A CACHE COHERENCE MECHANISM FOR SCALABLE, SHARED-MEMORY MULTIPROCESSORS

by

Steven L. Scott

Computer Sciences Technical Report #1002

February 1991

A Cache Coherence Mechanism For Scalable, Shared-Memory Multiprocessors

Steven L. Scott

Department of Computer Sciences
University of Wisconsin - Madison
1210 West Dayton Street
Madison, WI 53706
(608) 262-6614
sls@cs.wisc.edu

Abstract

In order to build very-large-scale multiprocessors while retaining high processor efficiencies, a *scalable* design must be used. While intuitively clear, there is no consensus on the exact meaning of scalability. We provide a useful, working definition of scalability that lends insight into the suitability of a design for implementation with a very large number of processors. We then propose a topology (the *k*-ary *n*-cube) and a cache coherence mechanism (*pruning cache* directories) for use in large scale systems. Using scalability arguments, we demonstrate that a system based on the *k*-ary *n*-cube topology and pruning caches, will scale properly to very large numbers of processors.

Steven Scott was supported by a fellowship from the Fannie and John Hertz Foundation.

1. INTRODUCTION

Very-large-scale multiprocessors offer the possibility of enormous computing power. However, the straight forward extension of existing system configurations to large numbers of processors can lead to significant per-processor performance degradation. Intuitively, these systems need to be *scalable*, in order to be efficient with large numbers of processors. Scalability, however, is not a well defined property. Though the term is widely used, there is no consensus as to its meaning. We claim that while a precise (but useful) definition of scalable may not be possible, a useful working definition can be given. We find that "scalable" is generally used to mean "suitable for very large systems". This requires that the system avoid bottlenecks that preclude implementations with large numbers of processors.

Scalability is not an end goal, in itself. For a given implementation, cost and performance are the primary concerns. However, as system size increases, a scalable design will eventually provide higher performance than a non-scalable design. For example, a ring interconnect may provide higher performance than a 3-dimensional mesh for small system sizes, due to smaller switching delays. However, the traffic over a link in the ring (assuming uniform communication) is proportional to the system size, while the traffic over a link in the mesh is proportional to the cube root of the system size. A 3-dimensional mesh, therefore, will perform better than a ring for sufficiently large systems. Even a 3-dimensional mesh will saturate eventually, however. We are interested in designs that can be efficiently implemented with thousands, or tens of thousands, of processors. How a design scales can lend valuable intuition as to its behavior for such large system sizes.

This paper focuses on shared-memory multiprocessors with hardware-maintained cache coherence. It is widely agreed that these machines present a convenient abstraction to the programmer, and, while not the only choice, they are at least an important class of multiprocessors. The bottlenecks in such systems can be due to software, hardware (communication topology) or the communication protocol (including the cache coherence mechanism). As a minimum, the software must have a parallel component that can be grown while keeping the serial component fixed [Gust88]. However, the three components cannot always be considered in isolation, as their interaction can significantly affect scalability. This will be addressed more fully in later sections of the paper.

In this paper, we suggest a topology and coherence mechanism that we believe is appropriate for very large systems. The notion of scalability is developed in order to gain insight into how this and other systems will behave as system sizes increase. In section 2, we present a working definition of scalability, complete with several caveats concerning its meaning and application. In section 3, we examine the scalability of multiprocessor topologies, and choose a topology (the k-ary n-cube) for use as a framework for discussing cache coherence mechanisms. In section 4, we examine the scalability of various cache coherence mechanisms. We propose a novel mechanism (*pruning caches*) for maintaining hierarchically-distributed directories that has excellent scaling properties. Finally, concluding remarks are given in section 5.

2. SCALABILITY

As was mentioned in the introduction, there is no consensus on a precise definition for scalability. Amdahl [Amda67] originally pointed out that the efficiency of a parallel computer will decrease as more processors are used to solve a fixed-size problem. This is due to the (reasonable) assumption of a fixed serial portion of the algorithm. As the number of processors executing the parallel portion of the algorithm increases, the relative time spent doing the serial computation increases. This eventually dooms attempts to speed up the execution of a program by using more processors. Gustafson [Gust88] suggested that instead of increasing the number of processors used to solve a fixed-size problem, the execution time should be held constant and the parallel portion of the problem should be increased with the number of processors. We agree that this is a necessary condition to maintain high processor efficiency as system size increases.

There have been several suggestions as to what scalability means. Patton [Patt85] stated that a scalable design "can be adjusted up or down in size without loss of functionality to scale effects". Goodman, et al [Good89], define a scalable *algorithm* as one whose serial portion does not grow with problem size and whose parallel portion contains parallelism at least proportional to the algorithm's complexity. They define a scalable *system* according to the speed at which it executes a scalable algorithm. Johnson [John90] presents a rigorous set of definitions for scalable hardware which is independent of the workload. Using asymptotic behavior, he defines 11 classes of scalability, including *uniformly, architecturally* and *implementationally scalable*. Hill [Hill90], on the other hand, questions whether scalability can be usefully defined at all. He challenges the technical community to either define the term rigorously, or stop using it altogether. Still others [Leno90, Cher89, Hage89] use the term without accompanying definition.

We believe that a rigorous definition of scalability may be of little use, but that we *can* arrive at a useful working definition. Several qualifications need to be offered at the start, however. First, scalability is not a simple binary property. Certain designs may be more or less scalable than others, however that does not preclude some designs from being clearly not scalable. Second, scalability, in and of itself, says nothing about the cost or performance of a given, fixed-size system. It is useful primarily for providing insight into the behavior of a system as the number of processors grows very large. Third, even though we are interested in the performance for very large systems, asymptotic behavior of a system may not be of any interest. The state of the art is pushing thousands of processors. The behavior of systems with billions of processors is simply not relevant for the foreseeable future.

