modest performance gains. The latency due to communication is reduced by roughly 5% to 15% depending on the application; the improvement in execution time will of course be less. Request forwarding will be a useful addition to a protocol only if it can be easily implemented, otherwise, it is probably not worth the effort

3.6 Summary and Conclusion

In order to choose a coherency protocol, designers must weigh the value of protocol features against their implementation costs. In this chapter we have employed trace-driven simulation to quantify the performance effects of three protocol enhancements. We have used the number of network messages as an architecturally independent metric

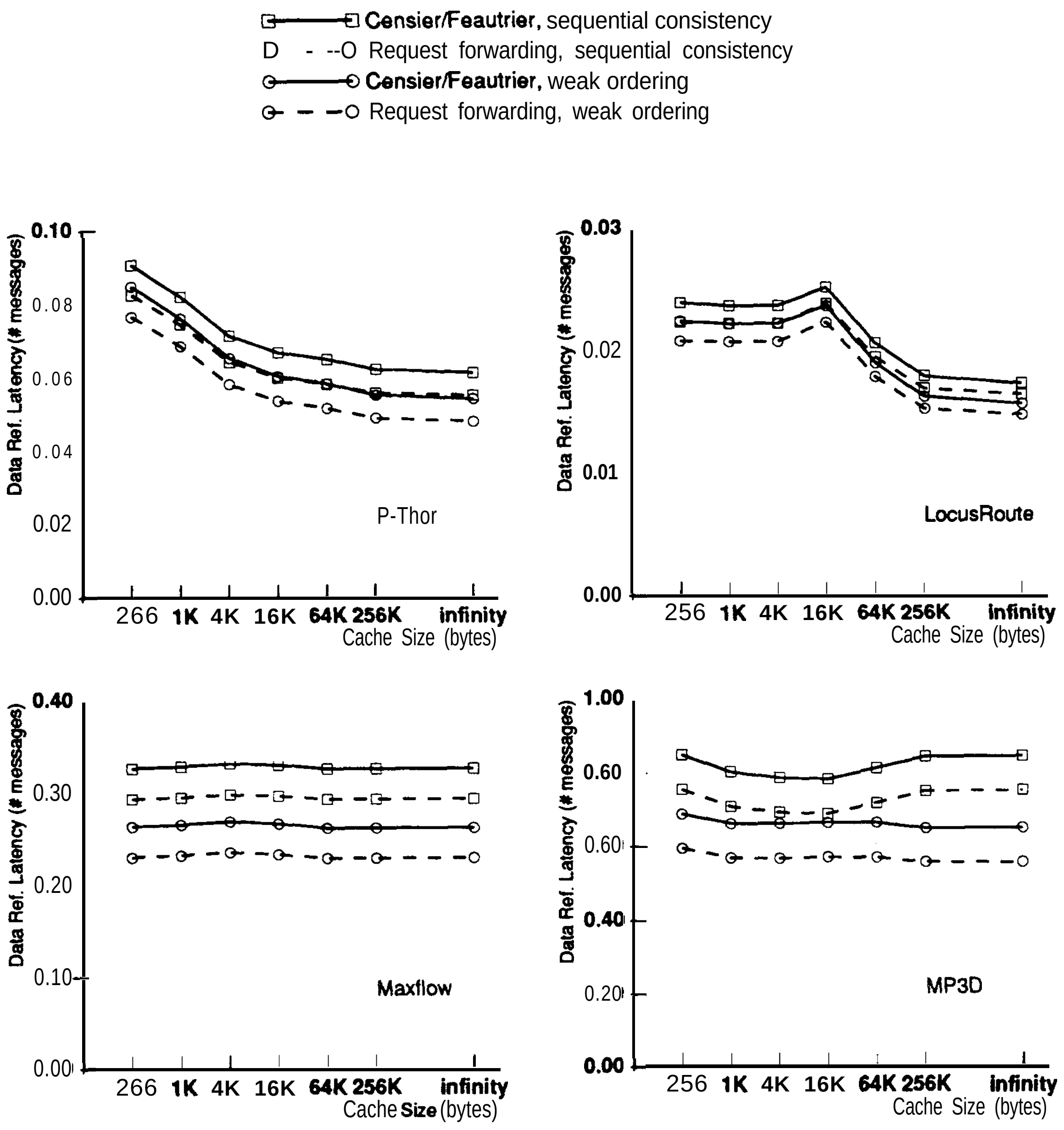

Figure 3.4: Average latency per data reference, expressed as the number of network messages required before a reference can be satisfied by the local cache. Note that the scales of the vertical axes differ.

of reference latency and network traffic.

Our performance results for the three protocol features make the virtues of simplicity clear. The additional complexity associated with two of our enhancements, the clean/exclusive state and request forwarding, seems to buy very little performance for our trouble. Even removing write hit requests, a staple of most published consistency protocols, proves to have only a slight negative impact on the network traffic for systems with small cache blocks, making the simplification worthwhile. Of course, simplification can always be taken too far: with large cache blocks, removing write hit requests can substantially increase network traffic.

Chapter 4

Directory and Protocol Implementation

In previous chapters we examined some of the interesting high-level directory and protocol design issues. We will now shift our focus to the trade-offs that arise at the implementation level, focusing on fundamental issues that must confront anyone designing directory hardware. We begin by briefly describing a very straightforward processing node. The remainder of the chapter then uses this design to demonstrate the relevant correctness and performance issues. In particular, we will show how to implement multiple-threaded directories, avoid deadlock, and handle subtle cases in the coherency protocol.

4.1 A Basic Node Design

As mentioned in Section 1.3, we assume that each processing node contains a processor and a cache, as well as an interconnect controller and a portion of main memory with accompanying directory information. Directory-based cache coherency protocols operate by passing messages between caches and directories in the system, using the interconnection network as necessary. So within a node, we are primarily interested in the controllers for the cache, directory, and interconnect. *The directory controller (DC)* maintains the contents of the directory bits. *The cache controller* (CC) maintains the internal cache directory (comprised of tags, valid bits, dirty bits, etc.) and satisfies memory requests from the processor. *The interconnect controller (IC)* implements the interface between the node and the interconnection network. For our purposes we can view the interconnect

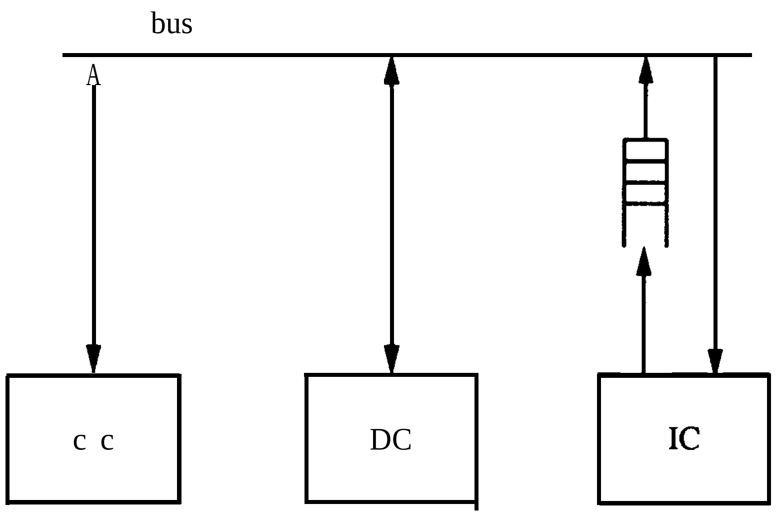

Figure 4.1: Interconnecting the directory, cache, and interconnect controllers.

controller abstractly as a mechanism that reliably sends and receives messages.

We define the following types of messages used by the protocol. A cache issues a request to memory using a cache-to-main-memory (abbreviated C→MM) command, to which the directory issues a *reply.* Before issuing the reply, the directory may need to issue one or more main-memory-to-cache (abbreviated MM-C) commands, to which caches reply. This implies that each controller within a node must be able to communicate with the other two controllers. For instance, the cache controller sends messages to either the local directory or to a remote directory via the interconnect controller. Similarly, messages from the directory controller may go to the local cache or to remote caches. Finally, messages arriving at the node through the interconnect controller may go to the cache or the directory. Figure 4.1 shows a basic means of interconnecting the three controllers using a bus. The interconnect controller feeds incoming messages into a queue so that arriving messages can be accepted from the network immediately, i.e., without waiting for a shared resource on the node to become available. Otherwise, a global resource (the network) may block due to local contention on the intra-node bus.

To form a basis for the correctness and performance issues we address in this chapter, we must flesh out the design further. To do this, we next examine the hardware inside the directory controller. We then describe how this hardware is used to achieve cache

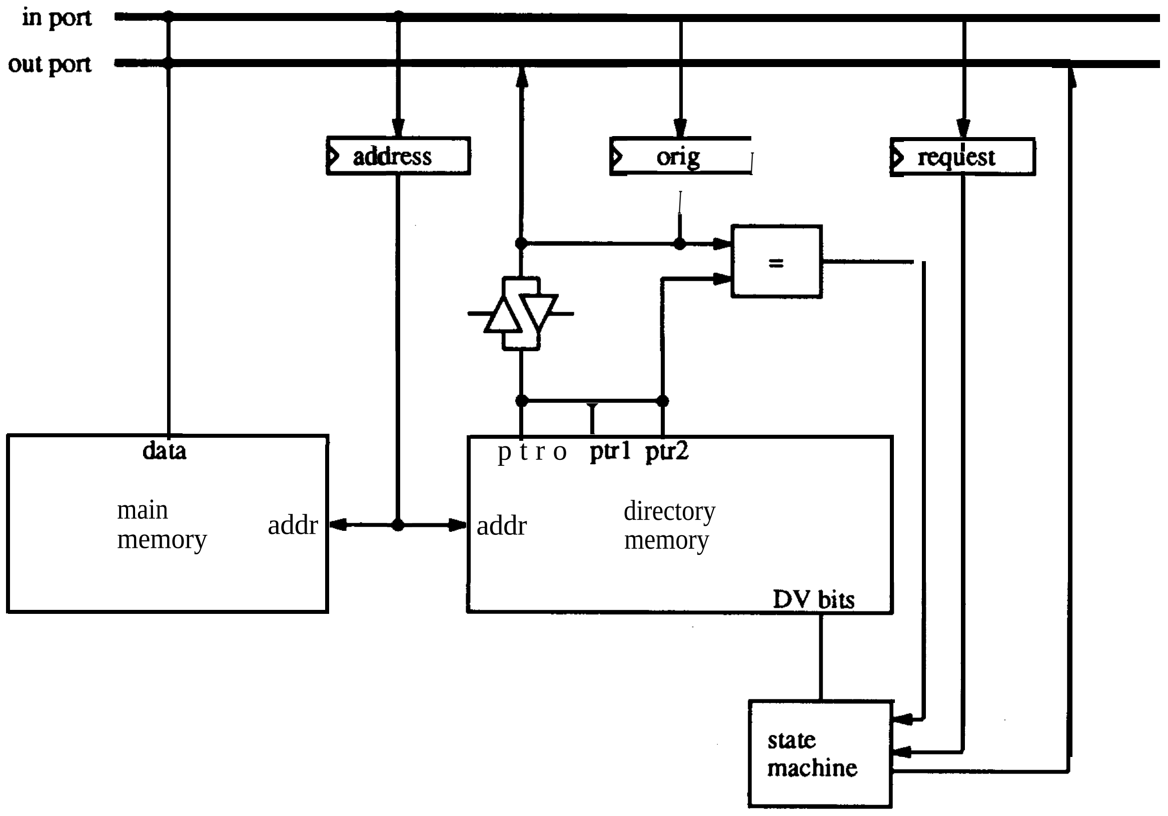

Figure 4.2: The directory datapath.

consistency by defining the necessary commands and replies, and developing a state machine to drive the directory hardware.

4.1.1 Directory Datapath

The basic internal **datapath** of the directory controller is shown in Figure 4.2. In general, a directory accepts a stream of messages from an *in port* and produces a stream of messages on *an out port.*[1] All data transfer to and from the directory controller occurs on these ports. Main memory is shown abstractly as a "black box" with address and data ports. The **datapath** assumes a no-broadcast limited pointers directory with three pointers per entry. There is one valid bit per pointer to indicate if the pointer is in use, and a single dirty bit per entry to indicate if the block is currently dirty in a cache. The **datapath**

[1] A single bidirectional port may be used instead, as in Figure 4.1.

controller can be implemented as a state machine. Figure 4.2 shows only a subset of the signals the state machine reads and writes; later we will present state transition tables that define the state machine more precisely.

When a message arrives, the datapath controller latches the address field from the in port into an address register that drives the memories. If the message is a C→MM command, then the identifier of the cache that originated the message is stored into a register, since the directory will need to reply to that cache. A comparator is included to compare this register against the contents of the pointers. On a write hit to a clean block, this comparison is used to prevent the directory from sending an invalidation to the cache that issued the write request.

The path between the ‘originating node register and the pointer bits of the directory is needed to write a new cache identifier into a directory pointer. The transceiver is used to isolate the original node register from the pointers when the comparator is being used. Outgoing messages can be sent to either the cache indicated by a pointer or to the cache contained in the originating node register, corresponding to MM→C commands and replies to C→MM commands, respectively.

4.1.2 Command and Reply Messages

Having defined the basic structure of the node hardware, we can now define the specific commands and replies that are required to implement our basic coherency protocol. We need four different C→MM commands:

- *readlnon-exclusive (read/non-ex)*. The cache issues this command on a read miss in order to get a copy of the block from main memory.
- *read/exclusive (readlex).* In the case of a w-rite miss, the cache uses this command to get an exclusive copy of the block from main memory.
- *exclusive (ex)*. When the cache encounters a write hit on a clean block, it must ask the directory for exclusive access to this block; it will then be the single cache with the dirty copy of the block.

- *writeback.* This command is used by the cache to write back a dirty block to main memory if, for example, it is replacing the block with another one.

As we mentioned, a directory may need to have other caches take action when one of these **C→MM** commands is received. To do this, the directory sends **MM→C** commands. We define three **MM→C** commands as follows:

- *copyback.* This command tells the target cache to **copyback** the indicated block to main memory. The block need not be invalidated, however. This command is required when there is a read miss in a cache on a block that is dirty in another cache.

- *flush.* A cache receiving this command should copy the specified block back to main memory and invalidate the block. This command is necessary when a cache has a dirty copy of the block and there is a write miss on the block in another cache.

- *invalidate.* This command should cause the receiving cache to invalidate the indicated block. It is issued when another cache requests exclusive access to a block, either through a *read/exclusive* or *exclusive* **C→MM** command.

Note that some of these commands require a *reply.* We define the following two replies:

- *return data (retdata).* This reply is issued in response to a **C→MM** ***read/exclusive*** or *read/non-exclusive* command, and is used to return the requested block of data to the cache.

- *copyback data (cbdata).* This reply is the response to a **MM→C** ***copyback*** **or** ***flush*** command, and carries the requested data from the cache back to main memory.

4.13 State Machine

Using the messages we have just defined, we can now write the state transition table describing the state machine that controls the directory datapath. We use a very simple coherency protocol. Any requests for blocks dirty in another cache are first retrieved from that cache and then supplied to the requesting cache. Requests for clean blocks are supplied immediately; if an exclusive copy is requested, invalidations are sent to the appropriate caches as indicated by the pointers.

We also make the simplifying assumption that the directory is *single-threaded,* that is, it may have transactions pending for no more than one C→MM command at a time. For instance, if a single-threaded directory is processing a *read/non-exclusive* command, C→MM commands for other addresses cannot be processed until the directory sends the ***return_data*** reply, even if the directory must first issue a ***copyback*** or *flush* command to retrieve the data from a cache. This results in a directory that sits in one of two states between messages, *idle* or *blocked.* If the directory is idle, new C→MM commands may be processed when they arrive. If the directory is blocked, the directory can process only replies it is awaiting.

Table 4.1 shows the resulting state transition table. While the table makes no allowances for timing considerations, such as the number of cycles necessary to access the directory bits, all of the necessary flow of data is indicated. Some incoming commands result in multiple outgoing commands; these are denoted with multiple lines in the output side of the table.

The table shows four inputs to the state machine. The first two are the current state of the block and the incoming request type. The next input, used only for replies to MM→C commands, indicates whether the directory must now reply to a *read/non-exclusive* or *read/exclusive* command. The next input is the dirty/valid (DV) state bits for the block. In this column, the condition ***fp*** stands for free *pointer.* This condition is true if there is at least one unused pointer, that is, [illegible] one of the valid bits is 0.

The first output in the table is the outgoing request type for commands and replies sent by the directory controller. The next column shows the processor number to which the message should be sent. The abbreviations in the processor number column are defined as follows:

- ***orig.*** This indicates the node that originated this sequence of messages with a C→MM command.

	inputs:				outputs:					
	state	incoming message	previous message	DV bits	outgoing message	P#	DV bits	free ptr	next state	other actions, conditions
1	idle	writeback	x	x			zero	•	idle	memory ← data
2	idle	read/nonex	x	$\overline{\text{dirty}} \wedge$ fp	retdata	otig	(add)	orig	idle	supply data
3	idle	read/nonex	*x*	$\overline{\text{dirty}} \wedge \overline{\text{fp}}$	retdata	orig				supply data
4					invalidate	rptr	*	orig	idle	
5	idle	read/nonex	*x*	dirty	copyback	dptr	*	*	blocked	
6	idle	read/ex	x	$\overline{\text{dirty}}$	retdata	orig				supply data
7					*invalidate*	ptr0				send if ptr0 valid
8					invalidate	ptr1				send if ptr1 valid
9					invalidate	ptr2	dirty	orig	idle	send if ptr2 valid
0	idle	read/ex	x	dirty	flush	dptr	•	•	blocked	
11	idle	ex	x	x	invalidate	ptr0				send if ptr0 valid & ≠ orig
2					invalidate	ptr1				send if ptr1 valid & ≠ orig
3					invalidate	ptr2			idle	send if ptr2 valid & ≠ orig
4	blocked	cbdata	read/nonex	*x*	retdata	orig	(add)	orig	idle	memory ← data, supply data
5	blocked	cbdata	read/ex	x	retdata	orig	dirty	orig	idle	supply data

Table 4.1: Basic state transition table for the datapath controller. An "x" indicates a "don't care" condition, and a "*" means the field should remain unchanged from its previous value. Other abbreviations are described in the text.

