SCALABILITY AND ITS APPLICATION TO MULTICUBE

by

James R. Goodman
Mark D. Hill
Philip J. Woest

Computer Sciences Technical Report #835

March 1989

Scalability and Its Application to Multicube

James R. Goodman, Mark D. Hill, and Philip J. Woest

Department of Computer Sciences
University of Wisconsin - Madison
Madison, Wisconsin 53706

Abstract

In parallel processing, scalability is an important issue for both applications and systems, with scaling arguments being presented for most large-scale multiprocessors. Even with such widespread interest, the notion of what constitutes a scalable multiprocessor remains ambiguous.

In this paper we first present a realistic, quantitative definition of scalability based on a multiprocessor's ability to execute a problem in roughly constant time, given that the number of processors scale in proportion to the complexity of the problem. We then use the definition to demonstrate that the Multicube architecture is scalable by showing that (1) sufficient bus bandwidth is provided to support scalable applications, and (2) broadcast invalidations and non-uniform memory accesses are handled efficiently. We discuss, for example, how the propagation of broadcast invalidations may be limited by the use of special *pruning* caches, and how Multicube's cache coherency hardware and synchronization primitives can be used to limit the degradation from non-uniform memory access patterns.

Keywords: *scalability, multiprocessors, Multicube, pruning cache.*

1. Introduction

It is well-known that *scalability* is a desirable property of a multiprocessor system. While the basic notion is intuitive, there is some ambiguity regarding what constitutes a scalable system. One of the first papers to use the term, for example, stated that "a design is scalable if it can be adjusted up or down in size or number without loss of functionality to scale effects" [Patt85]. We will define scalability with respect to how execution time varies as both problem size and system size are increased.

We begin by considering what scalability is *not*. First, scalability of a system does not imply that increasing the number of components in the system must proportionately reduce the time taken to complete a given problem, such as a benchmark. Such an interpretation would suggest that a problem could be solved in an arbitrarily short time, given enough processors. This interpretation is also given credibility by the term *speedup*, sometimes used to evaluate scalability, which directly implies a reduction in time to solve a fixed problem. We propose a somewhat more realistic requirement, that a system be able to solve larger problems within appropriate time limits to be regarded as scalable.

Second, scalability does not imply that all programs can take advantage of all the components, even for sufficiently large problem size. Some algorithms are inherently serial, and cannot be parallelized. Thus scalable systems need only show reasonable performance for algorithms that can be parallelized sufficiently.

Third, scalability does not usually address the memory requirements of a problem, though it does address memory bandwidth and network contention. All that can be said regarding memory size is that the memory of the system must be adequate for the problem size. Since memory requirements rarely grow faster than the complexity of the algorithm, it is not reasonable to insist that memory capacity grow faster than the number of processors for a scalable system. Results showing superlinear speedups because of reduced paging or other effects of increased memory size should not be confused with the issue of scalability.

In this paper we seek to provide a realistic, quantitative definition of scalability that permits evaluation of a system against a standard which is not likely to be exceeded, but may be approachable, even for very large scaling factors. We then apply this definition to the Multicube architecture, and demonstrate the scalability of Multicube.

The remainder of this paper is organized as follows. In section 2 we lay out the requirements of scalability, and give a precise definition of scalability. In section 3 we review the Multicube architecture. Then in section 4 we apply the definition of scalability to the Multicube model, and show that Multicube is scalable. In the final section we summarize our results.

2. Scalability

Scalability is an attribute of systems that can be implemented over a range of sizes. Size is measured by system resources such as processors. Asymptotic (or theoretical) scalability compares the performance of a system of fixed size with one that is arbitrarily large, while limited (or practical) scalability is concerned with system growth by small factors, less than 100. Below we discuss the more stringent demands of asymptotic scalability. In many cases, however, building a system for limited scalability is sufficient, since few designs are practical (and near optimal) at both ends of much more than a factor of ten change in size. While an architecture may scale by as much as two orders of magnitude, a given design is likely to scale less than one order.

As part of analyzing system scalability, we need to determine if and how the problem to be solved changes with growing system size. The simplest assumption is to hold the problem size fixed and assume that scalability requires larger systems to solve the problem faster (*e.g.* as is assumed with Amdahl's law [Amda67] and most benchmarks). Experience has shown, however, that people not only use larger systems to solve the same problem faster, but they also use them to solve larger problems. Furthermore, the second case is probably more important. This observation led Gustafson to redefine Amdahl's law based on a fixed execution time instead of fixed problem size [Gust88].

Both Amdahl and Gustafson assume (simplistically but usefully) that algorithms have serial and parallel parts. Amdahl makes no further assumptions regarding the serial part, but requires the parallel part to possess arbitrarily large parallelism. Gustafson, on the other hand, adds the requirements that the time required to execute the serial part be independent of problem size and that the parallel part exhibit linear complexity and parallelism. This precludes the scalability of any sorting algorithm, since the serial complexity of sorting is $s \log s$ for problem size s.[1] The serial complexity for an algorithm, $f(s)$, gives the number of operations required to perform a computation as a function of the problem size, or alternatively, the time required to execute the algorithm on a single processor. We drop the linearity requirement in the following definition:

> A *scalable algorithm* is a parallel algorithm whose serial portion requires constant execution time, regardless of problem size, s, and whose parallel portion contains parallelism at least proportional to $f(s)$, the algorithm's serial complexity.

While the execution of non-scalable algorithms may be important, no system can be expanded to execute larger and larger versions of a non-scalable algorithm in constant time.