2.1. A Working Definition of Scalability

We define scalability in terms of three metrics: cost, latency, and bandwidth. Each of these can be approximated as growing by some order of N, the number of processors in the system.

Cost: The cost of the system is measured in terms of the required hardware. A full crossbar interconnect, for instance, requires $O(N^2)$ switching elements. An Omega network requires $O(N \log N)$ switching elements. A ring requires only $O(N)$ switching elements. Another effect on the cost metric is the size of memory needed to store directory information. If a directory scheme requires that every line of memory be accompanied by a tag that is proportional to N in size, then the tag memory must grow as $O(N^2)$ (we assume that the size of main memory grows linearly with N). Some costs should be ignored, as they are practically constant for all but truly asymptotic behavior. For instance, the number of bits used to specify a processor grows as $O(\log N)$. But if a single word (32 bits) is used, this effect is not seen until the number of processors exceeds 2^{32}. This sort of behavior can be ignored for scalability considerations. For a system to be considered scalable at all, we require that the cost be less than $O(N^2)$. By this measure, a full crossbar is not considered scalable. Beyond that, a system may scale better or worse than another regarding cost. For example, a 2-dimensional mesh scales better in cost ($O(N)$) than an Omega network ($O(N \log N)$).

Latency: The latency metric is the *average* latency of a memory request. This is affected by the topology of the system, and can also be affected by the communication protocol and the workload. Ideally, we would like the average latency to remain $O(1)$ as system size increases. Realistically, this is not possible for more than incremental growth of the system. Several interconnection networks (such as the Omega network) provide $O(\log N)$ latency upon first approximation. Eventually, however, wire lengths grow, and latency must increase at least as $O(N^{1/3})$, due to propagation of signals in 3-dimensional space. A three dimensional cube topology, which can be implemented with constant length wires, provides latency of $O(N^{1/3})$.

We argue, however, that asymptotically, the latency of any system must increase as at least $O(N^{1/2})$. Consider an idealized sphere of processors with radius r. The sphere contains $O(r^3)$ processors. The communication distance between any two processors is $O(r)$. The bisection area of the sphere is

$O(r^2)$. The traffic across the bisection (assuming uniform communication) is proportional to the number of processors, or $O(r^3)$. Thus the traffic density across the bisection increases as r. This cannot be supported asymptotically. If the traffic density is kept constant, then the sphere may only be packed with $O(r^2)$ processors. The communication latency is then $O(N^{1/2})$. As an example of this phenomenon, the Tera Computer System populates a 3-dimensional mesh of size r^3 with only r^2 processors [Alve90].

Considering this, the tightest absolute requirement that we can impose is that latency cannot grow faster than $O(N^{1/2})$. Even for very large systems, however, we may be able to ignore the asymptotic latency of a design, so certain topologies may scale better than others ($O(N^{1/3})$ versus $O(N^{1/2})$, for instance). Recall, also, that for a given size (even a very large one) a specific system may perform better than another, even if the other has better scaling properties.

Bandwidth: The bandwidth metric is the maximum of the average traffic over each wire and through each node resulting from every processor issuing a single memory request. The traffic can be caused by the requests themselves, by associated data, or by extra traffic generated by the cache coherence mechanism (to retrieve data from another processor or to invalidate lines after a write, for instance). For limited scalability, a system may be able to handle $O(N^{1/2})$ or $O(N^{1/3})$ traffic, but eventually this will saturate the available bandwidth. Thus, we require the traffic to be $O(\log N)$ or less in order for the system to be considered scalable.

2.2. The Effect of Software on Scalability

The three scalability metrics are affected by a combination of hardware, software and the communication protocol. The default assumption about software is the *uniform workload* model. The processors make a uniform distribution of requests to different memory modules. In addition, the percentage of accesses that are writes to shared variables remains fixed as system size increases. This means that the number of invalidations per request per processor remains roughly constant. If the scalability of a topology is discussed without mention of the software, it is the uniform workload model that is being assumed.

Under a *hot-spot workload*, processors make a higher than average fraction of their requests to a particular memory location or module. This will prevent a system from scaling unless the hardware or communication protocol takes explicit steps to handle it. Concurrent *read* requests are expected to occur frequently, such as after a barrier synchronization or after a widely read variable is invalidated. Concurrent write requests are not expected to occur often.

In contrast to the hot-spot workload, a *conspirator workload* can make a multiprocessor scalable that otherwise would not be scalable. For example, if the workload has high geographic locality of communication that matches the physical layout of the system, and the communication protocol allows for localized communication, then the average message will traverse only a fraction of the distance across the system, and an otherwise non-scalable system may scale. Another example of a conspirator workload is one whose fraction of writes to shared variables decreases with system size. This makes the rate of invalidations per request per processor decrease as system size grows larger. This sort of workload occurs, for instance, when shared variables are read by all processors between successive writes.

3. A SCALABLE TOPOLOGY

Using our definition, it is easy to identify certain topologies as non-scalable. A ring will not scale (for all but extreme conspirator workloads) because both the traffic across each link and the average communication latency increase as $O(N)$. A single shared bus will not scale, because the traffic on the bus increases as $O(N)$. In addition, though it may not be immediately apparent, the latency of shared bus accesses increases with N (apart from the queueing delay). This is because the physical length of the bus, as well as its capacitance, increases with N. This suggests an important point: a ring of point-to-point links can be used to simulate a bus. Both have the same scaling properties, but point-to-point links can be clocked at a higher rate (independent of N), leading to higher performance. A broadcast operation is simulated by passing a message completely around the circle. This does not provide global event ordering, however, so modifications are necessary for snooping protocols that depend on this.