- *rptr.* This stands for *replacement pointer.* The processor number saved in the pointer that is about to be replaced is to be used. The determination of which of the three pointers this refers to depends on the pointer replacement policy.
- *dptr.* This is an abbreviation for *dirty pointer. The* processor number saved in the single pointer that is valid when the block is dirty is to be used.
- *ptr0, ptr1, ptr2.* These refer to the processor number saved in one of the three pointers, labeled pointer0, pointerl, and pointer2.

The next output column shows the changes that should be made to the dirty/valid bits in the directory entry. The notation *(add)* means the valid bit corresponding to the pointer that is becoming valid should be set, thereby "adding" the pointer to the existing state. Also, *(add)* implies that the dirty bit should be cleared if it is set. The notation *zero* indicates that all of the valid bits should be cleared. The next output, free *ptr,* shows when the processor number in the originating node register should be saved in a pointer that is free or has been made free by some action taken by the datapath controller. The *next state* column indicates the new state of the directory. The final column shows other actions that must be taken, or any additional conditions that must be satisfied to send the outgoing message.

4.1.4 Correctness and Performance

We have described a simple node design, including the intra-node interconnect, the directory datapath, the types of messages that must be supported by the coherency protocol, and the state machine to implement the protocol at the directory. Throughout the remainder of this chapter we will use this basic design to demonstrate the fundamental correctness and performance issues with which directory designers must be concerned. In the next three sections we show how to easily make the directory multiple-threaded, and how the node design can be modified to prevent deadlock troubles. Section 4.5 then demonstrates several subtle problems in our coherency protocol and suggests appropriate solutions. Having addressed these correctness and performance issues, in Section 4.6 we return to the basic node design we have discussed and incorporate our suggested improvements, thereby producing a better design for a node.

4.2 Multiple Threads

As we said in Section 4.1.3, the basic node design we have described is single-threaded. A single-threaded directory only supports a single outstanding transaction at a time, leaving the memory blocked while the directory waits for a reply to a MM-C command it has sent. This restriction is not placed on multiple-threaded controllers. While waiting for caches to respond to MM→C commands, a directory with multiple thread: can proceed with C→MM requests for other blocks For thrs reason, multiple-threaded directories have a performance advantage over single-threaded directories.

To implement multiple threads, we must save state for each C→MM command that cannot be satisfied immediately by the directory because one or more MM→C commands must be sent (e.g., to retrieve a dirty block from a cache). When replies to the MM→C commands are later received, this state allows the directory to reply to the original request. To identify the state that must be saved, we need to determine the information needed by the directory to send this reply. For instance, to send a *return data* reply after retrieving a dirty block from a cache, the directory needs the processor number and address from the original request for the block. While bits could be added to each directory entry to save this state, there is a better solution that requires no additional storage. The state can be easily maintained by encapsulating it in the **MM→C** commands that are sent, with the understanding that the receiving caches will simply echo the state back in their replies. Of course, the address must be sent anyway, so the only extra data in the message is the processor number. An extra bit must be added to the directory entry as well to indicate that a transaction is pending so that unrelated **C→MM** messages for that address will not be processed until the pending transaction is complete. This extra bit encodes the *idle* and *blocked* states indicated in the state transition table we presented (see Table 4.1). In the basic single-threaded design, *idle* and *blocked* indicated the state of the entire directory; in a multiple-threaded directory, this state is maintained on a per-block basis.

4.3 Avoiding Reply Deadlock

The basic node design of Section 4.1 does not consider the possibility of deadlock. Unfortunately, the design can deadlock quite easily. When the directory sends a MM-C command to retrieve a dirty block, the directory does not process further commands for that address (or *any* further commands if it is single-threaded) until the reply to that command is received. If any further commands are received by the node, they must wait in the input queue (recall Figure 4.1). Unfortunately, they will wait forever, since the reply will become stuck behind them in the same queue. As designers, we would like to know in general whether deadlock may result from prohibiting a given message type from acquiring a resource while that resource awaits a reply to a message it has sent. In this section, we develop sufficient conditions for avoiding this type of deadlock, which *we* call *reply deadlock.*

To determine these conditions, we first examine the different sequences of message types that can occur for a given request. We can arrange these sequences in a tree structure, as in Figure 4.3. In all cases, a request begins with a **C→MM** command, shown at the root of the tree. In the case of a request to a clean block, the left branch is

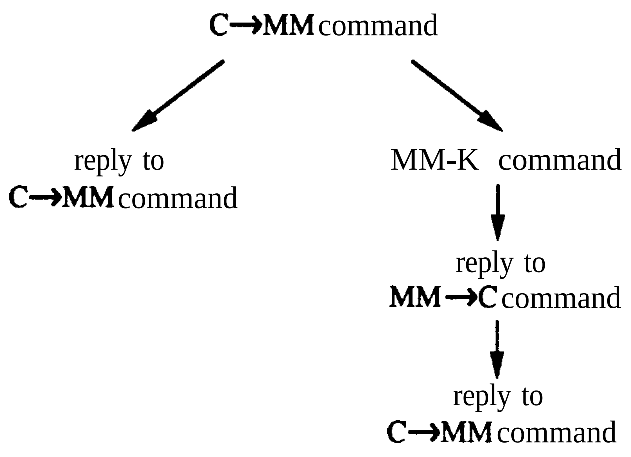

Figure 4.3: Possible sequences of messages for a given request.

followed: the directory sends the reply to the request. If the request is for a dirty block, however, then the right branch is followed: the directory must send a **MM→C** command and receive a reply before it can send a reply to the original **C→MM** command. The tree shows four types of messages, that is, two types of commands and two types of replies. Given this tree, we can state a sufficient condition for avoiding reply deadlock:

Condition 1 *If a message type B exists in any subtree for which message type A is a root, then messages of type B cannot be postponed by a unit because it is waiting for a reply to a message of type A.*

In other words, if a hardware unit expects a reply to a message it has sent, it should not block message types that may be needed for the reply to arrive. If it does, the unit may deadlock with another unit trying to do the same. The proof that the above condition is sufficient is as follows:

Theorem 1 *If Condition I holds, then no request fails to complete due to a unit blocking messages while awaiting a reply.*

Proof Define a *request* to be a sequence of messages beginning at the root of the tree in Figure 4.3 and completing at a leaf. We denote the set of active requests at a given time as $\{r_i\}$. Within an active request r_i, the active message is at height h_i in the tree, where the height of a node is the "length of the longest path from the node to a leaf" [4]. The proof proceeds by induction: we will first show that all requests $\{r_i \mid h_i = 0\}$ must complete. We will then show that if all requests $\{r_i \mid h_i \leq j\}$ must complete, then all requests $\{r_i \mid h_i = j + 1\}$ must complete.

Basis Consider the requests $\{r_i \mid h_i = 0\}$. Figure 4.3 indicates the current message for these requests must be a reply to a C→MM command. Since all possible subtrees contain a reply to a C→MM command, Condition 1 ensures that no unit may block these messages due to pending replies. Therefore, all requests $\{r_i \mid h_i = 0\}$ must complete.

Induction Assume all requests $\{r_i \mid h_i \leq j\}$ must complete, and consider the requests $\{ r_i \mid h_i = j + 1)$. Condition 1 implies that the latter requests may only be postponed by pending replies for message types with height less than or equal to j. However, it is given that requests of this height must complete. All requests r_i for which $h_i = j + 1$ must therefore eventually change state to $h_i = j$. From this point they must complete due to the inductive hypothesis. □

We now have a general framework in place for deciding when resources may block messages without risking reply deadlock. We can avoid this deadlock at each of the system resources by ensuring that each satisfies Condition 1. In the next several sections we look at some of the ramifications on each of the major resources, that is, the cache, directory, and interconnection network.

4.3.1 Cache Reply Deadlock

The only case in which a cache awaits a reply is when it sends a C→MM command. Since this command is at the root of the tree in Figure 4.3, Condition 1 implies that the cache controller must not block any message types that may arrive simply because it awaits a reply. To illustrate the consequences of ignoring this rule, assume the cache controllers were designed to block incoming MM-C commands while awaiting a reply to a **C→MM** command. Now imagine two processors each issue a *read/non-exclusive* to a block held dirty in the other's cache. The directories for the blocks will issue ***copyback*** commands to the caches to retrieve the dirty blocks. But if these caches must postpone *the **copyback*** until *they* receive the reply to the *readlnon-exclusive* commands they sent, then they will wait forever, because neither reply can be returned by the directories until the ***copyback*** commands have been completed.

4.3.2 Directory Reply Deadlock

The directory awaits a reply whenever it sends certain MM-C commands, such as *copyback.* According to Condition 1, this wait must not cause the directory to block any message of the types found in the **subtree** in Figure 4.3 for **MM→C** commands. This may have several ramifications in a typical design. For instance, it is obvious that if the directory were to block replies to **MM→C** commands, it would never see the reply for which it waits. Unfortunately, this is sometimes the case for our basic node design of Section 4.1. In the basic single-threaded design, all **C→MM** commands block at the directory when a reply is forthcoming; in the multithreaded design, those **C→MM** commands for addresses with outstanding transactions block (see Section 4.2). While Condition 1 permits the directory to block these commands, the result is that replies may also be blocked. This is because our design provided only a single queue for messages arriving at the node (see Figure 4.1); replies may get stuck behind blocked **C→MM** commands in this queue, resulting in deadlock. We will describe one solution to this problem in the next section.

4.3.3 Network Reply Deadlock

Condition 1 affects the network design in several ways, where the network in this case includes both the channels of the interconnection network between nodes and the **intra-**node communication paths. In the inter-node network, no link used to transfer a **C→MM** command may be held for the eventual reply if that might block **MM→C** commands, since MM-C commands are in the **subtree** of **C→MM** commands in Figure 4.3. For most networks, this requires a split-transaction strategy (i.e., separate network transactions are required for a command and its reply).

Within a processing node, care must be taken to ensure that one controller does **not** block certain messages destined for another controller simply because the first awaits a reply. For instance, having sent a MM-C command, a directory awaiting a reply must not prevent a different MM-C command arriving at the node from being delivered to the local cache. This is because MM-C commands reside in the subtree for MM→C commands in Figure 4.3 (since the root of a subtree is also a member of the subtree). To demonstrate this, consider a machine with directories designed to block a MM→C command arriving at the node whenever they await a reply to their own MM-C command. Deadlock may occur if each directory on two nodes sends a MM→C command to the other node, for the commands may be blocked from the destination cache by the local directory. This can happen in our basic node design, since the MM-C command can become stuck behind a blocked C→MM command in the node's single input queue.

To solve this problem in our basic node design (as well as the similar problem described in Section 4.3.2), we can add an additional queue that only holds C→MM commands arriving at the node, as shown in Figure 4.4. In this scheme, all arriving C→MM commands enter this additional queue, while all other messages enter the other queue. This strategy is effective since C→MM commands, the only messages that may block while a controller awaits a reply, now cannot prevent other types of messages from reaching their destinations. An alternative would be to **negative** *acknowledge* (NAK) any C→MM commands that block at the directory, but this policy may have a negative impact on performance. In any case, the additional queue may also be used to reduce the incidence of store-and-forward deadlock, which we examine next.

4.4 Avoiding Store-and-Forward Deadlock

In general, the directory controller and cache controller may have to block when trying to send an intra-node message to another unit if that unit is not ready to accept another message. This creates *the* potential for *store-and-forward deadlock* [49], which occurs if the directory and cache controllers each need to send a message to the other, but neither can accept more messages until their own message has been sent. Unless an

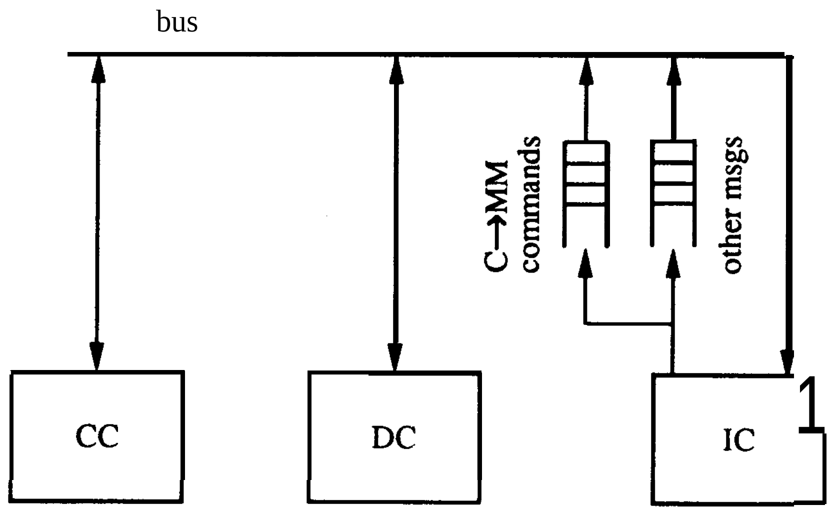

Figure 4.4: Adding an additional queue to prevent reply deadlock.

entire message can be transferred between units on a single clock edge, this deadlock can be prevented only by adding buffer storage between the output of one unit and the input of the other.[2] We can modify the solution we used to prevent reply deadlock in Section 4.3.3 to help solve this problem as well. To do this, we move the additional queue we added for C→MM commands to the input of the directory controller, as shown in Figure 4.5, This configuration allows C→MM commands from the cache controller to the local directory controller to be queued, greatly reducing the probability of deadlock. Of course, deadlock could occur due to replies from the cache controller as well; for this reason, we have also added a queue for replies to the input of the directory controller.

Of course, by adding the queues between the cache and directory controllers, we have reduced the probability of deadlock substantially., but the probability is still non-zero. This is because the cache and directory controllers do not constitute a closed system; either can accept messages arriving from the network through the interconnect controller. In addition, the cache may itself introduce messages due to cache misses. It is possible for these externally generated messages to fill up the buffer storage, leaving the system in essentially the same state as without the buffer.

There are several approaches to dealing with this problem. By adding sufficient

[2]An analogous problem in computer programming is swapping the values of two variables x and y. An intermediate storage variable t must be introduced: $t \leftarrow x; x \leftarrow y; y \leftarrow t$.

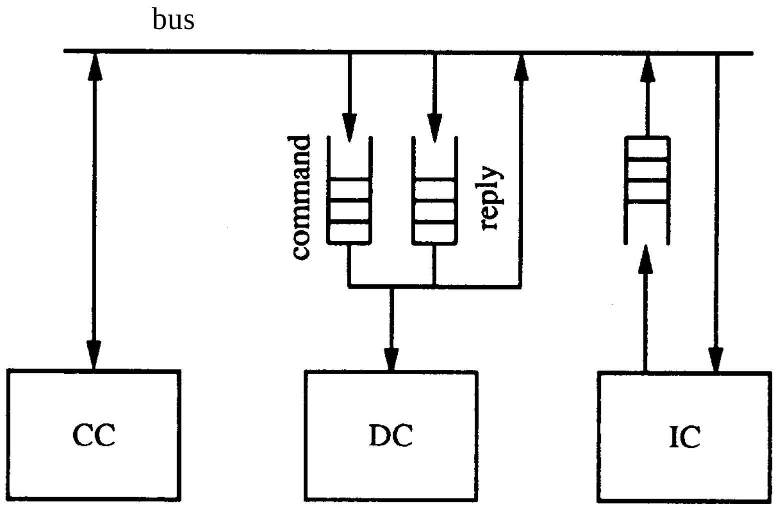

Figure 4.5: Moving queues to the input of the directory controller to help prevent store-and-forward deadlock.

queueing depth, systems in which the directory and cache can each process commands at the maximum rate they can arrive through the network can avoid the deadlock problem. However, directories that serially generate multiple invalidation messages in response to a single write will have trouble meeting this requirement. An alternative is to provide queueing depth equal to the maximum number of messages that can ever be introduced into a node at one time. While tighter bounds will exist for many systems, an upper limit on the number of messages is the product of the maximum number of outstanding memory requests per processor and the number of CPUs. The final avenue we propose is to detect when deadlock *might* occur in the near future; when this condition is detected, the node would not accept any additional messages from the local cache or network. The detection is accomplished by monitoring the directory input queues. When a queue becomes nearly full, the cache is inhibited from making further requests and arriving network messages destined for the queue are NAKed until free space in the queue becomes more plentiful. While this obviously may reduce the performance of the machine by causing message retries and possibly stalling the local processor, this situation should only occur rarely.

We have focused on the interaction between the cache and directory controllers as a source of store-and-forward deadlock. Interaction with the interconnect controller may or may not contribute to this problem, depending on how the interconnect controller is integrated into the inter-node network. In order to prevent deadlock, the network architecture may guarantee that the interconnect controller can always place a new message on this network in finite time without first being required to accept an arriving message from the network. In this case, store-and-forward deadlock cannot occur within the node due to the interconnect controller, because its ability to consume a message from one of the other two controllers does not depend on its ability to send a message to one of them. Although the cache or directory controller may have to block while waiting for the interconnect controller to accept a message, the wait will eventually end since the interconnect controller will ultimately place all other pending messages on the network.

4.5 Protocol Implementation Details

The cache coherency protocol used in our basic design is defined by the state transition table for the directory controller (see Table 4.1). However, there are a number of problems not addressed by that protocol that must be resolved to ensure proper operation. These problems are due to transit delays between caches and directories and the distributed nature of the coherency state in the system. One consequence of the delays is that data values written by a processor are not instantly available to other processors; this ideal model of parallel execution must therefore be replaced with a policy that defines the correct operation of the machine for both architect and programmer. Another difficulty is that the actual cache state may become temporarily inconsistent with the state indicated by the directory, which can result in incorrect operation if care is not taken. In this section we describe how each of these issues impacts the coherency protocol. Afterwards, we will return to the basic design and update the state transition table to reflect the additions to the protocol.

4.5.1 Model of Parallel Execution

In any multiprocessor with cacheable shared data, an effort must be made to provide the programmer with a reasonable model of parallel execution and cache consistency. Such a model allows the programmer to know when a value written by one processor is guaranteed to be available to other processors accessing the same address. For instance, some models specify that a processor may issue no further references until its writes have been seen by all of the other CPUs. Others make guarantees only at synchronization points. The basic protocol described in Section 4.1 guarantees only that all written values are *eventually* accessible by all processors, making it difficult to program many algorithms. In this section, we describe the protocol support necessary to provide a more usable model.

Several possible execution models have been proposed [42, 21], such as strong ordering, weak ordering, release consistency, etc. Though a full treatment of this topic is beyond the scope of this thesis, the semantic distinctions between the models cause hardware differences at the processors, not in the memory system. For all models, the memory system must (1) ensure that no processor may access a newly-written value (except the writer) until all processors may access it, and (2) provide each processor with the information necessary to determine at any given time whether all of *the invalidate* commands sent due to the previous writes by the processor have completed in the target caches. This allows the processor to implement a *fence* operation, which is performed by the processor at various times, depending on the model of execution. A fence operation prevents the CPU from issuing new references until its associated invalidations are known to have completed. After describing the mechanism for informing processors that their invalidations have completed, we will show how it may be used to implement the fence operation for several sample models of execution.

