PERFORMANCE OF PRUNING-CACHE DIRECTORIES FOR LARGE-SCALE MULTIPROCESSORS

by

Steven L. Scott and James R. Goodman

Computer Sciences Technical Report #983

December 1990

Performance of Pruning-Cache Directories for Large-Scale Multiprocessors

Steven L. Scott and James R. Goodman

Computer Sciences Department
University of Wisconsin - Madison
1210 W. Dayton Street
Madison, WI 53706

Abstract

Shared-memory multiprocessors have been successfully implemented with single-bus, snooping cache schemes, known as *multi*s. However, multis are limited to a small number of processors. As systems grow beyond a single bus, the bandwidth requirements of broadcast operations limit scalability, and hardware support to provide cache coherence without the use of broadcast can become very expensive. We describe an approach for maintaining coherence using approximate information held in special-purpose caches, called *pruning caches*, that provides robust performance over a wide range of workloads. The scheme works because the caches hold information necessary to limit broadcasts to those parts of the hierarchy that need to receive it, so information purged from the cache simply results in increased network traffic, not incorrect behavior.

We compare the pruning cache approach to the more conventional inclusion cache for providing Multi-Level Inclusion (MLI) in the cache hierarchy, and show that pruning caches are more cost-effective, and more robust. We show, through both analysis and simulation, that the *k*-ary *n*-cube topology provides scalable, bottleneck-free communication for uniform, point-to-point traffic. We conclude that pruning caches can be employed to build a scalable system for a broad range of workloads.

Steven Scott was supported by a fellowship from the Fannie and John Hertz Foundation. James Goodman was supported in part by National Science Foundation grant #CCR-892766.

1. INTRODUCTION

Achieving very high performance in computing today implies parallel processing. Pushing the frontier means expanding the number of processors while, at the same time, the performance of the candidate individual processors is growing rapidly. Shared-memory multiprocessing, one of several competing models for parallel computing, has established a beachhead in the marketplace with the introduction and wide acceptance of the *multi* [Bell85], a single-bus, snooping-cache-based multiprocessor. Unfortunately, as processor speeds increase, the feasible number of processors that can be connected through a single bus decreases. The medium of a bus is also less than ideal, and degrades as more devices are added to it. This means that the total bandwidth of a bus actually shrinks as more processors are added. For these reasons, it probably makes more sense today to talk about a 4- or 8-processor multi than a 32- or even 64-processor version, as was the case five years ago.

A shared bus has the natural property that all communications are broadcast. This explains the attractiveness of snooping cache protocols, where a simple broadcast model allows a system to guarantee cache coherence. Every cache node observes every bus transaction, and so can detect when shared data is modified. This also explains why there is no obvious extension to the snooping cache model beyond a single shared bus. Without a shared communications medium, modifications to shared data require either an explicit broadcast to all processors in the system, or a mechanism to selectively invalidate (or update) shared copies of the data.

A number of multiple-bus extensions to the multi have been proposed [Wils87, Mudg87, Wins88, Good88]. Among those that limit the number of processors per bus, only Multicube (a k-ary n-cube) expands the system bandwidth at a sufficient rate to allow processors to generate memory requests at a rate independent of system size [Good89]. To achieve this goal, the bandwidth *per node* must increase at the same rate as the average message path length, *i.e.*, the number of buses through which a message must pass. Even this very aggressive rate of growth is adequate only for point-to-point messages. Information that requires broadcast throughout the entire system requires a much greater rate of growth (total bandwidth proportional to the square of the number of processors). Thus, any scheme that must broadcast often is limited in its scalability[1].

Directory-based schemes are faced with the choice of either broadcasting invalidations (or write data) or maintaining records of all nodes that must receive such information. While the former approach doesn't scale because of bandwidth constraints, the latter poses difficulties because of the possibly very large amount of data necessary to keep track of heavily shared cache lines. We will address this problem in detail in section 2 of this paper, where we assume the use of a cube topology, and present a coherence mechanism, *pruning-cache directories*, that blends these two approaches. Rather than broadcasting invalidations to all processors in a system, invalidations are *multicast* to appropriate subsets of the system by using directory information.

Another opportunity provided by the broadcast nature of a bus occurs when reading shared data. Multiple reads of the same cache line can be combined while waiting to access the bus, allowing for the data to be transmitted over the bus only once. While this ability has found little applicability in single-bus systems, it is inherently appealing in larger systems, where the contention for shared data may be higher. Hardware combining

[1]We use the term "scalable" in its intuitive sense throughout the paper. Network utilization should remain roughly constant as system size grows and memory request latency should grow sublinearly, the slower the better. For a more formal treatment of scalability and what it means, refer to [Scot91].

mechanisms have been proposed by several researchers for multiprocessors based upon multistage interconnection networks [Gott87, Pfis85]. In section 2, we present a mechanism for hierarchical read combining, based upon the read combining in multis, for a *k*-ary *n*-cube.

The remainder of this paper is organized as follows. In section 2, we present a topology, cache coherence mechanism and read combining mechanism designed to scale to very large systems. We present analysis demonstrating the performance potential of these mechanisms for invalidations and concurrent read requests. In section 3, we present a simulation study that examines the performances of these mechanisms in the context of a fully operating system. In section 4, we present a summary and concluding remarks.

2. INGREDIENTS FOR A SCALABLE SYSTEM

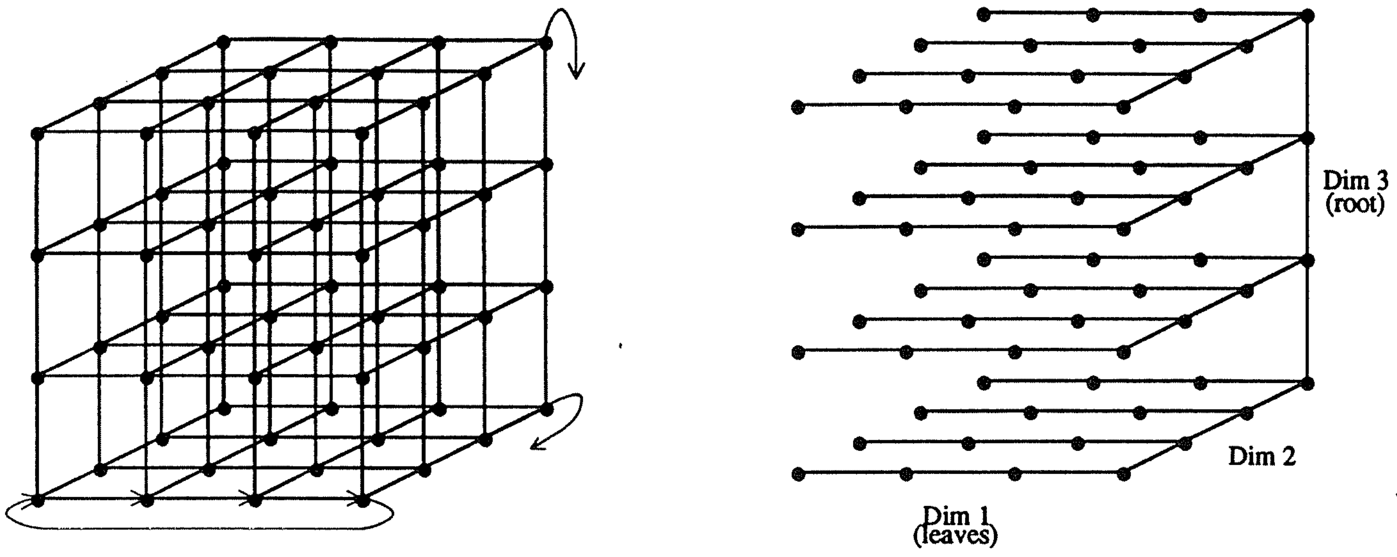
In this section, we present a three-fold approach to designing very large cache-coherent multiprocessors. We first describe a topology that provides bandwidth (for uniform communication) proportional to the number of processors in the system times the average distance between processors. We then propose a cache coherence mechanism that requires only $O(N \log N)$ directory information, and yet prevents invalidation traffic from growing with system size. Finally, we present a hierarchical read combining mechanism that allows concurrent read requests to be serviced in approximate constant time.

2.1. Topology

As was mentioned in the introduction, we will assume the use of a *k*-ary *n*-cube topology. If grown in the proper way, a cube interconnection will provide bandwidth proportional to the number of processors in the system times the average distance between processors. While a cube can be implemented with either buses or point-to-point links, we believe that performance considerations favor the use of links, and will assume links for all analysis in this paper. The electrical properties of direct links allow for faster clocking of data, and their one-way nature allows for clock periods independent of wire lengths, decoupling throughput from latency.

A *k*-ary *n*-cube multiprocessor contains $N = k^n$ processors, each connected to n rings (one for each dimension), with an input and output link for each ring (see figure 1). The total number of links in the system is Nn. A random point-to-point message traverses $O(nk)$ links. Thus, if each processor sends a message to some other processor, the average number of resulting messages per link is $O(k)$. The bandwidth of a cube will scale, then, given that the radix of the network (k) is kept sufficiently small. This requires larger systems to be implemented with higher dimensional networks in order to support the same traffic rate as smaller systems. The latency of messages in a *k*-ary *n*-cube may grow faster than $O(nk)$, particularly if n is greater than the number of physical dimensions in which the network is implemented. See[Agar90] for a more thorough analysis of this phenomenon.

We assume that memory is distributed amongst the processors and interleaved across the system by lines. Each line of memory has a *home* location, where it and its associated directory entry are stored. An important property of the cube topology is that, given a home memory location, an *n*-level tree is formed with the memory at the root (see figure 1(b)). This allows protocols based upon a hierarchy — a tree of buses, for instance — to be implemented on the cube. A system implemented with a single tree will not scale for uniform workloads due to contention for the root of the tree [Scot91]. However, a *k*-ary *n*-cube, by providing k^{n-1} distinct trees, avoids this bottleneck.