One topology that has been widely studied [Wils87, Wins88, Vern89, Baer88] for large numbers of processors is the bus hierarchy (see figure 1). This topology will not scale, however, for uniform workloads. Each time a line is written, if a copy exists in any of the other primary subtrees of the hierarchy, an invalidate or other message will have to travel over the root bus. Thus the traffic over this bus is $O(N)$, and the system does not scale. For a conspirator workload, where the fraction of writes to shared variables decreases with N, the traffic over the root bus can remain $O(1)$. A recent analysis of hierarchical bus-based multiprocessors [Vern89] concluded that the systems scale well only if the average fraction of processors that read a shared line between writes, f_{RI}, remains large as system size increases. This can only be accomplished with the conspirator workload mentioned above. If the fraction of writes to shared variables, f_{WS}, remains constant, then the average number of processors that read a shared line between writes is limited to $\frac{1-f_{WS}}{f_{WS}}$. Thus, f_{RI} is limited to $\frac{1-f_{WS}}{Nf_{WS}}$, which decreases with system size (the paper accidentally used a workload where both f_{WS} and f_{RI} remained fixed as system size increased). Their analysis showed that when f_{RI} is small, the system is extremely inefficient for large numbers of processors (due to contention for the root bus).

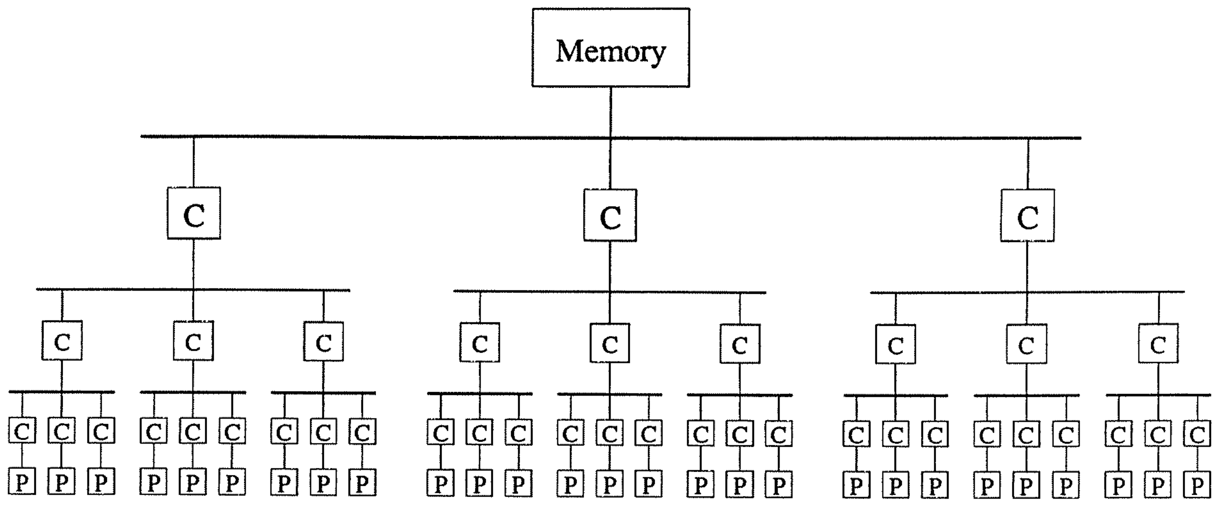
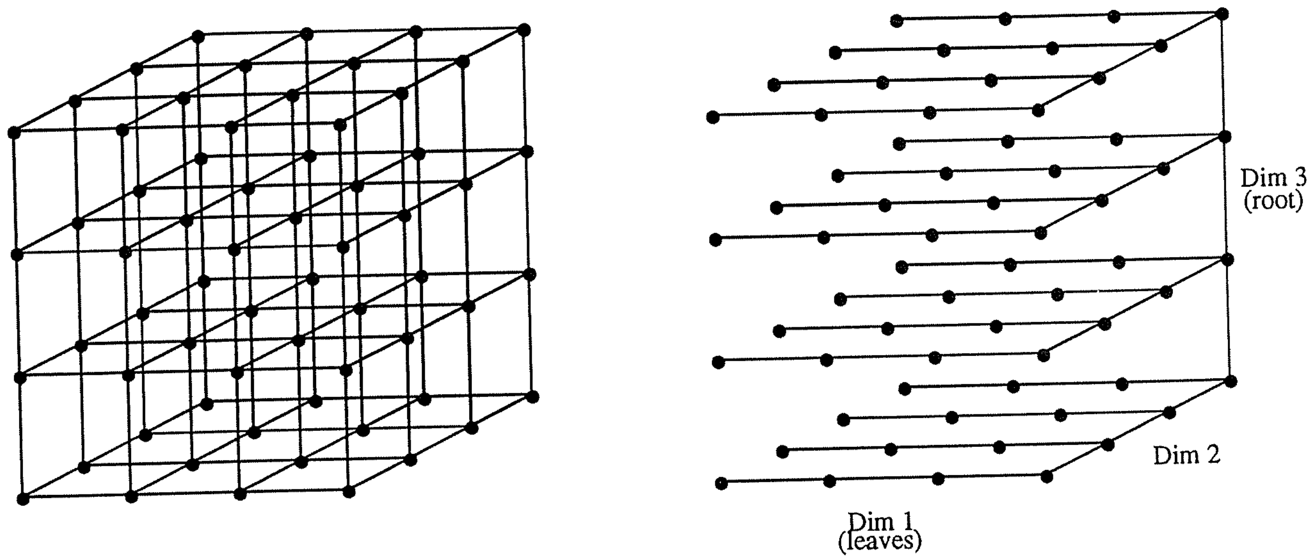


Figure 1: A Hierarchical Bus-based Multiprocessor

Another widely advocated topology is the *k*-ary *n*-cube. Rings, 2-dimensional toruses, 3-dimensional toruses and hypercubes are all sub-classes of this topology. The *k*-ary *n*-cube has *n* dimensions, with *k* processors in each dimension, for a total of $N = k^n$ processors. For example, a 2-dimensional torus with 64 processors is a 8-ary 2-cube. While a torus implies point to point links, an *k*-ary *n*-cube can also be implemented with buses (with *k* processors per bus and *n* buses per processor). Figure 2 shows a 4-ary 3-cube, implemented with buses. The buses may either be actual buses or logical buses implemented with rings of point-to-point links.