In our approach, the caches notify the directory as they complete *invalidate* commands they have received; once the directory receives all notifications for a given write, it informs the processor that performed the write. To do this, we add two new message types to those we described in Section 4.1.2. To indicate that *an invalidate* command has completed, we require caches to reply with *an invalidate acknowledge* to the directory. Once all the *invalidate acknowledge* replies have been received, the directory sends *an invalidates done* command to the cache that issued the corresponding *exclusive* or *read/exclusive* command. This separate *invalidates done* command is useful because some models of execution *allow* the directory to send a reply such as *return data* before the invalidations have completed, yielding higher performance. Other incoming requests for the data must block at the directory until all *invalidate acknowledge* replies have been received; this guarantees that no processor may access the block until all the stale cached data has been purged.

Although we could send an *invalidates done* command back to a cache for every *read/exclusive* or *exclusive* command it issues, we can improve performance by recognizing that *invalidates done* commands are not needed in several common situations. If a cache issues a *read/exclusive* command and the block in question is dirty in another cache, then the reply cannot be sent until the block has been written back anyway. At other times, the block may not exist in any other caches, requiring no invalidations. For these situations, we do not want to send *invalidates done* commands that tie up the network and the target caches needlessly. However, a cache cannot know *a priori* whether or not its command will cause invalidations to occur. To solve this problem we can include an extra bit in the encoding of every reply to a C→MM command. This bit indicates one of two situations: *waif,* in which an *invalidates done* command will be forthcoming from the directory, or *nowait*, in which no *invalidates done* command will be sent. The bit is then used at the processor to keep track of whether there are still invalidations outstanding.

An example of an execution model that can be implemented using these commands is sequential consistency [31]. Under this model, a fence operation must occur prior to issuing each memory reference. Since exactly one reference occurs between fence operations, the processor actions needed for implementing this restriction are very straightforward. Whenever a cache receives a reply indicating the *wait* condition, then no more references from the processor on that node can be issued until an *invalidates done* command is received.

Fences can also be used to implement weak ordering [42]. In this case, a fence is not required before each reference, but rather immediately before and after each access to a synchronization variable [21]. Before any references can be issued after a fence, it must be known that no more *invalidates done* commands are expected from directory controllers. A simple way to ascertain this condition is to use a hardware counter at each cache whose value at any time is the number of expected *invalidates done* commands not yet received. This counter is incremented each time the cache receives a reply indicating the ***wait*** condition, and decremented each time ***the*** cache receives ***an invalidates done*** command. To implement a fence, the processor halts further references until the value of the counter is zero.

There is an alternative to first collecting ***invalidate acknowledge*** replies at the directory and then sending ***an invalidates done*** command to the cache. Instead, we could have caches forward their ***invalidate acknowledge*** replies directly to the cache. In this case, the directory sends to the cache the number of invalidations sent, and the cache is responsible for counting the ***invalidate acknowledge*** replies. From a performance standpoint, this scheme has the advantage that the cache will (in most cases) receive earlier notification that the invalidations have completed than if the acknowledgements are first sent to the directory. However, this scheme may complicate the hardware considerably. For instance, to reasonably implement weak ordering, a counter is needed for each block in a cache. This is because a request may arrive at a cache for a dirty block before all invalidations associated with its write have completed. If the block is supplied to another cache, that cache also becomes responsible for keeping track of the outstanding invalidations to maintain proper ordering.[3] Another option is to include only a single counter per cache; however, in practice this prevents caches from servicing external requests for dirty blocks until all outstanding ***invalidate acknowledge*** replies associated with ***all*** of the cache blocks have been received. By first sending ***invalidate acknowledge*** requests to the directory, we avoid these problems since other requests for the block stall at the directory until all invalidations have completed in their target caches.

Collecting ***invalidate acknowledge*** replies also impacts our strategy for implementing multiple threads. Recall from Section 4.2 that rather than save state at the directory, we send the required state along with the $MM \rightarrow C$ commands and have the caches echo the state back in their replies. Unfortunately, in order to send the ***invalidates done*** command at the appropriate time, the directory needs to maintain additional state indicating the number of ***invalidate acknowledge*** replies it has received for each block. This count can be kept only at the directory since ***the invalidate acknowledge*** replies are collected there. The count could be maintained by adding bits to each directory entry, or by keeping a "counter cache," a memory containing the counter value for each address in a

[3]Gharachorloo et al. [21] explain this problem in much greater detail.

corresponding tag store. In both cases, the counter value would be incremented when an ***invalidate*** command is sent, and decremented when an ***invalidate acknowledge*** reply is received.

An alternative to using storage dedicated to keeping this count is to simply maintain the count in some fashion using the directory entries themselves. This is possible since the entry is in ***the blocked*** state while the directory awaits ***invalidate acknowledge*** replies. For most pointer-based directory organizations, a feasible strategy is to represent the count of outstanding ***invalidate acknowledge*** replies by the number of pointers "in use" in the directory entry.[4] When an ***invalidate acknowledge*** reply arrives, the directory marks a previously "in use" pointer "not in use." For instance, in a limited pointers directory, a pointer's previously set valid bit is cleared. For a dynamic pointer allocation directory, the first pointer from the block's list is returned to the free list. When no "in use" pointers remain in ***the*** entry, the directory ***can*** send the ***invalidates alone*** command. An example that demonstrates how this technique may be implemented in a dynamic pointer allocation directory is given in Appendix D.

4.5.2 *Invalidate Before Exclusive Acknowledge*

A situation caused by transit delay that we must take care to handle correctly occurs when a block sits clean in several processors' caches, and two of the processors need to write the block at about the same time. Archibald [7] covers this case in detail for a directory organization with a full valid bit vector per entry; we now tailor his approach to better match organizations based on pointers, such as limited pointers and dynamic pointer allocation. Roth caches proceed by issuing an ***exclusive*** command to the directory in charge of the block. It is clear that only one of the writes should proceed without intervention, that is, the write corresponding to the first ***exclusive*** command to reach the directory. However, the basic design we described in Section 4.1 allows both writes to proceed immediately. This action is incorrect, because if the two writes are to different words in the block, the value written to one of the blocks will be lost when the blocks

[4]Sometimes ***invalidate*** commands are sent to all but one of the caches indicated by the directory entry ***(e.g.,*** in response to ***an exclusive*** command). In these cases, one pointer is immediately marked "not in use" to properly initialize the count of outstanding ***invalidate acknowledge*** replies for the entry.

are written back to main memory. To prevent this, we introduce another new type of reply to the message types described in Section 4.1.2. When a directory receives an ***exclusive*** command, it now returns an ***exclusive acknowledge*** reply; a write to a clean block must not proceed in a cache until its ***exclusive*** command has been acknowledged by the directory.

With these acknowledgements in place, how does the protocol operate when two processors both need to write a block? The first cache to reach the directory with its ***exclusive*** command receives an ***exclusive acknowledge*** reply, and proceeds with its write. Sending ***an exclusive acknowledge*** reply to the second cache is not useful, however, since the cache will have already received an ***invalidate*** command for the block due to the first ***exclusive*** command. The problem is to cheaply (in terms of both time and hardware) discover this condition when the directory processes the second cache's ***exclusive*** command.

One straightforward approach is to first check the dirty bit at the directory. If the dirty bit is set, then it must be the case that the block now resides dirty in a cache other than the one that issued the ***exclusive*** command. The directory can get the block back (with a ***flush*** command) and return the data to the requesting cache with a ***return data*** reply, as if the cache had issued a ***read/exclusive*** instead of an ***exclusive*** command. Unfortunately, if the cache containing the dirty block writes it back to main memory before the directory processes the ***exclusive*** command, then the dirty bit will not be set. If the dirty bit is not set, then the simplest option is to process the ***exclusive*** command as usual by sending an ***exclusive acknowledge*** reply and the appropriate ***invalidate*** commands. If the cache finds it no longer contains ***the*** data when it receives the ***exclusive acknowledge*** reply, it then issues a ***read/exclusive*** command. A better alternative is to use the directory information to determine whether the block has been invalidated from a cache that issued an ***exclusive*** command This requires checking the directory data that indicates which caches contain the block. In a directory organization based on pointers, each pointer must be checked against the identity of the cache issuing the ***exclusive*** command. In a limited pointers directory, this would mean either including a separate comparator for each pointer field or using a single comparator to check the pointers serially. In a dynamic pointer allocation directory, the pointers must be checked serially.

The key to performing these comparisons without significant extra hardware or delay is noticing that the directory must send ***invalidate*** commands serially to each cache contained in the pointers anyway. Furthermore, these pointers already have to be compared against the cache issuing ***the exclusive*** command, since ***an invalidate*** command must not be sent to that cache. So by comparing each pointer with the cache identifier as the ***directory*** sends ***the invalidate*** commands, the directory can determine whether the cache that issued this ***exclusive*** command still has a valid copy of the block.

This technique spawns two obvious strategies for handling ***exclusive*** commands at the directory if the dirty bit is not set. ***The*** first is to send all necessary ***invalidate*** commands, checking the pointers along the way. Once these commands have been sent, the directory sends either ***an exclusive acknowledge*** or ***return data*** reply, depending on the outcome of the pointer comparisons. While this solution is straightforward, it delays the reply to the ***exclusive*** command, to the detriment of the common case in which an ***exclusive acknowledge*** reply is returned. A better alternative is to have the directory return an ***exclusive acknowledge*** immediately, on the assumption the cache probably still contains the data block. If after sending ***invalidate*** commands the directory determines the cache in fact no longer contains the block, the directory then sends a ***return data*** reply, supplying the block from main memory. If a cache receives an ***exclusive acknowledge*** reply for a block marked invalid in the cache tags, it simply discards the reply and waits for the ***return data*** reply to arrive.

There is a similar situation that may occur in a no-broadcast limited pointers (Dir_i NB) or dynamic pointer allocation directory. In these directories, processing a ***read/exclusive*** command may require a pointer to be freed by sending an ***invalidate*** command. However, the cache that ***receives*** the ***invalidate*** command may have already sent an ***exclusive*** command for that address to the directory. Once again, the block is invalidated before the cache receives an ***exclusive acknowledge*** reply. This case can be handled as before. ***The*** directory sends ***an exclusive acknowledge*** immediately, which will be discarded by the cache. When the directory finds while sending ***invalidate*** commands that the cache no longer contains ***the*** data, ***the*** directory sends a ***return data*** reply.

An important point to recognize ***Is*** that our scheme relies on the ***invalidate*** command arriving at the cache before the ***exclusive acknowledge*** reply to that cache's ***exclusive*** request. This condition will be true for systems with interconnection networks that always deliver messages from a given node to another in the same order they were sent, since the ***invalidate*** is always sent before ***the exclusive acknowledge.*** The same condition will also be true for systems that collect ***invalidation acknowledgment*** replies at the directory, since the directory will not process the ***exclusive*** command until all ***invalidate*** commands have been completed. The problem is more difficult for systems that both forward ***invalidation acknowledgement*** replies directly to caches and may deliver messages out of order. If the caches maintain per-block invalidation counters, then the directory can keep the block locked until receiving notice from the cache that all invalidations have completed. Otherwise, for each ***exclusive*** command the directory must compare all directory pointers against the identity of the sender before sending a reply.

Notice also that the situation described in this section can be avoided altogether by eliminating ***exclusive*** commands from the protocol. This is done by requiring caches to issue a ***read/exclusive*** command if a clean block must be written. This also removes the need to compare pointers against the cache identity at the directory, since ***invalidate*** commands are always sent to every cache indicated by the pointers. The primary draw-back is the increase in network traffic due to the fact that a cache block must always be returned. We measured the magnitude of this increase for different system configurations in Section 3.4.

4.53 *Writeback* Before *Copyback* or *Flush*

Transit delays can also result in the directory trying to retrieve a dirty block from a cache that has already replaced the block. Say a cache issues a read command ***(read/exclusive*** or ***read/non-exclusive)*** on a block that is dirty in another cache. At roughly the same time the cache containing the dirty block replaces the block and sends the data back to memory via a ***writeback*** command. Assume the read command reaches the directory first. The directory, which still shows the block dirty in a cache, sends a ***flush*** or a ***copyback*** command to that cache. Of course, the cache no longer contains the data.

There are several possible approaches for solving this problem. One option is to adopt the policy that when a cache initiates a ***writeback*** command, it must not invalidate that line in its cache until it has received an acknowledgement from the directory. By doing this, the cache is capable of supplying the data as necessary until the ***writeback*** command has been processed at the directory. Unfortunately, this scheme has several drawbacks. First, network traffic is increased for the case when a cache initiates a ***writeback*, *since an*** acknowledgement is required. Second, the cache issuing a ***writeback*** cannot proceed with its replacement operation until it has received the acknowledgement from the directory. Since these negative effects impact all ***writebacks*,** we advise against this scheme.

Archibald suggests a strategy in which the directory considers ***the writeback*** command to ***be the*** reply to the *copyback* or *flush* command. To prevent deadlock in a system with a command queue that blocks on commands for addresses with pending replies, this solution must encode ***writeback*** commands as reply messages so that they bypass the blocked command queue (see Section 4.3.2). However, this strategy makes a multiple-threaded directory difficult to implement. This is because the ***writeback*** reply (née command) does not contain the state needed by the directory to respond to the read command. The directory would therefore need to save that state in the directory entry, requiring additional directory storage.

We recommend a scheme that is similar to Archibald's in operation, but easier to incorporate in a multiple-threaded directory. When a cache receives a *flush* or *copyback* command for a block it has already written back, the cache responds with a new type of reply known as *copyback* ***without data.*** In addition, all ***writeback*** commands are encoded as replies, allowing the ***writeback*** to complete in the directory before the *copyback* ***without data*** reply arrives.[5] Then, when the *copyback* ***without** data* reply arrives, the original read request can be satisfied from main memory.

4.5.4 Supporting Out-of-Order Message Delivery

Up to now, we have been assuming in-order message delivery on the interconnection network. That is, for any two processing nodes i and j, messages sent from i to j arrive at j in the same order they were sent from i. While this property is true of many

[5]The *writeback* is guaranteed to arrive before the *copyback without data* reply in networks that deliver messages between two nodes in the same order they were sent. We will examine this scenario again for networks with out-of-order delivery in Section 4.5.4.

networks, it is often not guaranteed by networks with adaptive routing. In this section, we present the changes and additions to the protocol that are necessary to maintain coherency across a network with out-of-order message delivery. Since all transactions for a block are still serialized through the directory, there are only a few additional cases to handle. In some of these cases, requests for data arrive in advance of the data itself. Although these requests could be buffered, we will recommend NAK/retry solutions for simplicity, since messages will frequently arrive in order.

Let us consider the following sequence of transactions. A directory sends a reply to a cache in response to a ***read/exclusive***, ***read/non-exclusive,*** or ***exclusive*** command. For brevity, let us call this message pair the ***greedy*** command and reply. Then, another request for the block arrives at the directory, causing the directory to send a ***copyback, flush***, or ***invalidate*** command to the first cache. We call this the ***spoiler*** command. With out-of-order message delivery, the spoiler command may arrive at the cache in advance of the greedy reply. At the very least, we would like the reference that caused the greedy command to be able to complete before the spoiler command must be carried out, to ensure the processors continue to make forward progress. This means that the spoiler command cannot complete in the cache until the greedy reply has been received and one reference has been satisfied. The easiest course of action for the cache is to NAK the spoiler command, causing the directory to send it again later. A more complicated alternative is to buffer the information necessary to execute and reply to the spoiler command, and not do so until after the greedy reply has been received. This buffering would probably take the form of either a small buffer cache or additional bits on each cache tag. If it is found that spoiler commands do not often arrive before greedy replies, then NAKing all spoiler commands that arrive too early is probably a reasonable approach.

Let us now return to the circumstances we described in Section 4.5.3, in which a read request causes a directory to send a ***copyback*** or ***flush*** command to a cache that no longer has the ***data*** due to a ***writeback*** that is still in transit. Recall that the cache sends a ***copyback without data*** reply, and that ***writebacks*** are encoded as replies so that they complete before the ***copyback without data*** reply arrives. Out-of-order message delivery adds another wrinkle to this situation: the ***copyback without data*** reply may arrive at the directory prior to the ***writeback.*** In this case, the requested data cannot be supplied when the ***copyback without data*** reply arrives, because the data is still in transit. ***The copyback without data*** reply contains the state necessary to respond to the original read request; we do not want to add storage or complexity to buffer this state until ***the writeback arrives.*** However, these circumstances will occur infrequently, so a sensible strategy is to NAK the original read request. Once the ***writeback*** has completed, a re-sent read request will succeed at the directory.

We must also take care in handling the case in which a cache receives an ***invalidate*** command on a block for which it is awaiting an ***exclusive acknowledge*** reply (see Section 4.5.2). Recall our suggestion for handling ***exclusive*** commands at the directory: if the dirty bit is not set, then an ***exclusive acknowledge*** reply is sent immediately. If the directory later determines the cache no longer contains the block, the directory supplies the data in a ***return data*** reply. With out-of-order message delivery, this ***return data*** reply could ***arrive*** prior to the ***exclusive acknowledge.*** Since the ***exclusive acknowledge*** may be delayed for an indefinite period of time, it could conceivably falsely acknowledge a future ***exclusive*** command from the cache. A straightforward way to prevent this false acknowledgement is to not satisfy the processor reference that caused an ***exclusive*** request until ***the*** cache receives the ***exclusive acknowledge*** reply, even if a ***return data*** reply arrives sooner. Of course, if the block is dirty in another cache, then the directory will retrieve it and send a ***return data*** reply; this reply must be encoded differently to indicate to the cache that a separate ***exclusive acknowledge*** reply will not be sent. We rename this reply an ***exclusive acknowledge with data*** reply.