(a) The complete 3-cube (b) One of 16 distinct trees

Figure 1: A 4-ary 3-cube Multiprocessor (N=64, k=4, n=3)

Each processor is connected to n rings (with an in and out link for each ring), and each ring consists of k processors (if implemented with buses, each processor would connect to n buses, and each bus would connect to k processors). Although not explicitly shown, all rings include end-around connections as illustrated in figure 1(a). Each processor is accompanied by a portion of main memory and one or more levels of cache. Memory is *interleaved* by cache lines amongst the memory modules. For any given memory module, a tree of rings (or buses) is formed as shown in 1(b). This tree is equivalent to a conventional tree hierarchy, given that each parent node (those in the rightmost plane of figure 1(b)) is allowed to be one of its children. This allows communication protocols based on a hierarchical topology to be implemented.

2.2. Pruning Caches (Scalable Directories)

Although a cube provides sufficient bandwidth (within a constant factor) for uniform, point-to-point traffic, invalidation traffic may grow with system size if not handled properly. In order to scale to very large systems, the fraction of traffic due to invalidations must be kept roughly constant as system size increases. Invalidation latency is also an important issue, as it can affect execution time for certain types of algorithms. We assume that invalidations require acknowledgments in order to provide for some level of sequential consistency [Lamp78, Dubo86, Adve90]. This increases invalidation traffic and may increase invalidation latency, especially if acknowledgements have to be collected serially.

A naive extension of a single bus multiprocessor protocol might broadcast invalidations to all processors in the cube. This would cause the rate of invalidation requests arriving at a processor to increase with N, assuming workloads with a fixed rate of invalidations per instruction. This would place an increasingly heavy burden on the interconnection network and the processors as system size increased.

Alternatively, the locations of shared lines could be maintained, such that invalidations could be sent only where needed. This is the philosophy behind directory-based cache coherence schemes. One of the first directory schemes proposed [Cens78] uses a bit vector of size N for each line of main memory. A bit in a directory entry is set when the corresponding processor obtains a copy of the line associated with the directory entry. When a line is invalidated, invalidation messages are only sent to those processors whose bit is set in the directory entry. While distinct directory entries can be accessed concurrently, accesses to a single directory entry are serialized. This can force the time for m processors to read a line, or the time to invalidate m shared copies of a line (due to acknowledgement collection), to grow as m. This may create significant degradation in large systems for certain workloads. Another disadvantage of the global bit-vector scheme is that the directory requires $O(N^2)$ memory, assuming the size of the memory to be $O(N)$. This becomes prohibitively expensive for very large systems.

In systems with a hierarchical topology and a multicast capability, the directory can be partitioned hierarchically, as follows, to better fit the topology. Let us assume the k-ary n-cube topology discussed above. At each level of the hierarchy, a directory entry consists of a *pruning vector* of length k. A bit in the vector is set if the corresponding subtree beneath the vector may contain one or more copies of the line. When a line is invalidated, the invalidate is placed on the root ring along with the top-level pruning vector, and it is propagated only to those subtrees that may contain a copy of the line. This process is repeated at lower levels by looking up pruning vectors in *sub-directories*. A total of $n-1$ levels of sub-directory accesses are needed (the lowest level is arguably not needed, as it only reduces traffic when a copy resides in the parent of a leaf node, but not in any of its children).

This structure allows read requests to be combined on their way to memory, as a bit in a directory entry is set when *any* processors in the corresponding sub-directory read the line; it is not necessary to know which ones or how many. Thus, when a request reaches a node where there is currently an outstanding read request for the same line, the request may be dropped, and the result decombined when the first request completes (a bit vector corresponding to subtrees waiting for the result can be kept in the cache line while waiting for the data). In addition, latency is reduced by avoiding serialization of invalidation message issue. This also leads to a reduction of traffic over the global (non-hierarchical) directory, because multiple invalidates share part or all of their path through the network. The hierarchical directory, however, still does not scale in cost. It uses $[N-1]\left\lceil \frac{k}{k-1} \right\rceil$ bits, distributed over several directory structures, for each directory entry. Thus, just as the global bit vector, this scheme requires $O(N^2)$ storage for the directories.

A novel way of scaling hierarchical directories is to limit their size and manage them as caches (*pruning caches*) [Good89]. We no longer ***require*** a sub-directory to contain the pruning vector for a given line when it is accessed. If it contains the entry, then we proceed as with the hierarchical directory. If it does not, then we must make the conservative assumption that any subtree may contain a copy of the line (a pruning vector of all ones) and propagate the invalidate to all subtrees. The performance of this mechanism depends upon the hit ratio, h, of the pruning caches. When $h=1$, the pruning caches act identically to a full hierarchical directory. When $h=0$, invalidation traffic is less than with full broadcasts (because the top-level pruning vectors are stored in their directories), but the traffic does not scale; that is, invalidation traffic becomes an ever larger fraction of total traffic as system size increases.

A coherence mechanism that is similar to pruning caches in some ways is the *multi-level inclusion* (MLI) property [Lam79, Wils87, Baer88]. The multi-level inclusion property requires that a cache in a hierarchy contain a superset of all lines residing beneath it in the hierarchy. When the line is invalidated, a parent only propagates the invalidate to its children if it has a copy of the line. This prunes the broadcast invalidation in much the same way as a hierarchical directory or pruning caches. In order to save space, the parent may be required only to have directory information about all lines beneath it in the subtree. This can be kept in *inclusion caches.*

The key difference between inclusion caches and pruning caches, is that MLI *requires* an inclusion cache to contain entries for all lines beneath it. This means that either an inclusion cache must be built such that all lines that can possibly reside simultaneously in the subtree beneath it can also reside simultaneously in it [Baer88], or when an inclusion cache has to throw out a line, it must invalidate that line in the subtree beneath it [Wils87]. The first solution is only feasible in a single tree system where the number of caches decreases at each higher level. The second solution, however, is feasible in a cube-based system. We will compare pruning cache and MLI systems via simulation in section 3.

In order to analyze the performance of pruning caches, we must first make some assumptions regarding the operation of the system. We will assume two lengths of messages: one for messages containing a line of data, and one for address-only messages. Let these messages require T_{data} and T_{addr} cycles to traverse a link (where T_{data} and T_{addr} are the respective message sizes divided by the link width). Further assume that when a message traverses a ring, the message travels from the source to the target on the ring and an echo packet (of the same size as an address packet) is returned around the ring from the target to the source. The echo packet is used for fault tolerance and is similar to that used in SCI [IEEE90]. While the latency to traverse a ring depends upon the distance between the source and target on the ring, the bandwidth used by an address-only message on a ring is always kT_{addr}. A message containing data will use less bandwidth on a ring if the source and target are close, because the message is larger than the echo packet.

Recall that to provide sequential consistency, we must know when an invalidation has been seen by all processors. This requires returning acknowledgements from all processors that received the invalidate. In a hierarchical system, acknowledgements from a broadcast or multicast invalidation can be combined at each level; a parent propagates a single acknowledgement up after receiving the acknowledgements from all its children. When a parent propagates an invalidate onto a leaf ring, it may send the acknowledgement as soon as the invalidate has completed its circuit.

With these assumptions in mind, we can now calculate the expected traffic resulting from a broadcast invalidation. The number of rings in a broadcast tree (see figure 1(b)) is $\left[\frac{k^n-1}{k-1}\right]$, so the traffic from the invalidate packets is $\left[\frac{k^n-1}{k-1}\right]kT_{addr}$. For all but the root ring in this tree, an acknowledgement packet must be passed up to the next highest ring when the invalidations below the ring have completed. However, for one ring out of k, the acknowledgement packet will have to travel no distance on the next highest ring. Thus the traffic from acknowledgements is $\left[\frac{k^n-1}{k-1}-1\right]k\left[\frac{k-1}{k}\right]T_{addr}$, and the total traffic caused by the invalidation is

$$T_{BC} = \left[\left[\frac{k^n-1}{k-1}\right]k+(k^n-k)\right] T_{addr} \tag{1}$$

Given the hit rate, *h*, we can calculate the the expected traffic resulting from an invalidation using pruning caches. Assume *m* shared copies of a line distributed randomly throughout the system, and label the rings as in figure 1(b). A subtree at level *i* contains the k^i processors that are located at or below a level *i* ring. The probability that one of the shared copies resides in a given level *i* subtree is

$$P_C(i,m) = 1 - \frac{b(k^n - k^i, m)}{b(k^n, m)} \qquad (2)$$

where $b(x,y)$ is the binomial function. If we exclude the k^{i-1} processors in the level i subtree whose path to the root of the tree does not traverse the level i ring, then the probability that one of the m shared copies resides in the level i subtree is equivalent to the probability that an invalidate packet *must* traverse the level i ring on an invalidation, and is given by

$$P'_C(i,m) = 1 - \frac{b(k^n - k^i + k^{i-1}, m)}{b(k^n, m)} \qquad (3)$$

The expected traffic from an invalidation using pruning caches with a hit rate of h is now

$$T_{PC} = \left[\sum_{i=1}^{n} k^{n-i} P_{inval}(i,m) k + \sum_{i=1}^{n-1} k^{n-i} P_{inval}(i,m)(k-1) \right] T_{addr} \qquad (4)$$

where $P_{inval}(i,m)$ is the probability that an invalidate packet traverses a ring at level i, which is given by