The *k*-ary *n*-cube is scalable, according to our definition, if it is grown in the proper way. The cost of the system is O(Nn), where $n = \log_k N$. There are N processors and $\frac{Nn}{k}$ buses (Nn links). The average distance between two processors is O(n) bus hops (O(nk) link hops). Thus, assuming uniform traffic, the latency of a communication request is O(nk), and the traffic over a bus or link is O(k). The cost and latency metrics do not preclude scaling, by our definition, regardless of how the system is grown. The bandwidth requirement, however, can only be met by increasing the dimensionality (n). Thus a fixed dimension network (a 2-dimensional torus, for instance), is not scalable, due to increasing link traffic.

Dally [Dall90] concludes that for optimum performance, both *k* and *n* must be increased as system size increases (in fact, he increases *k* primarily). His work assumes, however, that the network is being implemented on a single VLSI chip, and is thus limited by wiring density. He constrains the bisection



(a) The complete 3-cube

(b) One of 16 distinct trees

Figure 2: A 4-ary 3-cube Multiprocessor (N=64, k=4, n=3)

Each processor is connected to n buses, and each bus is connected to k processors (if implemented with point-to-point links, each processor would have $2n$ links, one incoming and one outgoing in each dimension, and each "bus" would be a ring of k processors). 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 busses is formed as shown in 2(b). This tree is equivalent to the one shown in figure 1, given that each parent node (those in the rightmost plane of figure 2(b)) is allowed to be one of its children. This allows communication protocols based on a hierarchical topology to be implemented.

width of the network to be held constant for a given system size as the dimensionality of the network is chosen. The constant bisection constraint causes the width of the links to be substantially reduced as the dimensionality increases for a given system size. For a multiple-board-level multiprocessor, however, a constant pinout per processor constraint may be more realistic. Agarwal [Agar90] studies k-ary n-cubes with the constant pinout per processor constraint, and also models processing overhead in the switching elements, which Dally's work did not. He does not specifically address how a system should be grown, but finds that higher dimensional systems should be chosen under the constant pinout per node constraint than under the constant bisection constraint. He further finds that heavier traffic calls for higher dimensionality. This agrees with our observation that the traffic over a link is O(k) and supports the claim that the systems should be grown by increasing n.

4. A SCALABLE DIRECTORY FOR CACHE COHERENCE

The basic function of a cache coherence mechanism is to make sure that after a memory location is modified, subsequent reads of that location return the new (modified) value. Unfortunately, in a multiprocessor system, it is difficult to specify the exact time of a write; because of nondeterministic delays to access memory, requests to different locations and from different processors may complete in a different order than that in which they were issued. In addition, as system sizes increase, so does the interval of time between when an event occurs, and when that event can be observed in other parts of the system. The cache coherence mechanism need only make the following two guarantees. First, writes to a given memory location by a given processor, cannot be observed by any processor to occur out of program order. Second, there must exist some global ordering of writes to a given memory location, such that no processor observes any other order.

In addition to cache coherence, a multiprocessor may guarantee some level of *sequential consistency*, which makes some guarantee about the global ordering of writes to *different* memory locations [Lamp78, Dubo86, Adve90]. Providing sequential consistency can significantly impact the cache coherence mechanism. In order to make any assertions concerning a global ordering of writes, the system must be able to determine when all processors have seen the new value of a write (or at least can no longer see the old value). This paper does not address the issue of sequential consistency other than by placing this constraint upon the cache coherence mechanism.

We assume that coherence is maintained over cache lines, and that each line of memory has an associated state (such as *modified, shared* or *private*) in each processor cache where it resides and possibly in main memory as well. When a line is written, all shared copies must be either updated or invalidated. The choice of whether a *write update* or *write invalidate* protocol should be used has been analyzed by Eggers and Katz for single-shared-bus multis [Egge89]. They concluded that write invalidate protocols were preferable for the workloads they analyzed. We further argue that write invalidate protocols become more appropriate as system size increases. As systems grow beyond a single bus, the bandwidth used to broadcast a modified block to all processors (or all processors that previously read the block) becomes increasingly larger relative to the bandwidth used by a read request. Thus, in situations where write updates must be *broadcast* (or where only a small fraction of processors that currently have a copy of the line will by reading the new data) it is potentially far more economical to broadcast (or multicast) the smaller invalidation message. For these reasons, we will assume the use of invalidates rather than write updates.

4.1. Broadcast Invalidate

One simple coherence mechanism is to broadcast an invalidate to all processors whenever a shared line is written. This is the approach used for single bus multiprocessors, where a broadcast requires only a single bus transaction. We can immediately see, however, that this approach does not satisfy our notion of uniform scalability. As the system size increases, each processor will receive invalidations at a rate proportional to N. This will saturate the network or the processors at some point, regardless of topology, and therefore this mechanism is not appropriate for very large systems.

Formally, if the fraction of requests that are writes to shared variables remains fixed (the uniform workload assumption), then $O(N)$ invalidations are generated on each request, each of which must be sent to every processor. From a latency viewpoint, queueing of messages in the network implies that the average latency of a request will grow as $O(N)$, clearly violating the requirement for scalability. From a bandwidth viewpoint, since the number of input lines to any processor can grow by at most $O(\log N)$ (the cost requirement for scalability), and the number of incoming invalidation messages per request is $O(N)$, the traffic per line per request grows as at least $O(N/\log N)$, again, clearly violating the requirement for scalability.

