Hierarchical Cache / Bus Architecture for Shared Memory Multiprocessors

Andrew W. Wilson Jr.

Encore Computer Corporation
257 Cedar Hill St., Marlborough, MA. 01701

Abstract

A new, large scale multiprocessor architecture is presented in this paper. The architecture consists of hierarchies of shared buses and caches. Extended versions of shared bus multicache coherency protocols are used to maintain coherency among all caches in the system. After explaining the basic operation of the strict hierarchical approach, a clustered system is introduced which distributes the memory among groups of processors. Results of simulations are presented which demonstrate that the additional coherency protocol overhead introduced by the clustered approach is small. The simulations also show that a 128 processor multiprocessor can be constructed using this architecture which will achieve a substantial fraction of its peak performance. Finally, an analytic model is used to explore systems too large to simulate (with available hardware). The model indicates that a system of over 1000 usable MIPS can be constructed using high performance microprocessors.

Introduction

Although the computation speeds of conventional uniprocessors have increased dramatically since the first vacuum tube computers, there is still a need for even faster computing. Large computational problems such as weather forecasting, fusion modeling, and aircraft simulation demand substantial computing power, far in excess of what can currently be supplied. While uniprocessor speed is improving as device speeds increase, the achieved performance levels are still inadequate. Thus researchers have been seeking alternative architectures to solve these pressing problems.

Many of these proposed solutions involve the construction of multiprocessors, systems which link a large number of essentially Von Neumann machines together with a high performance network[2,3]. Unlike multicomputers, which have also been proposed, multiprocessors provide a shared address space, allowing individual memory accesses to be used for communication and synchronization. All multiprocessors require an interconnection mechanism which physically implements the shared address space. Numerous proposals for such structures appear in the literature, covering a wide range of performance, cost and reliability[1,5,8,14].

In a multiprocessor there are two sources of delay in satisfying memory requests, the access time of the main memory and the communication delays imposed by the interconnection network. If the bandwidth of the interconnection network is inadequate, the communication delays are greatly increased due to contention. Both the bandwidth and access time limitations of interconnection networks can be overcome by the use of private caches. By properly selecting cache parameters, both the transfer ratios (the ratio of memory requests passed on to main memory from the cache to initial requests made of the cache), and effective access times can be reduced[4]. Transfer ratio minimization is not the same as hit ratio maximization since some hit ratio improvement techniques, (i.e. prefetch) actually increase the transfer ratio.

While private caches can significantly improve system performance, they introduce a stale data problem (often termed the multicache coherency problem) due to the multiple copies of main memory locations which may be present. It is necessary to ensure that changes made to shared memory locations by any one processor are visible to all other processors. One solution is to use a central cache controller to arbitrate the use of shared cache blocks[3,13] and thus prevent the persistence of obsolete copies of memory locations. While the central controller enforces multicache coherency, it constitutes a major system bottleneck of its own, which makes it impractical for large multiprocessor systems.

When a single shared bus is used for processor to memory communication, each private cache is able to observe the requests generated by other caches in the system. Thus the use of a shared bus allows the possibility of distributed cache coherency control algorithms. Recently, several proposals for multicache coherency algorithms which utilize a common shared bus have been published[6,7,10,11]. In these systems each cache monitors the transactions taking place on the shared bus and modifies the state of its cached copies as necessary. The important feature of the new multicache coherency algorithms is that no central cache controller is required, rather coherency control is distributed throughout the system. Furthermore, the overhead due to the extra bus traffic required for coherency control is negligible. Since there is only one bus, the ultimate expandability of the system is limited.

The research described in this paper is sponsored by the Defense Advanced Research Projects Agency (DOD), DARPA contract N0039-86-C-0158. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Because the published multicache coherency algorithms are limited in expandability by the need for a common shared bus, it is desirable to extend the algorithms to multiple bus architectures. This paper proposes one such extension which allows a large architecture to be built. It consists of a hierarchy of buses and caches which maintain multicache coherency while partitioning memory requests among several buses. As will be shown, the benefits of the shared bus, multicache coherency algorithms are maintained, while much larger systems are made possible.

Multicache Coherency Algorithms for Shared Buses

Before introducing the hierarchical approach for large multiprocessor systems, a brief review of shared bus, multicache coherency algorithms is in order. Such algorithms attempt to keep all copies of shared memory locations identical, at least to the extent that no transient differences are visible to the processors. If a processor modifies its cache's copy of the memory location, all other copies must be invalidated. With non shared bus switching schemes the traffic due to invalidation messages can become quite large. As will be seen, shared buses provide these messages implicitly.

Write-Through

The simplest scheme for avoiding coherency problems with a shared bus multiprocessor is to use write-through caches. With a write-through cache, each time a private cache's copy of a location is written to by its processor, that write is passed on to main memory over the shared bus. As indicated in the state diagram of Figure 1, each memory location has two states: the *valid* state, which indicates that a copy resides in the cache, and the *invalid* state, where the only copy is in main memory. Transitions to the valid state occur every time a location is accessed by the cache's associated processor. Transitions from the valid state occur every time a cached location is replaced by a different location, and every time a write from another processor for the memory location is observed on the backplane bus.

The transition to invalid when a bus write is observed for a cached location is the mechanism by which cache coherency is maintained. If any caches contain a copy of the memory location being written to, they will invalidate it. If none of the other caches ever actually use that location again, then the coherency scheme produces no additional bus traffic. Of course, one or more of the processors whose caches were purged of copies of the location may request that location later and extra main memory reads will result. Simulation experiments reported later in this paper show that these extra reads are infrequent and contribute very little extra bus traffic.