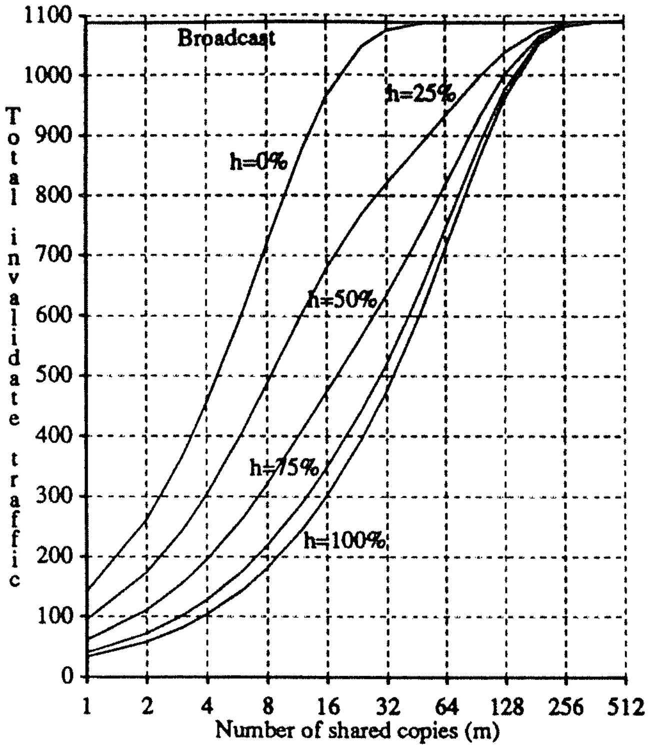
$$P_{inval}(i,m) = P'_C(i,m) + \sum_{j=i}^{n-2} \left(\left[P'_C(j+1,m) - P_C(j,m) \right] (1-h)^{j-i+2} + \left[P_C(j,m) - P'_C(j,m) \right] (1-h)^{j-i+1} \right)$$

$$+ \left[P_C(n-1,m) - P'_C(n-1,m) \right] (1-h)^{n-i} \qquad (5)$$

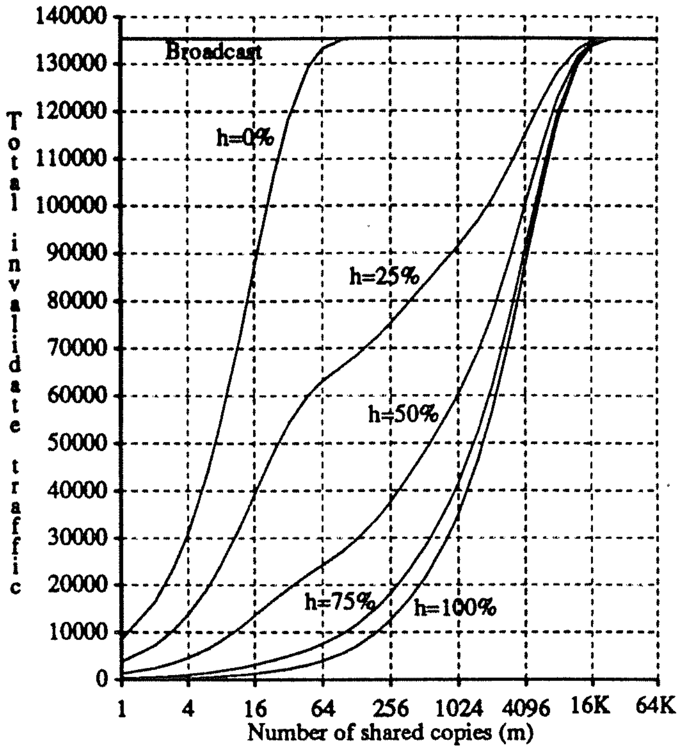
The first term of $P_{inval}(i,m)$ is the probability that an invalidate was *supposed* to traverse the ring at level i, and is the only term that would be non-zero if the hit rate were 1. The remaining terms account for the probability that an invalidate reaches the ring due to pruning cache misses. The first term inside the summation is due to invalidates that were supposed to traverse the level $j+1$ ring, but were not supposed to reach the level j subtree. The second term inside the summation is due to invalidates that were supposed to reach the level j subtree, but not traverse the level j ring. The last term of $P_{inval}(i,m)$ is equivalent to the second term inside the summation for a value of $j=n-1$. For this value of j, the first term inside the summation does not exist because the top-level directory never misses.

If $h=1$, then all traffic due to pruning cache misses goes away, and equation (4) gives us the expected traffic used by a full hierarchical directory. If $h=0$, then the pruning caches are ineffective and traffic is reduced from a full broadcast only by the effect of the pruning vectors in the top level directory. Figure 2 plots the expected traffic of an invalidation versus m, for two example systems (T_{addr} is assumed to be 1). The traffic is shown for a full broadcast (equation 1) and for pruning cache systems with various values of h (equation 4). As indicated in the figure, pruning caches with a modest hit rate significantly reduce the invalidation traffic.

The assumption of a random distribution gives somewhat conservative results, as shown in figure 3. This figure plots the best-case, random, and worst-case traffic caused by a completely pruned invalidation versus the number of shared copies, m, in the system, for a 4096 processor system ($k=8$ and $n=4$). In the best-case distribution, shared copies are clustered along leaf rings and such that higher-dimension rings are traversed as little as possible. This results in traffic of



(a) 512 processor system (k=8, n=3)



(b) 64K processor system (k=16, n=4)

Figure 2: Pruning Cache Performance

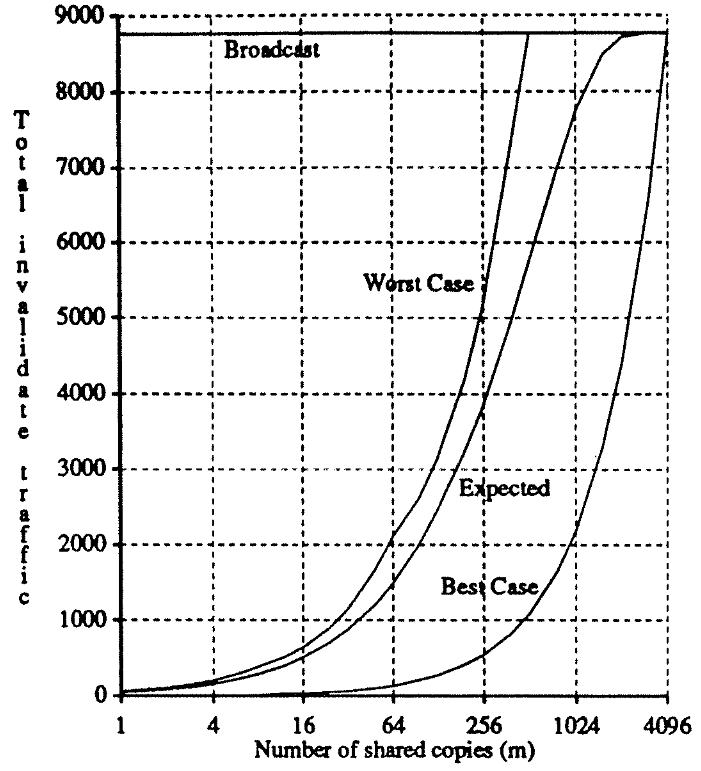


Figure 3: Distribution's Effect on the Traffic of Pruned Broadcasts
(N=4096,k=8,n=4,h=1)

$$T_{best\ case} = \left[\sum_{i=1}^{n} \left\lceil \frac{m-k^{i-1}}{k^i} \right\rceil k + \sum_{i=1}^{n-1} \left(\left\lceil \frac{m}{k^i} \right\rceil - \left\lceil \frac{m}{k^{i+1}} \right\rceil \right) k \right] T_{addr} \tag{6}$$

In the worst-case distribution, shared copies are spread out among different leaf rings and such that higher-dimension rings are traversed as much as possible. This results in traffic of

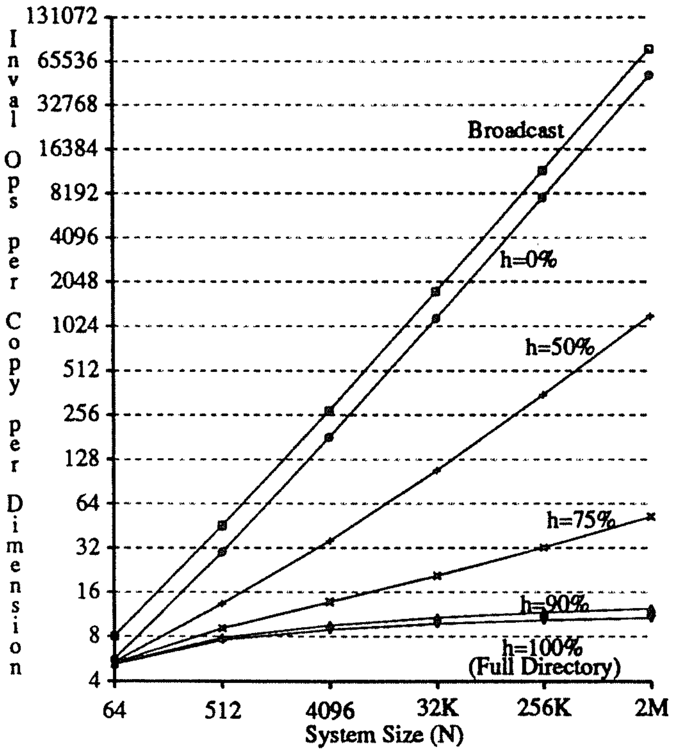
$$T_{worst\ case} = k \left[1 + \sum_{i=1}^{n-1} \left[min\,(k^i, m) + min\,((k-1)k^{i-1}, m) \right] \right] T_{addr} \tag{7}$$

The random distribution assumed in figure 2 gives traffic estimates very close to the worse case, indicating that there is possibly much to gain by organizing sharing along leaf rings when possible. As an added incentive, sharing along leaf rings would increase the efficiency of pruning caches and read combining in the network. This will be validated by simulation results in section 3.

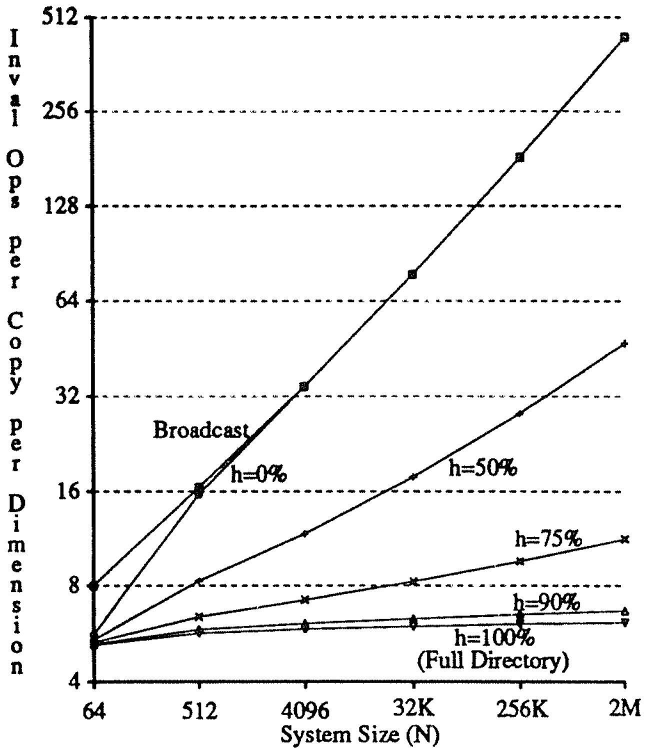
In order for the system to scale in size without invalidation traffic becoming a larger and larger proportion of network traffic, the average invalidation traffic over a link that is caused by each processor making one request, must be O(1). This implies that the traffic caused from an invalidation, divided by the dimensionality of the system and the number of shared copies being invalidated, must also be O(1). For a system that uses a single message for each shared copy, an invalidation requires m messages, each of which requires O(nk) link operations, so this requirement is met. For a system that always performs full broadcast invalidations, however, this requirement would only be met if, on the average, the number of shared copies on an invalidation was a large fraction of N.