The above definition implies that some super-linear algorithms are scalable. Thus system scalability cannot require that execution time remain constant when problem size scales in proportion with system size. A definition of scalability that could include super-linear algorithms is to require execution time to remain constant when the value of problem serial complexity, $f(s)$, rather than the problem size, s, grows linearly with system size. Thus for a problem of serial complexity $f(s)$, increasing problem size from s_0 to s_1 requires system size to increase by a factor of $f(s_1)/f(s_0)$. Doubling the problem size of linear and quadratic algorithms, for example,

[1]Recall that the problem size is the amount of memory necessary to specify the input to the problem.

requires system size to increase by factors of two and four, respectively. Conversely, doubling the system size for linear and quadratic algorithms allows problem sizes to increase by factors of 2 and $\sqrt{2}$, respectively.

Consequently, a desirable goal for a *scalable system* (software and hardware) is that the execution time for a scalable algorithm not increase when the value of $f(s)$ grows proportionally with system size. Unfortunately, due to physical limits, this goal is not realizable for algorithms whose communication requirements (*e.g.* memory requests or messages) are of the same order as their serial complexity. When this is the case, the goal of constant execution time requires that individual communication be serviced in constant time, regardless of system size. For small changes in system size (*e.g.* adding a few more processors to a bus), this requirement can be approximately met. For larger changes, it cannot be met without purposefully degrading the communication latency of the smaller system. For example, when multistage interconnection networks are postulated, scaling a system by a factor of k increases latency proportional to $\log k$. Ultimately, however, due to the propagation of light in three-space, communication latency increases at least with the cube-root of the system size. In a multistage interconnection network, latency grows from logarithmic to at least cube root when the secondary effect of longer wires between stages can no longer be ignored. This leads to the following definition for a scalable system:

> A *scalable system* allows $f(s)$ for a scalable algorithm to grow proportionally with system size, and guarantees that the execution time (1) grows logarithmically for limited scalability and with the cube root for asymptotic scalability, for algorithms limited by communication, and (2) is constant, otherwise.

In addition to the latency requirements described above, a scalable system places two kinds of demands on the bandwidth of its communication network. First, it requires that the deliverable bandwidth (i.e., bandwidth as seen by the processors) must grow with the demands of scalable algorithms. A scalable algorithm of serial complexity $f(s)$ makes requests at a rate no greater than proportional to $f(s)$. Since we assume $f(s)$ scales linearly with the number of processors, we have that the request rate per processor does not increase with increasing problem and system size. However, due to increasing system size, each request can require more network resources to reach a more distant destination.

The second bandwidth demand placed on the communication network of a scalable system is that bandwidth through specific system resources (*e.g.* memory locations or message receipt points) also scales as needed. For a fixed-size system, Pfister and Norton call such a resource a *hot spot* and define it be the target of a ''higher access rate superimposed on a background of uniform traffic'' [PfNo85]. For an asympotically scalable system, we propose that a *hot spot* be any resource that receives requests at a faster rate for larger problem sizes. Such a hot spot will always meet Pfister and Norton's definition, for sufficiently large system and problem size.

After reviewing the Multicube architecture, we will show that it meets the above criteria, and hence is scalable.

3. The Multicube Architecture

The recently proposed Multicube architecture [GoWo88, LeVe88, GoVW89] uses a multidimensional grid of buses to provide efficient hardware cache coherency and high interprocessor bandwidth. The architecture provides for a multi-level cache structure: a first-level, or *processor*, cache for reducing memory latency and a second-level, or *snooping*, cache for minimizing bus traffic. The second level caches are envisioned as being very large (a minimum of 64 DRAMs), suggesting that for typical applications, most cache misses will result from accesses to shared data recently modified by another processor. Coherency is maintained between the two levels of cache by using a write-through strategy and imposing the MultiLevel Inclusion property [BaWa87]. Both memory and I/O devices are distributed among the processors. Because of the symmetry of the organization, bus traffic can be distributed uniformly across the buses, avoiding bottlenecks in the global interconnect. The Multicube project includes the design and implementation of a two-dimensional first generation prototype, the *Wisconsin Multicube*, shown in Figure 1.

Multicube is an attractive architecture for developing parallel applications. While providing a view of a single shared memory to the programmer, it imposes no notion of geographical locality. This ensures that applications developed for *multis* [Bell85] can be easily converted to this architecture. Thus, the Multicube is intended to be a general purpose multiprocessor architecture which supports a large range of applications, such as high-transaction database systems, large-scale simulation models, and artificial intelligence, as well as numerical applications.

High speed processors generally require caches to achieve high performance. In a multiprocessor, this introduces the problem of *cache coherency*. Software solutions typically require invalidating cache entries, which results in lower hit ratios and additional bus traffic. Cheong and