The final situation we must consider involves keeping track of outstanding invalidations in order to enforce a model of parallel execution (see Section 4.5.1). In a typical sequence of events, a cache sends a ***read/exclusive*** or ***exclusive*** command to the directory, which sends its reply, either a ***return data*** or ***exclusive acknowledge.*** Later, when all invalidations have been completed, the directory sends ***an invalidates done*** command to the cache. This command may actually arrive before the reply to the original request. If the cache is keeping track of outstanding ***invalidates done*** commands with a counter, as in the weak ordering example of Section 4.5.1, then the counter will be decremented before being incremented, due to the early arrival of the ***invalidates done*** command. This is dangerous, since the counter may be decremented to zero even though there are still outstanding acknowledgements for a different block. To correct for this, we must modify the operation of the counter as follows. The counter should be incremented whenever ***the*** cache sends a ***read/exclusive*** or ***exclusive*** command. The counter should be decremented whenever the cache receives a ***return data*** or ***exclusive acknowledge*** reply with the ***nowait*** condition, indicating that no ***invalidates*** done command is forthcoming. As before, the counter should also be decremented when the cache receives ***an invalidates done*** command. With this policy, the increment always occurs before the decrement, even if the ***invalidates done*** command ***arrives*** prior to the ***return data*** or ***exclusive acknowledge*** reply.

By reducing congestion, networks with adaptive routing may improve the average latency and throughput over their non-adaptive counterparts. However, this performance advantage is not cost-free, due to the fact that these networks may deliver messages between two nodes out of or&r. As we have seen, out-of-order message delivery sometimes causes requests for data to arrive at a cache or main memory before the data itself has arrived. This leads to either increased hardware cost to buffer these requests or greater protocol inefficiency due to NAKs and retries. Since messages will usually arrive in order in spite of adaptive routing, NAK/retry strategies are probably best from a cost/performance standpoint.

4.6 A Better Node Design

We have spent the lion's share of this chapter suggesting a number of specific improvements to the basic node design from Section 4.1. The purpose of this section is to step back and bring those enhancements together to yield a better design for the node. To do this, we will first briefly summarize the changes we have suggested for multiple threads and the queueing within a node. We then review the additional types of command and reply messages we have added to the basic set described in Section 4.1.2. With these message types in place we can specify the data fields that make up the format of each message, and examine implementation constraints on the order of those fields in the format. Finally, we incorporate the protocol modifications we have suggested into the state transition table we presented in Section 4.1.3.

The multiple-threaded directory we propose blocks commands for addresses with outstanding requests from the directory, but commands for other addresses may proceed. The state required per block to accomplish this is the identity of the node that sent the pending C→MM request, the address of the block, and a count of any ***invalidate*** commands not yet acknowledged. We avoid storing this state in the directory by sending the node identifier and address with MM→C commands, to be echoed by the caches in their replies (see Section 4.2), and by using the pointers themselves in the directory entry to count invalidations (see Section 45.1).

In order to prevent deadlock, we added two queues to the input of the directory controller (recall Figure 4.5). One queue accepts C→MM commands; the directory simply stops processing commands from the queue if the first item is a command for a block with outstanding requests. The other queue accepts replies to MM-C commands. This queue allows replies to bypass blocked commands so the command queue may eventually become unblocked. Furthermore, since all paths from the local cache to the directory go through a queue, the probability of store-and-forward deadlock between the cache and directory controllers is reduced. To then eliminate store-and-forward deadlock, we prevent new commands from entering the node when either queue becomes nearly full.

To remedy correctness problems in the protocol of the basic node design, we added several message types. We introduced several types of replies:

- ***invalidate acknowledge (invack).*** This reply is sent by a cache to acknowledge it has performed a MM-C ***invalidate*** command requested by the directory (see Section 4.5.1).

- ***exclusive acknowledge (exack).*** This reply is sent to acknowledge a C→MM ***exclusive*** command. On receipt, the processor may proceed with its write (see Section 4.5.2).

- ***copyback without data (cbnodata).*** This reply is issued in response to a MM→C ***copyback*** or ***flush*** command if the cache no longer contains the requested data due to replacement (see Section 4.5.3).

- ***exclusive acknowledge with data (exackdata).*** This reply, used only if out-of-order message delivery must be supported, is issued in response to a C→MM ***read/exclusive*** or ***exclusive*** command. The message contains the contents of the data block; on receipt, the processor may proceed with its write (see Section 4.5.4).

We also introduced one new MM→C command:

- ***invalidates done (invsdone).*** This command is sent to a cache to indicate that all ***invalidate*** commands caused by a write in that cache have completed (see Section 4.5.1).

In addition to the new commands and replies, we have included an extra bit in all replies to C→MM commands (see Section 4.5.1). This bit indicates a ***wait*** and ***nowait*** situation. If the bit indicates ***wait,*** then invalidations were necessary and a ***invalidates done*** command will therefore be forthcoming from the directory.

4.6.1 Message Format

Having defined all of the message types, we can now determine the data fields that make up a message and examine the constraints for arranging these fields into a message format. The fields are as follows:

- ***Source node.*** This field indicates the processing node sending the message. The source node information allows the message to be NAKed if necessary.

- ***Destination node.*** This field indicates the processing node to which the message is to be delivered.

- ***Address.*** This field contains the address of the block to which the message applies.

- ***Request type.*** This field indicates the type of command or reply that is being sent. A useful request type encoding would probably use separate bits to indicate whether the message is intended for the cache or directory, whether the message is a command or a reply, and for replies to C→MM commands, whether or not there is a forthcoming ***invalidates done*** command (the ***wait*** or ***nowait*** condition described in Section 4.5.1).

- ***Originating node.*** This field indicates the processing node where the cache miss (or write hit to a clean block) occurred that caused the C→MM command to be sent. In some cases, this field is used only to hold this information for later use by the directory (see Section 4.2).

- ***Data.*** This field contains the data words accompanying the message. Since some message types do not transfer data, this field is not always necessary.

For both inter-node and intra-node transmission of messages, these fields will probably be packed into words that are sized according to the width of the interconnection network paths and the width of the queues. Choosing a message format simply entails packing the fields as efficiently as possible into the words so that a message occupies the minimum possible number of words. However, this packing must be done under certain constraints. For instance, the design of the interconnection network constrains the position of the destination node field. For most networks, it will be necessary to include the destination node in the first word of each message, so the network can route the message appropriately without having to add additional buffering stages. To facilitate intra-node routing, the two bits from the request type indicating a command or reply message intended for the cache or directory should also be included in the first word if possible. These two bits allow the node to determine whether the message must be NAKed to ensure deadlock does not occur (see Section 4.4). The source node field should probably fall next in the message format, since it would be the first field of the return message if the arriving message must indeed be NAKed.

If the node accepts the arriving message, it is desirable to discard the destination node and source node fields before the message is delivered to the cache controller or one of the directory controller input queues. This allows the address field, which should be placed next in the message format, to reside in the first word of the message reaching the directory. Since C→MM commands may stall in a queue if the directory already has an operation pending on the same memory block, placing the address in the first word allows the directory to determine if the command must stall before removing any words of the message from the queue

Another advantage to placing the address in the first word of messages reaching the directory is that the memory read cycle for the directory state and pointer bits can begin as soon as possible. A memory cycle can also begin at the same time on main memory data if the implementation allows the read/write specification to occur later, and if the cycle can be aborted later if it is found that no memory access is necessary. This may be possible since DRAMs first require the row address to be latched (with the $\overline{\text{RAS}}$ signal) before the data read or write cycle occurs; the ease of implementation will partially depend on the interface with the DRAM controller.

4.6.2 State Machine

We can now take the original state transition table for our basic node design (see Table 4.1) and incorporate the improvements we suggested in Section 4.5. The resulting state transition table, which assumes the network delivers messages from one node to another in the same order they were sent, is shown in Table 4.2. We assume the directory is multiple-threaded, ***so*** the ***idle*** and ***blocked*** states are maintained for each block. We have written a simulator with network delays that uses this table to describe the actions of the directory. Using the address traces described in Chapter 2 as stimulus, no protocol errors were discovered.

The same conventions we used in Table 4.1 apply here as well; however, there are several additions. In the dirty/valid (DV) bits input column, we define the conditions ***zero, one,*** and ***many,*** which are true if no pointers, one pointer, and more than one pointer, respectively, are in use in the block's directory entry. We also add another input column that indicates whether the count of outstanding invalidations for the block is equal to one. If the count equals one when ***an invalidate acknowledge*** reply is received, then there are no more outstanding invalidations.

On the output side of the table, the outgoing request type for commands and replies now includes a w or ***nw*** designator that specifies ***the wait*** or ***nowait*** condition (see Section 4.5.1). Besides the abbreviations used in Table 4.1, the column indicating the processor number to which the message should be sent also uses the abbreviation ***vptr,*** which stands for ***validpointer.*** This applies only when the block is using a single pointer,

	inputs:					outputs:					
	state	incoming message	previous message	DV bits	inv count = 1?	outgoing message	P#	DV bits	free ptr	next state	other actions, conditions
1	idle	writeback	x	x	x			zero	*	idle	memory ← data
2	blocked	writeback	x	x	x			zero	•	blocked	manory ← data
3	idle	read/nonex	x	$\overline{\text{dirty}} \wedge \text{fp}$	x	retdata/nw	orig	(add)	orig	idle	supply data
4	idle	read/nonex	x	$\overline{\text{dirty}} \wedge \overline{\text{fp}}$	x	retdata/w	orig				supply data
5						invalidate	rptr	*	•	blocked	set inv count = 1
6	idle	read/nonex	x	ditty	x	copyback	dptr	•	*	blocked	
7	idle	read/ex	x	zero	x	retdata/nw	orig	diny	orig	idle	supply data
8	idle	read/ex	x	one ∨ many	x	retdata/w	orig				supply data
9						invalidate	ptr0				send if ptr0 is valid
0						invalidate	ptr1				send if ptr1 is valid
1						invalidate	ptr2	*	*	blocked	send if ptr2 is valid
2	idle	read/ex	x	dirty	x	Rush	dptr	*	*	blocked	
3	idle,	ex	x	one	x	exack/nw	orig	dirty	*	idle	do only if valid ptr = orig
4						invalidate	vptr				do only if valid ptr ≠ orig
5						retdata/w	orig	*	*	blocked	do only if valid ptr ≠ orig
6	idle	ex	x	many	X	exack/w	orig				
7						invalidate	ptr0				send if ptr0 valid & ≠ orig
8						invalidate	ptr1				send if ptr1 valid & ≠ orig
9						invalidate	ptr2				send if ptr2 valid & ≠ orig
:0						retdata/nw	orig	*	*	blocked	send only if Vi ptri ≠ orig
:1	idle	ex	x	dirty	x	flush	dptr	*	*	blocked	
:2	blocked	invack	x	x	no					blocked	decrement invcount
:3	blocked	invack	read/nonex	x	yes	invsdone	orig	*	orig	idle	
.4	blocked	invack	$\overline{\text{read/nonex}}$	x	yes	invsdone	orig	dirty	orig	idle	
.5	blocked	cbdata	read/nonex	x	x	retdata/nw	orig	(add)	orig	idle	memory ← data, supply data
.6	blocked	cbdata	$\overline{\text{read/nonex}}$	x	x	retdata/nw	orig	dirty	orig	idle	supply data
:7	blocked	cbnodata	read/nonex	x	x	retdata/nw	orig	(add)	orig	idle	supply data
:8	blocked	cbnodata	$\overline{\text{read/nonex}}$	x	x	retdata/nw	orig	dirty	orig	idle	supply data

Table 4.2: Improved state transition table for the datapath controller. An "x" indicates a "don't care" condition, and a "*" means the field should remain unchanged from its previous value. Other abbreviations are described in the text.

and represents the processor number saved in that pointer.

Transition Table Highlights

There are several points of interest in the transition table, including the special cases we discussed in Section 4.5. To correctly support a model of parallel execution (see Section 4.5.1), we return the ***wait*** or ***nowait*** condition with each reply to a C→MM command. In addition, the directory collects ***invalidate acknowledge*** replies (lines 22-24) and sends ***an invalidates*** done command when there are no more outstanding acknowledgements.

Now consider ***the invalidate*** before ***exclusive acknowledge*** case we examined in Section 4.5.2. Line 21 shows the response to an ***exclusive*** command for a block that has already become dirty in another cache: a ***flush*** command is sent to retrieve the dirty block. If the dirty block has already been written back, then lines 13-20 apply. An ***exclusive acknowledge*** reply is sent immediately, and is later followed by a ***return data*** reply if necessary. Note here that we distinguish between the case in which one pointer is valid and more than one pointer is valid. This ***allows us*** to set the ***wait*** or ***nowait*** condition appropriately to match the most common circumstances. If only one pointer is valid, then there will usually be no invalidations, and the ***nowait*** condition is returned. The opposite is true if more than one pointer is valid.

To correctly handle the ***writeback*** before ***copyback*** or ***flush*** situation we described in Section 4.5.3, ***we*** encode ***writeback*** commands as replies and therefore allow them to complete even if the state is blocked (see line 2). We also must handle ***copyback without data*** replies, shown in lines 27-28. By the time a ***copyback without data*** reply arrives, the ***writeback*** will have completed, and the data is supplied from main memory in a ***return data*** reply.

Note that the directory's actions when a reply to a MM-C command arrives depend on whether or not the outstanding C→MM command was a ***read/non-exclusive (see*** lines 23-28). This is because the directory must set the dirty/valid bits and the pointers according to whether the cache requested an exclusive copy of the block. An alternative to checking the outstanding C→MM command is to set the dirty/valid bits and the pointers appropriately earlier when the MM-C command is sent. However, this precludes using the valid bits in the directory entry to count ***invalidate acknowledge*** replies as described in Section 4.5.1, since these acknowledgements arrive after the MM-C command is sent. The information indicating whether the outstanding C→MM command was a ***read/non-exclusive*** command can be easily encoded in each MM→C command as an extra bit that is echoed in each reply.

Out-of-Order Message Delivery

To include support in the **datapath** controller for a network with out-of-order message delivery, we must address each of the cases we discussed in Section 4.5.4. First consider the ***writeback*** before ***copyback*** or ***flush*** scenario: recall that with out-of-order message

	inputs:					outputs:					
					inv						
		incoming	previous		count	outgoing		DV	free	next	other actions.
	state	message	message	DV bits	= 0?	message	P#	bits	ptr	state	conditions
27	blocked	cbnodata	read/nonex	dirty	x	retdata/nw	orig	(add)	orig	idle	supply data
28	blocked	cbnodata	$\overline{\text{read/nonex}}$	$\overline{\text{dirty}}$	x	retdata/nw	orig	dirty	orig	idle	supply data
29	blocked	cbnodata	x	dirty	x	NAK	orig	*	*	idle	

Table 4.3: Modifications to the state transition table to support out-of-order message delivery.

delivery, the ***copyback without data*** reply may arrive at the directory before the ***write-back.*** We detect this situation at the directory by checking the dirty bit; if it is set, the ***writeback*** has not arrived and we must NAK the original read request. The corresponding modifications to the state transition table are shown in Table 4.3.

Now consider ***the invalidate*** before ***exclusive acknowledge*** situation. Depending on the circumstances, the directory may reply to an ***exclusive*** command with an ***exclusive acknowledge*** reply, a ***return data*** reply, or an ***exclusive acknowledge*** followed by a ***return data*** reply. As we described in Section 4.5.4, if both replies are sent then ***the return data*** may arrive at the cache first. We must therefore alter the encoding of some ***return data*** replies so that the cache can determine whether or not ***an exclusive acknowledge*** reply is forthcoming. We do this by introducing the ***exclusive acknowledge with data*** reply. We can now think of replies to all read requests as containing permission to write the data ***(exclusive acknowledge),*** the data itself ***(return data),*** or both ***(exclusive acknowledge with data).*** This adds no new entries to the state transition table of Table 4.2; we simply change the ***return data*** message encoding to ***exclusive acknowledge with data*** in lines 7, 8, ***15, 26,*** and 28.

Finally, we must account for the possibility that a ***spoiler*** command may arrive at a cache before a ***greedy*** reply, as discussed in Section 4.5.4. To handle this case, the only requirement is that the sending node be able to m-send spoiler commands NAKed by a cache. The node must be able to re-send NAKed messages anyway to prevent store-and-forward deadlock (see Section 4.4), so this requires no additional hardware.

4.7 Dynamic Pointer Allocation Implementation

The proposed node design we have pursued uses a no-broadcast limited pointers directory. Interestingly, the implementation considerations that are the focus of this chapter are fairly orthogonal to the organization of the directory. To demonstrate this, let us examine the modifications to our design that would be required to implement a dynamic pointer allocation directory.

Recall the diagram of the directory datapath shown in Figure 4.2. The unit labeled ***directory memory*** accepts an address and allows the corresponding pointers to be read or written. Although we assumed a limited pointers directory, this structure fits the dynamic pointer allocation directory as well. The implementation of dynamic pointer allocation we described in Chapter 2 also accepts an address and allows the corresponding pointers to be accessed (see Figure 2.15). We can therefore simply use that implementation as the realization of the unit labeled ***directory memory*** in Figure 4.2. Furthermore, the state transition table we presented in Section 4.6.2 need not be changed for a dynamic pointer allocation directory.[6] This is aided by the fact that both the no-broadcast limited pointers and dynamic pointer allocation directories handle the situation in which no free pointers remain similarly. For both directories, the state transition table indicates the appropriate action: the data is returned to the requesting cache, a pointer is replaced by sending an invalidation, and that pointer is set to identify the new cache. In general, dynamic pointer allocation makes basic operations (e.g., storing the originating node field into a pointer associated with the block) slightly more complicated, but the necessary steps are performed at a lower level. These steps are described using pseudo-code in Appendix D.

4.8 Summary and Conclusion

In this chapter we have examined the principal problems involved in the implementation of directory-based cache coherency and presented solutions to these problems. To provide

[6]The semantics of the ***empty*** and dirty bits associated with each block would be changed to encode the four states indicated by the DV bits in Table 4.2: zero, ***one***, many, and dirty. Also, with dynamic pointer allocation we modified the coherency protocol to include replacement notifications (see Section 2.3.2); handling this new type of message will require another entry in the table.

a framework for demonstrating and discussing these problems, we began by laying out a basic design for a processing node by describing the interconnect within the node, the datapath for the directory, the types of messages required to maintain coherency, and the state machine used by the directory to respond to those messages.

We found that multiple threads can be easily added to the directory by encapsulating the necessary per-block state into the messages sent by the directory and the subsequent replies from the caches. We also identified a sufficient condition for preventing deadlock. To satisfy this condition, our experience is that the protocol must be designed in concert with the message delivery mechanisms. In particular, the strategy for queueing and transferring messages within a processing node must be carefully planned.

We then looked at protocol implementation details, refining previously proposed schemes for handling subtle situations in order to make the schemes more amenable to a directory based on pointers. We also described the protocol support needed to implement a consistent model of parallel execution, and to tolerate out-of-order message delivery by the interconnection network. Each of these protocol improvements manifests itself in our state transition table for the directory controller.

