On the Performance of a Bus-based Multiprocessor Cluster Architecture *

Craig Anderson and Jean-Loup Baer
Department of Computer Science and Engineering
Box 352350
University of Washington
Seattle, WA 98195-2350
{craig, baer}@cs.washington.edu

TR UW-CSE-95-12-01

December 4, 1995

Abstract

The focus of this paper is on the evaluation of a hierarchical cluster architecture where a cluster consists of a single bus shared-memory multiprocessor and where the interconnect is a tree hierarchy of busses.

We first outline a cache coherence protocol for a UMA architecture. We then introduce a variation of the protocol for sector caches where the coherence and transfer units are not the same. We evaluate, through cycle by cycle simulation, the UMA architecture under these two protocols. We simulate six benchmarks with varied memory access patterns and show that the clustering concept is beneficial as long as architectures are balanced. The subblock protocol works well and is more stable than protocols with fixed block size that can behave very well on some applications and very poorly on others.

In the second part of the paper, we modify the UMA architecture into a NUMA architecture. We discuss several ways to perform memory allocation in this new context. We test some of these allocations on two of the benchmarks and report that improvements are largest when the memory allocation does not induce much extra computations and when architectures are balanced.

*This work was supported in part by NSF Grants CCR-94-01689, CCR-91-23308 and by Apple Computer, Inc.

1 Introduction

In order to meet the needs of computationally intensive applications, shared-memory processors must be scalable. The single shared-bus architecture that has been so successful in the 1980's cannot meet the bandwidth requirements of current and future large applications. One architecture that has been proposed is based on clusters of processors connected via some interconnection network. Table 1 summarizes variations on this basic scheme. In this paper, the focus is on the evaluation of a hierarchical cluster architecture, where a cluster consists of a single bus shared-memory multiprocessor, i.e., processors with private level 1 caches and a shared level 2 cache, and where the interconnect is a tree hierarchy of busses.

Table 1: Hierarchical Systems

Feature	Architecture						
	Wilson	DDM	KSR	Stanford DASH	Galactica Net	Convex Exemplar	Hector
Cluster Interconnect	Bus	Bus	Ring	Bus	Bus	5X5 Crossbar	Bus
Top Interconnect	Tree of Busses	Tree of Busses	Tree of Rings	Mesh	Mesh	Four Rings	Two Level Ring
Cluster Cache Coherence	Snoopy	Snoopy	Snoopy	Snoopy	Snoopy	Directory	Snoopy
Top Cache Coherence	Snoopy	Snoopy	Snoopy	Directory	Hybrid	SCI	None
Cluster Cache?	Yes	Dir.†	Dir.†	Yes	Yes‡	Yes	No
Inclusion?	Yes	Yes	Yes	No	N/A	Yes	N/A
Memory Organization	UMA	COMA	COMA	NUMA	NUMA	NUMA	NUMA
Reference	[Wil87]	[HLH92]	[FIR93]	[LLJ+92]	[WTP+92]	[Con94]	[VSLW91]

†Dir.: Only directory entries are cached, not data.
‡Each cluster shares a portion of main memory, which can be used as a cache of other clusters' pages.

Several factors impact the scalability of this type of architecture. In a hierarchical setting, it is important to be able to determine the proper number of processors sharing a common bus in a cluster, and the number of clusters that may share an inter-cluster bus before that bus becomes saturated. As a corollary, any artifact that reduces the communication

overhead caused by cache coherence will be welcomed since it will postpone reaching the saturation points of the various busses. Another critical aspect is that the shared-memory programming paradigm must recognize the effects of non-uniform memory latency. In the context of a hierarchical bus based multiprocessor system, this implies the efficient scheduling and compilation of parallel programs, and means to selectively control the placement and caching of shared data.

We start this paper (Section 2) by introducing the base architecture, a Uniform Memory Access (UMA) system, then continue by outlining a simple (but not simplistic) snoopy-based cache coherence protocol, and terminate this section by describing our evaluation methodology. In Section 3, we propose improvements on the coherence protocol using subblocks as units for coherence. These improvements have been proven effective in non-hierarchical systems and we evaluate them, with appropriate modifications, in this new context. In Section 4, we modify the UMA architecture by placing memory at the cluster level. This NUMA architectural variation is evaluated with different memory allocation procedures. Finally, in Section 5, we summarize our findings on the effectiveness of cluster architectures.

2 Base Architecture and Protocol

2.1 Base Architecture

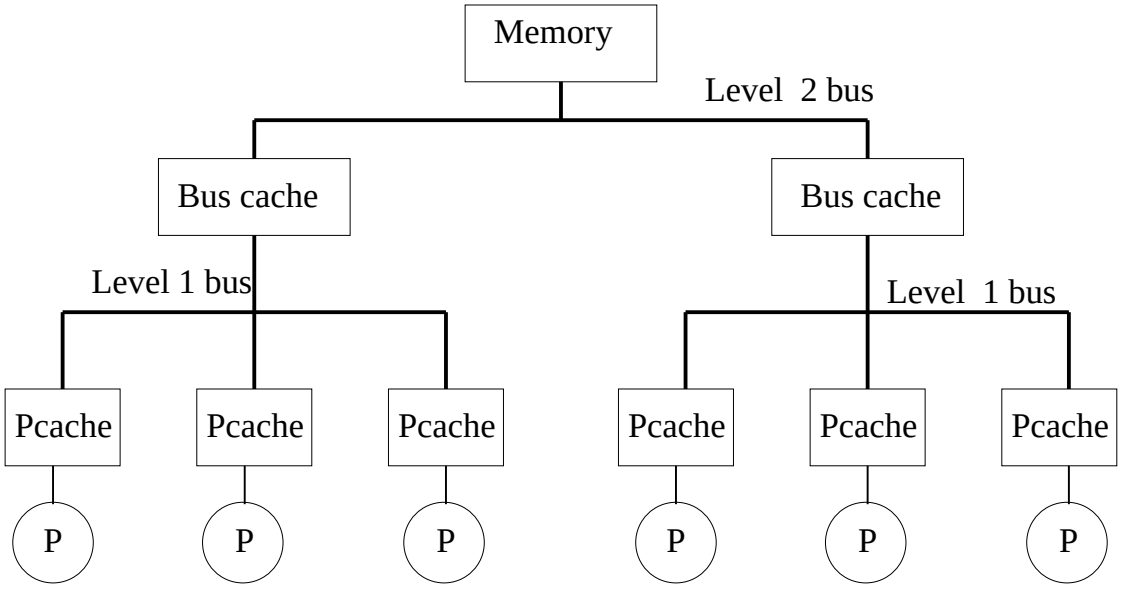

Figure 1: Base (UMA) Machine Architecture

The base architecture is built on a hierarchy of busses as shown in Figure 1. Although the figure and the remainder of the paper limit themselves to two levels, the basic protocol that we developed works on a machine that may have from two to an arbitrarily high number of levels.

At the bottom of the hierarchy (level 0) are processors along with their caches. The processors are grouped into clusters of some number of processors per cluster, typically two to eight. Each processor cache has a dual directory which permits simultaneous processor access and cache coherence checks initiated by bus requests. The intra-cluster processors are connected to one another via a conventional bus, which we call the level 1 bus. Also connected to the intra-cluster bus is an intra-cluster cache/bus connector, or *bus cache*. This component both connects each level 1 bus to the corresponding level 2 bus, and also caches data for the processors in the cluster below it. To reduce bus traffic, all caches in the system are write back caches. Bus caches satisfy the *inclusion* property, that is, each cache at a given level contains a superset of the contents of the caches below it in the hierarchy [BW88]. Inclusion is guaranteed through the actions of the protocol, as in [Wil87]. When a bus cache replaces a block, it sends a message down that forces all processor caches below it to invalidate the block.