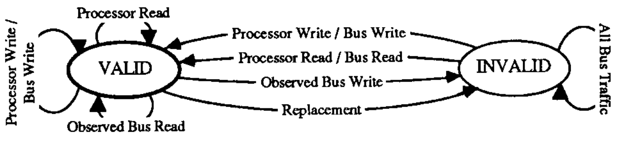

Processor Read
Processor Write / Bus Write
VALID
Processor Write / Processor Read / Bus Read
Observed Bus Write
Replacement
INVALID
All Bus Traffic
Observed Bus Read
Processor Write / Bus Write

Figure 1: State Diagram for a Cache Location in a Multiprocessor Utilizing Private Write-Through Caches

A second copy of each cache's tag store may be required to prevent saturation of the private caches while monitoring bus traffic. Because the caches are write-through, the amount of traffic on the bus will never be less than the sum of all the processor-generated memory writes, which typically comprise 15%-20% of memory requests. Write-through caching is highly effective where limited parallelism (on the order of 20 medium speed processors) is required and is simple to implement.

Write-deferred

Rather than pass all write requests on to main memory, a cache can simply update its local copy and defer updating main memory until later (such as when the modified location is replaced by a new location). Uniprocessor designers have found that write-deferred caches can significantly reduce the amount of main memory traffic from that of write-through caches. Current practical cache sizes produce reductions two to four over write-through caches. It should be expected that two to four times as many processors could be added to a shared bus multiprocessor utilizing such caches. In a multiprocessor, the necessity of coherency maintenance results in a more complicated system with higher bus utilization rates than a pure write-deferred system, but still much lower utilization per processor than write-through.

There are presently several known variations of shared bus oriented write-deferred caching algorithms which maintain cache coherency[12]. One of the first is the write-once scheme[7] which utilizes an initial write-through mode for recently acquired copies to invalidate other caches in the event of a local modification to the data. Figure 2 presents a diagram of the state transitions which occur for a memory location with respect to a cache. A given memory location is in the *Invalid* state if it is not in the cache. When a main memory location is initially accessed it enters the cache in either the Valid state (if a read) or Reserved state (if a write). A location already in the *Valid* state will enter the *Reserved* state if a processor write access occurs. A processor write which causes a transition into the *Reserved* state will be passed through the cache and onto the shared bus. Subsequent processor writes to that location will place it in the *Dirty* state, indicating that the cache's copy is the only correct copy of the location.

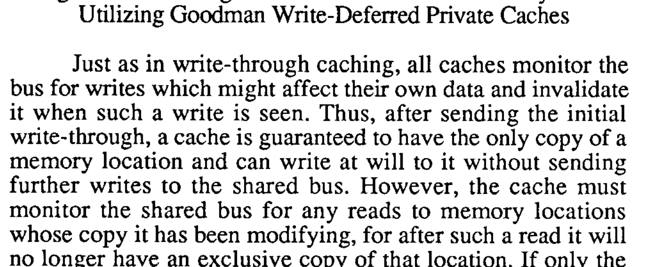


Figure 2: State Diagram for a Cache Location in a System Utilizing Goodman Write-Deferred Private Caches

Just as in write-through caching, all caches monitor the bus for writes which might affect their own data and invalidate it when such a write is seen. Thus, after sending the initial write-through, a cache is guaranteed to have the only copy of a memory location and can write at will to it without sending further writes to the shared bus. However, the cache must monitor the shared bus for any reads to memory locations whose copy it has been modifying, for after such a read it will no longer have an exclusive copy of that location. If only the initial write-through write has occurred, then the only action necessary is for the cache which had done the write to forget that it had an exclusive copy of the memory location. If two or more writes have been performed by the cache's associated processor, then it will have the only correct copy and must somehow update main memory before main memory responds to the other cache's read request.

There are several variations on this basic protocol which could be used to achieve cache coherency with write-deferred caches. The initial write-through can be replaced with a special read cycle, which returns the most recently modified copy of the memory location, and invalidates all others[6]. An additional bus wire can be added which is asserted by any cache which has a copy of the location when a read request is observed, allowing the requesting cache to transition directly to the *Reserved* state if no other cache reports having a copy[9].

Extensions for Even Larger Systems

While a single high speed shared bus can support quite a few processors when private write-deferred caches are used, the bus eventually becomes a bottleneck. A method of extending the above mentioned cache coherency schemes to configurations of multiple shared buses will now be developed. The method involves the use of a hierarchy of caches and shared buses to interconnect multiple computer clusters.

Hierarchical Caches

The simplest way to extend shared bus based multiprocessors is to recursively apply the private cache - shared bus approach to additional levels of caching. As shown in Figure 3, this produces a tree structure with the higher level caches providing the links with the lower branches of the tree. The higher level caches act as filters, reducing the amount of traffic passed to the upper levels of the tree, and also extend the coherency control between levels, allowing system wide addressability. Since most of the processor speedup is achieved by the bottom level caches, the higher level caches can be implemented with slower, denser dynamic RAMs identical to those used by the main memory modules. Average latency will still be reduced, since higher level switching delays will be avoided on hits. To gain maximum benefit from these caches, they need to be large, an order of magnitude larger than the sum of all the next lower level caches which feed into them. But since they can be made with DRAMs, this will not be a problem.

The second and higher levels of caches in the hierarchical multiprocessor require some extensions to maintain system wide multicache coherency. The most important is the provision that any memory locations for which there are copies in the lower level caches will also have copies in the higher level cache. As shown in the state diagram of Figure 4, this is accomplished by sending invalidates to the lower level caches whenever a location is removed from the higher level cache. Because all copies of memory locations contained in the lower level caches are also found in the higher level cache, the higher level cache can serve as a multicache coherency monitor for all of the lower level caches connected to it.