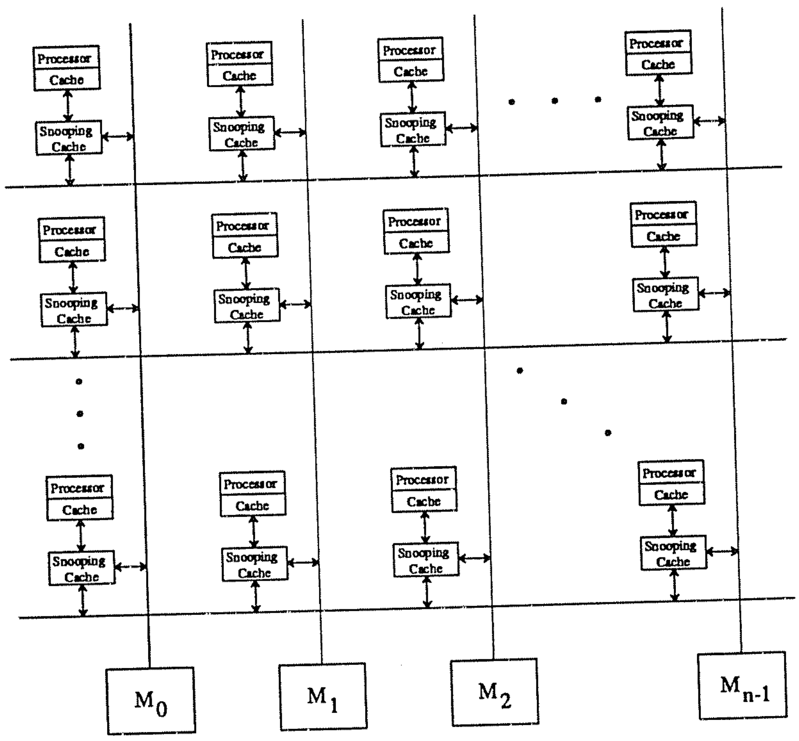

Figure 1. The Wisconsin Multicube.

Veidenbaum [ChVe88] have pointed out that indiscriminate invalidation of the entire cache is fast but results in the lowest hit ratios, while selective invalidation of entries is time consuming. They argue that current techniques are inefficient and that adding special hardware to the cache to support software coherency mechanisms yields the best of both techniques. In addition, their results indicate that block sizes larger than one word significantly increase cache performance, although such larger sizes complicate the cache coherency software and/or hardware.

Hardware cache coherency schemes, on the other hand, relieve the programmer and/or compiler from having to detect potential conflicts in accessing shared variables, while incurring the overhead of maintaining coherency (*i.e.* flushing cache entries to main memory) only when actually called for. These schemes fall into two categories: (1) directory schemes [Tang76, CeFe78, ArBa84] and (2) snooping cache schemes [Good83, Fran84].

Snooping cache protocols are characterized by the requirement that each processor be able to observe every bus transaction. Thus they are most appropriate for single bus multiprocessors. For large-scale systems that perform writes on an exclusive copy of a line (*e.g.* Multicube), some form of directory is useful for storing a pointer to the cache holding that line. The two-dimensional Multicube is a special case in that these directories can be distributed among the columns, with an entry for an exclusive copy replicated for every row, allowing a straightforward extension of the snooping cache protocol to two dimensions.[2] The net effect is that a request for a line held exclusively can be routed directly to the cache holding that line, rather than always to main memory, thereby reducing memory latency for these accesses.

Unfortunately updating a distributed directory in systems having more than two dimensions requires limited, but nevertheless expensive, broadcasts. Thus, the Multicube architecture, in general, will employ cache coherency hardware with directories maintained in main memory,

[2]Details of the snooping cache protocol are given elsewhere [GoWo88].

though distributed caches to aid in reducing bus traffic and latency are planned. Agarwal [ASHH88] indicates that for directory-based schemes, directory bandwidth is not a severe constraint. Later we propose a method for limiting the propagation of broadcast invalidates. We believe that such arguments make a strong case for providing cache coherency in hardware, rather than software, in spite of additional hardware costs.

Synchronization in the Multicube is based on locks [GoVW89], where a single lock bit is logically associated with each line in main memory,[3] with mutual exclusion provided through a *test_and_set* operation. Efficient (*i.e.* local) busy-waiting is supported by "shadow" locks which may be individually allocated in the local caches of any or all of the processors in the system.

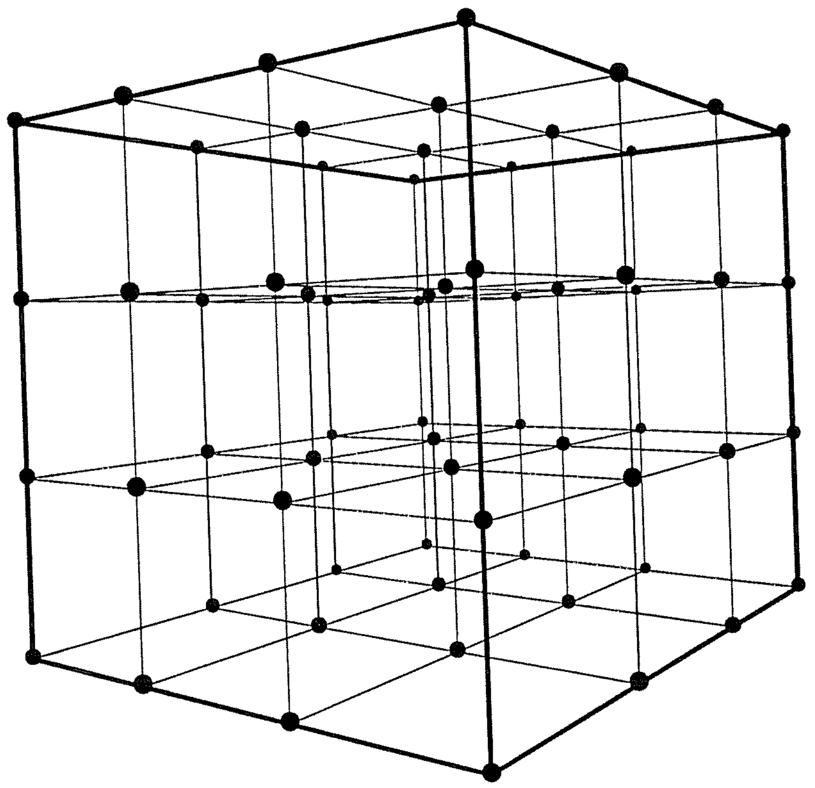

Figure 2. A Three-Dimensional Multicube.