Each bus cache has a dual directory so that operations on both busses it is connected to can proceed in parallel. Each bus cache also has a number of bus request buffers that hold both bus requests that have been enqueued waiting for access to either bus, as well as information about read/write/swap operations in progress that are being acted on in other parts of the system.

At the top of the hierarchy is memory. Like in single bus snoopy protocols, blocks found in memory have no state. Requests for blocks that are not found in any cache progress up the hierarchy until they reach the top, where they are satisfied by main memory.

2.2 Overview of the base protocol

The base protocol is designed to minimize response time and bus traffic, especially at the higher levels of the hierarchy. To do this, requests are satisfied as close as possible to the requesting processor. The protocol will attempt to satisfy a request using caches at the same level as that as that of the requester; the request is forwarded to higher levels only when necessary. In some cases, a request may need to travel up the hierarchy, then back down to another branch of the tree; the reply retraces the same path back to the request originator. In addition, lines are written back only to the next higher level of the hierarchy. Only when a cache at the top the hierarchy writes a line back does the line migrate to memory.

The processor level protocol most closely resembles the Illinois protocol with INVALID, VALID EXCLUSIVE, READ SHARED, and DIRTY cache block states. There are also transitional states that are needed for the correctness of the protocol (see below).

The blocks in the bus caches are in one of the five following states (with, of course, the possibility that they are in transitional states; see [And95] for a complete protocol description):

- INVALID: The block is not present in or below the cache.
- READ SHARED: The block is present in the cache in a clean state and may exist elsewhere in the system in a clean state.

- VALID EXCLUSIVE: The block is present in the cache, but is not guaranteed to be up-to-date. The block may exist below this cache in a dirty or clean state.
- DIRTY SHARED: The block is present in the cache and is dirty with respect to memory. This cache is responsible for writing the value back to memory. In addition, the block may exist lower in the hierarchy in READ SHARED state only.
- DIRTY OWNED: The block is present in the cache, is dirty with respect to memory, and is the only cached copy in the system.

Having different states for non-processor caches is usual for hierarchical architectures [Wil87], [YTB92]. The introduction of the DIRTY SHARED state is motivated by its effect of reducing level 1 traffic in inter-cluster requests.

Bus requests are generated by processor actions (read, write, swap) and cache states. In addition to the usual single-bus multiprocessor transactions, there is a need for transactions traveling "down" the hierarchy in the case of inter-cluster transfers and for maintaining inclusion (e.g., purge transactions).

When a request misses in a processor cache, the processor is forced to wait until the request is satisfied. The processor's cache allocates a block for the data (if necessary) and enqueues the appropriate bus request.

Because conflicting request can be generated "at the same time" by different processors, and because there is no implicit serialization device such as the bus in a single-bus system, it is possible that a request generated by a miss will fail. If the request succeeds, the processor is restarted; otherwise the processor will wait a fixed amount of time after the failed request has been returned to it, then retry the request (we experimented with an Ethernet-like back-off policy but it did not present obvious advantages).

In addition to the above states, the protocol uses transition states to inhibit additional requests for a block when the state of the block is in flux. This helps to greatly reduce the complexity of the state space of a given block. For example, suppose a processor wishes to write to a given block which is not in its cache (call it cache **X**). The block exists in a cache (call it cache **Y**) in another cluster in state DIRTY. To obtain the block, cache **X** must send a request which propagates up and then back down the hierarchy to cache **Y**. Assuming the request succeeds, the up-to-date value of the cache block won't be found in any cache until the reply has reached cache **X**. Because of this, all other requests for this block must be canceled and retried until the block is in a stable state in cache **X**. This is done by allocating a block (if necessary) on each cache on the path from **X** to **Y** and assigning a transitional state to that block. Then, when under the normal cache snoop mechanism a cache detects a request on the bus(es) it is attached to that matches a block in a transition state, it signals that the request should be canceled. The request is then sent back to the originator, which will try the request again after waiting a period of time.

Occasionally, a request from a cache's upper bus and another request from its lower bus will attempt to access the same cache block on the same cycle. When this occurs, the request from the lower bus will be delayed until the request from the upper bus has been processed.

The reason that the request on the upper cache is given preference is that it is likely that the upper request has traveled further than the lower request, hence making it wait for the lower request will likely force the upper request (rather than the lower) to be retried. This is a more expensive operation than retrying the lower request.

In a similar vein, two (or more) conflicting block requests may exist waiting for access to the same bus at the same time. If all the conflicting requests are in lower cache bus buffers, then the first request to obtain the bus has priority over all other requests. However, any upper cache bus request has priority over any lower cache bus request, for the same reason given above that an upper bus transaction has priority over a lower bus transaction if they access the same block on the same cycle.

2.3 Evaluation methodology

Our architectural evaluations are performed on a cycle by cycle execution driven simulator, a modification of Cerberus [BAD89, AB93]. The processor module simulates (at a crude level) the pipeline delays of a RISC processor. All instruction references and all private data references are assumed to hit in the processor cache with no additional delay. Shared references that hit in the simulated cache and require no further action by the memory system also require a single cycle to finish.

All busses in our simulations are 64 bits wide and split-transaction on non cache-to-cache transactions. Arbitration for bus access is done on a round-robin basis, and requires one cycle. We assume the bus has separate address lines so that addressing information may be transmitted at the same time as data for the same transaction.

In the baseline architecture, the processor caches (for shared data) were 8K bytes, 2-way set-associative and had either an 8 byte or a 64 byte block size. Bus caches were either 32K or 128K bytes, 4-way set associative and had the same block size as the processor caches. If a shared reference requires the use of the memory system, the delays are 6 cycles plus one cycle per 8 bytes for a hit in the bus cache (latency of 14 cycles for a 64 byte block). If the reference misses in the processor and bus caches, the delays are 20 cycles plus two cycles (one for the upper and one for the lower bus) per 8 bytes for a memory access (latency of 36 cycles for a 64 byte block). These figures assume no contention on the busses.

We used 6 applications to test and evaluate the proposed designs. We chose these benchmarks because together they exhibit a wide variety of reference patterns. In all cases, we gathered statistics only for the parallel section of the program.

The Gauss – a gaussian elimination program without pivoting [Dar88]– and Cholesky – sparse matrix factorization from the SPLASH suite [SWG92] – benchmarks both have good spatial locality and exhibit high hit rates. MP3D – a 3-dimensional molecule simulator also from the SPLASH suite – exhibits a great deal of true sharing, and was written to avoid false sharing. Both the Topopt – topological optimization on VLSI circuits using a parallel simulated annealing algorithm [DN87] – and Pverify – a verification program which compares two circuits and reports if they are functionally equivalent [Egg89] [MDWS87] – applications exhibit both true and false sharing, even using moderate block sizes. Lastly, the Barnes

application – another of member the SPLASH suite simulating the effect of gravity on a system of bodies – does relatively few shared accesses.

3 Evaluation of the UMA architecture

In this section, we evaluate the effectiveness of the UMA hierarchical architecture. This evaluation is done under two protocols: (1) the base protocol of the previous section, and (2) a protocol that, in general terms, uses a large block size (64 bytes) for read transfers and a subblock size (8 bytes) as a unit for coherence. The motivation for the second set of experiments is that the subblock protocol was found to be successful when applied within a cluster [AB95]. We first describe briefly this second protocol.