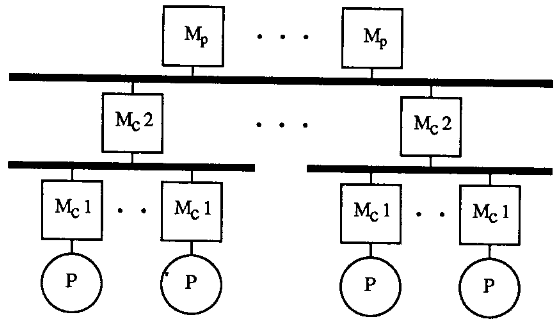
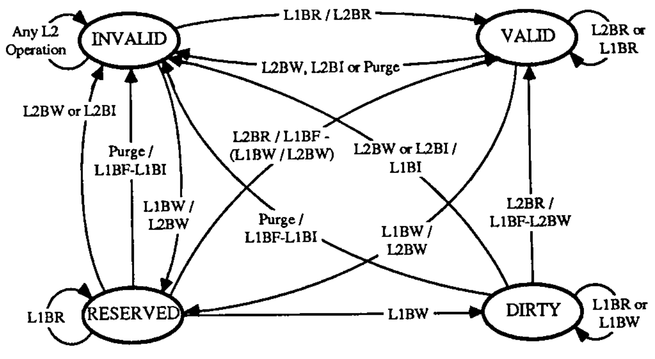


Figure 3: Hierarchical Multiprocessor with two or more Levels of Private Caches and Shared Buses



KEY

L1BR:	Cluster Bus Read	L2BR:	Global Bus Read
L1BW:	Cluster Bus Write	L2BW:	Global Bus Write
L1BI:	Cluster Bus Inval	L2BI:	Global Bus Inval
L1BF:	Cluster Bus Flush	Purge:	Replacement of cache entry

Figure 4: State Diagram for a Second Level Cache Using the Extended Goodman Multicache Coherency Algorithm

Figure 4 shows how the Goodman algorithm is extended for higher level caches. Each cache location still has four states, just as in the basic Goodman cache (see Figure 2). However the cache will now send invalidation or flush requests to the lower level caches when necessary to maintain multicache coherency. An invalidation request is treated in the same way as a bus write by the lower level cache, and a flush request is treated in the same way as a bus read.

To understand how multicache coherency control is achieved in a hierarchical shared bus multiprocessor using the Goodman cache coherency protocol, consider the operation of a two level structure when confronted with accesses to shared data. As indicated in Figure 5, when processor P1 issues an initial write to memory, the write access filters up through the hierarchy of caches, appearing at each level on its associated shared bus. For those portions of the system to which it is directly connected, invalidation proceeds just as described for the single level case. Each cache (such as Mc12 connected with processor P2) which has a copy of the affected memory location simply invalidates it. For those caches at higher levels of the hierarchy, the existence of a particular memory location implies that there may be copies of that location saved at levels directly underneath the cache. The second level cache Mc22 in the figure is an example of such a cache. When Mc22 detects the write access from P1 on bus S20, it must not only invalidate its own cache but send an invalidate request to the lower level caches connected to it. This is readily accomplished by placing an invalidate request on bus S12, which is interpreted by caches Mc16, Mc17 and Mc18 as a write transaction for that memory location. These caches then invalidate their own copies, if they exist, just as though the invalidate request was a write from some other cache on their shared bus. The final result is that only the first and second level caches associated with the processor which generated the write (Mc11 and Mc20) have copies of the memory location. Subsequent writes will stay in the first level cache, or filter up to the second level cache if local sharing or context swapping occurs.

Other shared bus coherency protocols can be modified to work in a hierarchical multiprocessor. For example, the exclusive access read of the Synapse scheme can serve to invalidate other cache copies in the same way as the Goodman initial write. An additional benefit is that the exclusive read

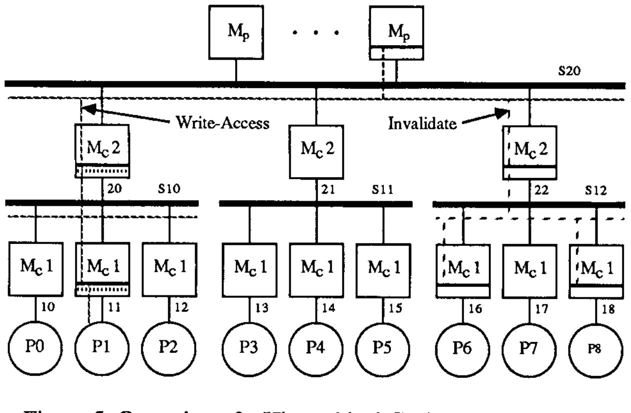


Figure 5: Operation of a Hierarchical Cache Structure when Initial Write-Through Occurs

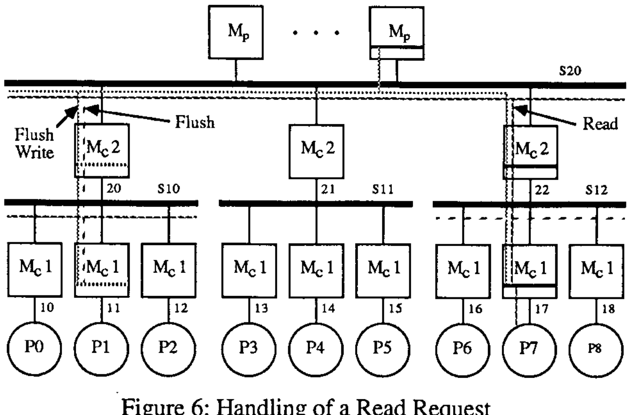


Figure 6: Handling of a Read Request in the Presence of *Dirty* Data

returns the very latest copy of the memory location, so that read/modify/write operations automatically give correct results.