It should be noted that for a conspirator workload in which the fraction of writes to shared variables decreases as $O(1/N)$, a broadcast invalidate protocol can scale. In addition, for sufficiently small implementations, a broadcast invalidate protocol may be quite feasible and relatively simple to implement.

4.2. Global Directory

Another possible coherence mechanism is to keep track of all shared copies of a line in a global directory. The directory can be distributed along with the memory of the system (as opposed to a "centralized" directory, which would create a bottleneck in the system). Censier and Feautrier [Cens78] proposed keeping a bit vector of size N for each line, with the corresponding bit set for every processor that has a copy of the line. When the line is invalidated, individual messages are sent to each processor whose bit is set. This does not violate the bandwidth requirement for scalability, assuming that general read traffic scales, because the number of invalidation messages is directly proportional to the number of original read requests that caused the directory bits to be set. However, it clearly violates the cost requirement. The amount of memory storage needed to implement this directory is $O(N^2)$ (we assume the size of the memory is $O(N)$). In addition, the latency of such an invalidate grows linearly with the number of shared copies, which precludes scalability for workloads with heavy read sharing.

Another way to maintain a global directory is to have a fixed number, i, of processor pointers in each directory entry. Either the number of shared copies of a line must be limited to i, or when the number exceeds i, this fact must be recorded and a broadcast invalidate must be issued when the line is modified. Agarwal, et al [Agar88] suggested a label of Dir_iB or Dir_iNB to represent the versions of this protocol which do, and do not, use broadcast, respectively. We restrict our attention to Dir_iB, as Dir_iNB does not allow data to be globally shared.

In order to scale to very large systems, the number of processor pointers that are stored in each directory entry cannot be significantly increased (our cost requirement for scalable systems allows it to increase only as $\log N$). Therefore, the bandwidth and latency requirements will only be met if the number of shared copies of a line is well behaved for very large systems. This means that upon invalidation, the number of shared copies of a line must be less than or equal to i, or it must be on the order of N. If a fixed fraction of the invalidations require broadcasts when the number of shared lines is only slightly larger than i, then the bandwidth of the system will not scale, as discussed in section 4.1. Certain workloads may display the correct behavior for large systems, but it is not clear whether programs will behave correctly in general.

Weber and Gupta [Webe89] analyzed five parallel traces and recorded the distribution of the number of shared copies that needed to be invalidated on a write, for 4, 8 and 16 processor systems . They found that the number of shared lines that needed invalidation on a write was typically low, but that the distribution *did* spread out as the system size increased. It is difficult to say what the distribution would look like with 100, 1000, or 64K processors.

Agarwal, et al [Agar88], evaluated Dir_1NB and Dir_0B as well as two snooping protocols (Write-Through-With-Invalidate and Dragon) for three 4-way-parallel traces and found Dir_0B to be competitive with the Dragon protocol. Unfortunately, their traces were of limited parallelism and the topology simulated was a single shared bus (where there is little difference between a point-to-point message and a broadcast), so it is difficult to draw conclusions from this work regarding the scalability of the directory protocols.

4.3. Hierarchical Directories

In systems with a hierarchical topology (such as cubes and trees) and a multicast capability, the directory can be partitioned hierarchically in order to fit the topology. As a framework for discussing such systems, we will use a n-dimensional cube topology, as discussed in section 3. We will assume the use of buses with k processors per bus, but, as was emphasized in section 3, the buses can easily be implemented as rings of point-to-point links. Recall that every address in our system has a home memory, and, for each address, there exists a n-deep tree with branching factor k, that is rooted at the home memory and includes all processors (figure 2(b)).

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 bus 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 subdirectory 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).

There are several advantages to such an approach for very large systems. First, invalidation bandwidth is reduced over a global (non-hierarchical) directory, because multiple invalidates share part or all of their path through the network (recall, however, that the bandwidth of a global directory still scaled). Second, invalidation latency is reduced by not serializing invalidation messages. Lastly, the hierarchical organization allows the directory to be maintained while concurrent read requests are combined in the network. With a global directory, each read request must be routed to the directory. This will not scale for workloads with concurrent read contention. The cube topology, however, allows requests to be combined on their way to memory. 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 obtained when the first request completes. This is compatible with a hierarchical directory, as the second request would have been from the same subtree as the first request, and thus would not have affected any directory entries above the point at which combining took place.

The hierarchical directory, however, still does not scale in cost. It uses O(N) bits ($[N-1]\frac{k}{k-1}$) 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.

4.4. Multi Level Inclusion

A coherence mechanism that is similar to hierarchical directories in some ways is the *multi level inclusion* (MLI) property [Lam79, Wils87, Baer88]. The MLI property requires that a cache in a hierarchy contain a superset of all lines residing beneath it it 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 a broadcast invalidate in much the same way as a hierarchical directory. 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*. An inclusion cache simply contains tags for each cache line residing within the corresponding subtree. The logical *inclusion bit* for a line is considered to be set if there is an entry for that line in the inclusion cache, and cleared if there is no entry.

The VMP-MC system[Cher89] enforces multi level inclusion within a VMP node, but uses directory entries similar to pruning vectors rather than inclusion bits. A pruning vector at a given node is simply the collection of its childrens' inclusion bits for the same line. The directory entries in VMP-MC are associated directly with memory and cache lines, so MLI is required for the actual data, as opposed to just the tags.

The directory overhead required to enforce MLI does not grow as O(N^2), because inclusion information for a line is not maintained in subtrees that do not contain the line. Rather, the required directory space grows as O($N \log N$). Each of the O(N) lines in the leaf caches requires inclusion information in at most O($\log N$) parent caches. This overhead is further reduced if data in the leaf caches is shared. Although the directory overhead for MLI scales, it can still be quite large. This will be addressed further in section 4.5.4.

4.5. Pruning Caches - A New, Scalable Solution

We now propose a variation of hierarchical directories that is similar in flavor to MLI. We implement hierarchical directories as described in section 4.3, but 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 on an invalidate. If it does contain the entry, then we proceed as with the hierarchical directory. If it doesn't contain the entry, 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.