3.1 Subblock protocol for two level architecture

The subblock protocol is intended to work with sector caches where blocks are subdivided into subblocks (in our case, blocks of 64 bytes are divided into 8 subblocks of 8 bytes each). In the sector caches, both blocks and subblocks have states.

At the intra-cluster level the protocol is similar to the one described in [AB95]. There are 4 block states and 4 subblock states. Cache to cache transfers are favored. On read misses, as many valid subblocks in the block as possible are transferred. On writes to clean subblocks and write misses, only the subblock to be written is invalidated. The motivation is to avoid false-sharing as much as possible.

When expanding the single bus subblock protocol to a hierarchy of busses, we strove to make a minimal amount of changes to the single bus protocol in order for it to work at the cluster level in a hierarchical system. At the same time, we based the upper level subblock protocol on the "standard" hierarchical protocol given in the previous section. Unlike the lower level protocol, the block state in a bus cache (INVALID, CLEAN SHARED or DIRTY SHARED) merely describes what type of subblocks may be found in the block. The block state makes it simpler to preferentially replace blocks that will not require a write back.

Given an unlimited number of state bits, we would have chosen to use 5 subblock states, corresponding to the 5 upper level states in the standard hierarchical protocol. Concerns about the number of state bits required forced us to discard a state. Of the five states, the choice came down to discarding either the DIRTY OWNED or DIRTY SHARED state. Because upper bus bandwidth is usually the bottleneck in hierarchical bus systems, it should be saved in preference to saving lower bus bandwidth. Hence, we decided to discard the DIRTY SHARED subblock state. A complete protocol description for both levels can be found in [And95].

3.2 Evaluation of the UMA architecture

We simulated ten different combinations of processors and clusters, ranging in size (and cost) from the 2 clusters of 2 processors (2X2) configuration to the 8 clusters of 4 processors

configuration. Two of the organizations (2X8 and 2X16) are unbalanced because they have many processors per cluster and few (i.e., 2) clusters. Not only must the cluster bus support the traffic of 8 or even 16 processors, but also the processors will suffer more cache misses when the sum of the sizes of the processor caches approaches or exceeds (in the 32K bus case) the size of the bus cache, possibly causing many forced replacements as the bus cache maintains inclusion. Nevertheless, we still include those configurations in our results.

In the following paragraphs, we give detailed results for the six benchmarks we studied. For those applications for which results are plotted, we present the ten configurations from left to right in order of increasing number of processors; and for a given number of processors, in increasing number of clusters. In the legend, the bars are labeled with the block size (8b and 64b for the standard hierarchical protocol with small – 8 bytes – and large – 64 bytes – block sizes and Sb for the subblock protocol; we also call these protocols 8b, 64b, and Sb). All figures are configurations with 128KB bus caches and speed-up is versus a one processor configuration with an 8 byte size block and 128KB bus cache (in that case a second level cache).

Barnes The Barnes application sped up well on the cluster architecture, achieving near-linear speedup up to 16 processors, and a speedup of about 27 using 32 processors. Because of Barnes' small data set, there was little difference between using the small or large bus cache. And since Barnes did comparatively few shared references, varying the block size had only a minor effect on overall performance. Using the small block size did have a slight (up to 5 percentage points) advantage over using the large block size, particularly when 16 or 32 processors were used. When we examined the statistics for the number of times a bus transaction had to be retried, we found that the number of retried bus transactions was relatively large when a large number of processors were used. This was especially true in the 64 byte blocks case.

Running the subblock protocol had very little effect on program performance. In some cases, using the Sb-protocol increased performance by 1–2%, while in others it had the same speedup as the 8b-protocol.

Gauss Gauss performs best when using a large block size because of its spatial locality. This is quite evident from Figure 2. Though the speedup for the 8b-protocol didn't get much above 10 for any configuration, the 64b-protocol achieved excellent speedup on nearly all configurations; in some cases, an apparent superlinear speedup was obtained because the additional processors and bus caches could hold more of the working set, and because the single processor used for the comparison has an 8 byte line size. Not surprisingly, those configurations that had more clusters had better speedups than those with few clusters, though the differences were small except for the 2X16 configuration. In that case, the lower bus had a high utilization rate which led to a drop-off in performance.

The reason for the 64b-protocol's superior performance is that it used far fewer transactions, about 85% less, to move data around. The number of bytes transferred on the upper bus was slightly larger, from 10 to 15% depending on the configurations, indicating that most

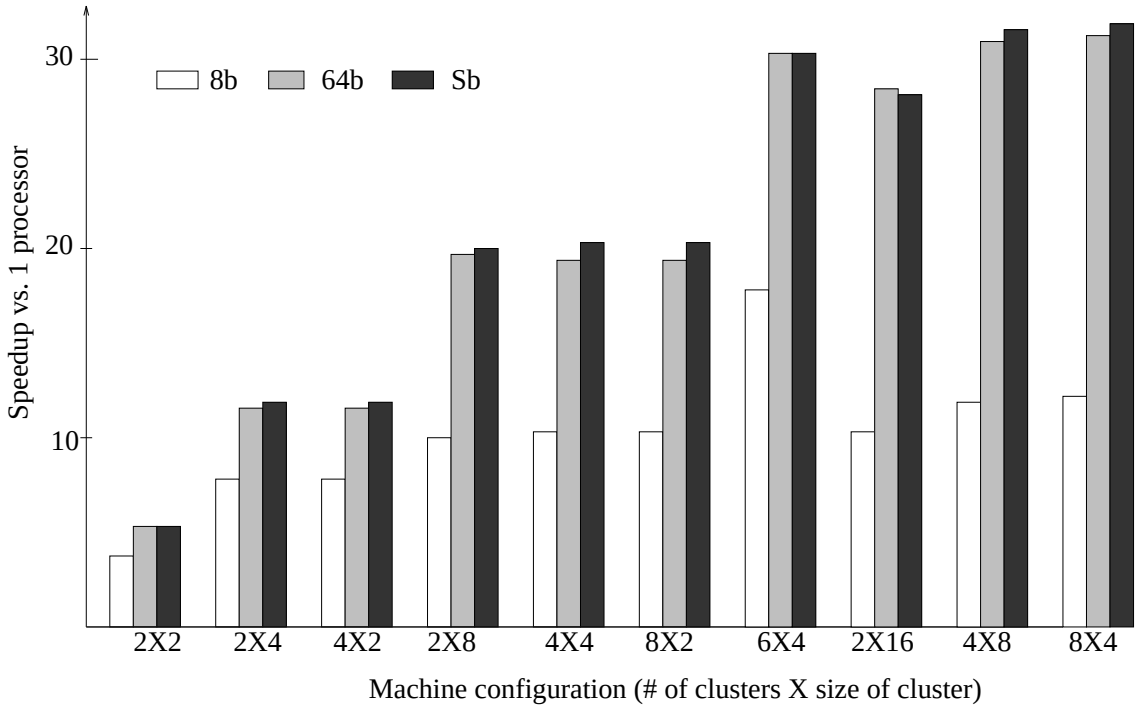

Figure 2: UMA architecture – Gauss speedups

of the data transferred in each 64 byte block was needed (sooner or later) by the processor, and that false sharing was not a problem.