Figure 4 shows the normalized invalidation traffic (total traffic of an invalidation divided by the dimensionality of the system and the number of shared copies) for broadcast and pruning cache-based systems, as system size increases. Part (a) shows the scaling behavior with a fixed, small number of shared copies. Part (b) shows the scaling behavior when the number of shared copies grows as the root of the system size. We see that with a pruning cache hit rate of 100%, the normalized traffic appears almost constant as system size increases. In fact, this traffic *is* constant asymptotically. With even a modest hit rate of 75%, invalidation traffic increases very little with system size. With m fixed at 8, traffic increases by a factor of 10 as system size increases by a factor of 32K. By contrast, broadcast invalidation traffic increases by a factor of 10000. This indicates that, given high enough hit rates, pruning cache-based systems will scale to very large numbers of processors.

The question that remains to be addressed is whether pruning caches require $O(N)$ storage (making the total storage requirement $O(N^2)$) to maintain the same level of performance as system size increases. This can be answered with the following argument, which shows that the size of a pruning cache must grow only as $O(n)$.



(a) Small number of shared copies (m=8)



(b) Large number of shared copies ($m=\sqrt{N}$)

Figure 4: Scalability of Pruning Cache Systems (k=8, n=2....7)

The only entries that need be present in the pruning caches are those for actively shared lines. Entries for private, read-only and inactive shared lines may drop out of the pruning caches without harm (in fact, entries for private and read-only lines, if they can be identified, do not have to be placed in pruning caches at all). Assume that each processor cache contains a size-S set of actively shared data. Because memory is interleaved amongst the memory modules, we can assume that the home memory modules of the shared data are roughly spread out throughout the system. A given pruning cache, therefore, must contain approximately $\frac{S}{n}$ entries for each of the k caches along its level 1 ring (refer to figure 1(b)), $\frac{S}{k^2}$ entries for each of the k^2 caches along its level 2 ring and below, ..., and $\frac{S}{k^{n-1}}$ entries for each of the k^{n-1} caches along its level $n-1$ ring and below. The total number of entries needed in a pruning cache is thus $(n-1)S$ and pruning caches should retain the same level of performance if they scale in size as O(n).

2.3. Pruning Cache Management

Pruning caches are maintained as read results and invalidates propagate down through their respective trees. When a parent supplies a line to a child in response to a read request (the parent could have either had a copy of the line being requested, or have propagated the request up the tree and just received the result now) it must do two things. First, it looks up the corresponding pruning vector in its own pruning cache and includes it with the line (recall that if the cache misses, an all-ones vector is assumed). Second, if the pruning vector was in the cache, the bit corresponding to the child is set. When the child receives the line, if its bit in the supplied pruning vector is zero, then it knows that no cache in its subtree previously had a copy of the line, and it can create an all-zero pruning vector for the line in its pruning cache. If it in turn passes the line down to one of its children, then the appropriate bit in this newly created, all-zero vector would be set. The top-level pruning vector for a line is kept in the directory with main memory, and thus is always present. On an invalidation, memory clears its pruning vector, as do all pruning caches that the invalidate passes through. An actual implementation may use the pruning cache entries to combine acknowledgements. In this case, a pruning cache entry would be locked into its cache when an invalidate for its line passed down. As acknowledgements propagated up, the corresponding bits would be cleared and an acknowledgement propagated up to the next level when the vector became zero.

There are several important issues related to pruning cache management that are beyond the scope of this paper. One is their replacement policy. Since the potential penalty for losing a pruning vector is greater the nearer the vector is to the root, a reasonable choice might be to base replacements upon dimension; priority would be higher for higher-dimension vectors. An LRU policy would also be reasonable, as vectors for higher dimensions would tend to be accessed more frequently, and we want rarely used vectors to drop out of the cache. The simulations presented in this paper assume an LRU policy.

Another important issue is the policy regarding all-zero pruning vectors. In the ideal case of perfect hit rates, an all-zero vector is never needed, as any invalidation for the corresponding line would be pruned by a pruning cache at a higher level or the top level directory. This might suggest that all-zero vectors should have the lowest placement priority in pruning caches, never replacing a non-zero entry. However, in the realistic case where pruning caches do miss, all-zero vectors may provide a firewall to prevent errant propagation of an invalidation that missed in a higher level pruning cache. The issue at hand is whether the positive benefit of the "firewall" outweighs the negative effect on hit rates caused by placing all-zero vectors in the pruning caches. The simulations presented in this paper assume that all-zero vectors are not placed in a pruning cache if they require displacing another entry.

2.4. Read Combing In Cube Networks

Contention for data and synchronization objects is a potential problem that must be addressed when designing very large systems. Synchronization can be handled in a variety of manners, both in hardware and in software. Examples include software combining [Yew87], software queueing [Mell91] and hardware queueing [Good89a]. We do not address synchronization issues in this paper other than to say they are important and must be dealt with in some way. We do address the problem of read contention, however, as we feel it is closely related to the cache coherence mechanism. There are several situations in which concurrent read requests to the same line are likely; after a widely shared data object is invalidated, for example, or after a barrier synchronization, as processors read data for a new iteration. If these read requests are serialized at the memory, then latencies may become extremely long in large systems. The hierarchical structure of the cube topology and pruning caches, however, allows concurrent read requests to be combined in the network.

In a system with read combining, reads to private data proceed as normal. Reads to shared data access the data caches along their route to memory. Only the parent caches are checked (those at nodes where the path changes dimensions). If the data is found in a parent cache, the data can be returned immediately, else a shadow line is allocated in the parent cache, and the request continues towards memory. If another read request for the same line arrives at the same parent cache before the data arrives, this request can "combine" and be dropped. A bit vector indicating which subtrees are awaiting the data (called a *pending vector*) is kept in the shadow line and used to distribute the data when it arrives. If more than 1 subtree is waiting for the data when it arrives, the data is *multicast* on the ring to each of the waiting subtrees. This simply means that the data is sent around the ring along with its pending vector, and each processor whose bit is set retains a copy. When the last processor has taken its copy, the ring echo is sent around the remainder of the circuit.

A possible disadvantage of such a combining mechanism is cache pollution caused by parents allocating lines in their cache for their children's reads. Thus, it might be beneficial to keep an auxiliary structure (called a *pending cache*) to store pending vectors. This cache could be quite small, as it would only contain entries for currently outstanding read requests to shared lines. If a pending vector were lost (from either a pending cache or a data cache) before the corresponding data arrived, then the worst-case assumption that all children had requested a copy would have to be made. Thus the data would be broadcast below the processor where the pending vector was lost.

Without read combining, the expected traffic resulting from m concurrent read requests is simply m times the expected traffic resulting from one read request, and is given by

$$T_{reg} = m\left[n\,(k-1)T_{addr} + n\left\lceil\frac{k-1}{2}\right\rceil(T_{addr}+T_{data})\right] \tag{8}$$

The expected latency, assuming no other traffic in the network, is dominated by the contention for the memory module or link entering the memory module, whichever is slower. This latency is

$$L_{reg} = E_{min\_links}(m) + T_{addr} + (m-1)max\,(T_{addr},L_{mem}) + L_{mem} + n\left\lceil\frac{k-1}{2}\right\rceil + Tdata \tag{9}$$

where $E_{min\_links}(m)$ is the expected minimum number of links between the memory module and the nearest participating reader (see Appendix A). The traffic caused by the concurrent reads does not present a problem, but the latency may.

With hierarchical read combining, the traffic and latency of m concurrent reads are both significantly reduced when m is large. To calculate the traffic, we must first make some assumptions regarding combining on the root ring. On all lower levels, we assume that the read requests arrive from below before the data from the first read request arrives from above (*ie:* full combining). Combining can take place on the root, as well, if multiple requests arrive at the memory module before it has retrieved the data for the first request. We will make the conservative assumption, however, that no combining takes place on the root ring. That is, that up to k requests (from the k subtrees at level $n-1$) arrive at the memory and are serviced sequentially. The traffic from the requests is now

$$T_{comb} = \sum_{i=1}^{n}\left[k^{n-i}T_{addr}(k-1)P_C(i-1,m)k\right] + \sum_{i=1}^{n-1}\left[k^{n-i}T_{MC}(P'_C(i,m),(k-1)P_C(i-1,m))\right]$$

$$+ (k-1)T_{MC}(Pc(i-1,m),Pc(i-1,m)) \qquad (10)$$

The first term corresponds to the requests propagating up in the tree. The second term corresponds to the data packets being transmitted on the root ring. The last term corresponds to data packets transmitted on lower levels. $T_{MC}(p,c)$ is the expected traffic from a multicast of a data packet on a single ring, where p is the probability that the multicast will have to be transmitted on the ring at all, and c is the expected number of recipients of the multicast that reside in the $(k-1)$ non-local (different than the processor initiating the multicast) processors on the ring. $T_{MC}(p,c)$ includes the traffic from the data packet as well as the ring echo, and is given by

$$T_{MC}(p,c) = pk\left[T_{addr}\left[\frac{1}{1+c/p}\right] + T_{data}\left[1-\frac{1}{1+c/p}\right]\right] \qquad (11)$$