The overall results of our efforts are encouraging. Although devising a protocol that correctly handles all uncommon as well as common situations is not a simple task, it is certainly tractable. And just as importantly, the final protocol specification is straightforward to implement. Only a few message types are needed, resulting in a simple datapath for the directory that requires only a few registers, drivers, and a comparator in addition to the directory storage itself. Furthermore, the state machine controlling the flow of data is fairly small. Our conclusion is that careful protocol design allows cache coherency to be efficiently implemented at a hardware cost that is dominated by the cost of the directory storage itself and the pointer management mechanism (which can be significant for some organizations, such as dynamic pointer allocation).

Chapter 5

Conclusion

We have presented and evaluated many of the trade-offs that must be made in the design of a directory-based cache coherence mechanism for a large-scale, shared memory multiprocessor. In general, the primary challenge is achieving simplicity without sacrificing performance or correctness. This chapter summarizes our findings about meeting this goal in organizing the directory information, choosing a coherency protocol, and implementing the directory and protocol. We also consider several areas of future research.

5.1 Directory Organization

We began at the highest level of the design by considering the organization of the information in the directory. The traditional approach that maintains a full valid bit vector per directory entry is unsuitable for large-scale machines due to high storage overhead. We have proposed several alternate organizations. First, ***limited pointers*** directories replace the full valid bit vector with several ***pointers*** indicating those caches containing the data. Analytic modeling demonstrates that this scheme performs well across a wide range of workloads. However, unlike the full valid bit vector, its performance does not improve as the read/write ratio becomes very large. In some programs, this performance gap can be significant, often due to a few memory blocks that are read frequently but rarely written. To address this drawback, we have proposed a ***dynamic pointer allocation*** directory as an alternative. This directory allocates pointers from a pool to particular memory blocks as they are needed. This scheme is limited only by the number of pointers on a memory module; since the pointers may be allocated to any block on the module, the probability of running short is much smaller than for the limited pointers directory.

Others have also proposed alternative directory organizations that perform better than the limited pointers strategy for blocks with high read/write ratios. While each of these schemes has its own set of merits and drawbacks, it is apparent from both our work and others' that directories can be built that provide nearly the performance of a full valid bit vector scheme without its prohibitive storage cost. Among this set of directory alternatives, the dynamic pointer allocation directory lies at a particularly attractive cost/performance point. The circumstances under which its performance degrades should rarely occur, the maximum performance degradation is modest, and its implementation is straightforward and inexpensive. Dynamic pointer allocation is therefore an appealing solution to the directory organization problem.

5.2 Coherency Protocol Design Options

Following our examination of directory organization, we evaluated the performance impact of several coherency protocol features. In general, protocol enhancements often improve performance only marginally. This was true for adding a clean/exclusive state for reducing the time required to write a clean block, and also for using request forwarding to transfer a dirty block directly to another cache that has requested it. Furthermore, for small cache block sizes, write hits to clean blocks can be simply treated as write misses without incurring significant extra network traffic. Our conclusion is that protocol features designed to improve performance must be examined carefully, for they often complicate the protocol significantly without offering substantial benefit.

5.3 Directory and Protocol Implementation

We finished by demonstrating the issues that must be addressed in building a directory-based coherency mechanism. To do this, we first presented a basic design for a processing node, including the necessary hardware elements of the directory datapath. This design, which was incorrect, served to illustrate implementation problems that must be solved. For instance, deadlock may occur due to blocking while awaiting a reply message. To address this problem, we identified a sufficient condition for avoiding this type of deadlock in the design of the caches, directories, and network. In addition, we suggested several methods for preventing deadlock caused by each of two controllers within a processing node running short of storage to buffer a message from the other.

In addition to deadlock, there are a number of correctness issues that arise when implementing the coherency protocol. These difficulties are fundamentally caused by the fact that directory state may be temporarily inconsistent with cache state because of transit delays between caches and directories. This complexity is compounded if the interconnection network does not guarantee that messages from one node to another are delivered in the same order they were sent. We solved each of the resulting correctness problems by modifying the coherency protocol appropriately. Although the problem cases are typically subtle and hard to discover, the required protocol alterations are straightforward and only slightly increase the hardware cost of the directory controller.

Since blocking the directory could degrade performance, it is desirable for the directory to be multiple-threaded, that is, to allow multiple outstanding requests at the same time. By encapsulating the state of outstanding requests in the messages themselves, multiple-threading can be achieved without significantly increasing the directory storage required to save the state of outstanding requests. Finally, we combined this technique with the solutions to deadlock and the other correctness problems, and presented the resulting state transition table for the directory datapath controller. The compact size of this table illustrates the overall simplicity of the directory mechanism.

By using the techniques described herein, hardware-based cache coherence can be added to large-scale multiprocessors in an inexpensive yet effective manner. During the development of our ideas we found that for any given solution to some problem, we could almost always think of a more complicated solution that might yield slightly higher performance. Though potentially higher performance is always tempting, even seemingly low-cost enhancements should be evaluated carefully. Though the incremental hardware cost may be low, the additional design complexity can be significant. As we have demonstrated, even very simple protocols are difficult to implement correctly in the presence of non-zero message latencies and message buffering in the memory system. Our experience suggests that many (perhaps even most) design enhancements add at least one unanticipated sequence of events that results in a subtle error in the initial specification of the coherency protocol. Since the complexity of a protocol directly affects the ease of proving or verifying its correctness, the performance improvement of a given enhancement must be large enough to justify not only the extra hardware cost, but also the additional design time and grief.

5.4 Future Work

This thesis examines many aspects of the design of cache coherency for large-scale multiprocessors. However, the design space is large, and we have made several important assumptions in order to limit the scope of our consideration. A natural extension of this work would be to see if the performance or cost characteristics of the design may be improved by relaxing or eliminating some of the assumptions we have made.

We have assumed that cache coherency is maintained by the machine hardware. For programs written with more restrictive programming models than those assumed by this thesis, researchers have recently made substantial progress in defining compiler-based and compiler-aided techniques for maintaining cache coherence [17, 15, 1]. Further refinement may expand the set of programs to which these software algorithms can be fruitfully applied. If shared data can be partitioned into distinct classes of sharing behavior (as several papers have proposed [53, 10]) at compile-time, then the communication required to maintain coherence can be optimized for minimum latency and traffic on a per-class basis. Nonetheless software-based coherency remains a difficult problem, since compilers must often be conservative due to the lack of dynamic flow information. But even if these schemes cannot replace hardware-based coherency for general-purpose workloads, they may still be useful to optimize the performance of particular data items with reference patterns that are readily determined by the compiler. This idea is appealing since the increase in performance is not accompanied by an increase in hardware cost or complexity

We have also assumed that the coherency protocol operates by invalidating, rather than updating, stale data that is cached. Although a pure update-based protocol would generate far too much network traffic, hybrid update/invalidate protocols may be feasible. These are often called ***adaptive*** protocols, because they adapt to the current reference behavior to a block by switching "on-the-fly" between invalidation and update coherence strategies. While most of the work to date has been targeted at snooping-based schemes [8, 29], similar techniques can be applied to directory-based coherence as well. For instance, caches could send updates to the directory for forwarding to the other caches containing the block. To limit unnecessary traffic, caches not actively using the block could invalidate the block and notify the directory to no longer send the updates. Each cache could detect whether it is finding the updates useful by maintaining a bit for each block that indicates whether its processor has accessed the block since the last update. These protocols that selectively update cached data have the potential to increase the amount of sharing that can be well-supported by a system, since the increase in invalidation misses with greater sharing is curbed. However, the amount of traffic may be substantially increased, for both high and low levels of sharing. This would tend to offset the performance advantage unless a more costly, higher bandwidth network is provided. The hybrid strategy will also add substantial complexity to the protocol design, so the performance improvement must be sizable if the strategy is to be worthwhile.

It is possible to take adaptive protocols a step further by using different protocols for different classes of data, depending on the usual access patterns exhibited by references to each class [53, 10]. While this can be done to a limited extent in hardware (e.g., not maintaining consistency for instructions, private data, and write-once data), much of this work is more applicable to software-based coherency approaches.

Another assumption we have made is that the directory maintains coherence across a number of single-processor nodes. An alternative is to use a two-level (or more generally, multi-level) coherency protocol that groups processors into ***clusters*** at the lowest level.' A cluster is defined here as a small number of nearby processors. In this scheme, one strategy (e.g., snooping caches) is used for maintaining coherence within a cluster, while another (e.g., directories) is used across clusters. One advantage of two-level protocols

'These levels may also be reflected in the network topology (as in DASH [33], for instance), but this is not a requirement.

is that directory resources may be shared across the processors of a cluster, potentially reducing hardware cost if the intra-cluster coherence mechanism is relatively inexpensive. In addition, clustering allows distant memory blocks to be fetched quicker if they are already cached by another processor in the same cluster. A drawback of two-level protocols is that they increase the design complexity of the machine, because two protocols and their interactions must now be managed. Also, reducing the latency of misses to blocks cached in the cluster may not significantly improve the overall performance of a parallel program if its algorithm does not map naturally onto the multi-level topology defined by the protocol.

One of the primary results of this thesis is that the design complexity of simple directory-based coherency protocols is significant yet still manageable. Before deciding to enhance the protocol (e.g., by using adaptive or multi-level schemes), it is imperative to carefully weigh the benefits against the potential to greatly complicate the design.

Appendix A

Performance Model

In Chapter 1 we presented results from simple performance models in order to demonstrate the usefulness of caching shared data (see Figures 1.1 and 1.2). The results are also useful in providing perspective as to the granularity of sharing that we can expect large-scale machines to support well under the best and worst of application sharing patterns (i.e., nearest-neighbor sharing and randomly distributed sharing, respectively). This appendix describes the assumptions behind the models and presents the equations that make up the models.

A.1 Machine Assumptions

Although disguised under the more pretentious appellation of "models," our performance calculations are actually closer in spirit to "back-of-the-envelope" scribblings, designed not to exactly represent a particular system in minute detail, but rather to grossly and easily yield an idea of the performance we can expect from large-scale machines. Here we state our simplifying assumptions about the machine so that the results can be properly interpreted.

The machine we are studying is made up of n processing nodes, each with a single processor, its associated cache, and l/n-th of the globally-shared main memory. The nodes are interconnected by a network with a conventional two-dimensional mesh topology; the boundary links of the mesh "wrap-around," thereby forming a torus. Each network link allows bidirectional transmission to allow each node to rapidly communicate with any of its neighbors. We do not consider the effects of non-uniform delivery times in the network due to contention; each message incurs a fixed per-message delay d_{msg} plus a per-link delay d_{link} for each link it must traverse to reach its destination.

We consider simple RISC-style processors capable of issuing up to one instruction per cycle, and having no more than one data reference outstanding at a time. The CPU/cache configuration is such that an instruction reference and a data reference can be satisfied in the same cycle. We also make the simplifying assumption that the CPU is never stalled due to an invalidation occurring in its cache. The caches exhibit a miss ratio of m_{pvt} for both instructions and data, not including misses due to data sharing. There are f_{data} data references per instruction.

For simplicity, all sharing occurs between pairs of processors.[1] Under this assumption, directory organization is not an issue; we presume the directory maintains full information about the locations of cached data. The coherency protocol is the base Censier/Feautrier protocol described in Section 3.1. There are no write buffers, so processors must stall on a write hit to a clean, shared block. This is not the case for private data, which we assume is not coherently cached. Weak ordering allows the directory to send acknowledgements to processors (allowing them to continue) before sending invalidations. Once a memory request has reached the node where the target memory is located, the memory access itself incurs delay d_{mem}.

The workloads we examine exhibit two extremes of geographic locality. First, to maximize geographic locality, ***we*** look at ***nearest-neighbor*** sharing. In this case, we assume that sharing occurs only between adjacent processors in the network, and furthermore, that the shared data is located on the same processing node as one of the CPUs. With nearest-neighbor sharing, the processor utilization is independent of the number of processors since memory latency remains constant as the machine is scaled. The second workload behavior ***we*** examine is ***random sharing,*** which exhibits no geographic locality. In this case, sharing occurs between randomly selected pairs of processors for data located in main memory at a third randomly selected node. Since memory latency between two randomly chosen nodes depends on the size of the system, processor utilization decreases

[1]This is also the most common case observed in our benchmark applications.

as n is increased.

For the graphs in Chapter 1 (i.e., Figures 1.1 and 1.2), we assume the following machine parameters: d_{msg} = 10 cycles, d_{link} = 2 cycles, d_{mem} = 20 cycles, f_{data} = 0.33, and m_{pvt} = 0.02.

A.2 Utilization

Both the models for cached and uncached shared data present per-processor utilization as a fraction of uniprocessor utilization. By this metric, a result of 1.0 would indicate that all CPUs are running as fast as a uniprocessor constructed of a single processing node. In this section we describe how the uniprocessor utilization is calculated; following sections derive multiprocessor per-CPU utilization.

Each instruction requires one cycle plus some number of extra cycles due to cache misses on instructions and data. The processor utilization is therefore

$$u = \frac{1}{1 + i_e + d_e} \tag{A.1}$$

where i_e and d_e represent these extra cycles. The extra number of cycles due to instruction misses is

$$i_e = m_{pvt} d_{mem} \tag{A.2}$$

since we assume instruction misses are satisfied by each node's local memory. For a uniprocessor, the extra number of cycles due to data misses is:

$$d_e = f_{data} m_{pvt} d_{mem} \tag{A.3}$$

We are now in a position to compute uniprocessor utilization. To compute multiprocessor utilization, we must revise our expression for d_e to account for additional cycles required to satisfy shared data misses. In the next two sections we derive d_e for a systems with and without caching of shared data.

A.3 Model for Uncached Shared Data

If shared data is not cached, then all shared references must be satisfied by main memory. Let us call f_{sh} the fraction of data references that are to shared data, and l the average latency to satisfy such a reference. We can now modify the uniprocessor expression for d_e (A.3) as follows:

$$d_e = f_{data}[(1 - f_{sh})m_{pvt}d_{mem} + f_{sh}l] \qquad (A\text{-}4)$$

The expression for l depends on the degree of geographic locality in the workload. For nearest-neighbor sharing, half of the references to the block in main memory are from the CPU on the same node on average, while the other half are from an adjacent processor. A given reference is therefore satisfied with latency

$$l = \frac{1}{2}d_{mem} + \frac{1}{2}(d_{mem} + 2d_{msg} + 2d_{link}) \qquad (A.5)$$

For random sharing, to first order all references are to main memory located on a different node. On average, $\sqrt{n}/2$ links must be traversed to reach that node.[2] Shared reference latency is therefore

$$l = d_{mem} + 2d_{msg} + \sqrt{n}d_{link} \qquad (A.6)$$

The multiprocessor utilization for uncached shared data is obtained by eliminating l, d,, and i_e by substitution from the system of equations formed by (A.l), (A.2), (A.4), and either (AS) or (A.6).

A.4 Model for Cached Shared Data

In the previous section we derived a model for uncached shared data that depends on f_{sh}, the fraction of data references that are to shared data. This makes sense if shared data is not cached, because every shared reference impacts performance negatively. However, if shared data is cached, then f_{sh} is not sufficient to characterize the communication activity due to sharing, since only a fraction of shared references require main memory access. We therefore introduce a new parameter m_{inv} that specifies the fraction of all data references that are invalidation misses. This parameter is an attractive measure of the amount of sharing in a workload, since invalidation misses occur on precisely those references that require inter-processor communication to take place if the most recently

[2]This assumes that $\sqrt{n}$ is an even number, which is true for the processor sizes we have evaluated in our graphs.

written value of the data is to be returned. Notice that our results now depend on both m_{inv} and f_{sh}. On the graph in Figure 1.2, we have shown m_{inv} on the horizontal axis and drawn each curve twice, once with $f_{sh} = 0.2$ and once with $f_{sh} = 0.8$, since f_{sh} will fall in this range for most programs.[3]

As before, we need to rewrite equation (A.3), which describes the number of extra cycles required to service data misses. We assume a migratory pattern of sharing between each pair of processors sharing a piece of data. In this pattern, each processor reads the data and then performs some number of references, including at least one write, to the data before the other sharing processor begins a similar sequence of references. Note that this results in exactly one invalidation when a processor writes the data the first time, which is the most common case observed in our benchmark applications. So one processor experiences a read miss to a block dirty in another CPU's cache and a write hit to a clean block. The other processor then undergoes the same experience, and the cycle is repeated. The significance is twofold: (1) each invalidation miss requires the directory to retrieve the block from another cache, and (2) following each invalidation miss there is a write hit to a clean block. This allows us to write an expression for ***d*,:**

$$d_e = f_{data}[(1 - f_{sh})m_{pvt}d_{mem} + f_{sh}m_{pvt}l + m_{inv}(2l) + m_{inv}l] \qquad (A.7)$$

where the latency l, representing a single round trip from a cache to main memory, is as given by either (A.5) or (A.6), depending on whether the sharing is nearest-neighbor or random. In (A.7), the first two terms within the square brackets represent the interference misses for private and shared data, while the last two terms correspond to invalidation misses and write hits to clean blocks. As in Section A.3, the multiprocessor utilization is derived by substitution.

[3]Across the set of programs, r and f_{sh} will be statistically correlated, since programs with a higher fraction of shared references will tend to incur invalidation misses more frequently. In Figure 1.2, high values of f_{sh} cause negative effects if invalidation misses are infrequent; however, few programs will exhibit both of these traits.

Appendix B

Limited Pointers Model

In this appendix we present the details of the limited pointers model used in Chapter 2. Recall from Section 2.2.1 that there are m processors that may access a block q, and that a sequence of references proceeds as shown in Figure 2.3. For all $1 \leq i \leq m$ we can calculate the probability that exactly i processors access q during a sequence, thereby occupying i pointers in the corresponding directory entry. Let us call this probability f_i. Note that this is a different value than the probability that q was accessed i times, since some processors could have accessed it more than once. In order to calculate f_i, we define some probabilities of interest:[1]

t_i = P(at least i processors read the data q at least once)
n_i = P(some processor that has not yet accessed the data q will eventually access q, given that i processors have already accessed q)
s_i = P(some processor that has not yet accessed the data q is eventually selected, given that i processors have already accessed q)
p_i = P(in a given selection, a processor is selected that has not yet accessed the data q, given that i processors have already accessed q)
r_n = P(a processor accessing the data q for the first time issues a read)
r_o = P(a processor accessing the data q, but not for the first time, issues a read)

[1] The notation $P(A)$ is used to indicate the probability of event A.

g_{n_i} = P(a selected processor that has not yet accessed the data q accesses q, given that i processors have already accessed q)

c_i = P(a selected processor that has accessed the data q in the past does not terminate the reference sequence, given that i processors have already accessed q)

g_{o_i} = P(a selected processor that has accessed the data q in the past accesses q, given that i processors have already accessed q)

Note that the subscripts n and o in the above events refer to new and old processors.