The Sb-protocol performed quite well on Gauss. For all configurations and both sizes of bus cache, the subblock protocol equaled or slightly exceeded the already excellent performance of the 64b-protocol. The lone exception to this was the (unbalanced) 2X16, 128K bus cache organization where performance dropped by 1%. Like the case with single bus machines [AB95], the subblock protocol's policy of using large transfers worked well in applications (like Gauss) with good spatial locality. The number of upper bus transfers under the Sb-protocol was nearly identical to the number used by the 64b-protocol, and the number of upper bus bytes transferred was less.

Cholesky Like Gauss, Cholesky performs best when using a large block size. Unlike Gauss, Cholesky did not achieve near linear speedup because of the limited size of the input matrix that restricts available concurrency [SWG92]. Maximum speedup of about 17.5 was obtained using the 64 byte block size and 32 processors (in the 8X4 or 4X8 organizations) and 128K bus cache size. The advantage of using the 64b-protocol over the 8b-protocol was 12–30% for large bus caches and much more for small bus caches. Generally, the differences were largest when there were more bus transactions, either because of small bus caches or larger numbers of processors. The improved performance was due to the reduced number of bus transactions needed by the 64b-protocol. The reduction was of the order of 80% on both upper and lower busses. Using the large block size reduced the total number of transactions

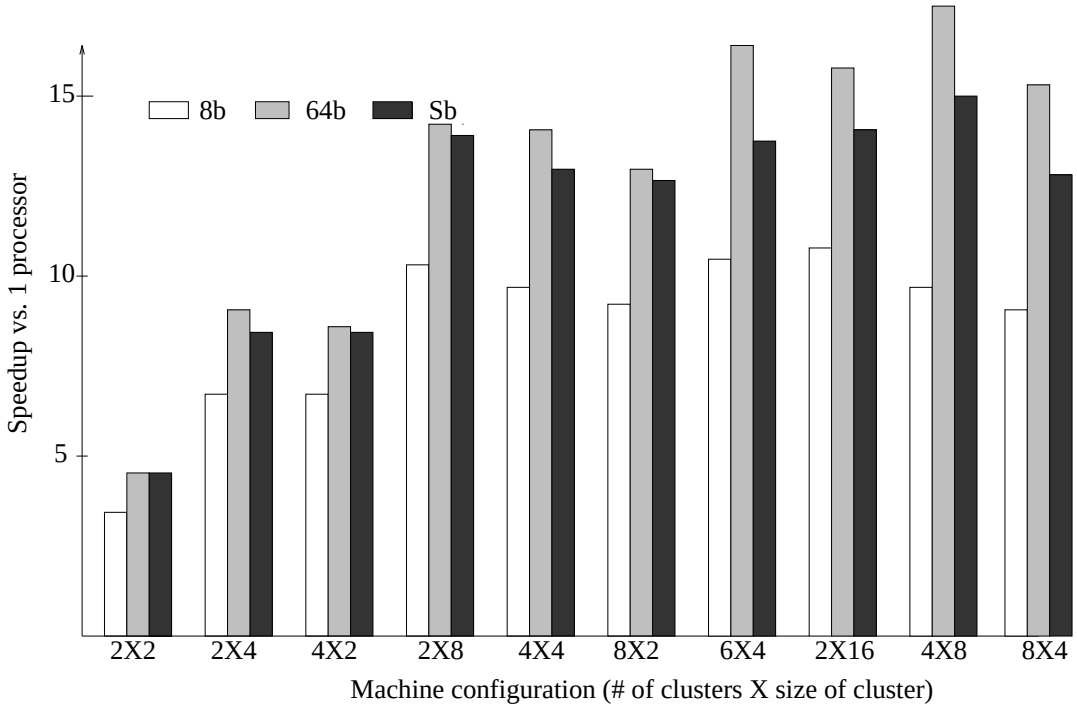

Figure 3: UMA architecture – MP3D speedups

on the lower busses as well.

The performance of the subblock protocol was better than that of the 8b-protocol (up to 15% for balanced configurations with a large number of processors) but not as good as that of the 64b-protocol (by about 15 to 25% in the same conditions). This is not a surprise, given the subblock protocol's performance on Cholesky was closer to the 8b than the 64b-protocol when using a single bus [AB95]. This stems from the nature of Cholesky's data access pattern to the main data structure.

MP3D MP3D performed best when using a 64 byte block size. Speedups reached a maximum of about 10 under the 8b-protocol and about 15 under the 64b-protocol. (cf. Figure 3). MP3D's medium speedup is largely due to a great deal of true sharing. The sharing caused a large amount of traffic on the upper bus, which was saturated when using 16 or more processors. For all but the 2X8 and 2X16 configurations, the upper bus was saturated before the cluster busses were saturated.

The reason for the difference in performance caused by increasing the block size from 8 bytes to 64 bytes is again due to the reduction in bus transactions on both upper and lower level busses. The gain in performance caused by this reduction was partially offset by a larger number of bytes transferred, sometimes as much as 2.7 times on the upper bus and 3.2 times on the cluster bus.

As in Cholesky, the performance of the Sb-protocol was in between those of the other two protocols. In contrast with Cholesky, though, the performance was much closer to that

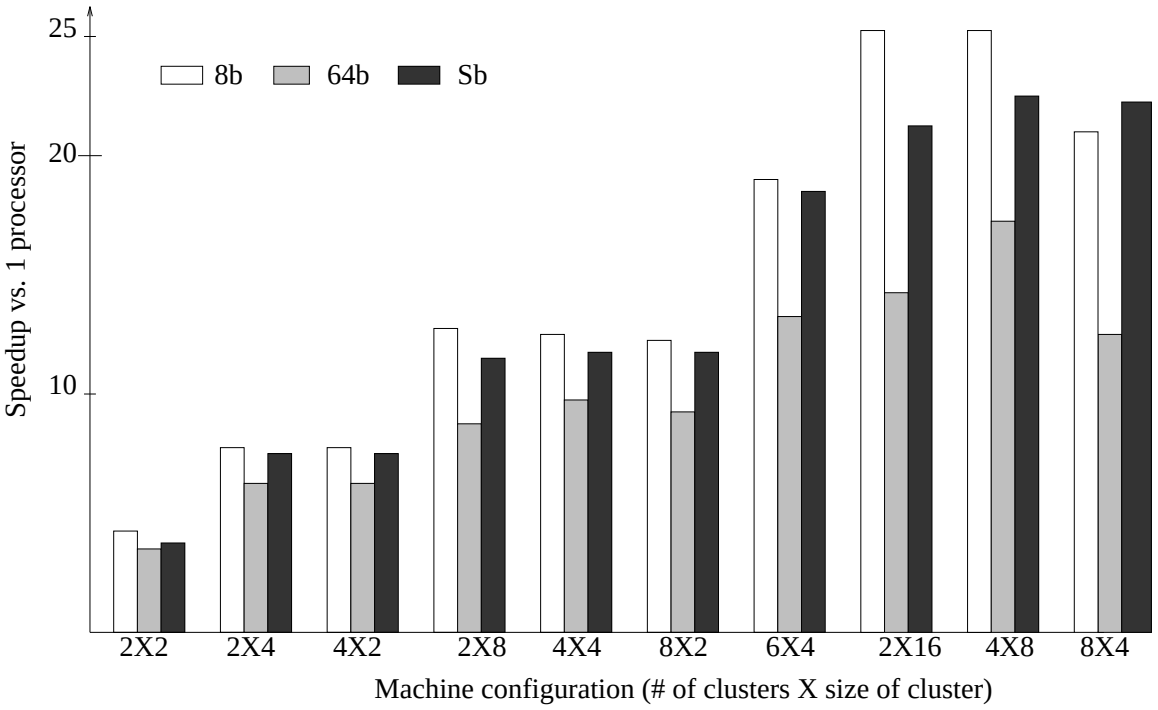

Figure 4: UMA architecture – Topopt speedups

of the best protocol.