4.5.1. Pruning cache performance

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, the top level pruning vector acts as a *coarse vector* [Gupt90]. The invalidation traffic in this case is less than with a broadcast scheme, but still does not scale. In order to demonstrate that a system with pruning caches will scale, we must demonstrate that the bandwidth and cost requirements are both met.

Recall that to provide sequential consistency, we must be able to know when an invalidate 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 invalidate can be combined at each level such that each parent propagates a single acknowledgement up, after receiving the acknowledgements from its children. Parents that propagate an invalidate down onto a leaf bus, may immediately propagate an acknowledgement up. When each of these acknowledgements is received by the next parent up, it will propagate an acknowledgement up to the next level. Except for the invalidate on the root bus, there is a one-to-one correspondence between invalidates and acknowledgements. However, because some parents and children are the same node in a cube-based hierarchy, $1/k^{th}$ of the acknowledgements will not actually have to be placed on a bus. The number of buses in a broadcast tree is $\frac{k^n-1}{k-1}$, so the bus load for a full broadcast invalidate is

$$B_{broad} = \left[\frac{k^n-1}{k-1}\right] + \left[\frac{k^n-1}{k-1} - 1\right]\left[\frac{k-1}{k}\right] = \left[\frac{k^n-1}{k-1}\right] + K^{n-1} - 1 \qquad (1)$$

Given the hit rate, h, we can compute the expected number of bus operations needed using pruning caches to invalidate m shared copies randomly distributed throughout the system. Label the buses as in figure 2(b). A subtree at level i contains the k^i processors that come at or below the dimension i bus in the tree. The probability that at least one copy exists 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 bus, 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 bus 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 load from an invalidate using pruning caches with a hit rate of h is

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

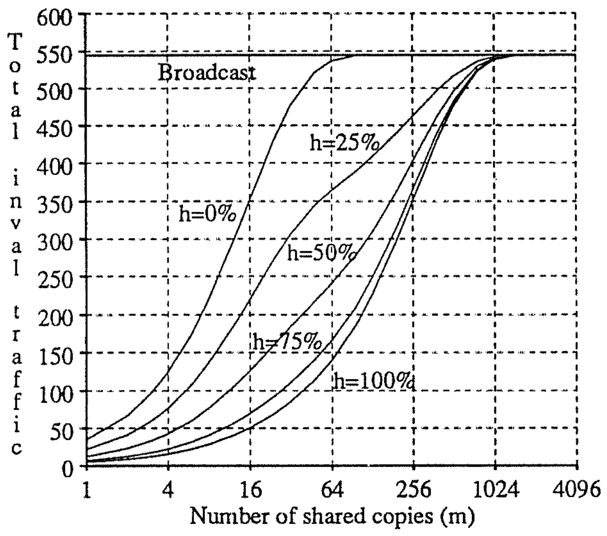
where $P_{inval}(i,m)$ is the probability that an invalidate reaches a bus at level i, which is given by

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

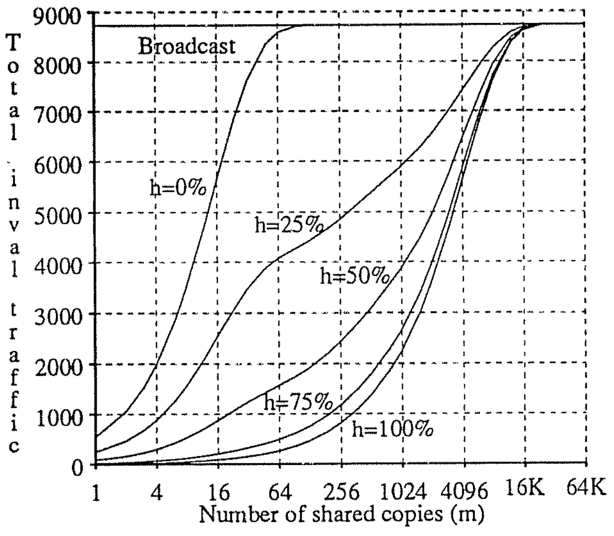
$$+ \left[P_C(n-1,m) - P'_C(n-1,m)\right](1-h)^{n-i} \qquad (5)$$

The first term of $P_{inval}(i,m)$ is the probability that an invalidate was *supposed* to traverse the bus at level i. The remaining terms account for the probability that an invalidate reaches the bus due to pruning cache misses. The first term inside the summation is due to invalidates that were supposed to traverse the level $j+1$ bus, 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 the level j subtree, but not traverse the level j bus. The last term of $P_{inval}(i,m)$ is equal 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 bus load used by a full hierarchical directory. Figure 3 plots the expected bus load of an



(a) 4096 processor system (k=16,n=3)



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

Figure 3: Pruning Cache Performance

invalidate versus m, for two example systems. The bus load is shown for a full broadcast (equation 1) and for pruning cache systems with various values of h (equation 4). Since pruning vectors for the root bus are kept in directories, a significant reduction in traffic is seen even when the pruning caches never hit.

The assumption of a random distribution gives somewhat conservative results, as shown in figure 4. This figure plots the best-case, random, and worst-case bus load caused by a completely pruned invalidate versus the number of shared copies, m, in the system, for a 4096 processor system with $k=8$ and $n=4$. In the best-case distribution, shared copies are grouped along leaf buses as much as possible. This gives a bus load of

$$B_{best\ case} = \sum_{i=1}^{n} \left\lceil \frac{m-k^{i-1}}{k^i} \right\rceil + \sum_{i=1}^{n-1} \left(\left\lceil \frac{m}{k^i} \right\rceil - \left\lceil \frac{m}{k^{i+1}} \right\rceil \right) \qquad (6)$$

In the worst-case distribution, shared copies are spread out among leaf buses as much as possible. This gives a bus load of

$$B_{worst\ case} = \sum_{i=1}^{n} min(k^{n-i},m) + \sum_{i=1}^{n-1} min(k^{n-i}-k^{n-i-1},m) \qquad (7)$$

The random distribution assumed in figure 3 gives bus load estimates very close to the worse case, indicating that there is possibly much to gain by organizing sharing along leaf buses when possible. As an added incentive, sharing along leaf buses would increase the efficiency of pruning caches and read combining in the network.