[3]A line is expected to be on the order of 16-32 bus words in the Wisconsin Multicube.

This mechanism reduces bus traffic (and memory contention) by removing the need to spin over the global interconnect for synchronization events. The Multicube architecture defines two separate operations for changing the status of shadow locks. First, a broadcast primitive, similar to an invalidate, may be used to clear a lock and all associated shadow locks. This mechanism allows for an efficient implementation of barrier synchronization and global event notification. Second, a special primitive queues a processor for first-come first-serve access to a lock bit. This queueing mechanism allows for an efficient implementation of, as well as fair access to, binary semaphores.

The synchronization primitives have been carefully designed to take advantage of hardware cache coherency. Thus they have efficient implementations in the Multicube architecture. In addition these primitives are sufficient to implement software combining for a general *fetch_and_add* [GoKr81] operation, with only local busy-waiting. This solution allows an architecture to scale to a large number of processors while avoiding the cost of hardware combining. A complete description of the synchronization primitives with a discussion of their implementation and scenarios for their use can be found elsewhere [GoVW89].

4. Multicube Scalability

The argument for Multicube scalability rests on demonstrating that the architecture correctly handles two important scaling issues. First, constant bandwidth must be provided per processor per dimension as the size of the system grows. This assumes that sufficient bandwidth is provided to handle all data requests and data transfers and that the propagation of broadcast invalidations can be limited, ideally, to the subset of processors that have actually requested shared copies of a line.

Second, the system must effectively handle concurrent requests to the same memory location, so that the effects of hot spot contention are distributed across system resources. This problem has two important cases, concurrent access to synchronization variables and read sharing, both of which can be solved by various forms of combining.

4.1. Bandwidth Scalability

The definition of scalability and accompanying arguments made earlier lead directly to the basic case for scalability of the Multicube architecture: as the number of processors increases, bus bandwidth increases sufficiently fast so as to accommodate the traffic offered by a scalable algorithm. In Section 2 we show that the per-processor request rate is constant or diminishes as problem and system size increase. Here we further assume that, at least to first-order, other causes of cache misses do not increase with system size, implying that per-processor requests occur at some constant rate r.

A multi using a single bus with bandwidth B, can support $n=aB/r$ processors, where $0<a\leq 1$ is the acceptable bus utilization. In a k-dimensional Multicube there are $N=n^k$ processors, where each processor is connected to k buses, and where the longest path from any processor to any other processor also involves k buses. Assuming r remains constant as the number of processors increases, the network must service $r\cdot n^k$ requests per second. If each request must go through no more than k buses, then the total bus bandwidth need service no more than $k\cdot r\cdot n^k$ operations per second.

Each of the N processors is connected to k buses, where each bus is shared by n processors, so there are $N\cdot k/n = k\cdot n^{k-1}$ buses. Thus the total available bandwidth is $k\cdot n^{k-1}\cdot B = k\cdot n^{k-1}\cdot(r\cdot n/a) = k\cdot r\cdot n^k/a \geq k\cdot r\cdot n^k$. Thus the bandwidth scales.

The above argument implies that communication latency in Multicube increases with the logarithm of the number of processors per bus, well under the requirement that it not increase more rapidly than the cube-root of the number of processors. Eventually, of course, increased delays due to packaging will cause latency to increase more rapidly than logarithmic.

Broadcast Invalidates

The bandwidth scalability argument above is based on the assumption that individual bus transactions go through no more than k buses (allowing k bus operations to reach the data and k more to bring it back). Requiring some requests for lines to incur an additional k short address-only bus operations (*i.e.* necessary to check a directory in main memory to determine the line's location) affects the required bandwidth by only a small, constant factor. However, one type of operation, broadcast invalidation, requires $1+n+n^2+\cdots+n^{k-1}=(N-1)/(n-1)$ bus operations. Although the bus operations are short -- containing no data -- for a single broadcast they cover more than one out of every k buses.

If a significant proportion of processor caches contain a shared copy of a line that needs to be invalidated, then a broadcast is the operation of choice. In fact the bus bandwidth consumed by the broadcast is small in comparison to the set of read requests that acquired the shared copies in the first place. This is because cache line transfers require long bus operations as opposed to short bus operations for invalidations. However, if on average, relatively few caches contain a copy then, as the number of processors increases, invalidations will consume proportionately larger amounts of bus bandwidth. Thus, there is a need to reduce or restrict most broadcasts.

In a directory-based hardware cache coherence scheme, validity bits may be maintained that record which caches contain shared copies of a line [Tang76, CeFe78]. As the number of processors increases, these bits may be replaced by a small number of cache pointers (tags). There is evidence [ArBa84, ASHH88] that a single tag is sufficient to handle the majority of invalidation requests, without broadcast, for small-scale multiprocessors. However, algorithms exist that frequently require more than a single shared copy of a line to be invalidated [WeGu89]. Some of these are due to invalidation of synchronization variables, while others are due to invalidating frequently read data. While using alternative algorithms may reduce the number of broadcast operations required, it may also be much slower or more complex.

In Multicube the buses traversed by a broadcast operation form a tree[4] with its root being the main memory module responsible for that line. Reducing the propagation of the broadcast is equivalent to "pruning" those subtrees that do not contain a shared copy of the specified line. Keeping a vector of presence bits (of length equal to the number of processors in the system) for every line in main memory requires a prohibitive amount of storage. Instead, a *pruning cache* may be used to hold tags for shared lines in each processor's subtree. Entries for private written data will never be entered into this cache, and entries for read-only data will eventually be replaced, leaving entries mostly for shared lines that are being actively read and written (invalidated). We hypothesize and are in the process of verifying that locality of reference to shared data will yield sufficient *locality of invalidation* to make low-cost pruning caches effective.