Topopt Topopt exhibits both true and false sharing. Thus, unlike the 3 previous applications, the best speedups are obtained under the 8b-protocol (cf. Figure 4). Since Topopt has a small working set size, speedups were similar for the small and large bus caches. Under the 8b-protocol and with large bus caches, the speedups were 7.8 for 8 processors, 12.7 for 16 processors, and 21–25 for 32 processors depending on the cluster configurations.

The effects of false sharing were noticeable by the fact that bus transactions (both read and invalidate) in the 64b-protocol increased by a factor of 1.5 to 2.1 over the 8b-protocol. The differences between the 2 protocols in terms of the number of transactions increased as the number of clusters increased. This is to be expected, since more of the false sharing traffic must cross the top interconnect when more clusters are used.

Since every read transaction using 64 byte blocks transfers 8 times as much data as when 8 bytes blocks are used, between 11 and 15 times more data was transferred on the upper bus in the 64b-protocol. Again, the disparities between the two protocols increased as the number of clusters was increased (keeping the number of processors constant). Because the subblock protocol (which does not suffer from false sharing) also transferred many times the data that the 8b-protocol did, part of the large increase in data transferred in the 64b-protocol was due to the fact that many of the additional words transferred with the requested word(s) were not used by the processor.

The performance of the Sb-protocol was very close to that of the 8b-protocol except when

the configurations were unbalanced (e.g., 2X16). Conversely, in balanced configurations, the Sb-protocol was very close to 8b or even better (8X4).

Pverify Since there is a large amount of false sharing in Pverify, we expected that like in Topopt the 8b-protocol would perform best. With large bus caches, the working set of the application was in-cache and hence good speedups (up to 28 for 32 processors) were attained. Speedups with the 64b-protocol were much more modest (speedup of 8 only for 32 processors) and the difference between the two protocols got wider as the number of processors increased causing saturation for the 64 byte block size on both the upper and lower busses.

The Sb-protocol performance was close, but inferior, to that of the 8b-protocol when the number of processors was limited. As soon as the number of processors attached to a cluster, or the number of clusters increased, the performance of the Sb-protocol dropped off considerably. Overall, though, it was always better than that of the 64b-protocol.

3.3 Summary

The cluster concept performed well on our varied mix of applications. Speedups were generally very good, except for Cholesky and MP3D. Cholesky did not speedup well because the data set we chose to simulate was small (to keep simulation times manageable) and lacked sufficient parallelism. MP3D, on the other hand, had so much sharing that it saturated the upper bus, long before it saturated the lower busses except when only two clusters were used. Like the case for single bus systems, using the "wrong" block size for an application substantially reduced performance because of increased bus traffic. The subblock protocol, which was designed to take advantage of an application's spatial locality while avoiding false sharing problems, functioned well on most of the applications. Though in only a few cases did it exceed the performance of the best choice of block size, its execution times were nearly always close to the faster of either the 8b or 64b-protocol. However, its behavior was not quite as good as it was for the single bus case [AB95]. Some explanations for this include:

- The subblock protocol lacked a DIRTY SHARED subblock state at the bus cache level. This caused some level of increase in the number of cluster bus transactions.
- The subblock protocol allocated cache space on a block (64 byte) by block basis. This works well if a good portion of the block is used, or if the cache is large enough that capacity/conflict misses are not a problem. This was often not the case for Topopt and Pverify, especially at the processor level where small caches (8K) were used.
- The small processor caches used in the hierarchical system increased the replacement of cache blocks. This had the effect of increasing the average number of valid subblocks per block. For most applications, this would speed execution, but it did just the opposite for Topopt and Pverify since the additional valid subblocks were often not used. In addition, the subblock protocol's policy of transmitting more than just the

requested subblock in response to a read request wasted bus bandwidth because in many cases the additional subblocks were not used by the requester.

It would be interesting to perform experiments where the subblock protocol would be used only intra-cluster and large blocks transmitted at the upper bus level. Similarly, one could expand the concept of subblock with the subblock in the bus cache having the size of a block in a processor cache. The savings in tag bits would be very large but the protocol would be slightly more complex.

One of the most important goals in designing a cluster architecture is that of *balancing* the traffic that will be carried on each level. If a system designed for a given number of processors has many processors per cluster, and few clusters, then traffic on the upper bus will be relatively small, since a given processor can access a good proportion of the other processors in the system merely by accessing the cluster bus. The biggest drawback is that the cluster busses can be saturated by the traffic from the many processors per cluster, while the upper bus is relatively unloaded. This often happened with the 2X16 configuration. On the other hand, if few processors per cluster are used, then the upper level bus becomes saturated long before the cluster level busses (this happened, in a less visible way, for the 8X2 configuration). In an ideal situation, cluster bus utilization should increase at the same rate as upper bus utilization. In practice, this is nealy impossible to achieve because various applications will load lower and upper busses in different ways.

To summarize our results, each program performed consistently well on three balanced configurations, 4X4, 4X8 and 8X4 when using either the best block size or the Sb-protocol. Since the tendency is towards larger block sizes, in order to save tag bits, our results are encouraging. To dwell a little more in detail, Barnes had so few shared references that both bus levels were lightly loaded, even in the unbalanced configurations. Much of Gauss's traffic was to and from memory, because of conflict misses. Therefore, upper bus utilization was high, even for the balanced architectures. The balanced architectures did not experience cluster bus saturation, unlike, for example, the 2X16 configuration. Cholesky's lack of speedup even on the balanced systems was due to the absence of application parallelism caused by the relatively small input data set. Bus utilization with the large blocks never exceeded 40% on either upper or lower busses. MP3D's poor speedup, on the other hand, was due to large amounts of true sharing which caused upper bus saturation for all but the most imbalanced of configurations long before linear speedup was reached. When using the 8 byte block size with Pverify, the 8X4 and 4X4 configurations had very well balanced bus loads. For these configurations, the Sb-protocol performed quite well. On Topopt, the 4X8 and 4X4 configurations overall were the most balanced, especially when the large (i.e., wrong) block size or the Sb-protocol were used.

4 From UMA to NUMA variations

The base architecture has a Uniform Memory Access (UMA) structure. Two factors argue for changing from the UMA model to a NUMA model where physical memory is distributed

among the processors or clusters in a system. First, this approach is more modular and hence expandability is achieved more easily. Second, from a performance viewpoint, some reduction in memory access times could be achieved if an application placed its data structures so that misses to memory could be satisfied by the memory located in the cluster.

To investigate data placement in a hierarchy of busses, we modified the hierarchical architecture detailed previously, moving physical memory into the clusters. These modifications are presented below. We also modified some of the applications, namely Gauss and MP3D, so that they allocated data in the cluster in which it is most heavily used. We report on simulation results for these two applications whose memory access patterns are most representative of those simulated in the UMA architecture.