Once a cache location has obtained exclusive access to a location, and has modified the contents of that location, another processor may request access to the location. Since the location will not reside in its cache, the read request will be broadcast onto the shared bus. As with the single level scheme all caches must monitor their shared buses for read requests from other caches and make appropriate state changes if requests are made for locations for which they have exclusive copies. In addition, if their own state indicates that there might be a dirty copy in a cache beneath them in the hierarchy, then they must send a "flush" request down to it. These flush requests must propagate down to lower levels of the hierarchy and cause the lower level caches to modify their state, just as though an actual read for that location had been seen on their shared buses. Figure 6 indicates what can happen in a typical case. Assume that caches Mc11 and Mc20 have exclusive access to a location as a result of the write sequence from the previous example. If processor P7 now wishes to read that location, the request will propagate up through the hierarchy (there will be no copies in any caches directly above P7 so the request will "miss" at each level). When it reaches bus S20, cache Mc20 will detect the need for relinquishing exclusive access and possibly flushing out a dirty copy of the memory location. It will send a flush request down to bus S10 where cache Mc11 will relinquish exclusive access and send the modified copy of the memory location back up the hierarchy. Depending on which flavor of write-deferred scheme is used the data will either return first to main memory or go directly to cache Mc22 and hence to cache Mc17 and processor P7. The copies in Mc20 and Mc11 will remain, but will no longer be marked as exclusive.

An important point to note is that only those lower branches that actually have copies of the affected memory location are involved in the coherency traffic. The section connected with Mc21 does not see any invalidates or flushes and thus sees no additional traffic load on its buses. Thus cache coherency is maintained throughout the system without a significant increase in bus traffic, and lower level pieces of the multiprocessor are isolated from each other as much as possible. The combined effect of traffic isolation at the low levels through multiple buses, traffic reduction at the higher levels through hierarchical caches, and limitation of coherency control to those sections where it is necessary results in a large multiplication of bandwidth with full shared memory and automatic coherency control.

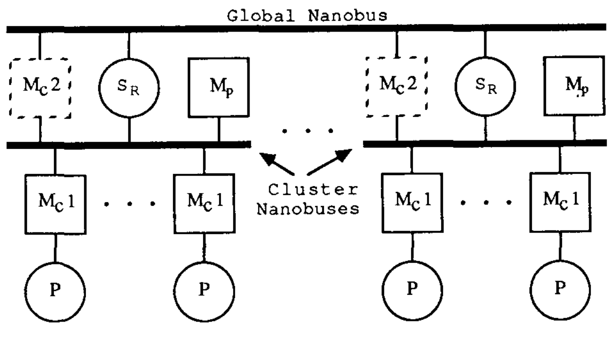
The Cluster Concept

Distributing memory amongst the groups of processors can significantly reduce global bus traffic and average latencies. Remote requests for local memory are routed through the local shared bus, using a special adapter board to provide coherency control. This later concept will be referred to as the cluster architecture, as each bottom level bus forms a complete multiprocessor cluster with direct access to a bank of cluster local memory.

There are several advantages to the cluster architecture. It allows code and stacks to be kept local to a given cluster, thus leaving the higher levels of the system for global traffic. Each process can still be run on any of the local cluster's processors with equal ease, thus gaining most of the automatic load balancing advantages of a tightly coupled multiprocessor, but can also be executed on a remote cluster when necessary. Because of the local cache memories, even a process running from a remote cluster will achieve close to maximum performance.

The cluster architecture can also help with the management of global, shared accesses as well. The iterative nature of many scientific algorithms causes the global accesses to exhibit poor cache behavior. But because the access patterns are highly predictable, it is often the case that globals can be partitioned to place them in the clusters where the greatest frequency of use will be. Thus the cluster approach can take advantage of large grain locality to overcome the poor cache behavior of the global data resulting in shorter access latencies and less global bus traffic than a straight hierarchical cache scheme.

As seen in Figure 7, accesses to data stored in remote clusters proceed in a fashion similar to that of a straight hierarchical cache system. The Cluster Caches form the second level of caches and provide the same filtering and coherency control for remote references as the second level caches of the hierarchical scheme. After reaching the top (Global) level of the hierarchy, the requests will be routed down to the cluster which contains the desired memory location, and will pass through that cluster's shared bus. Since the private caches on the cluster bus will also see this access, no special coherency actions are necessary. For those accesses which go directly to memory in the same cluster as the originating processor, additional coherency control is required. To perform this a special adapter card will be required to keep track of possible remote copies



Key:

P	Processor	M_P	Main Memory
M_C1	Processor Private Cache	S_R	Routing Switch
M_C2	Optional Cluster Cache		

Figure 7: Multiprocessor Structure Composed of Shared Bus Multiprocessor Clusters

and whether a remote copy has exclusive access or not. A diagram of the states required for each location in the adapter card is shown in Figure 8. If a local write is received, and the adapter card determines that a remote copy might exist, then an invalidate request is sent up the hierarchy to perform the function that the write request would perform in a pure hierarchy. If a local read is detected for a location for which the existence of a remote exclusive use copy is recorded, then a flush request must propagate up the hierarchy. With these extensions, the cache coherency schemes developed for hierarchical structures can be used to provide intercluster coherency as well.