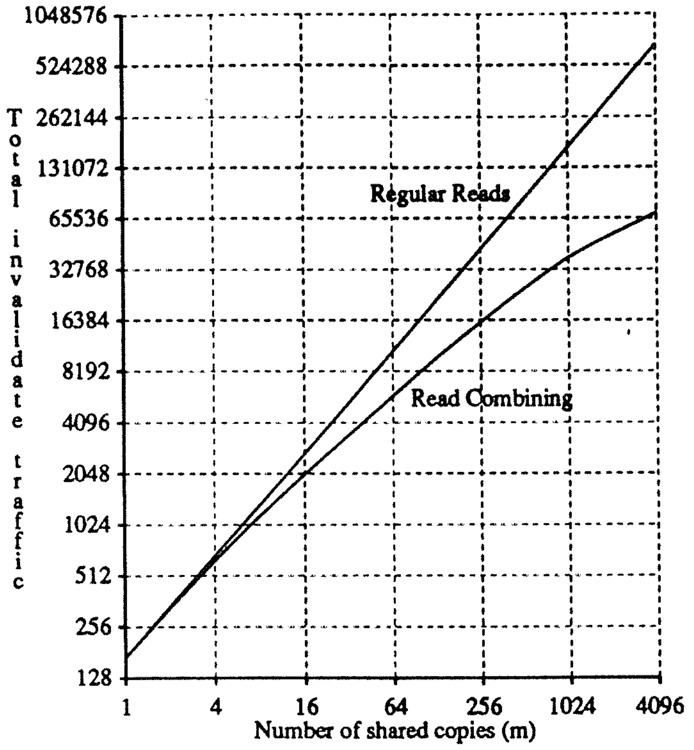
The latency of m concurrent read requests using read combining is given by

$$L_{comb} = E_{min\_links}(m) + E_{min\_dims}(m)L_{cache} + T_{addr} + max\,(L_{mem},T_{addr})(kP_C(n-1,m)-1)$$

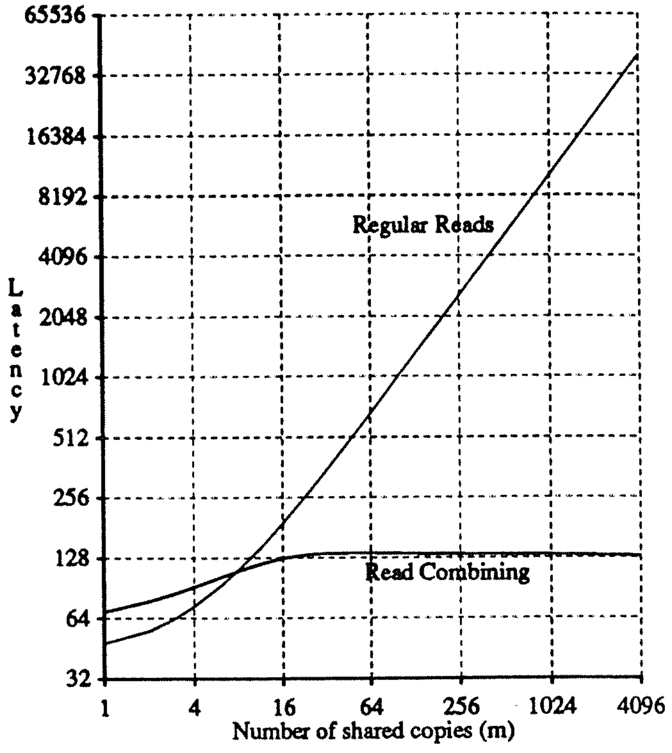
$$+ L_{mem} + E_{max\_links}(m) + E_{max\_dims}(m)L_{cache} + T_{data} \qquad (12)$$

where $E_{min\_links}(m)$, $E_{max\_links}(m)$, $E_{min\_dims}(m)$ and $E_{max\_dims}(m)$ are the expected minimum and maximum links and dimensions between the memory module and the participating readers (see Appendix A).

Figure 5 demonstrates the potential benefit of hierarchical read combining. We have assumed values of T_{addr}=1, T_{data}=9, L_{cache}=3 and L_{mem}=10. Note that the traffic with combining is always less than or equal to the traffic without combining, but that the latency with combining is sometimes greater than the latency without combining. This is due to the extra time needed to access caches on the way to and from memory. Thus, robustness in the presence of read contention comes at the price of performance in the presence of no contention. In an actual system, the relative difference between the two schemes for m=1 would be lessened, because both would suffer queueing delays for the links, which are not represented in figure 5(b). It may also be possible to overlap queueing delays with cache access, or even bypass the cache access entirely when there is no network contention.



(a) Total Read Traffic



(b) Latency for all Reads to Complete

Figure 5: Performance of Read Combining (N=4096, k=8, n=4)

3. SIMULATION STUDY

While the analysis in section two provides detailed performance metrics for certain isolated operations, it cannot provide overall system performance measures without a host of additional assumptions. Many of these assumptions, such as the pruning cache hit rate, would be quite difficult to make. In addition, the analysis does not address multi-level inclusion, and the effect of subtree invalidations on the hit rate of subsequent read requests. In order to address these and other issues, a detailed simulation study was performed. The goals were specifically to assess the effectiveness of pruning caches, to compare pruning caches against multi-level inclusion, and to validate our intuition about the scalability of such systems. All graphs of simulation results show a set of protocols executing a given workload with the performance metric along the y-axis and the system size along the x-axis. This gives an indication of how the protocols compare to each other, and how they are affected by increasing system size.

3.1. The Simulator

The simulator is driven by synthetic traces of second level cache accesses. Inter-reference times are are exponentially distributed, and represent the time spent processing and making first level cache hits. First level caches are assumed to contain subsets of the second level caches and to be write back. First level cache accesses and the interaction between the processor, first, and second level cache are not modeled, but contention for the second level caches between the processors and the network is modeled. Second level caches are 2-way set associative and consist of SRAM tag memory (3 cycle access) and DRAM data memory (10 cycle access), which can be accessed independently. A read hit requires 13 cycles while a write hit requires 3. Memory accesses are 10 cycles.

The interconnection network is a *k*-ary *n*-cube, with unidirectional, point-to-point links. Link transmission times are assumed to be one cycle. The line size is 64 bytes. Address packets are 8 bytes, and data packets 72 bytes. The link widths are 32 bits. The state of all cache lines in all data caches and the contents of all pruning caches and inclusion caches are explicitly maintained. The resulting memory demands dictated a second level cache size of only 128K. This is smaller than it would be in practice, but the workloads are similarly small.

3.2. Workload Characterization

The workload is characterized by some number of *segments*, each of which can have independent characteristics. For each segment, we specify the reference probability, size, maximum number of sharers, dimension along which sharers are arranged, read/write probabilities, and the spatial standard deviation of the second level cache references. Each processor has a virtual space consisting of its local segments, and the virtual spaces are woven together to form a single global physical address space. This system allows a large degree of flexibility in constructing traces with desired properties.

3.3. Protocols Simulated

Four different protocols were simulated. In all of the protocols, each line has a global state (maintained in the directory associated with memory) and each line residing in a cache has a local state. The global state is either shared or modified. If modified, memory has a pointer to the cache containing the line and memory's copy is invalid. If shared, memory's copy is valid, and memory has, in the case of pruning caches, the top level pruning vector, and in the case of MLI, the top level inclusion bit, stored in its directory. All four protocols treat write misses and accesses to private data in the same manner. Misses are routed directly to the home memory module, and the data is returned in modified state. If another cache has the current copy of the data, memory reroutes the request to that cache, which returns the data, forming a three leg transaction. Memory changes its pointer at the time the request is rerouted. The protocol is significantly complicated by certain race conditions, and is not given here in detail.

The four protocols differ in how they handle shared read misses, and write misses that find the data globally shared (requiring invalidations). Our base protocol (labeled *Direct* in figures 6-10) routes all read requests directly to memory, performing no read combining. Invalidations are broadcast to all processors, with acknowledgements hierarchically combined as described in section 2. The protocol labeled *Comb* uses read combining implemented with pending caches. Parent caches do not allocate space in their caches for children's read requests, but do maintain pending caches so that concurrent read requests can be combined. The *MLI* protocol enforces multi-level inclusion via inclusion caches. Parent caches allocate space in their cache for children's requests, and concurrent read requests are combined using the cache state. Invalidations are pruned using the inclusion information, and when entries are lost in the inclusion caches, the corresponding lines are invalidated in the subtrees beneath the caches. Lastly, the *PC* protocol uses pruning caches to prune invalidations, and performs read combining through the data caches as does MLI. Inclusion and pruning caches are 2-way set associative, using LRU replacement, and were simulated with sizes of 256 and 2048 entries.

3.4. Workloads Simulated

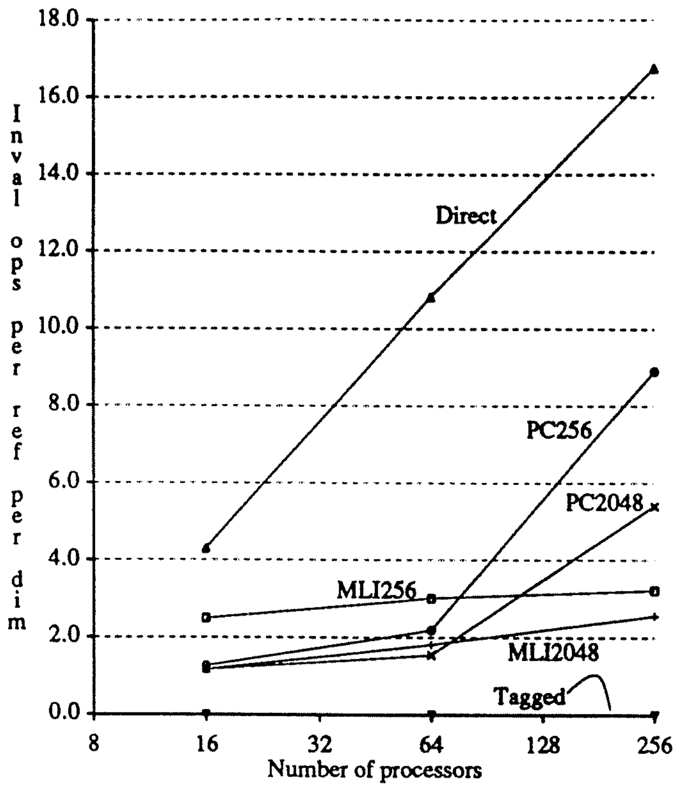
We simulated 5 different workloads in our experiments. All but the first workload included a 64K read-only code segment, an 8K private data segment (P_{write}=30%), and an additional data segment that determined the characteristic behavior of the workload.