4.1 Cluster Memory and Memory Allocation

We modified the base architecture of Figure 1 so that memory was distributed among the clusters. We show a two level system in Figure 5. We assumed that each cluster's memory bank could observe and respond to memory requests, both on the upper bus and the appropriate cluster bus. Further, associated with each memory block is a bit indicating if memory has an up-to-date copy of the block. When the system is initialized, all such bits are initially set to *true*. The bit is set to *false* when the memory block is accessed by a cache outside the memory's cluster; it is reset to true when the block is written back. It is necessary to turn the bit off on both read and read exclusive requests from remote caches, since a remote processor may write to a block which has been read from memory without a bus transaction, since the block is loaded in the VALID EXCLUSIVE state. Since we assumed that cluster memory banks could respond to requests from outside the cluster in the same manner as in the UMA case, non-local requests in the cluster system also require the same amount of cycles as before, i.e., 36 cycles. Local requests are satisfied in 23 cycles.

We changed the memory allocation mechanism from fine grain (i.e., a block) memory interleaving to coarse grain (i.e., a page, in our case 4K) interleaving. One problem with using coarse-grain interleaving and allocating data in "local" pages is that it can increase conflict misses. This occurs when the number of clusters is a power of 2 and when the bits comprising the page number are part of the cache index function, which happens when the cache size divided by the cache associativity is larger than the page size. Thus, when a processor explicitly allocates data in its cluster's memory, that data will be mapped to only $1/n$ of the cache's space, where n is the number of clusters.

Our solution to this underutilization of the cache was simple: we used additional address bits in the cache indexing function so that successive pages mapped in the same cluster would map to different parts of the cache. The additional bits were taken from the portion of the address just above the bits normally used to index the cache.

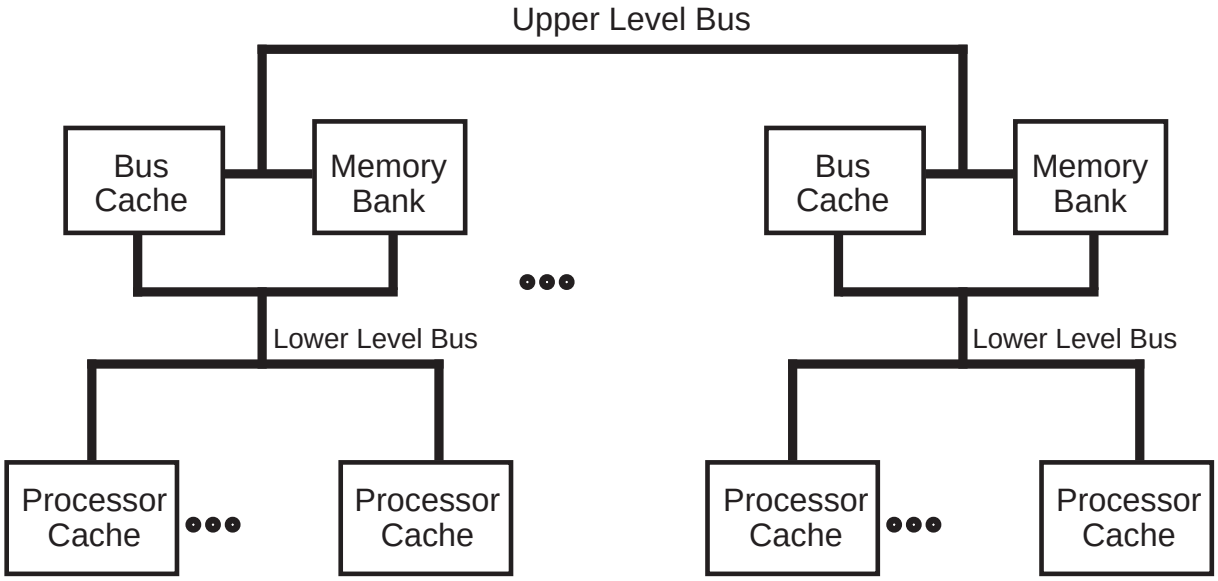

Figure 5: NUMA Cluster Architecture

4.2 Results of the evaluation

In the following, we first describe the modifications made to two applications, Gauss and MP3D, then present the results of running the program on the simulator. For comparison purposes, we varied several architectural parameters for our evaluation. One parameter is whether or not memory was distributed in clusters, i.e., the UMA architecture. Another parameter is whether or not the modified cache index was used. Finally, either the original or the modified version of the program could be used.

Table 2: Evaluated Architectural Variations

Program	Memory Clusters	Modified Index	Modified Program
Base (Gauss, MP3D)	No	No	No
Unmodified Cluster (MP3D)	Yes	No	No
Unmodified Cluster/MI (Gauss)	Yes	Yes	No
Modified Cluster (Gauss, MP3D)	Yes	Yes	Yes

Table 2 lists the configurations that appear in the graphs to follow. We simulated both the regular and modified index (MI) function for both applications. However, to reduce the

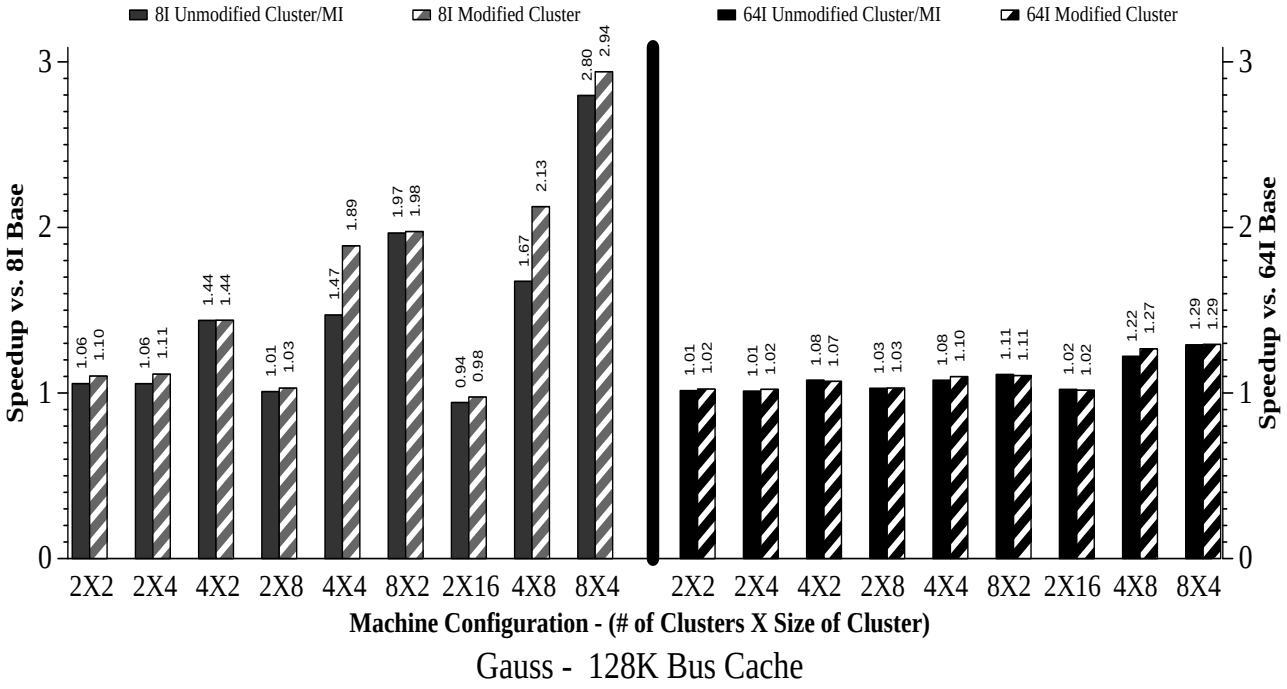

Figure 6: NUMA – Gauss Speedups

number of graphs presented, we will present only one data set for Unmodified Cluster (UC). For MP3D, we will present UC using the normal index function, since using the modified index function had little effect. On the other hand, we will show data for Gauss on UC with MI, since using MI made a significant difference in UC's performance. For the speedup curves, we normalized to the performance of the base architecture for the same configuration and block size. Hence a bar of length 1.0 indicates that the configuration performed the same as the base architecture.