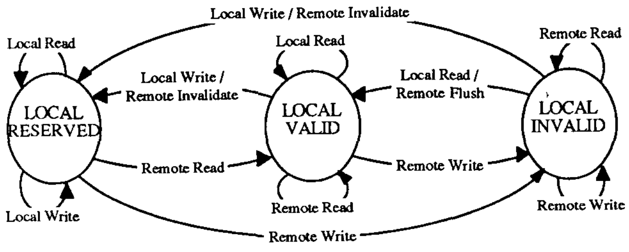


Figure 8: Cluster Adapter State Diagram for Extended Goodman Caching Algorithm

Simulation Experiments

Simulations were done to analyze the performance of medium and large scale, hierarchically clustered multiprocessors. The simulation techniques employed are described in detail in[15], and use statistics derived from address traces of actual benchmark programs. Three different size multiprocessors were examined: a single cluster system with 16 processors and an eight cluster system with 128 processors. The processors simulated are typical 1 MIPS, 32 Bit, microprocessors. These simulations indicate that the multiprocessor architectures developed in this paper can provide high performance parallel computation.

In performing this research, address traces of a number of different benchmark programs were first collected (from a VAX™). These traces were than statistically analyzed to develop approximate stochastic models of the processor's reference patterns. The stochastic models were then used to drive the simulator. Comparisons with the original traces indicate that the cache miss ratios tended to be overstated by up to a factor of three. In other words, the stochastic models did not capture all of the locality inherent in the original traces. On the other hand, trace driven cache simulations often understate miss ratios as compared to actual system measurements, so the results of this study err on the conservative side.

The results of measurements taken with three different benchmark programs are reported on here. One of the programs is a parallel, iterative, asynchronous partial differential equation (PDE) solver. The second is a parallel implementation of the Quick Sort algorithm, while the third benchmark program is the simulator itself. The goal of these experiments was to measure the reductions in performance due to hardware contention while ignoring algorithm inefficiencies. Thus, in the case of the Quick Sort algorithm, only that phase of the computation where all processors are engaged in sorting operations was modeled, eliminating the logarithmic start up phase.

Because the eight cluster multiprocessor architecture uses large (1 Megabyte) cluster caches, the benchmarks must be large. Hence the PDE traced was of the solution of a 30 by 480 element matrix, and the Simulator traced was of the simulation of a 16 processor multiprocessor. The PDE trace comprised 6.7 million memory references, while the Simulator trace comprised 4 million.

It is necessary to pick benchmarks which address a larger range of memory addresses than will fit in the cache. Otherwise an unrealistically high hit ratio will result. With 16 processors, the PDE global data matrix requires almost 2 Mbytes of memory, twice what the cluster level cache holds. Similarly, 16 processors executing the new Simulation benchmark require nearly 6.5 Mbytes of memory. Thus both of these benchmarks stress the entire multiprocessor architecture.

Single Cluster Results

A single cluster system was the first to be simulated, using three address traces derived from benchmarks. Two were true multiprocessor algorithms: a parallel Quick Sort and partial differential equation (PDE) solver. They were simulated both with and without the inclusion of coherency effects. The simulated system consisted of 16 processors and eight memory modules.

Benchmark	Simulated Time (sec)	Speedup	Bus Util.	Memory Util.
PDE	5.190	15.75	13%	6%
Quick Sort	1.285	15.05	16%	8%
Simulator	3.210	15.85	24%	12%

Figure 9: Results for 16 Processor Multiprocessor Simulations

Figure 9 summarizes the results for the single cluster systems. The percentage of bus and memory bandwidth utilized by each configuration is included to indicate where saturation is occurring. Note that with eight memory modules, the total memory bandwidth is twice that of the bus, so the bus utilization is twice that of the memory system utilization. In spite of the bus' reduced bandwidth the bus never becomes a bottle neck in these simulations, having at most a 24% utilization factor.

In general, it is desirable to maintain low required bus bandwidth. Contention for the shared bus causes dramatic increases in access delays when bus utilization exceeds 80%, due to queueing effects. Even at lower bus utilizations there are small but measurable increases in effective access times. Thus, the lower the bus utilization, the lower the additional delays due to bus contention. Since write-deferred caching results in significantly lower bus utilization, it is the caching protocol of choice for high performance systems.

Benchmark	Simulated Time (s)	Accesses per proc.	Speedup
PDE W-D			
1 proc. N/C	1.453	91942	1.00
16 proc. N/C	1.469	91942	15.83
16 proc. C	1.472	93769	15.79
QSort W-D			
1 proc. N/C	1.209	74857	1.00
16 proc. N/C	1.216	74857	15.91
16 proc. C	1.285	160108	15.05

Figure 10: Effects of Cache Coherency on Multiprocessor Performance

Two of the benchmarks are actual multiprocessor algorithms with shared global data. As such they actively share some regions of memory and require the services of the cache coherency algorithms to prevent stale data. The effects of the shared data on the performance of these two benchmarks can be seen from the data presented in Figure 10. Bus traffic increases almost 2%, but the amount of traffic increase is still so small that the execution time is essentially unchanged. There is a more pronounced effect with the Quick Sort benchmark, with simulated execution times of the benchmark increasing by five percent. Coherency caused invalidations contributed less than 10% to the total memory accesses per processor, thus producing a relatively small decrease in performance. This is a significant finding, since it indicates that the necessity of maintaining cache coherency will not prove to significantly penalize performance. This means that system designers need not worry about some inefficiency when actual data sharing occurs, provided that there is no penalty associated with exclusive access or read only sharing. Thus multi-cache coherency algorithms should concentrate on minimizing impact in the non-shared case, rather than the actively shared one.