- The first workload consisted of independent processes, each with its own 64K code segment and 128K data segment (P_{write}=10%). We simulated it with accesses marked as private to determine the scalability of the system for uniform traffic, and then with accesses non-marked (defaulting to shared), to compare how MLI and PC handle large data spaces with no active sharing, and to observe the performance degradation due to broadcast invalidations.
- The second workload used a fixed 512K segment shared among all processors. P_{write} was fixed at 10%, so contention for the data increased as system size increased.
- The third workload used a fixed 512K shared segment also, but scaled the write probability down with increased system size (P_{write}=$4/N$). A workload where most processors read a line between successive writes to the line would exhibit the same sort of behavior.
- The fourth workload used 4-way sharing with a local segment size of 64K. Each line was contained in the virtual space of four different processors, for a total global physical segment size of $N(16K)$. Sharing was arranged along the root dimension, which spreads out sharers among different leaf rings.
- The fifth workload used 4-way sharing, but with sharers arranged along the leaf dimension, which concentrates sharers on the same leaf rings as much as possible.

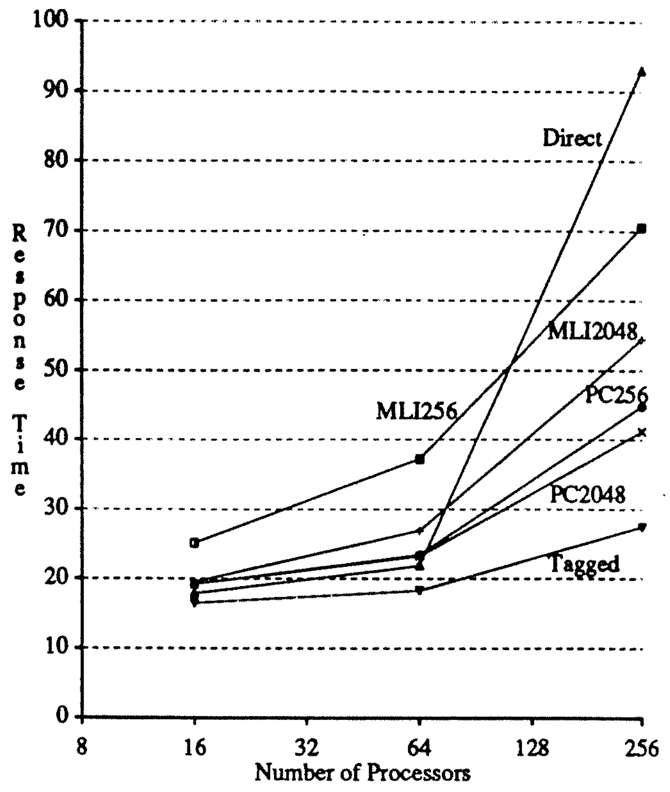
3.5. Results

Figure 6 shows results for independent workloads. For the curve marked *Tagged*, all requests are correctly tagged as private, and all protocols give the same performance. There are no global invalidations necessary, and we see in figure 6(b) that the cache response time increases only mildly as system size is increases from 16 processors (two dimensional) to 256 processors (four dimensional). This supports the claim that uniform traffic scales well in a cube network. When private accesses are not marked as such, however, performance degrades for all protocols. The normalized invalidation traffic soars for Direct, supporting our assertion that traffic from broadcast invalidations does not scale. Since the workloads are completely independent (and fairly large compared to the cache size), the inclusion and the pruning caches are not nearly large enough to maintain high hit rates. We see that this affects them in different ways, however. Invalidation traffic is fairly high for PC256. This is most likely due to misses for high-level pruning vectors, causing subsequent invalidations to be broadcast to many processors. MLI256 keeps the invalidation traffic lower than PC256, because it never has to broadcast invalidations, but it results in much higher cache response time due to misses caused by subtree invalidations. With sizes of 2048 entries, the pruning and inclusion caches perform much better, but not perfectly. We still see that the response time for PC2048 is significantly lower than the response time for MLI2048. This illustrates an important advantage of pruning caches over inclusion caches. When we can no longer keep track of a line, it is better to suffer increased invalidation traffic when the line is written than to prematurely invalidate the line.

If the data caches were big enough to hold the data that was marked as shared, but was in fact not actively shared, then pruning caches would have an additional advantage over inclusion caches. Entries for these lines



(a) Invalidate traffic



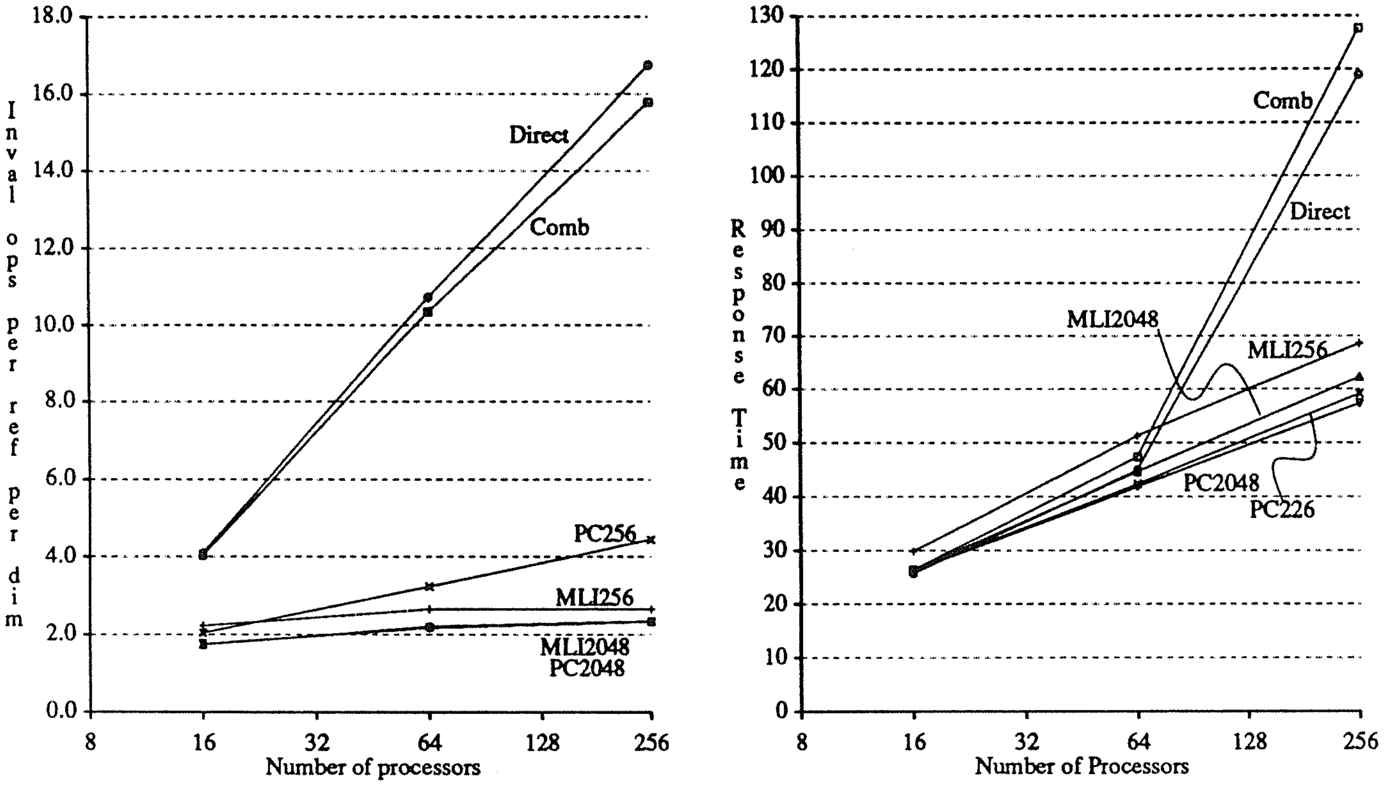
(b) Cache response time

Figure 6: Protocol Scalability
Independent processes

would drop out of the pruning caches, and the lines could remain in the data caches below. MLI, however, would require entries for these lines to reside in the inclusion caches, and thus would repeatedly invalidate lines as it replaced inclusion cache entries.

Figure 7 shows results for a fixed problem size with constant write probability. As system size increases, contention for data increases, but the sharing is very active and the average number of shared copies on an invalidation remains small. The normalized invalidation traffic for Direct and Comb, which use broadcast invalidations, increases dramatically with system size. By N=256, this traffic has caused the response time to twice that of the other schemes. Read combining does not take place much in this workload, and we see that Comb has a higher cache response time than Direct, due to the extra delay in accessing memory.

PC and MLI both reduce the invalidation traffic significantly over Direct and Comb. Unlike the previous workload, however, MLI does not have a significantly higher cache response time than PC. This is due to the active sharing, which reduces the demands on the inclusion caches and lessens the effect of subtree invalidations.



(a) Invalidate traffic

(b) Cache response time

Figure 7: Protocol Scalability
Fixed problem size, P_W=15%

Figure 8 shows results for a fixed problem size with a write probability that decreases with system size. This behavior corresponds to inactive write sharing, and might arise in a workload where processors repeatedly read an entire data set to write a small subset. Direct and Comb, which use broadcast invalidation, are not penalized for this because the frequency at which processors modify shared data decreases with system size. This same characteristic would allow such a workload to run efficiently on a single-tree-based multiprocessor.

The inactive nature of the sharing penalizes MLI, however. Pruning cache entries for inactive lines can be replaced without harm, but inclusion cache entries that are replaced cause invalidations that increase the data cache miss rate and drive up cache response time.