Gauss The Gauss benchmark solves the system $A \times x = b$. Rows are statically assigned to processors to work on. Since the memory for the rows is allocated dynamically, it is straightforward to allocate each row in the proper cluster. Though the b elements are also statically assigned to processors, we did not allocate them into a specific cluster, since each element would have required a separate call to the allocator, increasing the memory overhead for the b vector by a large amount.

One complication in examining the results for Gauss is the fact that it had many conflict misses, even when many 128K bus caches are used (i.e., many clusters). Reducing the number of conflict misses by using a different function to index the cache could increase system performance, irrespective of whether or not cluster allocation is used. In order to separate out this effect from the effect of the memory allocation procedure *per se*, when we ran the unmodified program on the cluster architecture, we used the additional address bits to index the cache. The results of the simulation are shown in Figure 6. The main

observations are as follows:

- the NUMA architecture yields extremely significant performance improvements when a small cache block size is used (up to almost a 200% gain) and non-negligible ones when a large block size is used (up to 30%). The higher gains for the small block size stem from two factors: (i) because of Gauss's high spatial locality, a miss to a large cache block brings in other needed data as well as the requested data, which reduces the number of bus transactions when compared to using a small block size (recall Section 3.2). Since cluster allocation potentially saves bus cycles on every memory transaction, it has more opportunities to improve performance when small block sizes are used; and, (ii) since the UMA architecture performed already so well with large block sizes, there wasn't a great deal of room for improvement.

  The gains when using 4 or 8 clusters were large enough that the small block performance came close to or exceeded UMA's performance when using large blocks. However, it was still 25 to 30% behind the performance of the NUMA with large block sizes.

- The best relative – and absolute – improvements occurred when the number of clusters was large and the NUMA architecture was balanced (4X8 or 8X4).

- Directing the memory allocation (i.e., using the modified program) brought minimal improvement for this application. This indicates that most of the NUMA benefits were due to localization of memory and reduction of conflict misses.

MP3D For MP3D, we cluster allocated the particles, since they are statically assigned to processors at the beginning of the program. Unfortunately, the original program allocated all particles as a contiguous single dimensional array, which cannot be spread out among memory clusters so that particles are allocated in the proper cluster. In addition, processors are not assigned a contiguous block of particles to manipulate; instead, small groups of particles are assigned to processors in a round-robin fashion. In order to maintain the same assignments of particles to processors, we converted the single-dimension array into a dynamically allocated, three-dimensional array of particles. This had the effect of increasing the total working set size, because additional storage was required for the arrays of pointers. The change also increased the number of instructions needed to access the particle structure. Thus, in order for the modified program to show a speed increase, it would have to reduce memory access times enough to make up for the additional cycles spent on accessing the particle data structure.

The results of the simulation are shown in Figure 7. In general terms they are the same as for Gauss but with less pronounced effects.

- When the number of processors was small, the NUMA architecture was slightly beneficial for the small block sizes (up to 10% performance increase) and had no impact on the large block size configurations. Performing the program modifications outlined above had a detrimental effect since the computational overhead was larger than the gain in memory access times.

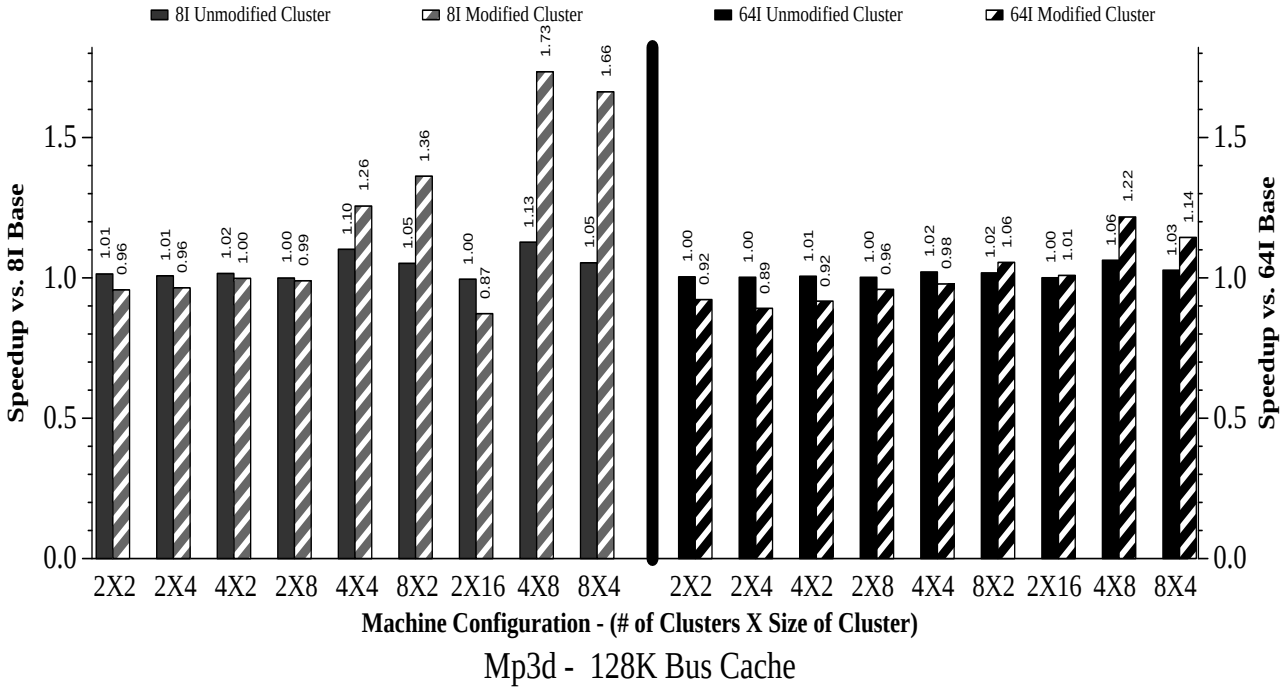

Figure 7: NUMA – MP3D Speedups

- When the number of processors was large (32), NUMA without program modification showed a small benefit. When program modifications were included the benefits were much larger for small blocks and larger for large blocks, provided that the architecture was balanced otherwise there could be a significant degradation (13% in 2X16 case for small blocks).

- When the modified program was run, there were in all cases significant improvements in upper bus traffic, both in terms of reduced number of transactions and number of bytes transferred. However, this did not always translate into overall performance improvements because of (i) the computation overhead due to the transformation in addressing structure, and (ii) there was a higher number of cluster bus reads thus saturation of cluster busses was reached more quickly (mostly in unbalanced cases like 2X16 configurations).

4.3 Previous work and summary

Singh *et al.* examined the effects of data placement on the DASH, where the granularity of data distribution to home nodes is the page [SJGH93]. A general technique they tried was the *first touch*, in which data was allocated in the home cluster which first accessed it. This technique has the potential to work well only when data structures are initialized in parallel by the processor which is going to access it the most. In most cases, first touch improved application performance over round-robin allocation, but occasionally it was worse.

Restructuring the application's data worked even better than the "blind" first-touch method, though it did make the code more difficult to write, and decreased performance in one instance when a small data set was used.