Eight Cluster Results

Simulations of the large, eight cluster architecture were performed. Since the Quick Sort benchmark was too small to adequately test the large multiprocessor system, it was dropped from the experiments. The large architecture was simulated with both the PDE and Simulator benchmarks.

Benchmark	Cluster Hit Ratio
PDE	.5740
Simulator	.6009

Figure 11: Cluster Cache Hit Ratios

Of particular interest is the hit ratios achieved with the large cluster caches. Figure 11 shows the cluster cache hit ratios achieved with the two benchmarks. Due to Virtual memory limitations of the computer system on which these simulations were done, only a 1 Megabyte cache rather than the 2 Megabyte cache anticipated was simulated. Also, as mentioned earlier, the statistical methods used do not adequately capture address stream's locality. The methods are especially pessimistic where Global data is concerned, which is the primary type of data seen at the cluster cache level. Thus the hit ratios shown in Figure 11 are rather pessimistic.

As pointed out in the section describing the proposed architecture, the cluster level caches must insure that all remote memory locations for which one or more local caches have copies must also exist in the cluster cache. They do this by placing an invalidation request on the cluster bus for any remote memory location copy which they are invalidating in their own cache (for instance, because the copy is being replaced by another). These invalidations could seriously impair the performance of a hierarchical multiprocessor by lowering local cache hit ratios if they occurred too frequently. Figure 12 shows the number of cluster invalidates generated by the cluster caches and indicates how the number of such invalidations compares to the total amount of memory requests produced by the local caches.

Benchmark	Req	Invals	Ratio
PDE	8,278	4,291	52%
Simulator	9,569	2,549	27%

(All Amounts in Thousands)

Figure 12: Relative Frequency of Cluster Level Invalidation Requests as Compared to Total Cluster Traffic

Figure 12 indicates that the cluster invalidations are a relatively large percentage of the total cluster traffic. Though this is partly due to the overstated cluster cache miss ratio, it is also due to the large traffic reduction achieved by the local caches. At first glance the large number of invalidation requests would appear to have a significant impact on local hit ratios. Since degradations in local hit ratios would significantly reduce system performance, the large number of invalidations might be a problem.

Benchmark	Simulated Time (sec)	Speedup	Bus Utilization Local	Global	Serial
PDE	6.384	102.0	26%	61%	58%
Simulator	3.909	104.3	37%	66%	60%

Figure 13: Eight Cluster Multiprocessor Results (Single Global Bus)

On further analysis, the cluster invalidates are seen not to be a problem. In the experiments only the shared globals are subject to cluster level invalidations. Experiments have shown[15] that shared globals already exhibit poor hit ratios. The cluster invalidates will only produce poor local cache hit ratios if the locations they are invalidating still reside in a local cache and if the processors connected with the local caches request the locations before they would have been replaced in the normal course of operation. The low initial hit ratios make the above sequence unlikely, so that any additional reduction in hit ratios should be small. Since hit ratios are already poor for shared globals, a small reduction in hit ratio will not significantly reduce performance.

The 128 processor results are shown in Figure 13. Measurements were taken of total simulated run time, cluster bus utilization, global bus utilization, and serial link utilization. Speedup is calculated as the time taken to execute a single processor version of the benchmark divided by the time taken to execute the 128 processor version and multiplied by the number of processors (128).

Examining Figure 13 it is evident that good performance is achieved under most circumstances. The simulations indicate speedups of around 100 out of 128. The global bus is heavily used, but is not saturating. Larger cluster level caches (with better hit ratios) would improve the performance of both systems.

There are some other possible architectural changes that might improve performance. Somewhat better speedups would occur with dual global buses, but much of the reduction in speedup is due to the increase in local bus traffic and not the existence of the global bus. An architecture in which each memory was dual ported and requests from remote clusters could avoid the local cluster bus would noticeably improve performance for write-through systems. However, the additional complexity of two port memories, especially with regard to cache coherency, would not be worth the additional performance gain.

Analytic Modeling

The simulations reported on in the previous section demonstrated that the proposed hierarchical multiprocessor architecture can achieve good performance with over one hundred processors. While simulations can provide more detailed, and hence more accurate prediction of behavior than analytic modeling, they require substantially more computing resources. In this research, resource limitations restricted the size of second level caches, as well as range of system parameters that could be simulated. In order to rapidly explore a larger design space, analytic models of the hierarchical multiprocessor system were constructed.

Model Description

The analytical models start with assumptions about the memory request rate of processors, then calculate the amount of traffic on all links and busses. Based on the traffic calculations, values for link and bus utilization are derived. A simple queueing model calculates average queueing delay from the utilization figures, which is then added to bus and link transport delays to determine overall read request latency. Finally, an estimate of the performance reduction due to the Ultra Multi architecture is calculated.

Models were developed of private and shared cache behavior, of Global Bus Watcher (GBW) and Check out Tag Store (CTS) behavior, and of bus and link behavior. A model of the traffic patterns of the CPU's and their response to the delays encountered in memory accesses is included. Though not all details of the system are modeled, an attempt has been made to have the models produce pessimistic results where ever approximations are encountered. A brief outline of the equations used in the models will now be given.

The models begin with calculations of traffic flow at the different levels of the architecture, based on offered processor traffic and cache transfer ratios. The traffic flow calculations are then used to determine bus and link utilization, and hence bus and link access delays. Finally, total access delays for cache read misses are calculated, and used to determine effective processor speed.