We now point out the important relationships between these probabilities. First, remember that we are seeking an expression for f_i:

$$f_i = t_i - t_{i+1} \tag{B.1}$$

In other words, the probability that exactly i processors read the data q before it is written is equal to the probability that at feast i processors read q minus the probability that ***at feast*** i + 1 processors read q. This is readily apparent, for once i processors have read q, then it is either the case that at least i + 1 processors will read q or exactly i processors read q before it is written (i.e., the $i + 1^{st}$ processor writes q).

We can also relate t_{i+1} to t_i. At least i + 1 processors access the data q without ending the sequence if at least i processors have done so, a new processor will eventually access q, and that access happens to be a read:[2]

$$t_{i+1} = t_i n_i r_n \tag{B.2}$$

We make the assumption that a reference sequence begins in a state in which one processor has already accessed q.[3] This reflects the fact that the write ending the previous sequence of references to q leaves the data cached in a single processor, that is, one pointer is already in use when a reference sequence begins. This assumption implies that t_1 = 1.

[2]The simple product is valid since the factors are the probabilities of independent events. This is also true for each of the terms in the upcoming equations (B.3) through (B.5). Also, equations (B.3) through (B.5) are simple sums-of-products; the sums are valid because the product terms are the probabilities of mutually exclusive events.

[3]A previous paper [44] describing this work did not make this assumption. The results in that paper assume a reference sequence begins in a state in which no processors have accessed q.

We can write an expression for n_i by observing that a new processor will eventually access the data q if a new processor is eventually selected and accesses q, or if a new processor is eventually selected but does not access q but another is later selected and accesses q, or if . . . , etc. This results in a simple series:

$$n_i = s_i g_{n_i} + s_i(1-g_{n_i})s_i g_{n_i} + s_i(1-g_{n_i})s_i(1-g_{n_i})s_i g_{n_i} + \ldots \tag{B.3}$$

In a similar vein, a new processor is eventually selected if a new processor is selected now, or if an old processor is selected now but does not end the sequence and a new processor is then selected, or if . . . , etc.:

$$s_i = p_i + (1-p_i)c_i p_i + (1-p_i)c_i(1-p_i)c_i p_i + \cdots \tag{03.4}$$

Finally, an old processor that has been selected does not end the sequence if it either does not access the data q or if it does access q and that access is a read:

$$c_i = (1-g_{o_i}) + g_{o_i} r_o \tag{B.5}$$

Equations (B.3) and (B.4) can be written in a simpler form as follows:

$$\begin{aligned} n_i &= \sum_{j=0}^{\infty} s_i g_{n_i}[s_i(1-g_{n_i})]^j \\ &= \frac{s_i g_{n_i}}{1-s_i(1-g_{n_i})} \end{aligned} \tag{B.6}$$

$$\begin{aligned} s_i &= \sum_{j=0}^{\infty} p_i[c_i(1-p_i)]^j \\ &= \frac{p_i}{1-c_i(1-p_i)} \end{aligned} \tag{B.7}$$

Since processors are selected at random, the probability of selecting a new processor if i of them have already accessed the data q is given by

$$p_i = \frac{m-i}{m} \tag{B.8}$$

We can now reduce the system of equations given by (B.1), (B.2), (B.5), (B.6), (B.7), and (B.8) by eliminating the variables c_i, p_i, s_i, n_i, and t_i, leaving us with the following

The model may be extended to calculate the number of extra invalidations incurred by the no-broadcast (Dir_i NB) directory. Although we do not show the details here, the basic idea is to add an input parameter that specifies the number of pointers per entry. The previous model indicates the frequency with which those pointers are not sufficient and we must begin invalidating data. For these circumstances, we construct a Markov chain whose states can be represented by a two-dimensional matrix. One dimension represents the number of extra invalidations that have occurred up to this point, and the other represents the number of caches that have received invalidations. It is then a straightforward matter to compute the state transition probabilities and therefore calculate the average number of extra invalidations per write.

there were no old processors. Since the processors are selected at random, the probability the first access to q will be from a secondary processor is

$$P(\text{first old CPU is a secondary CPU}) = \frac{(m-1)d}{b+(m-1)d}, \qquad \text{(B.10)}$$

since the single primary processor accesses q with probability b, and the $m-1$ secondary processors access q with probability d. The expression for the probability that the next processor making the new-to-old transition is also a secondary processor is similar: each occurrence of $m-1$ is replaced by $m-2$, since there is one fewer new secondary processor to choose from. So we can now write the expression for the probability of case 2:

$$P(\text{case } 2) = \prod_{j=1}^{i} \frac{(m-j)d}{b+(m-j)d}, \qquad \text{(B.11)}$$

where i is the number of old processors. This gives us the probability of case 1 as well, because $P(\text{case } 1) + P(\text{case } 2) = 1$. We are now in position to write expressions for g_{n_i} and g_{o_i}. First consider g_{n_i}, the probability with which a selected new processor accesses the data block q. If case 1 is true, then all new processors are secondary processors, and the selected one accesses q with probability d. If case 2 is true, then of the $m-i$ new processors, one is the primary CPU and the other $m-i-1$ are secondary CPUs. Therefore, we can write the expression for g_{n_i} as follows:

$$g_{n_i} = P(\text{case } 1)d + P(\text{case } 2)\left[\frac{m-i-1}{m-i}d + \frac{1}{m-i}b\right] \qquad \text{(B.12)}$$

Furthermore, we can easily derive g_{o_i} using a similar analysis, yielding

$$g_{o_i} = P(\text{case } 1)\left[\frac{1}{i}b + \frac{i-1}{i}d\right] + P(\text{case } 2)d \qquad \text{(B.13)}$$

Substituting (B.12) and (B.13) into (B.9) completes the model for the probability distribution of the number of pointers used. Not surprisingly, the resulting expression for f_i depends only on the ratio of the primary and secondary processor access probabilities b and d, so we replace them with a single parameter $a = b/d$. The model now takes four parameters as inputs, each with respect to the single block of data being modeled: the number of processors m accessing the data, the read probabilities r_n and r_o, and the ratio of the primary and secondary processor access probabilities a.

requests per cycle with reasonable latency.[1] To avoid exceeding this rate, the following constraint must be satisfied:

$$nrfm \leq q \tag{C.1}$$

Now let us introduce a dynamic pointer allocation directory at the memory module. The resulting miss ratio m_{dpa} is as follows:

$$\begin{aligned} m_{dpa} &= (1-f)m + f(m + m_{extra}) \\ &= m + f m_{extra} \end{aligned} \tag{C.2}$$

where m_{extra} is the extra misses that occur because the directory must invalidate blocks when it runs short of pointers. To calculate m_{extra}, we need to describe the directory. Assume the memory module has p times as many pointers as there are blocks in a processor cache. For instance, p = 4 for the 4:1 ratio of pointers to cache lines we calculated in Section 2.3.3.

The key to calculating m_{extra} is recognizing that the pointer/link store behaves as a cache, in that it contains information about recently accessed blocks. The analogous cache is fully associative with random replacement, since information pertaining to any address from the module may reside in any pointer/link pair, and pointer/link pairs are chosen at random for replacement when the available pointers are exhausted. The value m_{extra} is simply the miss ratio of this cache.[2] We can compute m_{extra} from m by relating the size and associativity of the processor cache and the pointer/link store. If the processor cache size is C bytes, then the pointer/link store will behave like a fully associative cache of size pC/n; the overall pointer/link store size pC must be divided by n since it is shared by n processors. We also assume the processor cache is direct-mapped, given the trend towards large, direct-mapped caches.

Many studies [46, 52, 2,401 have confirmed Chow's speculation that cache miss ratio is related to cache size by the expression m = kC^a [16], where k and a are constants for a given cache organization. The miss ratios m_1 and m_2 of any two caches of respective

[1] In other words, q is not the maximum available bandwidth of the memory module, but rather takes into account the fact that the module should not be 100% utilized.

[2] This is conservative since this assumes all "misses" in the directory would have otherwise hit in the processor cache, which is not the case.

Appendix C

Dynamic Pointer Allocation Model

In this appendix we calculate the expected increase in miss rate due to dynamic pointer allocation, under the assumption that a workload is running that exhausts the available pointers on a memory module. Obviously, if the module does not run short of pointers, then no miss rate increase is incurred. We limit our coverage of the model parameter space to those workloads that would run well with an unlimited number of pointers. Specifically, we exclude workloads with memory bandwidth demands that exceed the available bandwidth of the memory module, since these workloads will run poorly in any case.

Let us first characterize the workload. We consider a single memory module that contains data accessed by n processors. Each of these processors issues references at the rate of r references per cycle. We also assume that each accesses data from the module on a given reference with probability f. It is easy to see that this uniform distribution of references across the processors is the worst case, since this will cause the maximum number of pointers to be in use. Finally, let us state that all references miss in the processor's cache with probability m, assuming an infinite number of pointers were available. This parameter corresponds to the miss ratio. Using these parameters, requests arrive at the memory module at the rate $nrfm$.

By specifying the bandwidth of the memory module, we can now constrain the parameter space to include only those workloads that do not overwhelm the available bandwidth. Let us say the memory module is capable of satisfying requests at an average rate of q

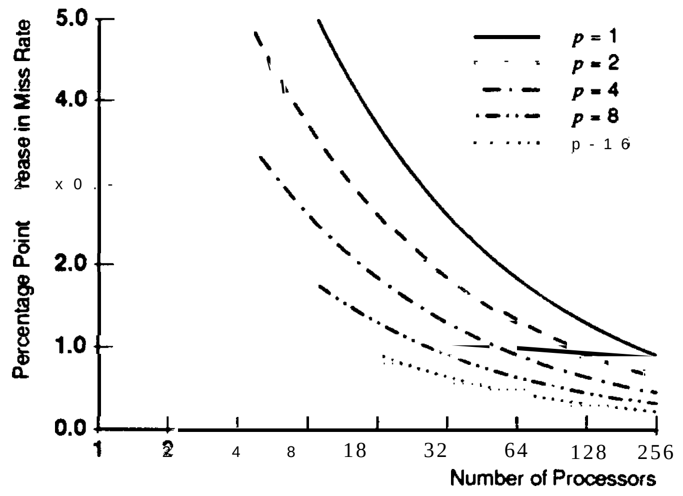

Figure C. 1: Increase in miss rate with $r = 0.167$ and $q = 0.035$.

We noted before that the model we have presented is only valid in those situations in which the directory has run short of pointers. This is because the model assumes the pointer/link store behaves as a fully associative cache *with random replacement.* Due to the free list mechanism, the directory actually performs much better than random replacement, since no useful data is ever discarded if there are any free pointers. Even during a time span when the directory's pointers are often exhausted, if any data sharing causes invalidations, some pointers will be at least temporarily available, resulting in a lower miss rate increase than the model predicts.

sizes C_1 and C_2 are therefore related as follows:

$$\frac{m_1}{m_2} = \left(\frac{C_1}{C_2}\right)^a \tag{C.3}$$

We can empirically determine a by examining measured miss ratios for different sized caches. Przybylski's data (see Figure 4-5 in [40]) shows a to be roughly -0.5, which corresponds very closely with the rule of thumb reported by Stone [48] that doubling the cache size reduces the number of misses by 30%. Of course, this assumes the two caches have the same organization. In our situation, the processor cache is direct-mapped, while the pointer/link store is fully associative. We can take this into account by observing that the miss ratio of a fully associative cache is roughly the same as that of a direct mapped cache that is twice as large (again, see Figure 4-5 in [40]). So m_{extra} is given by:

$$m_{extra} = m\left(\frac{2pC/n}{C}\right)^{-0.5} = m\sqrt{\frac{n}{2p}} \tag{C.4}$$

By substituting (C.4) into (C.2) we can generate an expression for the extra misses due to dynamic pointer allocation. We can now use the maximum miss ratio as given by (C.l) to solve for the maximum number of percentage points by which dynamic pointer allocation increases the miss rate:

$$\text{percentage point increase} = 100fm\sqrt{\frac{n}{2p}} = 100f\left(\frac{q}{nrf}\right)\sqrt{\frac{n}{2p}} = \frac{100q}{r\sqrt{2pn}} \tag{C.5}$$

Let us set r and q in order to demonstrate the miss increase. If a single instruction-per-cycle processor is 50% utilized and issues a data reference every third instruction, then $r = (0.5)(0.33) = 0.167$ references per cycle. If a non-pipelined memory module requires 20 cycles to satisfy a request up to about 70% utilization, then $q = (0.7)(0.05) = 0.035$ requests per cycle. The resulting percentage point increase in the miss rate as n and p are varied is shown in Figure C.l. If there are a large number of processors accessing the data, then the increase in misses is low since the base miss rate m must be very low to avoid exceeding the memory module bandwidth. For smaller values of n, the increase in the miss rate can be higher, but it is less likely to occur since the directory is less likely to run short of pointers. For the values of r and q *we* have chosen, the miss rate never rises by more than 3.3 percentage points given a 4:1 ratio of pointers to cache blocks, or by more. than 1.7 percentage points given an 8: 1 ratio.

first unlinks a pointer from the free list, and then inserts the pointer at the head of the list for the requested block. The pseudocode for this insertion is shown in Figure D.2. Since the pointer can be unlinked from the free list during the slow head links access, the actions described by Figure D.l require a time of $d_s + d_f + s$.

Figure D.3 shows the actions required for a write hit to a clean block. Invalidations must be sent to all of the caches indicated by the pointers on the block's list, except for the cache that issued the request. This will return all but one pointer to the free list. Rather than serially insert each pointer at the head of the free list, it is more efficient to write the id. of the requesting cache into the first pointer on the list, and then link the remainder of the list into the free list in one step after all the invalidations have been sent. These operations require at least $d_s + is + 2s$ time to complete, where i is the number of pointers on the list. An additional delay s is also incurred if the first pointer on the list is not the id. of the requesting cache. As the list is traversed, a state bit called *found* is set if the cache that sent the request is found on the block's list; this information is needed by the controller (see Section 4.5.2). The pseudo-code of Figure D.3 can be easily modified to handle a write miss to a clean block; the only difference is that all caches on the list receive invalidations.

Another means of handling a write hit to a clean block is shown in Figure D.4. This pseudo-code is used if invalidation acknowledgements are counted using the pointers themselves, as described in Section 4.5.1. Using this strategy, a block's pointers are not immediately returned to the free list after the invalidations are sent. Instead, the number of pointers on the list is used to represent the number of outstanding invalidation acknowledgements. When each of these acknowledgements is received, the directory returns a pointer to the free list, thereby decrementing the count of outstanding acknowledgements. This operation is shown in Figure D.5 (which performs one of its steps using the pseudo-code of Figure D.6). Both Figures D.4 and D.5 show two exit points, labeled *done* and *done_no_invacks*. The pseudo-code uses these two exit points to indicate whether there are still outstanding invalidation acknowledgements. The exit point done is used if further acknowledgements are expected; when they have all been received, *done_no_invacks* is used. The time used to send the invalidations is $d_s + d_f + is + 3s$, where i is the number of pointers on the list. The time required to handle each invalidation acknowledgement

Appendix D

Dynamic Pointer Allocation Operation

In this appendix we provide pseudocode that defines the sequence of sub-operations that must take place on the bus in Figure 2.15 in order to accomplish common directory operations. We also describe the amount of time required to execute these operations in terms of the access times of the head link storage and pointer/link store. We assume the head link storage is implemented with dynamic RAM, with an access time of d_s for random addresses and d_f for subsequent accesses to the same address. The pointer/link store is built with static RAM with an access time of s. The times we provide for each operation do not include controller delays or register-to-register transfer times, because these are highly dependent on the actual implementation and are probably dominated by the RAM accesses in any case.

Throughout this appendix we use the following conventions in our pseudo-code. In all cases, the main memory address register contents are supplied as the head links address by the multiplexer (see Figure 2.15) unless otherwise specified by the pseudo-code. Indentation is used in the pseudo-code to indicate the beginning and end of ***if-then-*** else clauses and *while* loop bodies. Multiple operations are delimited by a semicolon (;) if they may be executed in parallel. Pseudo-code sequences that are repeated in several places **are** sometimes moved into separate figures for clarity.

Figure D.l shows the pseudo-code that is executed at the directory for a read miss to a clean block, the most common directory request. In this case, a new pointer identifying the new reader must be added to the block’s list. To accomplish this, the pseudo-code

```
begin
    if (free list empty)                        # are free pointers available?
        goto no_pointers                        # no, go handle it (see Figure D.8)
    addr reg ← free                             # address first free pointer/link
    if (end-of-list)                            # is this the last pointer/link on free list?
        free list empty ← 1                     # yes, set free list to be empty
    else
        free ← link data                        # no, remove pointer/link from free list
    [insert addressed pointer at head of list]  # set up pointer to new cache (see Figure D.2)
end
```

Figure D.l: Pseudo-code for adding a new cache to a data block's list of pointers.

```
[insert addressed pointer at head of list]:
begin
    ptr data ← new cache id.;                               # set pointer to new cache
    if (not empty)                                          # is this block's list of pointers empty?
        link data ← head link data; end-of-list ← 0         # no, link to next pointer in block's list
    else
        link data ← mem block address; end-of-list ← 1      # yes, link back to head of list
    head link data ← addr reg                               # make new pointer/link head of list
end
```

Figure D.2: Pseudo-code for inserting the currently addressed pointer at the head of the current list. In addition, the pointer is set to the cache that issued this request.

is $d_s + d_f + 2s$, except for the acknowledgement that arrives last, which requires a time of $d_s + s$.

Figure D.7 shows the pseudo-code used to process replacement notifications (see Section 2.3.2). This involves searching the block's list of pointers for the id. of the cache that sent the notification, and returning the pointer that matches to the free list. The search itself requires time d_s + is, where the matching cache id. is stored in the i^{th} pointer on the list. The additional time to return the matching pointer to the free list is $d_f + s$ if $i = 1$ or $2s$ if $i > 1$.