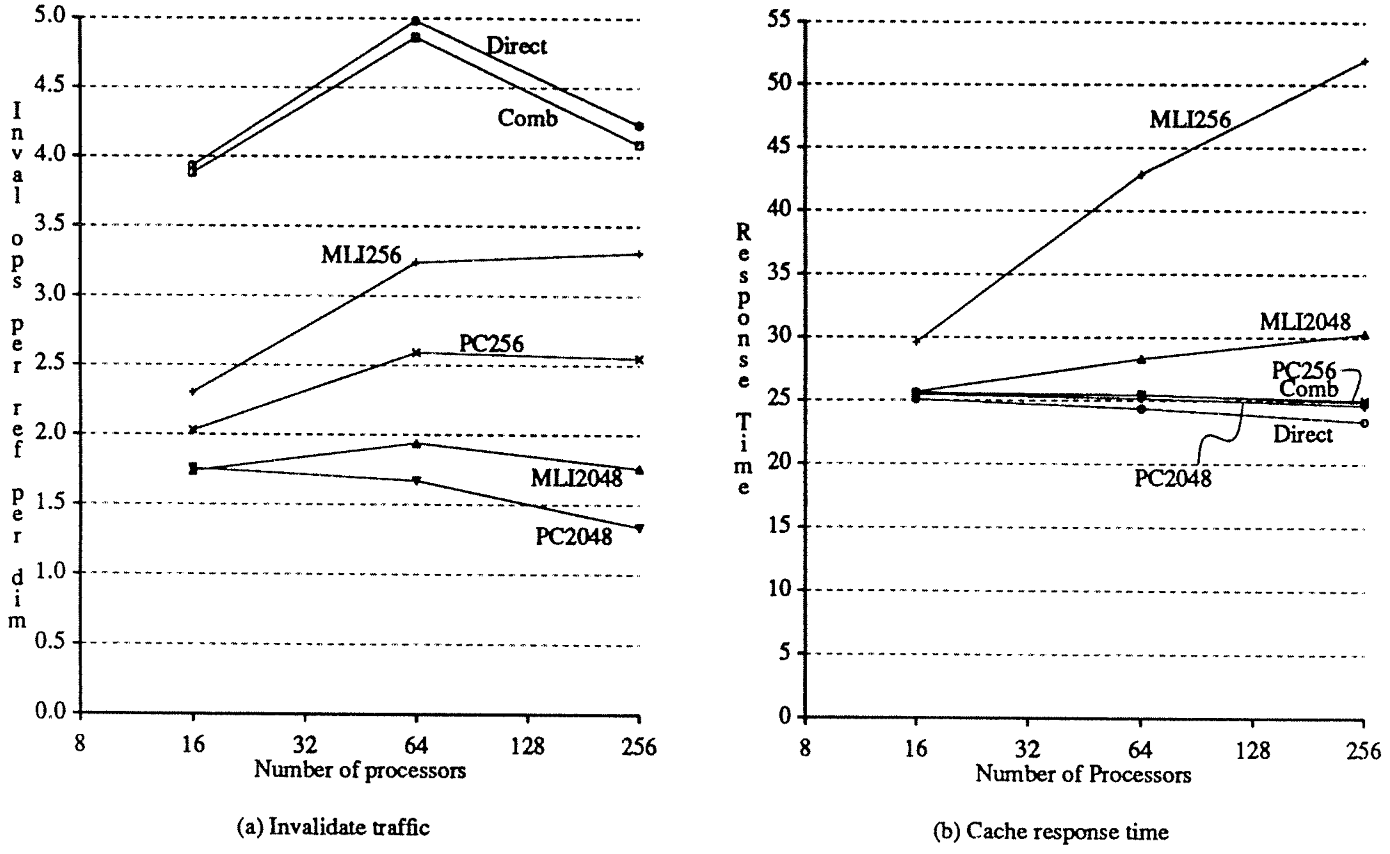
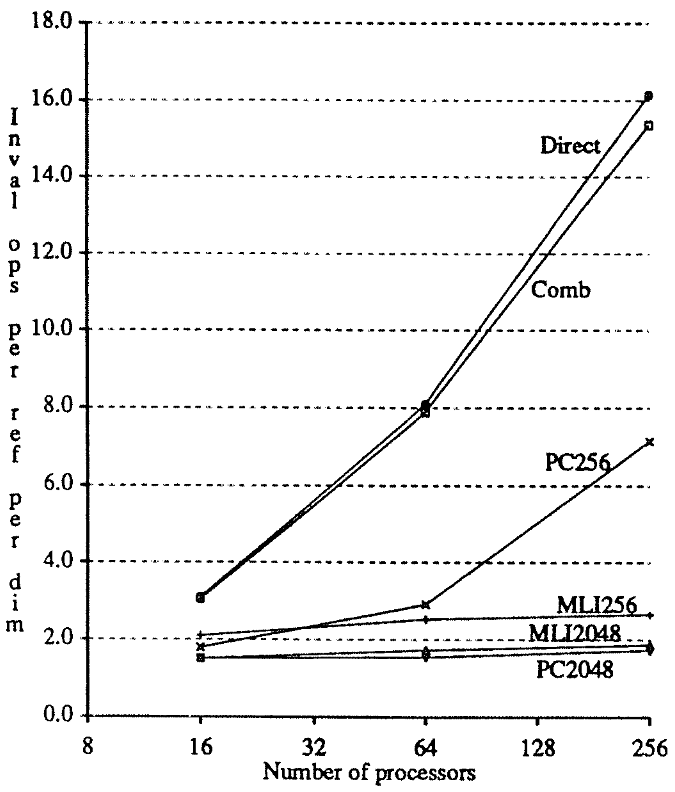


Figure 8: Protocol Scalability

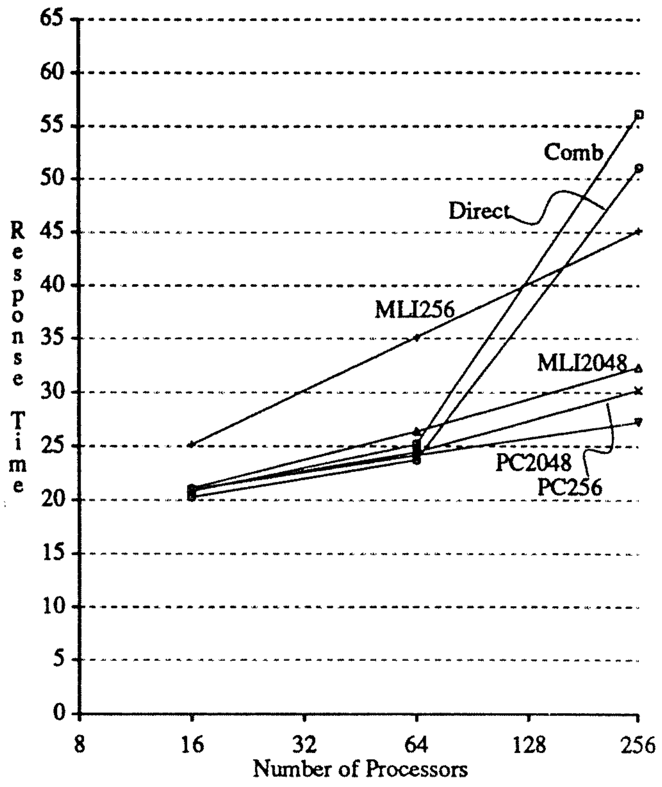
Fixed problem size, $P_{write}=\frac{4}{N}$

The workload presented in figure 9 scales in size with the number of processors. All sharing is 4-way and distributed along the root dimension (the worst-case distribution for inclusion and pruning caches). Since the local segment sizes and degree of sharing are independent of system size, changes in performance as system size increases should be due solely to scaling effects. The large address space and bad distribution result in a poor hit rate for PC256, resulting in heavy invalidation traffic for N=256. PC2048 maintains a good hit rate for all system sizes, and so invalidation traffic remains low. Although the 256-entry inclusion caches are also too small, it is difficult to tell from the invalidation traffic. It is readily apparent, however, by looking at the cache response times, which are significantly higher for MLI256 than for MLI2048, PC256 and PC2048.

Traffic caused by broadcast invalidations causes Direct and Comb to perform very poorly for N=256. Again we see that the combining mechanism did not improve performance, but rather worsened it by increasing the latency to memory for shared read requests.



(a) Invalidate traffic



(b) Cache response time

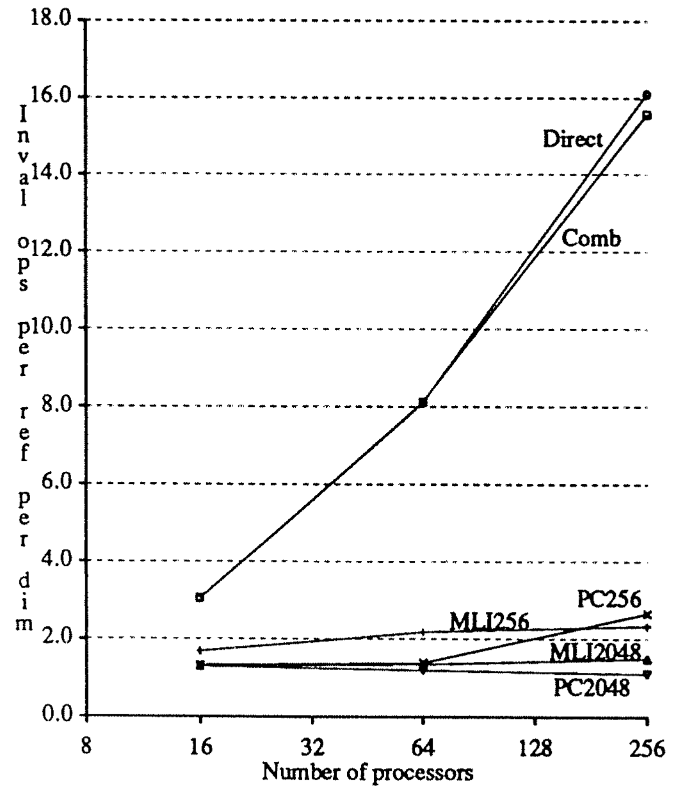
Figure 9: Protocol Scalability
4-way sharing (root dimension), Problem size αN, P_{write}=15%

The workload in figure 10 is identical to that in figure 9, save that the 4-way sharing is now arranged along the leaf dimension (the best-case distribution for inclusion and pruning caches). The performance of Direct is unchanged, but the performance of all other schemes improves. The response time of Comb lowers slightly, indicating that additional read combining is taking place. Comb still performs worse than Direct, however; there is not enough read combining in this workload to make up for the increased read latency.