Processor generated traffic is based on measured or estimated memory accessing frequency for given speed processors. The private cache hit and transfer ratios are extrapolated from actual system measurements and simulations. The rate of state change requests produced by GBWs and CTSs is also modeled. These state change requests are added to the total traffic on links and buses where they are placed.

Some of the requests generated by processors are assumed to be for the cluster in which the processor is located, and some for other clusters. The model takes a conservative approach and assumes that memory locations accessed by the processor are uniformly distributed throughout the system. Thus, in an eight cluster system, 7/8ths of the requests would be for other clusters. In reality, it may be possible to keep more memory locations in the processor's cluster, significantly reducing the fraction of non local requests.

Once the level of traffic at each link and bus has been calculated, access delays can be determined. Access delays due to links and buses are modeled as a transit delay plus a queueing delay. Since the links are assumed to be high speed serial connections, the transit time is mostly due to serialization time, which is proportional to message length. The queuing delay is calculated using the M/M/1 queuing delay formula and the calculated link utilization. For buses, a commercial bus was modeled, which has fixed transit delays and exhibits M/D/1 queuing properties.

Finally, the total effective access time as seen by processors is calculated. This time is used to determine the reduction in processor performance over an ideal, instant response memory system.

Performance Predictions

Processor performance parameters from several present and future microprocessors were used to analyze the Ultramax architecture. The processors range in performance from 2 MIPS to 13 MIPS. Private cache sizes commensurate with the technology level of the processor chips was assumed. Since RAM densities have been keeping pace with microprocessor improvements, it is expected larger caches sizes will be available at the same time that the faster processors are available. Figure 14 indicates some of the parameters used in the analytic modeling.

Processor Speed	Cache Type	Cache Size	Miss Ratio
2.0 MIPS	Wrt-Thru	64 K	.05
3.0 MIPS	Wrt-Thru	256 K	.025
5.0 MIPS	Wrt-Back	256 K	.025
7.5 MIPS	Wrt-Back	1024 K	.0125
13.0 MIPS	Wrt-Back	1024 K	.0125

Figure 14: Processor Parameters used in Modeling

The models have been used to analyze a number of system parameters, such as line size and link bandwidth. Of particular interest is the required cluster cache size. The cluster caches are used both to hide the latency of the interconnection network and to reduce global bus traffic to the point were a single bus of similar bandwidth to that of the local buses can be used. This implies that the cluster cache transfer ratio must be about 1/N, in a system with N clusters. The actual ratio required is 1/(N-1), since any global bus traffic shows up twice on the local buses. For a write-deferred cache with one of the multicache coherency protocols, simulations have shown that the miss ratio must be about 20% lower than the desired transfer ratio.

For an eight cluster system, the required transfer ratio is 1/7th, which can easily be obtained with a miss ratio of 1/9. At first glance, a miss ratio of 1/9 seems quite easy, since even 16 Kbyte caches can exceed that. However, the request stream that misses the private caches contains considerably less locality than a typical processor request stream. In fact, the intrinsic performance of the cluster cache has to be about 9 times better than that of the private cache to achieve a miss ratio of 1/9th. This effect is illustrated in Figure 15, where relative system performance (as a fraction of ideal performance) is plotted against cluster cache intrinsic hit ratios for the five candidate processors. The data of Figure 15 is for a ten cluster system. The faster processors, which have larger private caches, require substantially better intrinsic cluster cache performance to achieve acceptable results.

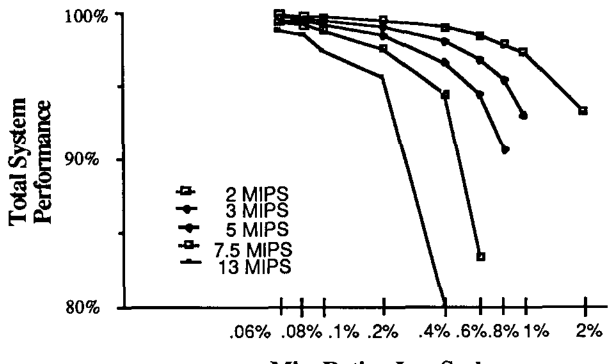


Figure 15: System Performance vs. Cluster Cache Miss Ratio

The data from Figure 15 indicates that the cluster cache should have an intrinsic miss ratio of less than .02 for the slowest processor to .003 for the fastest. Extrapolations from measured data for smaller caches indicates that the required intrinsic cluster cache hit ratios can be achieved with a caches ranging from around 8 Megabytes to 32 Megabytes in size. Since the cluster caches can be made with dynamic RAMs, such large cache sizes are technically feasible.

In Figure 16, the total performance for systems utilizing the 13 MIPS processors is shown. Each curve represents a different intrinsic cluster cache miss ratio. Even if the intrinsic cluster cache miss ratio is only .004, a system of 1000 real MIPS can be achieved.

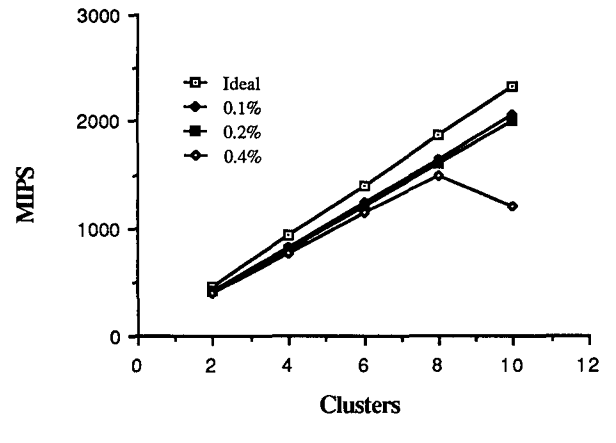


Figure 16: Total System MIPS vs. Number of Clusters

Conclusions

This paper has demonstrated the effectiveness of a hierarchical multiprocessor architecture. The simulation results obtained showed that the hierarchical multiprocessor computer structure achieves good speedup with suitable parallel algorithms. A simulated 128 processor system achieved a speedup of 104 executing the Simulator benchmark, a processor efficiency of 80%. Analytic models have predicted processor efficiencies as high as 98% compared to single