The steps required to process a read miss request when no free pointers remain are shown in Figure D.8. In this case, a pointer must be selected and freed by sending an invalidation to the cache identified by the pointer (see Section 2.3.2).[1] To do this, a pointer is first selected at random. If the selected pointer is at the end of a list, then it cannot be removed from its list since lists are singly linked. Instead, the back link is used to address the head link of the list, and the first pointer on the list is removed. The entire operation requires time $2d_s + 2d_f + 3s$. If the selected pointer is not at the end of a list, the list is traversed to find the last pointer, which is then removed. In this case, the time required is $d_s + d_f + (i + 3)s$, where i is number of links that must be traversed from the selected pointer to the pointer at the end of the list. The time d_s may actually be hidden, since the DRAM access can proceed while the list is being traversed.

'Other policies are possible as well. One option that would seem to make sense is to free multiple pointers in an attempt to forestall recurrences of this situation.

```
begin
    found ← 0                                              # haven't found sender pointer yet
    if (empty)                                             # is this block's list empty?
        goto done                                          # yes, we are done
    addr reg ← head link data                              # address first pointer/link on block's list
    while (true)
        if (ptr data ≠ sender id)                          # does pointer match sender?
            send invalidation to cache indicated by ptr data   # no, send invalidation
        else
            found ← 1                                      # yes, we have found sender on list
        if (end-of-list)                                   # is this last pointer/link on list?
            goto setup-count                               # yes, go set up for counting invack replies
        addr reg ← link data                               # no, move on to next pointer/link on list
setup-count:
    ptr data ← sender id                                   # write sender id into last pointer on list
    if (found)                                             # did we find sender on block's list?
        addr reg ← head link data                          # yes, address first pointer/link on list
        if (end-of-list)                                   # is there only one pointer/link on list?
            goto done_no_invacks                           # yes, must be sender, no invacks will arrive
        head link data ← link data                         # no, unlink first pointer/link from list
        [add addressed link to free list]                  # reclaim pointer/link (see Figure D.6)
    goto done                                              # go wait for first invack to arrive
donenoinvacks:
done:
end
```

Figure D.4: Pseudo-code for sending invalidations to all of the caches on a block's list of pointers, except for the cache that issued the request. This pseudo-code assumes the pointer list itself is used to count invalidation acknowledgements.

```
begin
    addr reg ← head link data                  # address first pointer/link on block's list
    if (end-of-list)                           # is there only one pointer/link on list?
        goto donenoinvack                      # yes, this is last invack
    head link data ← link data                 # no, unlink first pointer/link from list
    [add addressed link to free list]          # reclaim pointer/link (see Figure D.6)
    goto done                                  # count decremented, wait for next invack
donenoinvacks:
done:
end
```

Figure D.5: Pseudo-code for handling an incoming invalidation acknowledgement. This pseudo-code assumes that the pseudocode shown in Figure D.4 was used to send the invalidations.

```
begin
    found ← 0                                          # haven't found sender pointer yet (see text)
    if (empty)                                         # is this block's list empty?
        goto done                                      # yes, we are done
    addr reg ← head link data                          # address first pointer/link on block's list
    if (ptr data ≠ sender id)                          # does first pointer match sender?
        send invalidation to cache indicated by ptr data   # no, send invalidation
        ptr data ← sender id                           # set first pointer to sender id
    else
        found ← 1                                      # yes, we have found sender on list
    if (end-of-list)                                   # only one pointer/link on list?
        goto done                                      # yes, we are done
    templ ← link data                                  # no, save link to first pointer to free
    link data ← mem block addr; end-of-list ← 1        # link back to head of list
    addr reg ← templ                                   # move to second pointer/link on block's list
    while (true)                                       # loop until we explicitly branch out of it
        if (ptr data ≠ sender id)                      # does pointer match sender?
            send invalidation to cache indicated by ptr data   # no, send invalidation
        else
            found ← 1                                  # yes, we have found sender on list
        if (end-of-list)                               # is this last pointer/link on list?
            goto final                                 # yes, go return pointer/links to free list
        addr reg ← link data                           # no, move on to next pointer/link on list
final:
    if (-free list empty)                              # is free list empty?
        link data ← free; end-of-list ← 0              # no, link free list to end of invalidated list
    free ← templ; free list empty ← 0                  # insert invalidated list at head of free list
done:
end
```

Figure D.3: Pseudo-code for sending invalidations to all of the caches on a block's list of pointers, except for the cache that issued the request. This pseudo-code assumes the pointer list itself is not used to count invalidation acknowledgements.

```
no-pointers:
begin
    addr reg ← random value                                # select a pointer/link at random
    if (end-of-list)                                       # is this the last pointer/link on list?
        temp 1 ← link data, head links addr ← templ        # use return pointer to address head links
        addr reg ← head link data                          # address first pointer/link on this list
        send invalidation to cache indicated by ptr data   # get addr for invalidation from link data
        head link data ← link data; empty ← end-of-list    # unlink pointer/link from list
        head links addr ← main memory addr reg             # address block requested in miss
        [insert addressed pointer at head of list]         # set up pointer to new cache (see Figure D.2)
    else
        while (-end-of-list)                               # traverse list to last pointer/link
            templ ← addr reg                               # save address of this pointer/link
            addr reg ← link data                           # address next pointer/link on block's list
        send invalidation to cache indicated by ptr data   # get addr for invalidation from link data
        temp2 ← link data                                  # save link back to head of list
        [insert addressed pointer at head of list]         # set up pointer to new cache (see Figure D.2)
        addr reg ← templ                                   # address last pointer from the random list
        link data ← temp2; end-of-list ← 1                 # set link back to head of list
end
```

Figure D.8: Pseudo-code for adding a new cache to a block's list of pointers when no pointers remain on the free list.

```
[add addressed link to free list]:
begin
    if (free list empty)                                # is free list empty?
        end-of-list - 1                                 # yes, pointer/link will be end of free list
    else
        link data ← free; end-of-list ← 0               # no, pointer linked to rest of free list
    free ← addr reg; free list empty ← 0                # insert pointer/link at head of free list
end
```

Figure D.6: Pseudo-code for inserting the currently addressed pointer at the head of the free list.

```
begin
    if (empty)                                          # is this block's list empty?
        goto done                                       # yes, pointer not found, we are done
    addr reg ← head link data                           # address first pointer/link on block's list
    if (ptr data = sender id)                           # does first pointer on list match?
        head link data ← link data; empty ← end-of-list # yes, unlink first pointer/link from list
        [add addressed link to free list]               # reclaim pointer/link (see Figure D.6)
        goto done                                       # pointer/link reclaimed, we are done
    templ ← addr reg                                    # save address of this pointer/link
    addr reg ← link data                                # address next pointer/link on block's list
    while (true)                                        # loop until we explicitly branch out of it
        if (ptr data = sender id)                       # does this pointer on list match?
            temp2 ← link data, end-of-list              # yes, going to xfer link to previous pointer
            [add addressed link to free list]           # reclaim pointer/link (see Figure D.6)
            addr reg ← templ                            # address previous pointer/link on list
            link data, end-of-list ← temp2              # get link & end-of-list from reclaimed ptr.
            goto done                                   # pointer/link reclaimed, we are done
        templ ← addr reg                                # no match, save address of this pointer/link
        addr reg ← link data                            # address next pointer/link on block's list
done:
end
```

Figure D.7: Pseudo-code for searching a block's list of pointers for a particular pointer and returning that pointer to the free list.

than main memory blocks, multiple blocks must map into each entry. This mapping is done in the same way that cache addressing is accomplished: a subset of the bits that address main memory are used to address the directory. We call this subset the *directory index bits* of the address. Main memory blocks with identical directory index bits combine their directory information into a single entry. The combined information in the entry represents the union or logical-OR of the information that would be carried if the blocks had their own entries. For instance, in a pointer-based directory, the pointers would identify those caches containing at least one of the blocks. An exception is the dirty bit; in entry combining we continue to maintain a dirty bit for every block of main memory. In Section E.3 we will see that combining the dirty bit would substantially increase the performance penalty of entry combining.

Since we would like to minimize the frequency with which directory entries represent more than one cached block at a time, the directory index bits should be lower-order bits from the memory address. This prevents blocks that are nearby in the address space (and therefore likely to be cached simultaneously due to spatial locality) from mapping into the same entry.

There are several obvious ways entry combining may be used in conjunction with the width-reducing directory organization described in Chapter 2. In a limited pointers directory of a given size, the designer can double the number of pointers per entry for each halving of the length using entry combining. In a dynamic pointer allocation directory, entry combining can be used to reduce the length of the head links storage. Alternatively, the head links storage could be completely removed by directly indexing the pointer/link store using the directory index bits. In this case it would probably be best to make the pointer/link store large enough that those pointers addressable by the directory in&x bits could be reserved for head links. In this way, a pointer never needs to be freed to begin a list.

E.2 Directory Protocol

We now describe the coherency protocol used by the entry combining directory to maintain coherency. Assuming direct-mapped processor caches, the actions taken by the

Appendix E

Directory Entry Combining

In Chapter 2 we proposed limited pointers and dynamic pointer allocation directories. Both of *these* schemes reduce the directory *width,* or number of bits in each directory entry. We believe that these techniques should be sufficient to meet the performance goals of the directory under reasonable cost constraints. However, for a given implementation, a designer may feel that a width-reducing scheme does not go far enough in reducing the cost of the directory required for a given level of performance. For this situation, the width-reducing strategy may be used in concert with a technique for reducing the *length* of the directory, or the number of directory entries.

In this appendix we propose *directory entry combining,* a general method for reducing the directory length. Entry combining combines the data pertaining to multiple memory blocks (and therefore normally residing in multiple directory entries) in a single entry. The idea is to take advantage of the fact that most main memory blocks at a given time are in fact not cached in the system at all. Most entries, while potentially responsible for multiple blocks, therefore usually contain data for no more than a single block, thereby losing no efficiency.

E.1 Directory Organization

Entry combining does not dictate the organization of each directory entry; instead, it simply reduces the number of those entries in the directory. With fewer directory entries the requesting cache, and the data is written back to memory and sent to the requesting cache.

Write buck. When dirty data is written back, the directory clears the block's dirty bit and the pointer indicating the cache that had the data.

Of course, this description of the protocol is meant to be informal. Keep in mind that there are other cases that must be handled, such as when a block is invalidated from a cache before the cache's write hit request reaches the directory. These situations are orthogonal to the introduction of entry combining into a directory, and are therefore handled in the same way as for a directory without entry combining. For more details, see Chapter 4.

As in the dynamic pointer allocation scheme, accumulating pointers to caches that no longer contain the data degrades the performance of entry combining directories (see Section 2.3.2). These *stale* pointers, which are caused by clean replacements in caches, cause the directory to send a cache messages pertaining to a block the cache no longer contains. One option is to send explicit replacement notifications, as we did in the dynamic pointer allocation directory. The other option takes advantage of the fact that caches must check their tags and respond to each message received from a directory. In the response, the caches can easily indicate in their responses whether they contain a valid datum corresponding to the directory entry containing a pointer to them. When the directory receives a given reply, the pointer can be cleared if the cache indicated that it no longer has valid data corresponding to that directory entry.

The protocol described above may be easily modified to work with set-associative processor caches. The fundamental difference is that a cache can invalidate (or write back) a block that maps to a given directory entry, and still contain a block that maps to the *same* entry. When this occurs, the directory must not clear the pointer indicating the cache, even though an invalidation has occurred. To modify the protocol, all acknowledgements sent by a cache on writes must indicate whether the cache still contains a block that maps to the directory entry corresponding to the request sent to the cache. This is also true for write back messages as well. When an acknowledgement or write back message indicates the cache no longer contains such a block, then the directory frees the pointer identifying that cache.

Entry combining may also has an impact on the width-reducing scheme with which it is used. If pointer resources are exhausted, some width-reducing directories (e.g., no-broadcast limited pointers and dynamic pointer allocation) select a pointer and free it by sending an invalidation to the cache indicated by that pointer. However, using entry combining, there is no way the directory can determine the address (or addresses) corresponding to the selected pointer. In this case, the directory must send a special type of invalidation that includes only the directory index bits instead of a full address. On receiving such an invalidation, a cache must invalidate its block[2] that maps into that directory entry.

E.3 Performance

Since directory entry combining incurs no performance penalty in entries for which no more than one block is cached at a time, we expect configurations in which most entries satisfy this condition to closely approach the performance of the same organization without entry combining. For instance, an entry-combined limited pointers directory that has, say, one quarter as many entries as there are memory blocks should perform close to a standard limited pointers directory with the same number of pointers per entry. This is because a directory that large makes it unlikely that many entries will be shared by more than one cached block simultaneously. However, if the directory length is reduced further and is closer to the number of cache sets, then more entries will maintain pointers for multiple blocks simultaneously.

Even when an entry is required to store pointers pertaining to multiple blocks, there is no increase in the number of cache misses due to entry combining since no extra invalidations are performed. However, there is a traffic penalty from two sources. First, misses on blocks dirty in another cache require the directory to send requests to return the block to all cache identified by the pointers. Only one of these requests will be successful; the others waste network and cache bandwidth. Second, writes to clean blocks require

[2]Or blocks. in the case of a set-associative cache.

invalidations to be sent to all caches[3] identified by the pointers. Those invalidations directed at caches without the block also waste network and cache bandwidth.

Using trace-driven simulation to determine the performance of directory length-reducing measures such an entry combining is problematic. This is because the directory performance depends on the relative sizes of the application working set, the processor caches, and the directory. Address traces typically do not have enough references to properly exercise the large cache sizes of today. One approach is to scale down the simulated cache and directory sizes to match the smaller number of addresses touched by a trace: this methodology is suspect for determining the *absolute* performance of a large-scale machine since it is largely unknown how application behavior scales up. However, a scaled-down simulated directory does create plausible interference behavior within a given directory entry; we can use this behavior to asses the *relative* performance of different length-reducing directory schemes that map multiple addresses into the same direc tory entry.

Using the multiprocess address traces described in Section 2.2.2, we ran simulations to measure the increase in network traffic and memory latency incurred by entry combining and another length-reducing measure, *directory caches* [7, 26]. As in entry combining, multiple blocks map into the same entry in a directory cache. Unlike entry combining, directory cache entries may not be shared by several addresses simultaneously. Instead, the address of the block currently represented by a given entry is stored in a tag associated with that entry. If a request arrives at the directory and the corresponding entry is being used by another block, the entry must be cleared by invalidating that block from all caches indicated by the entry.

The simulator maintains cache and directory state, and modifies that state as each reference in the trace is processed. In other words, no effects due to network delay or cache/directory latencies are included in the study. The simulated caches contain both private and shared data, but no instructions. There are 16 directories and 16 caches. Each directory is sized to have the same number of entries as there are blocks in the cache. Each entry uses the full valid bit vector organization of Censier and Feautrier (see Section 1.1). The caches are direct-mapped, as are the directories in the case of directory

[3]Or all but one cache in the case of a cache hit.

caches. Shared data is allocated randomly to a given directory, while data private to process i is allocated to directory i. Nonetheless, all messages between directories and caches are counted equally without regard to whether the message destination is local or remote. To measure network traffic, we simply count the number of messages sent between caches and directories. To measure memory latency, we count the number of messages that must serially complete before a cache request is satisfied by memory. For instance, a read miss to a block dirty in another cache incurs a latency of four messages (i.e., two round trips in the network). We assume a weakly ordered model of execution; therefore, invalidation messages on write hits to clean blocks do not contribute to the latency.

The results of the simulations are shown in Figures E.l and E.2. The vertical axes show the percentage increases in the number of messages, relative to the traditional directory organization of Censier and Feautrier. The horizontal axes indicate the size of the data space touched by the trace, as expressed by the number of blocks that map into a single directory entry.[4] So moving to the right on the horizontal axes decreases the size of the caches and directories, thereby increasing the interference between different memory blocks in a given directory entry. The four graphs in each figure indicate the results for different applications.

Figures E.l shows the percentage increase in traffic due to the two directory strategies. Figure E.2 shows the percentage increase in latency for the directory cache; since there is zero increase in latency for entry combining, it is not shown on the graph. In general, we see that the basic entry combining directory, indicated by the solid lines, is quite competitive with the directory cache. Entry combining generates substantially less traffic and latency for P-Thor and LocusRoute, and slightly less for Maxflow. The directory cache generates less traffic than entry combining for MP3D. Even in this case, however, the percentage increase in traffic for entry combining appears to be leveling off at a modest 15%.

Though a study using more benchmarks would be beneficial, it appears from our data that entry combining is more robust from a performance standpoint than a direct-mapped

[4]Actually, the trace does not necessarily reference all of the blocks in the data space, since the granularity of data space allocation is 1024 bytes in these simulations.

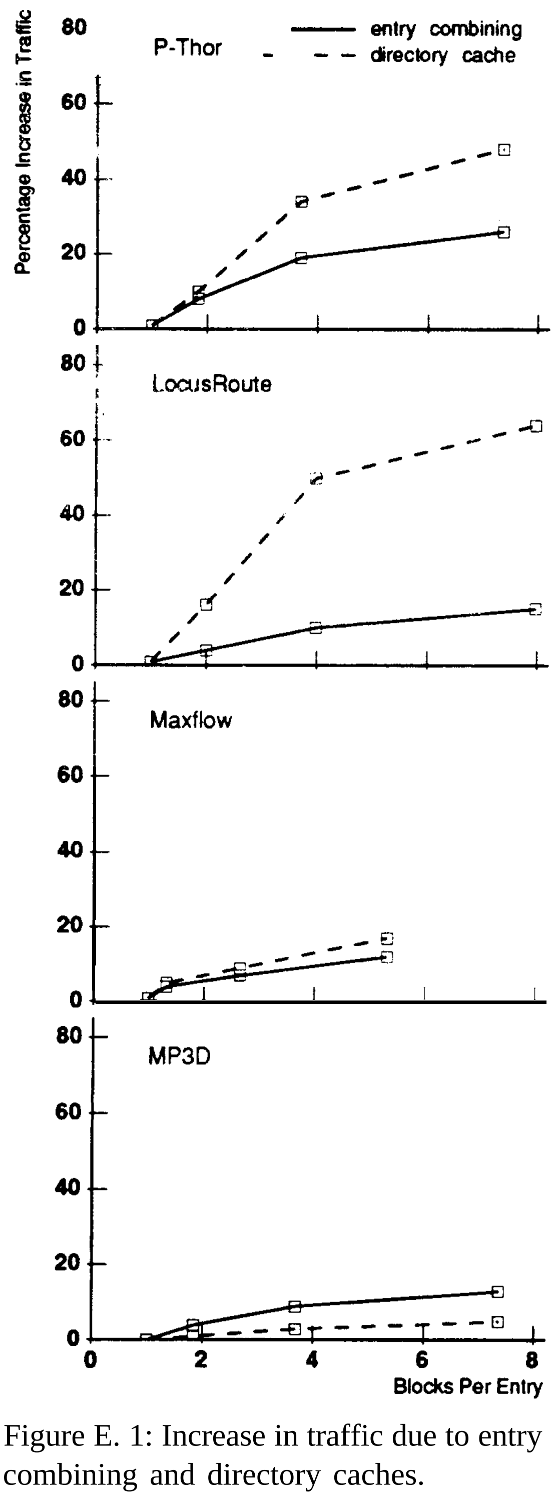

Figure E. 1: Increase in traffic due to entry combining and directory caches.