Erlichson *et al.* investigated the benefits of clustering in a NUMA architecture which differs from ours in the sense that a cluster was made up of processors sharing a first level cache and that clusters were connected by an interconnection network. Caches were made coherent via a full directory scheme [ENSO94]. They found limited advantage to clustering because of contention and larger access times to the shared cache.

In our case, we found that cluster allocation succeeded in reducing upper bus traffic on the two applications we simulated. It also increased application performance for all configurations when running the Gauss application, and for balanced architectures for MP3D. Using cluster allocation succeeded in reducing the number of transactions and bytes transferred on the upper bus for both applications. Reducing upper bus traffic is in general a good way to increase application performance, since saturation on the upper bus often limits application performance. However, lowering upper bus traffic is not always enough to increase application performance. At times other factors, such as changing the code and data structures to do cluster allocation, and changing the cache index function can have an adverse impact on cache hit rates and can increase the total number of instructions executed.

5 Conclusion

In this paper we have presented and evaluated several variations on a hierarchical cluster architecture where the interconnect is a hierarchy of busses.

Starting with a UMA architecture, we briefly introduced a basic hierarchical snoopy-based cache coherence protocol. We then extended this protocol to a subblock protocol for sector caches so that effects of false sharing could be minimized. We simulated, on a cycle by cycle basis, six benchmarks on both the standard and subblock protocols on a variety of two level cluster configurations, varying both the number of clusters and the number of processors per cluster.

Applications had very different performance, depending on the block size used with the standard invalidate protocol. Overall, though, the benchmarks performed well when using the "correct" block size. When configurations were used that balanced bus traffic between the upper and lower levels of the system, saturation of the upper bus sometimes occurred, but in all but one of the benchmarks did not noticeably impair application speedup. Only on MP3D did upper bus saturation (due to a great deal of true sharing between clusters) strongly affect performance. In spite of our good results using identical busses at both the upper and lower levels of the system, it would be advisable to have a "beefier" interconnect above the cluster level. Since it is unlikely that non-cluster busses will be shorter than cluster busses, clocking the non-cluster busses faster than the cluster busses will probably not be an option. Instead, wider busses (to improve bandwidth) or multiple (possibly interleaved) busses [BBW92] would allow more clusters (hence more processors) to be effectively connected in a hierarchical system.

The hierarchical subblock protocol performed relatively well on the six benchmarks. Unlike the conventional protocol used with a fixed block size, its performance across the tested benchmarks was much more consistent. The overall performance improvements of the subblock protocol were relatively less impressive than in the case of single bus system [AB95]. One of the reasons is that we wanted to minimize the number of states in the bus cache in order to make the standard and subblock protocols more comparable cost-wise. Another factor that should be investigated is whether limiting the subblock protocol at the intracluster level or using different block and subblock sizes for the private and bus caches would improve performance.

We then examined the issue of data placement in NUMA hierarchical systems when physical memory is distributed among the system's clusters. We found that providing a simple method for allocating data in the cluster nearest the processor was effective in reducing upper bus traffic in the two applications studied. The reductions in upper bus traffic led to consistent performance gains for only one of the two applications. For the other one, performance was slightly improved remained the same because the application's algorithm had to be changed to accommodate cluster allocation. The main conclusion of this part of our study is that NUMA architectures will work best when the architecture is balanced. However, this conclusion is based on a limited application sample. More applications should be simulated where tuning the memory allocation cannot be performed, e.g., because of the dynamic allocation of threads, or where the computation to communication ratio is different.

References

[AB93] Craig Anderson and Jean-Loup Baer. A multi-level hierarchical cache coherence protocol for multiprocessors. In *Proc. of 7th Int. Parallel Processing Symoposium*, pages 142–148, 1993.

[AB95] Craig Anderson and Jean-Loup Baer. Two techniques for improving the performance of bus-based multiprocessors. In *International Symposium on High-Performance Computer Architecture*, pages 264–275, 1995.

[And95] Craig Anderson. *Improving the Performance of Bus-Based Multiprocessors*. PhD thesis, University of Washington, 1995.

[BAD89] Eugene D. Brooks III, Tim S. Axelrod, and Gregory A. Darmohray. The Cerberus multiprocessor simulator. In G. Rodrigue, editor, *Parallel Processing for Scientific Computing*, pages 384–390. SIAM, 1989.

[BBW92] Jonathan Bertoni, Jean-Loup Baer, and Wen-Hann Wang. Scaling shared-bus multiprocessors with multiple buses and shared caches: a performance study. *Microprocessors and Microsystems*, 16(7):339–50, 1992.

[BW88] Jean-Loup Baer and Wen-Hann Wang. On the inclusion properties for multi-level cache hierarchies. In *Proc. of 15th Int. Symp. on Computer Architecture*, pages 73–80, 1988.

[Con94] Convex Computer Corporation. *Convex Exemplar Scalable Parallel Processing System*, 1994.

[Dar88] Gregory A. Darmohray. Gaussian techniques on shared-memory multiprocessors. Master's thesis, University of California, Davis, April 1988.

[DN87] Srinivas Devadas and A. Richard Newton. Topological optimization of multiple level array logic. *IEEE Transactions on Computer-Aided Design*, November 1987.

[Egg89] Susan Eggers. *Simulation Analysis of Data Sharing in Shared Memory Multiprocessor*. PhD thesis, University of California, Berkeley, 1989.

[ENSO94] Andrew Erlichson, Basem Nayfeh, Jaswinder Singh, and Kunle Olukotun. The benefits of clustering in shared address space multiprocessors: an applications-driven investigation. Technical Report CSL-TR-94-632, Stanford University, 1994.

[FIR93] S. Frank, H. Burkhardt III, and J. Rothnie. The KSR-1: bridging the gap between shared memory and MPPs. In *Proc. of Spring 1993 COMPCON*, pages 285–294, February 1993.

[HLH92] Erik Hagersten, Anders Landin, and Seif Haridi. DDM - a cache-only memory architecture. *IEEE Computer*, 25(9):44–54, September 1992.

[LLJ+92] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH prototype: Implementation and performance. In *Proc. 19th Annual Symposium on Computer Architecture*, pages 92–103, June 1992.

[MDWS87] H-K. T. Ma, S. Devadas, R. Wei, and A. Sangiovanni-Vincentelli. Logic verification algorithms and their parallel implementation. In *Proceedings of the 24th Design Automation Conference*, pages 283–290, 1987.

[SJGH93] Jaswinder Pal Singh, Truman Joe, Anoop Gupta, and John L. Hennessy. An emprical comparison of the Kendall Square Research KSR-1 and Stanford DASH multiprocessors. In *Supercomputing*, pages 214–225, 1993.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, pages 5–44, March 1992.

[VSLW91] Zvonko Vranesic, Micahel Stumm, David Lewis, and Ron White. Hector: A hierarchically structured shared-memory multiprocessor. *IEEE Computer*, 24(1):72–79, January 1991.

[Wil87] Andrew Wilson Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proc. of 14th Int. Symp. on Computer Architecture*, pages 244–252, 1987.

[WTP+92] Andrew Wilson, Marc Teller, Thoams Probert, Dyung Le, and Richard LaRowe. Lynx/Galactica Net: A distributed, cache coherent multiprocessor system. In *Proc. of the 25th Hawaii International Conference on System Sciences*, volume 1, pages 416–426, 1992.

[YTB92] Q. Yang, G. Thangadurai, and L. Bhuyan. Design of an adaptive cache coherence protocol for large scale multiprocessors. *IEEE TPDS*, 3(3):281–293, May 1992.