A Cache Coherency Protocol for Optically
Connected Parallel Computer Systems

John A. Reisner
Headquarters, Standard Systems Group
Automated Communications Systems Directorate
Tinker AFB, OK 73145-9155

ABSTRACT
A cache coherency protocol was developed for the
Multiple Channel Architecture (MCA), a proposed
computer architecture that uses an optical inter-
connection network to overcome many of the problems
associated with internode communication in massively
parallel systems. A directory-based protocol was
attempted, but testing revealed a serious scalability
problem associated with the collection of
acknowledgement packets. An alternative scheme, which
employs snooping on a special-purpose, time-division
multiplexed optical channel, performed well.
Additionally, the benefits of a separate cache for shared
data were examined. Simulations demonstrate that the
coherency protocol improves system performance and
scalability, and that additional performance gains may be
attained by customizing shared cache parameters.

1: Introduction to the MCA

The Multiple Channel Architecture (MCA) is a
proposed computer architecture that uses fiber optic
technology and tunable laser receiver/transmitters to
overcome many of the problems associated with
interconnection networks. All nodes in the system are
interconnected by a passive star coupler, which evenly
divides laser transmitter power to all receivers. Each node
has at least one tunable laser receiver and one tunable laser
transmitter. The receiver(s) remain tuned to a fixed channel
for the duration of a program epoch. For example, when a
processor needs to communicate with a memory node, the
processor tunes its transmitter to the frequency (channel) on
which the memory node is listening, and then transmits its
message. Figure 1 shows a sample logical configuration
with four processors and four memories, each tuned to
receive on a different channel.

The MCA can emulate many different inter-
connection topologies by varying which receive channels
are assigned to each node. Extensive simulations have
shown that allocating one processor or memory node per
channel greatly underutilizes the frequency. Therefore,
multiple nodes are typically tuned to a single channel. An
MCA with 64 processors and 64 memory nodes may be
configured such that each channel is used for either eight
processors or eight memory nodes. This configuration
would require 16 distinct channels. However, if this
configuration resulted in too many collisions on the network
for a particular application, it could easily be modified such
that a single channel serves only four nodes. Because the
receive channels can be tuned dynamically, this flexibility is
always available. In a multitasking environment, nothing
prevents Task A from assigning four nodes per channel
while Task B uses 12 nodes per channel, assuming the tasks
are assigned different nodes on the network. Lasers capable
of tuning to as many as 50 distinct high-bandwidth channels
have already been demonstrated, and future devices are
expected to be capable of tuning to hundreds of channels
[9]. Because of this, the MCA is an attractive design for a
flexible, high-performance, massively parallel computer
architecture. More information on the goals and potential
benefits of the MCA are detailed in [18].

2: Background on cache coherency

A cache coherency protocol is a set of rules that ensure
that cached data distributed among individual processors
remains coherent. Data is said to be coherent when a read
to any item returns the last value written to that item [11].
An effective cache coherency protocol will enforce
coherency while minimizing any burden incurred on the
interconnection network. Because overall system
performance in a multiprocessor environment is highly
dependent upon the effectiveness of its data caching
scheme, much research has been devoted to this problem
over the past decade [see 1, 3, 4, 8, 11, 12].

U.S. Government Work Not Protected by U.S. Copyright
The goal of this research is to design an effective cache coherency protocol for the MCA, and, if possible, exploit the unique aspects of the architecture in the protocol design.

There are many considerations to evaluate when designing a solution to the cache coherency problem. One such issue is selecting an overall technique for enforcing coherency. Cache coherency protocols have generally fallen into three categories: snooping, directory, and software enforcement. The snooping technique uses a common bus that is constantly monitored. When a write occurs, a packet containing the address of the modified block is broadcast over the common bus. Each processing node examines this address and determines if the modified block is present in its cache; if so, some action is taken to ensure coherency. Archibald and Baer described and evaluated several snooping protocols in [1], including the Synapse protocol, the Berkeley protocol, the Firefly protocol, and the Xerox PARC Dragon protocol.

Directory protocols use a centralized directory to track which processors have cached each block. When a block is modified, the directory is used to determine which processors need to be notified of the change. Three types of directories—limited, fully-mapped, and chained—are discussed in [2]. A fourth alternative, overflow directories, is described in [16]. A chief obstacle in implementing a directory-based protocol in a large-scale system is the size of the directory [12]. In particular, as a system becomes large, the overhead required for a fully mapped directory becomes prohibitive. The Stanford DASH addresses the size problem with a hierarchical approach [10].