The pruning cache maintains information about the location of shared lines. Each cache entry contains a bit vector which contains a single bit for each processor to which a broadcast invalidation must be forwarded by the next bus invalidate operation for that line. Thus each bit serves as a (possible) presence bit for the line in the subtree rooted at the corresponding processor. An invalidate, upon reaching a processor, will first check the pruning cache. If a hit occurs then the invalidation, which includes the bit vector, is placed on the next bus. Those processors whose corresponding bit is set repeat the same process until the invalidations reach the leaves of the tree. Those with a clear bit, however, simply ignore the invalidate request. If a miss occurs, the worst case is assumed (*i.e.* that every subtree contains a shared copy), and a bit vector of all ones is used.

The performance of the pruning caches can be analyzed as follows. Assume the tree formed by a full broadcast invalidate is split into numbered levels, with main memory and the bus which first receives the invalidate numbered as level 0, and the leaf nodes (which have no

[4]The global interconnect allows a broadcast to fan out like on a n-ary tree, but only if every parent node is allowed to be its own child and buses are traversed in an order well-defined for each line.

associated buses) numbered as level k. Then there are n^i buses at level i (for $0 \le i \le k-1$), with a subtree rooted at level i containing $t_i = N/n^i$ leaf nodes.

If m shared copies of a line are randomly distributed among X nodes, then the probability of some specific subgroup of x nodes *not* containing a shared copy is given by $q(x, X, m)$, and the probability of containing *at least one* shared copy by $p(x, X, m)$, where

$$q(x, X, m) = \frac{\begin{bmatrix} X-x \\ m \end{bmatrix} \begin{bmatrix} x \\ 0 \end{bmatrix}}{\begin{bmatrix} X \\ m \end{bmatrix}} \quad \text{and} \quad p(x, X, m) = 1 - q(x, X, m).$$

Thus, the total number of invalidate bus operations at a desired level can be calculated as the total number of buses at that level multiplied by the probability that a specific subtree contains a shared copy of the line. The sum of the bus traffic at each level yields the expected minimum number of bus operations required to invalidate m randomly distributed copies, and is given by

$$\sum_{i=0}^{k-1} n^i \cdot p(t_i, N, m)$$

Since a pruning cache may need to replace entries, additional invalidate bus traffic will result. Any bus that is part of a subtree which contains no shared copy of the line will actually be included as part of the broadcast if (1) the next largest subtree containing the smaller subtree does have a shared copy and (2) the pruning cache at the root of the larger subtree has lost the entry for that line. For example, an (unneeded) invalidate operation will traverse a bus at level $k-1$ if that level $k-1$ subtree has no shared copy, but the encompassing level $k-2$ subtree does and has lost its pruning cache entry, or if the level $k-2$ subtree has no shared copy, but the encompassing level $k-3$ subtree does and has lost its pruning cache entry, and so on. Including these factors yields the following equation for the total expected number of bus operations for a single broadcast invalidate using pruning caches.

$$\sum_{i=0}^{k-1} n^i \left[p(t_i, N, m) + \sum_{j=1}^{i} \left[(1-h_{j-1}) \cdot q(t_j, N, m) \cdot p(t_{j-1}-t_j, N-t_j, m) \right] \right]$$

The average number of bus operations can be calculated for a system given the number of dimensions k, number of processors per bus n, total processors $N=n^k$, number of shared copies m, and pruning cache hit ratios h_i (for each level i). It is assumed that the topmost bit vectors are stored in main memory with the lines ($h_0=1$), and thus are never subject to replacement. The analysis also assumes that shared copies are randomly distributed among the processors and that the chance of replacement is equally likely for all bit vectors ($h_1 = h_2 = \cdots = h_{k-1} = h$).

Results are plotted for a four-dimensional, sixteen processor-per-bus Multicube for various pruning cache hit ratios and numbers of shared copies. Figure 3 shows the number of total bus operations required to invalidate a single line as a function of the number of shared copies. The lowest curve, for a pruning cache hit ratio of one, represents the average minimum number of bus operations needed to reach every processor, assuming a random distribution of copies. The hit ratio for a pruning cache that always misses is not always equal to the number of bus operations in a full broadcast since main memory always retains full pruning information for the topmost level of the invalidation tree. Figure 4 shows the same information, except normalized to yield the number of bus operations required per shared copy. This number is greatest when there are few copies, and approaches $1/(n-1)$ as the number of copies increases.

These figures show that even low hit ratios (*i.e.* 0.50) substantially reduce the bus traffic per invalidation, and suggest that hit ratios of 0.99 or better will reduce bus traffic due to invalidations to less than that due to the read requests themselves. Furthermore, performance can be improved by having pruning caches favor replacing bit vectors “farther” from main memory, rather than giving all bit vectors equal priority.