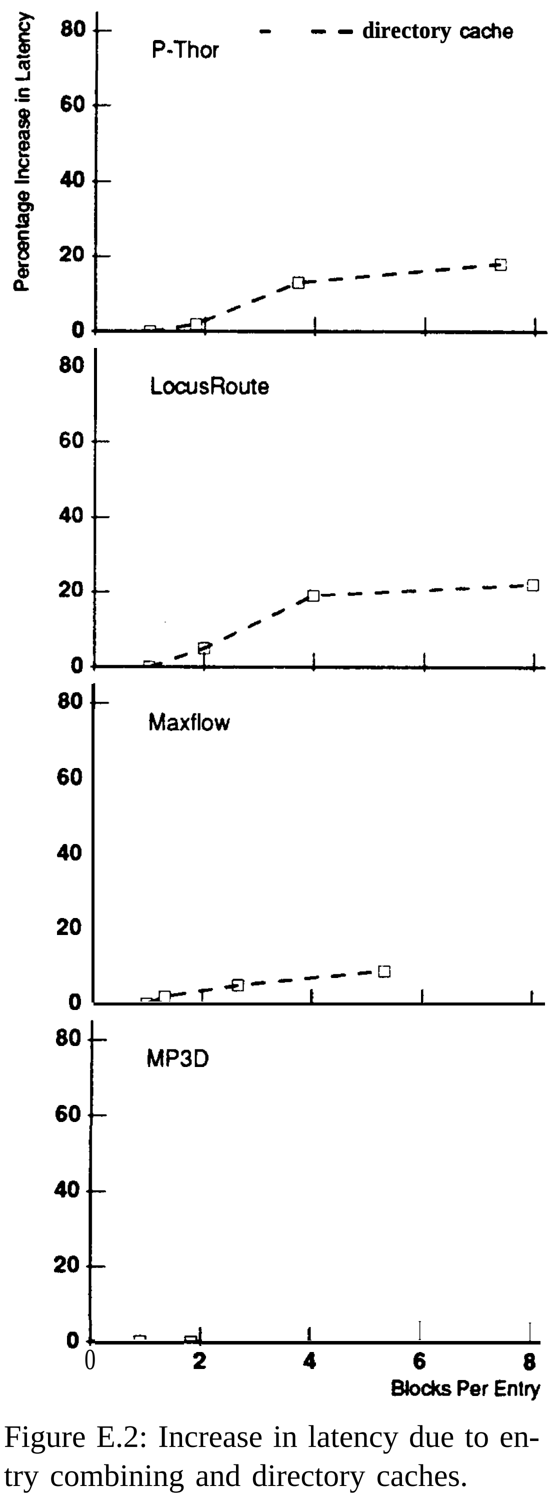

Figure E.2: Increase in latency due to entry combining and directory caches.

directory cache. Of course, the directory cache could be made set-associative; this will improve performance, but is more costly. One would hope that the directory is long enough that interference between addresses in a directory entry is minimal; in this case, both the entry combining and directory cache schemes will work very well, and the choice between them may be based on implementation concerns. In terms of implementation, the disadvantage of entry combining is slightly more complex controllers at the caches and directories. However, it does have the advantage that no extra storage structures are necessary, as opposed to the directory cache, which requires tags on each directory entry.

We also investigated the performance of a more aggressive entry combining scheme which also combines the dirty bits of the blocks that map into an entry, rather than maintaining dirty bits for every block of main memory. In a given entry, the combined dirty bit is set if at least one of the blocks mapping to the entry is dirty. Unfortunately, when a block is requested, the dirty bit may be set even if the block is clean. In this case the directory is forced to attempt to retrieve the block needlessly, resulting in additional latency and traffic. We found that this scheme more than doubled the percentage increase in traffic for three out of the four benchmarks, and incurred an increase in latency similar to the directory cache. In general, the basic entry combining directory that maintains a dirty bit per block of main memory performs substantially better. Since the memory overhead for this single bit is only 0.8% for 16-byte blocks, we do not recommend combining the dirty bits of multiple blocks.

E.4 Conclusion

Directory entry combining is a general directory length-reducing technique that combines directory information for multiple cached blocks into a single entry when necessary. Since it is compatible with most width-reducing measures, entry combining gives the designer wide latitude in choosing a combination of strategies that meet his or her directory performance and storage goals. For the cost of more complicated state machines at the directory and caches (but no additional storage structures), full-length directories can be shortened to any length from a small factor shorter than full length down to lengths on the order of the number of blocks in a cache.

In most cases, only a single address mapping to a given entry is cached at a time, resulting in directory behavior identical to a full-length directory. In the event an entry becomes shared by multiple addresses, we have shown that in some cases entry combining performs substantially better than direct-mapped directory caches, another length-reducing strategy. Unlike directory caches, entry combining does not increase the number of cache misses, and often results in lower network traffic. Overall, directory entry combining is a relatively inexpensive technique that helps reduce storage requirements while maintaining high performance in those situations where a width-reducing directory alone is deemed insufficient.

Appendix F

Trace Event Frequencies and Costs

In Chapter 3 we evaluated coherency protocol design options by measuring event frequencies from multiprocess address traces and assigning costs to each type of event. In this appendix we show the event frequencies that we measured in each trace, as well as the corresponding costs for protocols with and without the clean/exclusive state.

Table F. 1 gives the event frequencies measured using infinite-sized caches and a block size of 16 bytes. For the rows in the table specific to protocols with a clean/exclusive state, a non-aggressive protocol is assumed (see Section 3.3.2). If an aggressive protocol is used, the table is identical except that all *rm-drty-cx* events become *rm-drty-rwtcx* events and all *wm-drty-cx* events become *wm-drty-notcx* events.

Table F.2 shows the costs for each event, that is, the number of network messages required to handle a reference of each type. Even though the simulator differentiates between some events based on whether the directory shows the block clean/exclusive, we can use the same event frequencies to evaluate the base Censier/Feautrier protocol (which does not include the clean/exclusive state). Table F.2 demonstrates how this is done: for the Censier/Feautrier scheme, equal cost is assigned to any two events that are distinguished only by whether the directory shows the block clean/exclusive. For example, *the rm-cfn-cx* and *rm-cfn-rwtcx* events both cost two network messages. This makes sense, because the Censier/Feautrier protocol has no notion of a clean/exclusive state in its directory; therefore, the resulting cost of the event cannot depend on whether the simulated directory shows the block clean/exclusive.

Event Type	*Trace*			
	P-Thor	LocusRoute	Maxflow	MP3D
read	78.31	75.50	76.99	84.35
rd-hit	76.16	74.69	70.45	70.01
rd-miss	1.15	0.46	6.51	13.19
rm-cln-cx	0.22	0.12	0.02	0.95
rm-drty-cx	0.11	0.01	0.01	0.16
rm-cln-notcx	0.54	0.26	3.29	3.05
rm-drty-notcx	0.27	0.07	3.19	9.03
l-m-first-ref	1.00	0.35	0.03	1.15
write	21.69	24.50	23.01	15.65
wrt-hir	21.19	24.39	22.83	15.64
wh-drty	20.54	24.17	19.71	5.74
wh-cln	0.65	0.22	3.12	9.90
wh-cln-cx	0.34	0.13	0.01	0.19
wh-cln-notcx	0.31	0.08	3.11	9.71
wrt-miss	0.28	0.01	0.18	0.00
wm-cln-cx	0.01	0.00	0.00	0.00
wm-drty-cx	0.01	0.00	0.00	0.00
wm-cln-notcx	0.04	0.00	0.09	0.00
wm-drty-notcx	0.22	0.01	0.08	0.00
wm-first-rcf	0.22	0.11	0.00	0.01
wrt-cln-Oinv	0.34	0.13	0.01	0.19
wrt-cln-linv	0.25	0.05	1.82	8.43
wrt-cln-2inv	0.07	0.01	0.59	0.59
wrt-cln-3inv	0.01	0.01	0.34	0.68
wrt-cln-4inv	0.00	0.01	0.20	0.02

LEGEND

rd	read
wrt	write
ml	read miss
wh	write hit
wm	write miss
cln	clean
drty	dirty
cx	marked clean/exclusive at the directory
notcx	not marked clean/exclusive at the directory
first-ref	first reference by any processor
iinv	invalidations sent to *i* caches; 6inv through 15inv not shown

Table El: Event frequencies for infinite-sized caches. The numbers are shown as a percentage of all data references. The fractions in each sub-category add up.

protocol:	***Censier/Feautrier***		***Clean/exclusive state added***		
Event Type	sequential consistency	weak ordering	sequential consistency	weak ordering	weak ordering (aggressive)
read					
rd-hit					
rd-miss					
rm-cln-cx	2	2	4	4	L=2 T=4
rm-drty-cx	4	4	4	4	4
rm-cln-notcx	2	2	2	2	2
rm-drty-notcx	4	4	4	4	4
rm-first-ref					
write					
wrt-hit					
wh-drty					
wh-cln	2	2			
wh-cln-cx			0	0	L=0 T=2
wh-cln-notcx			2	2	2
wrt-miss					
wm-cln-cx	2	2	4	4	4
wm-drty-cx	4	4	4	4	4
wm-cln-notcx	2	2	2	2	2
wm-drty-notcx	4	4	4	4	4
wm-first-ref					
wrt-cln-Oinv					
wrt-cln-1inv	L=2 T=2	L=0 T=3	L=2 T=2	L=O T=3	L=O T=3
wrt-cln-2inv	L=2 T=4	L=O T=5	G-2 T=4	L=0 T=5	L=O T=5
wrt-cln-3inv	L=2 T=6	L=O T=7	L=2 T=6	L=0 T=7	L=O T=7
wrt-cln-4inv	L=2 T=8	L=0 T=9	L=2 T=8	L=O T=9	L=0 T=9
write back (replacement)	L=0 T=l	L=0 T=1	L=O T=l	L=O T=1	L=O T=l

LEGEND (see also Table F.1)

(blank, i.e., no value)	latency = traffic = 0 network messages
i (a single value)	latency = traffic = ***i*** network messages
L=i	latency = ***i*** network messages (used only if latency ≠ traffic)
T=*i*	traffic = ***i*** network messages (used only if latency ≠ traffic)

Table F.2: Event costs. The numbers indicate the latency and traffic for each event of a given type, expressed as the number of network messages.

Bibliography

[1] Sarita V. Adve, Vikram S. Adve, Mark D. Hill, and Mary K. Vernon. Comparison of Hardware and Software Cache Coherence Schemes. In *Proc. of the 18th Int. Sym. on Computer Architecture,* pages 298-308, May 1991.

[2] Anant Agarwal. *Analysis of Cache Performance for Operating Systems and Multiprogramming.* PhD thesis, Stanford University, Department of Electrical Engineering, 1987.

[3] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proc. of the 15th Znt. Sym. on Computer Architecture,* pages 280-289, May 1988.

[4] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[5] James Archibald and Jean-Loup Baer. An Economical Solution to the Cache Coherence Problem. In *Proc. of the 11th Int. Sym. on Computer Architecture,* pages *355-362,* May 1984.

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

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

[8] James K. Archibald. A Cache Coherence Approach for Large Multiprocessor Systems. In *Proc. of International Conference on Supercomputing,* pages 337-345, July 1988.

[9] Jean-Loup Baer and Wen-Hann Wang. On the Inclusion Properties for Multi-Level Cache Hierarchies. In *Proc. of the 15th Int. Sym. on Computer Architecture,* pages *73-80,* May 1988.

[10] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proc. of the 17th Int. Sym. on Computer Architecture,* pages 125-134, May 1990.

[1 1] Francisco Javier Carrasco. A Parallel Maxflow Implementation. Unpublished report, Stanford University, March 1988.

[12] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers, C-27(* 12): 1112-l 118, December 1978.

[13] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache Coherence in Large-Scale Multiprocessors. *Computer,* 23(6):49–58, June 1990.

[14] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proc. of the Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* April 1991.

[15] Hoichi Cheong. *Compiler-Directed Cache Coherence Strategies for Large-Scale Shared-Memory Multiprocessor Systems.* PhD thesis, University of Illinois, Urbana-Champaign, Department of Electrical Engineering, 1990.

[16] C. K. Chow. Determination of Cache's Capacity and its Matching Storage Hierarchy. *IEEE Transactions on Computers,* C-25(2): 157-164, February 1976.

[17] Ron Cytron, Steve Karlovsky, and Kevin P. McAuliffe. Automatic Management of Programmable Caches. In *Proc. of the 1988 Int. Conf. on Parallel Processing,* pages 11229-11238, August 1988.

[18] Michel Dubois and Fayé A. Briggs. Effects of Cache Coherency in Multiprocessors. In *Proc. of the 9th Int. Sym. on Computer Architecture,* pages 299-308, April 1982.

[19] Susan J. Eggers and Randy H. Katz. A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation. In *Proc. of the 15th Int. Sym. on Computer Architecture,* pages 373-382, May 1988.

[20] Susan J. Eggers and Randy H. Katz.' The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proc. of the Third Znt. Conf. on Architectural Support for Programming Languages and Operating Systems,* pages 257-270, April 1989.

[21] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. of the 17th Int. Sym. on Computer Architecture,* pages 15-26, May 1990.

[22] Stephen R. Goldschmidt. Simulating Multiprocessor Memory Traces. Unpublished report, Stanford University, December 1987.

[23] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proc. of the 10th Int. Sym. on Computer Architecture,* pages 124–130, May 1983.

[24] James R. Goodman and Philip J. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. *In Proc. of the 15th Int. Sym. on Computer Architecture,* pages 422-431, May 1988.

[25] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proc. of the 18th Int. Sym. on Computer Architecture,* May 1991.

[26] Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *Proc. of the 1990 Int. Conf. on Parallel Processing,* pages 1312-1321, August 1990.

[27] Shoji Iwamoto and Hisashi Horikoshi. Memory Control System in Multi-Processing System. U. S. Patent 3,581,291, May 25 1971.

[28] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi. Scalable Coherent Interface. *Computer,* 23(6):74–77, June 1990.

[29] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. Competitive Snoopy Caching. *Algorithmica,* 3(1):79–1 19, 1988.

[30] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a Cache Consistency Protocol. In *Proc. of the 12th Int. Sym. on Computer Architecture,* pages 276-283, June 1985.

[31] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers,* C-28(9):241-248, September 1979.

[32] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of the 17th Int. Sym. on Computer Architecture,* pages 148-159, May 1990.

[33] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash Multiprocessor. *Computer,* 25(3):63–79, March 1992.

[34] Daniel Edward Lenoski. *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor.* PhD thesis, Stanford University, 1991.

[35] Ewing Lusk, Ross Overbeek, et al. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, Inc., 1987.

[36] Jeffrey D. McDonald and Donald Baganoff. Vectorization of a Particle Simulation Method for Hypersonic Rarified Flow. In *AIAA Thermodynamics, Plasmadynamics and Lasers Conference,* June 1988.

[37] Sang Lyul Min and Jean-Loup Baer. A Performance Comparison of Directory-based and Timestamp-based Cache Coherence Schemes. In *Proc. of the 1990 Int. Conf. on Parallel Processing,* pages 1305-13 11, August 1990.

[38] Brian W. O'Krafka and A. Richard Newton. An Empirical Evaluation of Two Memory-Efficient Directory Methods. In *Proc. of the 17th Int. Sym. on Computer Architecture,* pages 138-147, May 1990.

[39] Mark S. Papamarcos and Janak H. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proc. of the 11th Int. Sym. on Computer Architecture,* pages 348-354, May 1984.

[40] Steven A. Przybylski. *Cache and Memory Hierarchy Design.* Morgan Kaufman Publishers, Inc., San Mateo, California, 1990.

[41] Jonathan Rose. LocusRoute: A Parallel Global Router for Standard Cells. In *Design Automation Conference,* pages 189-195, June 1988.

[42] Christoph Scheurich and Michel Dubois. Correct Memory Operation of Cache-Based Multiprocessors. In *Proc. of the 14th Int. Sym. on Computer Architecture,* pages 234-243, June 1987.

[43] Richard Simoni and Mark Horowitz. Dynamic Pointer Allocation for Scalable Cache Coherence Directories. In *Proc. of the Int. Sym. on Shared Memory Multiprocessing,* pages 72-81, April 1991.

[44] Richard Simoni and Mark Horowitz. Modeling the Performance of Limited Pointers Directories for Cache Coherence. In *Proc. of the 18th Int. Sym. on Computer Architecture,* pages 309-318, May 1991.

[45] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News,* 20(1):5–44, March 1992.

[46] Alan Jay Smith. Cache Evaluation and the Impact of Workload Choice. In *Proc. of the 12th Int. Sym. on Computer Architecture,* pages 64-73, June 1985.

[47] Larry Soule and Anoop Gupta. Characterization of Parallelism and Deadlocks in Distributed Logic Simulation. In *Proc. of 26th Design Automation Conference,* pages 81-86, June 1989.

[48] Harold S. Stone. *High-Performance Computer Architecture.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1990. Second Edition.

[49] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[50] 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.

[51] Manu Thapar and Bruce Delagi. Stanford Distributed-Directory Protocol. *Computer,* 23(6):78–80, June 1990.

[52] Dominique Thiébaut. On the Fractal Dimension of Computer Programs and its Application to the Prediction of the Cache Miss Ratio. *IEEE Transactions on Computers,* 38(7):1013–1026, July 1989.

[53] Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proc. of the Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* pages 243-256, April 1989.

[54] John C. Willis, Arthur C. Sanderson, and Charles R. Hill. Cache Coherence in Systems with Parallel Communication Channels & Many Processors. In *Proc. of Supercomputing ‘90,* pages 554-563, Nov 1990.

[55] Andrew W. Wilson, Jr. Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors. In *Proc. of the 14th Int. Sym. on Computer Architecture,* pages 244-252, June 1987.

[56] W. C. Yen and K. S. Fu. Coherence Problem in a Multicache System. In *Proc. of the 9th Int. Sym. on Computer Architecture*, pages 332-339, April 1982.