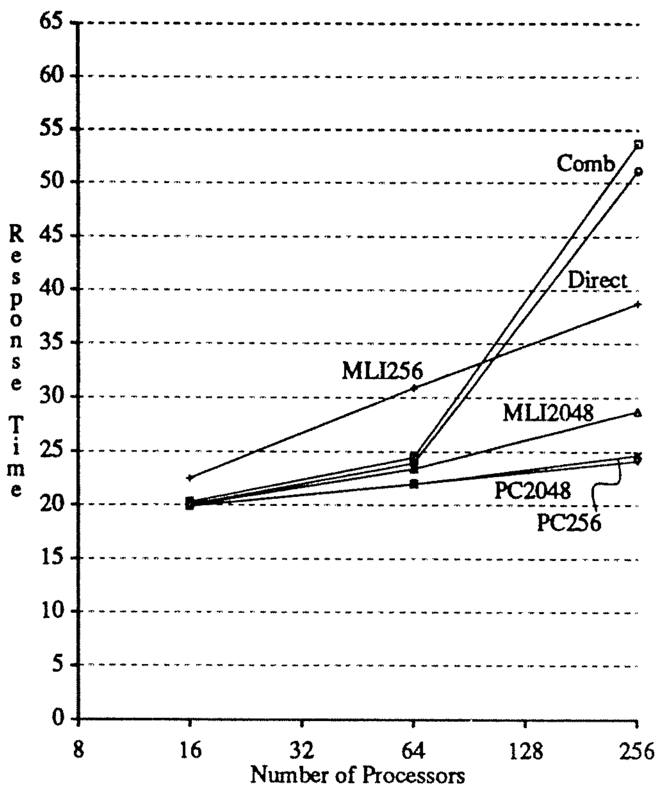
The better distribution has a positive effect on the performance of pruning and inclusion caches as well. Invalidation traffic is lower than with the bad distribution, as is cache response time. The response times of PC are still slightly lower than those of MLI, even though both the pruning and inclusion caches have very low replacement rates.

4. SUMMARY

We have shown, through both analysis and simulation, that the *k*-ary *n*-cube topology provides scalable, bottleneck-free communication for uniform, point-to-point traffic. Since operations requiring broadcast do not scale, we have investigated pruning caches as a method to exploit the benefits of broadcasting while attempting to



(a) Invalidate traffic



(b) Cache response time

Figure 10: Protocol Scalability
4-way sharing (leaf dimension), Problem size αN, P_{write}=15%

limit communication to the subset necessary to notify all interested parties of an operation. Pruning caches store information about where data does *not* reside, so the loss of a pruning cache entry simply results in extra broadcast traffic when (and if) the corresponding data is invalidated. In contrast, inclusion caches guaranteeing the Multi-Level Inclusion (MLI) property, store information about where a line *does* reside. When an inclusion cache entry is lost, the corresponding data must be invalidated in the subtree beneath the cache. We have compared pruning caches against inclusion caches and found that pruning caches are much more cost-effective: they can provide higher performance while storing less information.

Pruning caches perform more effectively than inclusion caches because the penalty for increased invalidation traffic is less than the penalty for prematurely invalidating data. Pruning caches, unlike inclusion caches, allow data that is shared but rarely modified (such as most segments of shared code) to remain in caches even though the directory information may be purged. Data which is declared shared but is not actively shared may also remain in caches while the pruning cache entries are purged. This is a potentially large class of data, including, for example, interior data in a partitioned grid algorithm.

We have shown that pruning caches provide robust performance over a broad spectrum of workloads, consistently providing better performance than an equivalent MLI scheme. While utilizing a simple directory scheme, they can nevertheless exploit the limited broadcast inherent in bus- or ring-based topologies, much as snooping cache protocols do. We conclude that pruning caches can be employed to build a scalable system for a broad range of workloads.

With the use of pruning caches, we found that performance is improved, but not greatly, by carefully distributing the data, suggesting that a major headache for the programmer, mapping the algorithm to the architecture, might be of minor importance. We also found little benefit from read-combining for the workloads investigated. We do not conclude, however, that such features are useless, since this study ignored synchronization effects in the workload. A problem that features barriers, for example, might benefit greatly by such read combining.

Acknowledgements

We would like to thank Phil Woest and Mark Hill for the original suggestion of pruning caches. We would also like to thank Mary Vernon, Ross Johnson and other members of the Wisconsin Multicube group for their comments and dicussion regarding this work.

References

[Adve90] Adve, S. V. and M. D. Hill, Weak Ordering - A New Definition, *Proc. 17th Annual International Symposium on Computer Architecture*, May 1990, 2-14.

[Agar90] Agarwal, A., Limits on Network Performance, *Submitted to IEEE Transactions on Parallel and Distributed Systems*, 1990.

[Baer88] Baer, J.-L and W.-H. Wang, On the Inclusion Properties for Multi-Level Cache Hierarchies, *Proc. 15th Annual International Symposium on Computer Architecture*, June 1988, 73-80.

[Bell85] Bell, C. G., Multis: a New Class of Multiprocessor Computers, *Science* **228**, April 1985, 462-467.

[Cens78] Censier, L. M. and P. Feautrier, A New Solution to Coherence Problems in Multicache Systems, *IEEE Transactions on Computers* **C-27**(12), December 1978, 1112-1118.

[Dubo86] Dubois, M., C. Scheurich, and F. Briggs, Memory Access Buffering in Multiprocessors, *Proc. 13th Annual International Symposium on Computer Architecture*, June 1986, 434-442.

[Good88] Goodman, J. R. and P. J. Woest, The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor, *Proc. 15th Annual International Symposium on Computer Architecture*, June 1988, 422-431.

[Good89a] Goodman, J. R., M. K. Vernon, and P. J. Woest, Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors, *Proc. ASPLOS III*, April 1989, 64-75.

[Good89] Goodman, J. R., M. D. Hill, and P. J. Woest, Scalability and Its Application to Multicube, Computer Sciences Technical Report #835, University of Wisconsin-Madison, Madison, WI 53706, March 1989.

[Gott87] Gottlieb, A. and et al, The NYU Ultracomputer -- Designing a MIMD, Shared Memory Parallel Machine, *IEEE Transactions on Computers* **C-32**(2), April 1987, 175-189.

[IEEE90] IEEE,, Scalable Coherent Interface, Logical Specification, IEEE Standard Specification P1596: Part II, November 1990.

[Lam79] Lam, C.-Y. and S. E. Madnick, Properties of Storage Hierarchy Systems with Multiple Page Sizes and Redundant Data, *ACM Transactions on Database Systems* **4**(3), September 1979, 345-367.

[Lamp78] Lamport, L., Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM* **21**(7), July 1978, 558-565.

[Mell91] Mellor-Crummey, J. M. and M. L. Scott, Synchronization Without Contention, *Proc. ASPLOS IV (to appear)*, 8-11 April 1991.

[Mudg87] Mudge, T. N., J. P. Hayes, and D. C. Winsor, Multiple Bus Architectures, *Computer* **20**(6), June 1987, 42-48.

[Pfis85] Pfister, G. F. and et al, The IBM Research Parallel Processor Prototype (RP3): introduction and architecture, *Proc. 1985 International Conference on Parallel Processing*, August 1985, 764-771.

[Scot91] Scott, S. L., A Cache Coherence Mechanism for Scalable, Shared Memory Multiprocessors, *Proc. 1991 International Symposium on Shared Memory Multiprocessors (to appear)*, April 1991.

[Wils87] Wilson, A. W., Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors, *Proc. 14th Annual International Symposium on Computer Architecture*, June 1987, 244-252.

[Wins88] Winsor, D. C. and T. N. Mudge, Analysis of Bus Hierarchies for Multiprocessors, *Proc. 15th Annual International Symposium on Computer Architecture*, June 1988, 100-107.

[Yew87] Yew, P.-C., N.-F. Tzeng, and D. H. Lawrie, Distributing Hot-Spot Addressing in Large Scale Multiprocessors, *IEEE Transactions on Computers* **C-36**(4), April 1987, 388-395.

Appendix A - Miscellaneous Formulas

$$P_{all\_in}(m,x) = \begin{cases} \prod_{i=0}^{m-1} \left(\frac{x-m}{k^n-m} \right) & \text{if } m \leq x \\ 0 & \textit{otherwise} \end{cases}$$

$$P_{none\_in}(m,x) = P_{all\ in}(m,k^n-x)$$

$$N_d(x) = \sum_{i=0}^{x} b(n,i)(k-1)^i$$

$$N_l(x) = \sum_{d_1=0}^{k-1} \sum_{d_2=0}^{k-1} \cdots \sum_{d_n=0}^{k-1} \delta(\sum_{j=1}^{n} d_j \leq x)$$

$$E_{min\_dims}(m) = \sum_{i=1}^{n} i \left[P_{none\_in}(m,N_d(i-1)) - P_{none\_in}(m,N_d(i)) \right]$$

$$E_{max\_dims}(m) = \sum_{i=1}^{n} i \left[P_{all\_in}(m,N_d(i)) - P_{all\_in}(m,N_d(i-1)) \right]$$

$$E_{min\_links}(m) = \sum_{i=1}^{n(k-1)} i \left[P_{none\_in}(m,N_l(i-1)) - P_{none\_in}(m,N_l(i)) \right]$$

$$E_{max\_links}(m) = \sum_{i=1}^{n(k-1)} i \left[P_{all\_in}(m,N_l(i)) - P_{all\_in}(m,N_l(i-1)) \right]$$