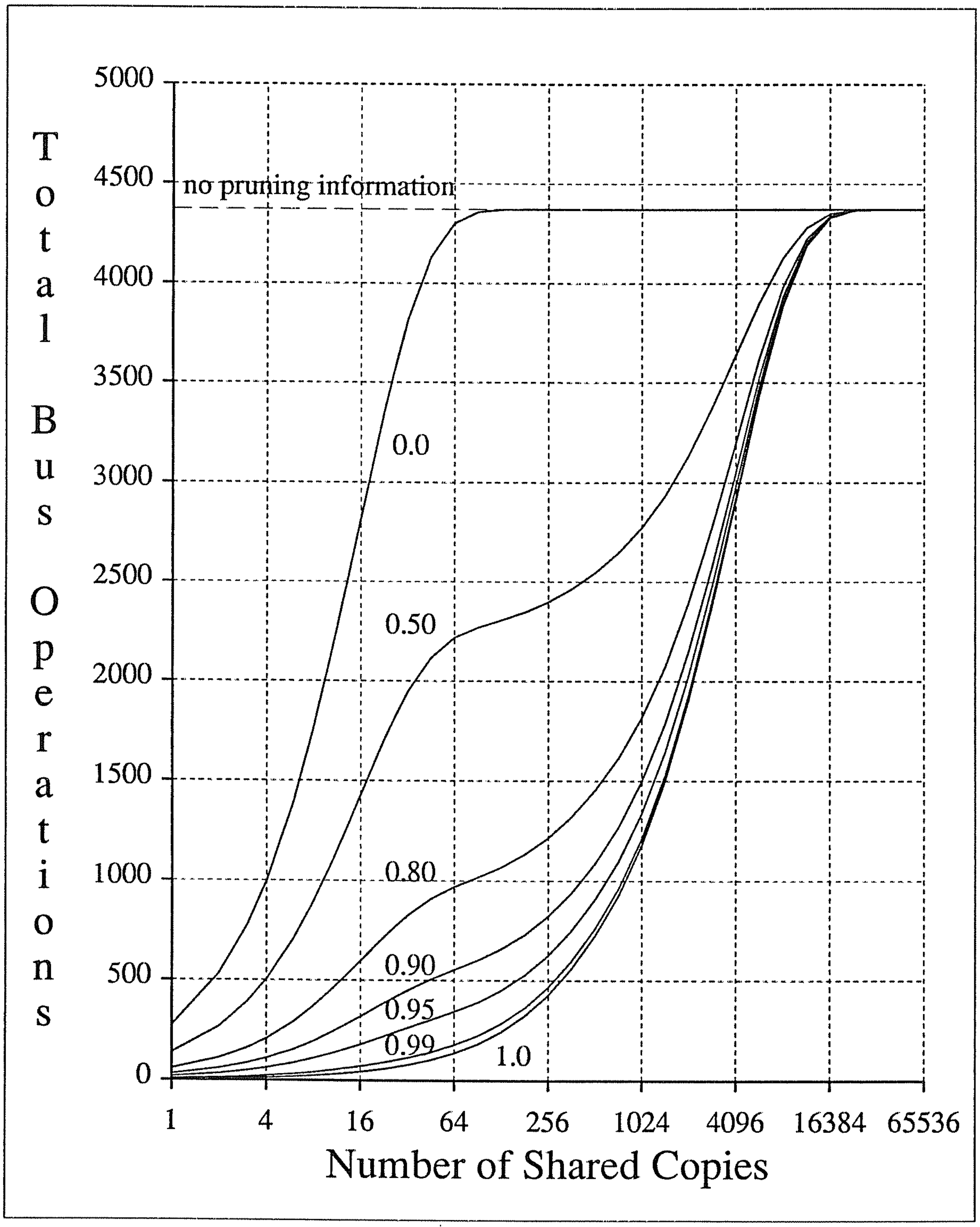

Figure 3. Pruning Cache Performance.