From the results, the following conclusions can be drawn about the multiprocessor architecture described in this paper:

- The use of hierarchical caches allows the expansion of the cache coherency techniques beyond a single cluster.
- The degradation in performance due to multicache coherency maintenance is small, even for very large systems.
- Hierarchical caching further reduces the global bus bandwidth requirements and effective access time, in most situations.
- Because of the poorer hit ratios and cache transfer ratios, the impact of shared global data on the global shared bus may still be a problem with very large systems.

The processors assumed in the simulation experiments are rated at approximately one million instructions per second. Extrapolating from the data for the PDE simulation shown in Figure 13, it appears that the hierarchical cache and bus structure simulated could support processors about three times faster (3 MIPS), provided a second global bus was used. This would result in a local bus utilization of 78%, a global bus utilization of 90% for each bus, and a serial link utilization of 87%. These utilization numbers are significantly overstated, since the simulated cluster cache was one eighth the size feasible with current technology.

The analytical modeling indicates that very high performance multiprocessors are possible. Using the 4 Mbit dynamic RAMS (allowing construction of 64 Mbyte cluster caches) and 13 MIPS processors that will soon be available, a shared memory multiprocessor of well over 1000 MIPS will be possible. Even including the effects of private and cluster cache misses, memory access times, and intercluster network delays, processor performance of 85% - 90% of optimum is obtainable. Figure 17 shows one possible configuration of a 128 processor system using the cluster architecture.

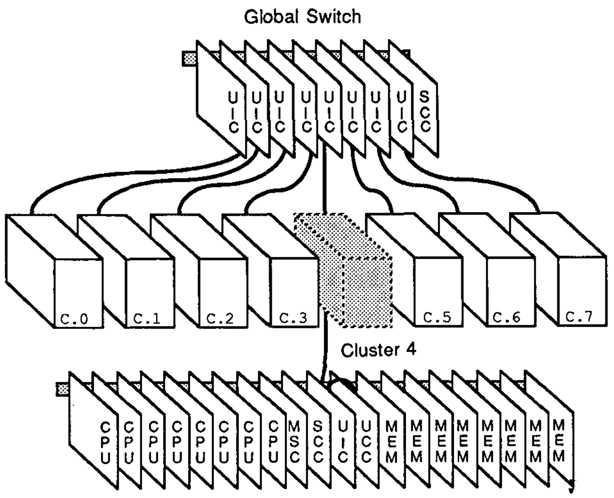


Figure 17: Cluster Connected, 128 Processor, Shared Memory Multiprocessor

References

[1] Anderson, G.A., and Jensen, E.D., "Computer Interconnection Structures: Taxonomy, Characteristics, and Examples," *Computing Surveys* 7(4):197-213, December, 1975.

[2] Baer, J.L., "A Survey of Some Theoretical Aspects of Multiprocessing," *Computing Surveys* 5:31-80, March, 1973.

[3] Censier, L.M. and Feautrier, P., "A New Solution to Coherence Problems in Muticache Systems," *IEEE Transactions on Computers* (C-27), December, 1978.

[4] Dubois, Michael, and Briggs, F.A., "Effects of Cache Coherency in Multiprocessors," *IEEE Transactions on Computers* C-31(11), November, 1982.

[5] Feng, T., "A Survey of Interconnection Networks," *Computer* 14(12):12-27, December, 1981.

[6] Frank, S.J., "A Tightly Coupled Multiprocessor System Speeds Memory-Access Times," *Electronics* 164-169, January, 1984.

[7] Goodman, J.R., "Using Cache Memory to Reduce Processor-Memory Traffic," in *10th Annual Symposium on Computer Architecture*. 1983.

[8] Haynes, L.S., Lau, R.L., Siewiorek, D.P., and Mitzell, D.W., "A Survey of Highly Parallel Computing," *Computer* 9-24, January, 1982.

[9] Papamarcos, M.S., and Patel, J.M., "A Low Overhead Coherence Solution for Multiprocessors with Private Cache memories," In *The 11th Annual Symposium on Computer Architecture*. 1984.

[10] Pohm, A.V., and Agrawal, O.P., *High-Speed Memory Systems*, Reston Publishing Company, Inc. 1983.

[11] Rudolph, L., and Segall, Z., "Dynamic Decentralized Cache Schemes for MIMD Parallel Processors," In *The 11th Annual Symposium on Computer Architecture*. 1984.

[12] Sweazey, P. and Smith, A.J., "A Class of Compatible Cache consistency Protocols and their Support by the IEEE Futurebus," In *The 13th Annual International Symposium on Computer Architecture*, pp. 414-423, June 1986

[13] Tang, C.K., "Cache System Design in the Tightly Coupled Multiprocessor System," In *National Computer Conference, Proceedings, AFIPS*, 1976.

[14] Thurber, K.J., "Interconnection Networks-A Survey and Assessment," In *National Computer Conference, Proceedings, AFIPS*, 1974.

[15] Wilson, A.W., *Organization and Statistical Simulation of Hierarchical Multiprocessors*, PhD. Thesis, ECE Department, Carnegie-Mellon University, Pittsburgh, PA, August, 1985.