To meet the bandwidth requirement for scalability, the invalidation traffic over each bus caused by each processor making one request must be O(1). This implies that the traffic caused from a single invalidate, 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(n) bus operations, so the bandwidth requirement is met.

Figure 5 shows the normalized invalidation traffic 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. For the global directory scheme, the normalized traffic would be two, for all system sizes, regardless of m. We see that with a pruning cache hit rate of 100%, the normalized traffic appears almost constant as system size increases (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

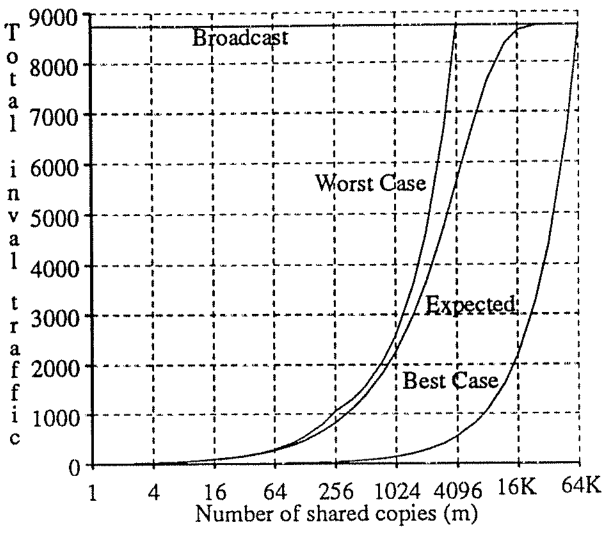
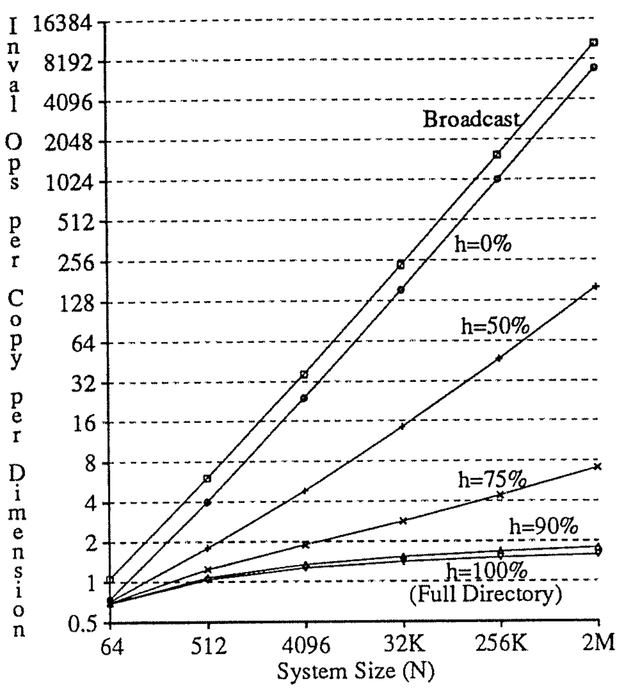


Figure 4: Distribution's Effect on the Bandwidth of Broadcasts (N=64K,k=16,n=4,h=1)

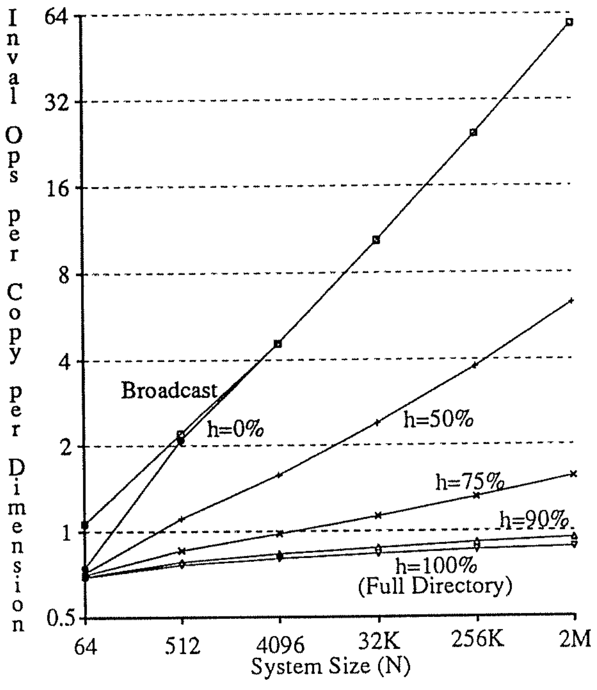
increases by a factor of 10000. This indicates that, given high enough hit rates, pruning cache-based systems will scale to very large numbers of processors.

4.5.2. Pruning cache space overhead

The question that remains to be addressed is whether the size of pruning caches can be kept to O(log$N$), while maintaining the same level of performance. This can be answered with the following argument, which shows that the size of a pruning cache must grow as O(n). 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 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 $\frac{S}{k}$ entries for each of the k caches along its level 1 bus (see figure 2(b)), $\frac{S}{k^2}$ entries for each of the k^2 caches along its level 2 bus and below, ..., and $\frac{S}{k^{n-1}}$ entries for each of the k^{n-1} caches along its level $n-1$ bus and below. The total number of entries needed in a pruning cache is thus $(n-1)S$ and pruning caches should retain the



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



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

Figure 5: Scalability of Broadcast and Pruning Caches (k=8, n=2....7)

same level of performance if they scale in size as O(n).