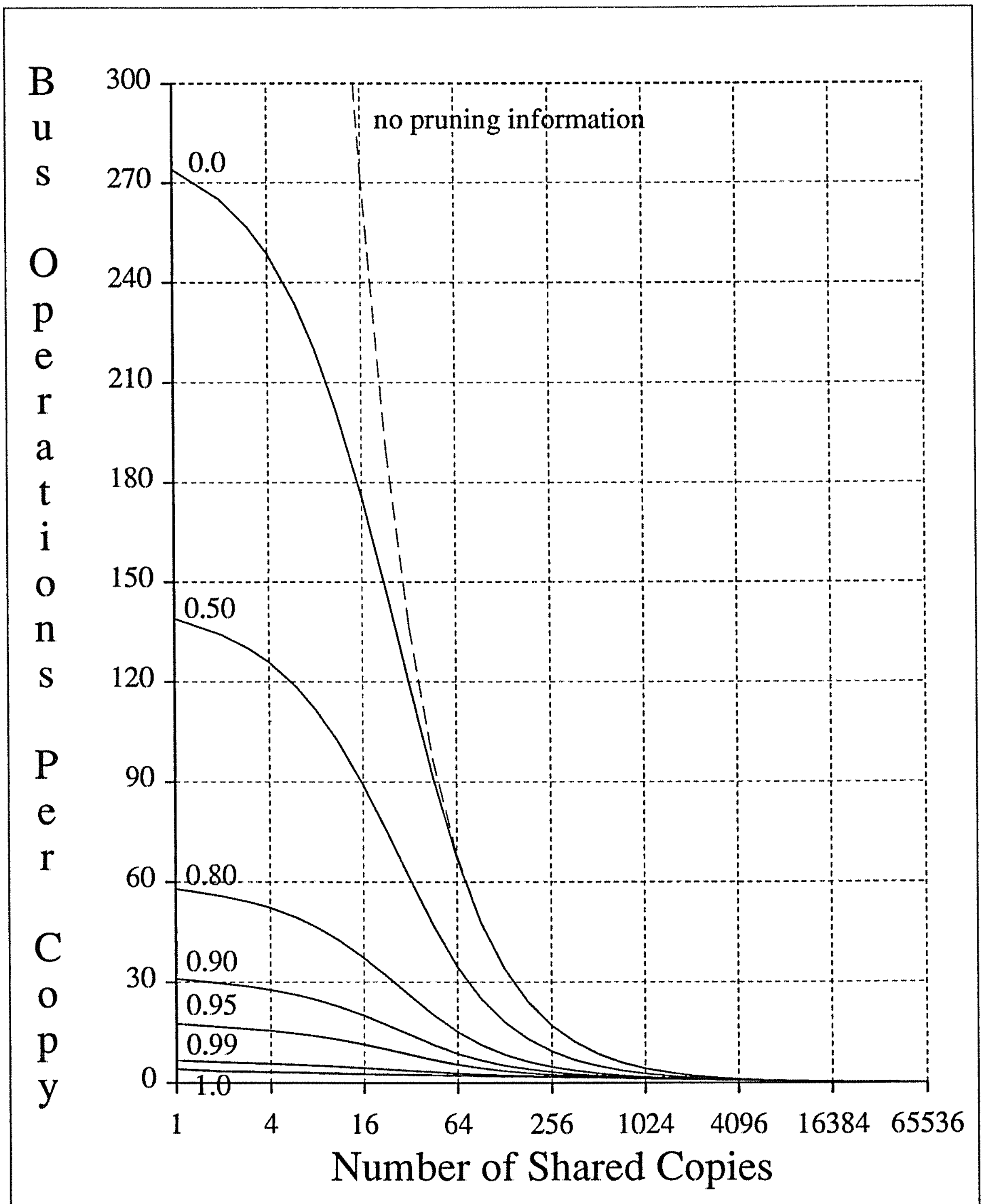

Figure 4. Normalized Pruning Cache Performance.

4.2. Hot Spot Contention

Non-uniformity in accessing memory can create bottlenecks in the global interconnect, known as *hot spot contention* [PfNo85]. While interleaving main memory can sometimes reduce these effects, this technique is not applicable to the case where there are multiple requests for the same memory location. Such requests may be further divided into two important cases: concurrent access to synchronization variables and read sharing. Each of these is discussed below.

Concurrent Access to Synchronization Variables

Serial access to synchronization variables is acceptable if contention for these resources is infrequent. However, for a number of important scenarios, such as barrier synchronization and work queues, satisfying requests serially can create hot spots that severely degrade performance. Thus, such solutions are unacceptable for a scalable system.

Scalable solutions have been proposed based on *fetch-and-*Φ operations, particularly fetch-and-add [GoLR83]. These solutions are free of critical sections, assuming that the fetch-and-Φ operations used to implement them can be combined. Unfortunately, hardware combining is expensive and is likely to penalize regular data accesses [PfNo85]. However, effective combining without additional hardware can be achieved by distributing hot spot addresses using a *software combining tree* [YeTL87]. With such a technique solutions for scenarios in which each processor busy-waits on a node in the tree (*e.g.*, barrier synchronization), are straightforward.

In the Wisconsin Multicube the idea of software combining trees has been extended to provide a general mechanism for combining fetch-and-add operations in software [GoVW89]. Using other synchronization primitives provided in the Multicube, simple solutions have been developed to general synchronization problems that require a relatively high level of contention, such as barrier synchronization and work queues. Since a combining fetch-and-add primitive is seen as the most likely means of avoiding the serial behavior inherent in other synchronization primitives, Multicube should scale for these types of problems.

Read Sharing

Heavy contention for lines that are not synchronized could potentially be caused by both write and read requests. There are three cases that must be addressed. First, applications that perform unsynchronized writes (and reads) to the same memory word at a high enough rate to cause global interconnect congestion are expected to be rare. Second, concurrent writes to different words in the same line actually represent a special situation called *false sharing*, which programmers (and/or compilers) should avoid to obtain high performance [EgKa88]. Avoiding false sharing is particularly important for large-scale shared-memory multiprocessors with cache line sizes much larger than a single word. This leaves the case of multiple read requests to the same line, a frequent situation called *read sharing*.

The problem with read sharing is that, although multiple copies of a line are allowed, concurrent requests for the same line (the same memory module) will be handled serially, either at main memory or at some point in the global interconnect. In the worst case every processor may attempt to read the same line or lines immediately after a barrier. This problem is particularly severe in implementations where the cache line size is many multiples of the bus word size.

Unfortunately, rewriting applications to avoid read-sharing may not be practical or desirable from a performance viewpoint. Many parallel algorithms, such as linear programming [Murt83, MaDe88], routinely re-distribute updated information to a large number of processors assigned to the task. The obvious solution to be applied to read-sharing is combining. However, hardware combining has already been rejected[5] and software combining is too slow and cumbersome for normal reads.

The Multicube architecture affords an effective means of combining read requests without significant additional hardware. When data is supplied as a result of a read request, processors

[5]Although hardware combining of read requests would require less hardware than that for fetch-and-Φ operations.

along the path from main memory to the destination node retain a copy of the line in their local cache. Subsequent read requests will check the caches at processors along the path to main memory. If the desired line is found, it can be supplied immediately. However, the main concern with the proposed scheme is in reducing the latency of read requests. While the benefits are large for situations in which the degree of sharing is high, all read requests will suffer the performance degradation of waiting for the cache lookups along the path to main memory. A promising trade-off is to allow read requests to wait for the lookup while waiting to access the appropriate bus. Thus, if bus contention is low, then the request will proceed as quickly as if there were no combining. On the other hand, if bus contention is high then the request will have to wait in either case. This is exactly the case where combining is likely and also where it is beneficial.

Alternative provisions may be made to reduce bus traffic, such as throwing away multiple requests from processors on the same bus or allowing processors to *snarf* lines that were recently invalidated in their cache [Good87, KMRS88].

5. Summary

We first presented a workable definition of system scalability, along with arguments for its applicability. This definition quantifies how rapidly larger versions of scalable systems must execute scalable algorithms operating on larger problem sizes. We next applied this definition to Multicube to show it is scalable. The discussion includes methods for mitigating problems caused by broadcast invalidates, synchronization variable accesses and read sharing.

6. Acknowledgements

The authors would like to acknowledge Garth Gibson, Ross Johnson, Debbie Joseph, Udi Manber, Mary Vernon and Paul Young for their helpful discussions. This work was supported in part by the National Science Foundation, under grant MIP-8604224, and by faculty initiation grants from the Graduate School at the University of Wisconsin-Madison.

7. References

[ASHH88] Agarwal, A., R. Simoni, J. Hennessy, and M. Horowitz, ''An Evaluation of Directory Schemes for Cache Coherence,'' *Proceedings of the 15th Annual International Symposium on Computer Architecture,* June 1988, pp. 280-289.

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

[ArBa84] Archibald, J., and J. Baer, ''An Economical Solution to the Cache Coherence Problem,'' *Proceedings of the 11th Annual International Symposium on Computer Architecture,* June 1984, pp. 355-362.

[BaWa87] Baer, J. L., and W. H. Wang, ''Architectural Choices for Multilevel Cache Hierarchies,'' *Proceedings of the 1987 International Conference on Parallel Processing,* August 1987, pp. 258-261.

[Bell85] Bell, C. G., ''Multis: A New Class of Multiprocessor Computers,'' *Science,* April 26, 1985, pp. 462-467.

[CeFe78] Censier, L. M., and P. Feautrier, ''A New Solution to Coherence Problems in Multiprocessor Systems,'' *IEEE Transactions on Computers,* December 1978, pp. 1112-1118.

[ChVe88] Cheong, H., and A. V. Veidenbaum, ''A Cache Coherence Scheme With Fast Selective Invalidation,'' *Proceedings of the 15th Annual International Symposium on Computer Architecture,* June 1988, pp. 299-307.

[EgKa88] Eggers, S. J., and R. H. Katz, ''A Characterization of Sharing in Parallel Programs and Its Application to Cache Coherency Protocol Evaluation,'' *Proceedings of the 15th Annual International Symposium on Computer Architecture,* June 1988, pp. 373-382.

[Fran84] Frank, S. J., ''Tightly Coupled Multiprocessor System Speeds Up Memory-Access Times,'' *Electronics,* January 12 1984, pp. 164-169.

[Good83] Goodman, J. R., ''Using Cache Memory to Reduce Processor/Memory Traffic,'' *Proceedings of the 10th Annual International Symposium on Computer Architecture,* June 1983, pp. 124-131.

[Good87] Goodman, J. R., ''Coherency for Multiprocessor Virtual Address Caches,'' *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II),* October 1987, pp. 72-81.

[GoWo88] Goodman, J. R., and P. J. Woest, ''The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor,'' *Proceedings of the 15th Annual International Symposium on Computer Architecture,* June 1988, pp. 422-431.

[GoVW89] Goodman, J. R., M. K. Vernon, and P. J. Woest, ''A Set of Efficient Synchronization Primitives for a Large-Scale Shared-Memory Multiprocessor,'' *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III),* to appear.

[GoKr81] Gottlieb, A., and C. P. Kruskal, ''Coordinating Parallel Processors: A Partial Unification,'' *Computer Architecture News,* October 1981, pp. 16-24.

[GoLR83] Gottlieb, A., B. D. Lubachevsky, and L. Rudolph, ''Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors,'' *ACM Transactions on Programming Languages and Systems,* April 1983, pp. 164-189.

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

[KMRS88] Karlin, A. R., M. S. Manasse, L. Rudolph, and D. D. Sleator, ''Competitive Snoopy Caching,'' *Algorithmica,* **3**, 1988, pp. 79-119.

[LeVe88] Leutenegger, S. T., and M. K. Vernon, ''A Mean-Value Performance Analysis of a New Multiprocessor Architecture,'' *Proceedings of the 1988 ACM SIGMETRICS Conference,* May 1988, pp. 167-176.

[MaDe88] Mangasarian, O. L., and R. De Leone, ''Parallel Gradient Projection Successive Overrelaxation for Symmetric Linear Complementarity Problems and Linear Programs,'' *Annals of Operations Research,* Vol. 14, 1988, pp. 41-59.

[Murt83] Murty, K. G., *Linear Programming,* Wiley and Sons, New York, 1983, Chapter 16.

[Patt85] Patton, P. C., ''Multiprocessors: architecture and applications,'' *IEEE Computer,* Vol. 18, No. 6, (June 1985), pp. 29-40.

[PfNo85] Pfister, G. F., and V. A. Norton, ''Hot Spot Contention and Combining in Multistage Interconnection Networks,'' *Proceedings of the 1985 International Conference on Parallel Processing,* August 1985, pp. 790-797.

[Tang76] Tang, C. K., ''Cache System Design in the Tightly Coupled Multiprocessor System,'' *Proceedings of 1976 AFIPS National Computer Conference,* June 1976, pp. 749-753.

[WeGu89] Weber, W.-D., and A. Gupta, ''Analysis of Cache Invalidation Patterns in Multiprocessors,'' submitted to the *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III).*

[YeTL87] Yew, P.-C., N.-F. Tzeng, and D. H. Lawrie, ''Distributing Hot-Spot Addressing in Large-Scale Multiprocessors,'' *IEEE Transactions on Computers,* April 1987, pp. 388-395.