4.5.3. Pruning cache maintenance

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 invalidate, 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.

An important issue relating to pruning caches 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 is to base replacements on 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.

4.5.4. Comparison to inclusion caches

The key difference between inclusion caches and pruning caches, is that multi level inclusion *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 practical in a single tree system (such as that in figure 1) where the number of caches decreases at each higher level. As was noted earlier, these systems do not provide scalable bandwidth. The second solution is feasible in a cube-based system, but provides inferior performance to a pruning cache-based system [Scot90]. The penalty for purging an entry from an inclusion cache is that the line must be prematurely invalidated in the subtree beneath the cache. This may cause subsequent cache misses within the subtree. The penalty for purging an entry from a pruning cache is that a later invalidation of the line may have to be broadcast underneath the entry, increasing the bus traffic. The former penalty is greater.

The practical upshot of the differences between pruning and inclusion caches is that inclusion caches must be considerably larger than pruning caches to afford the same level of performance [Scot90]. An inclusion cache must store information for *all* lines residing beneath it. A pruning cache need only store information for actively shared lines. Pruning vectors for lines that are not actively shared can fall out of their pruning caches while the data remains in the caches below.

5. CONCLUSIONS

We have provided a working definition of scalability and demonstrated its use on several topologies and cache coherence mechanisms. The basic notion is to keep the cost, communication latency and link traffic from growing too fast as system size increases. When used in the wrong way (considering scalability before performance, for instance), the concept of scalability can be abused, but when used correctly, scalability provides useful intuition about the behavior of very large systems. This intuition should prove valuable as efforts continue to build ever larger multiprocessors.

The *k*-ary *n*-cube is a promising topology for future, large-scale multiprocessors. When the dimensionality of cube networks is increased with size, the networks meet our criteria for scalability. They also provide a substitute for single-tree-based networks (which do not scale for uniform workloads) as a platform for hierarchical communication protocols. The equivalence of buses and rings, which came out of our definition for scalability, allows for cube networks to be implemented with either technology, and still employ the same basic protocols. It is likely that "simulated" buses, constructed from point-to-point links, will provide higher bandwidth than their conventional counterparts as logic speeds continue to outpace bus speeds.

The cache coherence mechanism used, greatly affects the scalability of a multiprocessor. A mechanism is needed that scales in both space, latency and bandwidth. The full hierarchical directory scales for latency and bandwidth. In addition, it allows for combing of concurrent read requests in the network, allowing for workloads with heavy read sharing to scale well. Unfortunately, it requires $O(N^2)$ space for the directory information. Pruning caches approach the performance of a full hierarchical directory, and yet require only $O(N \log N)$ space. Space is additionally conserved because pruning caches need only contain entries for actively shared data. As such, they should scale well to very large systems.

Acknowledgements

I would like to thank Jim Goodman, Phil Woest and Mark Hill for ideas and discussion that assisted 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.

[Agar88] Agarwal, A., R. Simoni, J. Hennessy, and M. Horowitz, An Evaluation of Directory Schemes for Cache Coherence, *Proc. 15th Annual International Symposium on Computer Architecture*, June 1988, 280-289.

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

[Alve90] Alverson, R., D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, The Tera Computer System, *Proc. 1990 International Conference on Supercomputing*, June 1990.

[Amda67] Amdahl, G. M., Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, *Proc. AFIPS Conference*, April 1967, 483-485.

[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.

[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.

[Cher89] Cheriton, D. R., H. A. Goosen, and P. D. Boyle, Multi-Level Shared Caching Techniques for Scalability in VMP-MC, *Proc. 16th Annual International Symposium on Computer Architecture*, May 1989, 16-24.

[Dall90] Dally, W. J., Performance Analysis of k-ary n-cube Interconnection Networks, *IEEE Transactions on Computers* **39**(6), June 1990, 775-785.

[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.

[Egge89] Eggers, S. J. and R. Katz, The Effect of Sharing on the Cache and Bus Performance of Parallel Programs, *Proc. ASPLOS III*, April 1989, 257-270.

[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.

[Gupt90] Gupta, A., W. Weber, and T. Mowry, Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes, *Proc. 1990 International Conference on Parallel Processing*, August 1990, I312-I321.

[Gust88] Gustafson, J. L., Reevaluating Amdahl's Law, *Communications of the ACM* **31**(5), May 1988, 532-533.

[Hage89] Hagersten, E. and Seif Haridi, The Cache Coherence Protocol of the Data Diffusion Machine, SICS Research Report R-89004, Swedish Institute of Computer Science, Kista, Sweden, May 1989.

[Hill90] Hill, M. D., What is Scalability?, *Computer Architecture News*, September 1990.

[John90] Johnson, R. E., Scalable Read-Sharing in SCI, Ph. D. Preliminary Exam, Department of Computer Sciences, University of Wisconsin-Madison, Madison, WI, August 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.

[Leno90] Lenoski, D., J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor, *Proc. 17th Annual International Symposium on Computer Architecture*, May 1990, 148-158.

[Patt85] Patton, P. C., Multiprocessors: architecture and applications, *IEEE Computer* **18**(6), June 1985, 29-40.

[Scot90] Scott, S. L., Performance of Pruning-Cache Directories for Large-Scale Multiprocessors, Computer Sciences Technical Report #983, University of Wisconsin-Madison, Madison, WI 53706, December 1990.

[Vern89] Vernon, M. K., R. Jog, and G. S. Sohi, Performance Analysis of Hierarchical Cache-Consistent Multiprocessors, *Performance Evaluation* **9**, 1989, 287-302.

[Webe89] Weber, W. and A. Gupta, Analysis of Cache Invalidation Patterns in Multiprocessors, *Proc. ASPLOS III*, April 1989, 243-256.

[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.