Software enforcement protocols use compile-time analysis to determine when a shared item may be safely cached. When a parallel program enters a program epoch in which a shared variable is modified, any cached copies of the variable are invalidated. Shared cached data only remains valid when compile-time analysis determines that no write to the block occurs anywhere in the program epoch. Cheong and Veidenbaum have proposed a software scheme in [3]; Min and Baer developed a more sophisticated scheme in [11] which uses clocks and timestamps.

Another issue to be considered is how notifications are implemented. After receiving notification that a cached block has been modified, a protocol may perform either an invalidation or an update of the block. An invalidation changes the state of a cached block to an INVALID state. If the processor performs a subsequent read to that block, a cache miss will occur and the block must again be fetched from memory. An alternative is to transmit the new value of the data inside the notification packet, and allow the cache to update the stale data. The Berkeley protocol uses invalidations [8], while the Firefly protocol uses updates [17].

Scalability is an extremely important concern to the protocol designer. According to [10], a system or subsystem is considered scalable if, as the number of processors increases, the performance improvement of the system is linear or near-linear. A cache coherency protocol designed for a high-performance architecture should be at least as scalable as the architecture itself, otherwise, the protocol will be a limiting factor for the entire system.

3: Designing a protocol for the MCA

In the early stages of designing the MCA coherency protocol, one study of extraordinary interest was [6]. In this research, Gupta and Weber analyzed cache
invalidation patterns. Specifically, they wanted to answer the question: When a write to a shared variable generates an invalidation, how many processors have previously cached the data? Because message traffic generated by cache invalidations can tax the interconnection network, it is useful to know if any patterns exist. Their research included numerous simulations using a test suite of five applications and a varied number of processors. Among other findings, their research included the following results, based on a 32-processor configuration:

- a. For all five applications, less than 30% of all the invalidations affected more than one processor.
- b. For two of the applications, less than 5% of the invalidations affected more than one processor.
- c. For four of the five applications, the percentage affecting more than two processors never exceeded 8%.
- d. Of the five applications, the largest average number of processors affected by an invalidation was 1.6.

In short, they found that relatively few processors are affected by a majority of the invalidations. Typically, only a few of the processors in the system have previously cached the data when an invalidation occurs, largely because shared data tends to be migratory. This is a significant discovery when considering a directory-based protocol. Fully mapped directories are not practical for large scale systems, but an alternative, such as an overflow directory, might be more viable. In an overflow directory, there are enough bits in the directory to track each block of data in the caches of N processors, where N is less than the total number of processors in the system. As long as no more than N processors have cached a given block, it is possible to know exactly where the block has been cached, and thus notifications can be sent only to those processors. When processor (N + 1) caches the data, an overflow bit is set, meaning that some sort of secondary notification scheme must be used. However, according to Gupta and Weber's research, this secondary notification scheme will not be needed very often even if N is relatively small.

With this in mind, we designed an overflow directory protocol for the MCA. Each block of data has an associated directory capable of tracking N channels, plus an overflow bit. When a processor reads from memory and caches a shared block, its receive channel is added to the directory. If a different CPU tuned to the same channel subsequently caches this same block, no action is necessary, because this processor node is already snooping a channel that is in the directory. Channels are added until there is no more room in the directory, and then the overflow bit is set.

Whenever a shared write is performed, the writing processor sends the updated copy of the block to the memory node. If the overflow bit is not set, a copy of the updated block is transmitted over each channel contained in the directory. All processor nodes tuned to these channels receive the message and proceed to update the block if it has been cached. Because this directory tracks channels rather than processors, and multiple processors typically tune to a single channel, the directory is even less prone to overflow than a directory that tracks each processor node individually.

The next task was to develop a secondary notification scheme to use when the directory overflows. When the overflow bit is set and a write to the block is performed, the secondary notification scheme ensures all processors are notified that the shared block has been changed. The goal was to design a secondary notification scheme that would incur a minimal performance penalty. One very attractive option was to use a barrier channel that had already been incorporated into the overall MCA design.

The barrier channel uses an extra receiver/transmitter at each node, configured as a time-division multiplexed (TDM) optical channel. (There is more than one way to construct an optical TDM channel, one possibility is described in [5]). This channel was originally designed as a way to efficiently meet and clear the barriers used in parallel programming.

The TDM channel works as follows: all processor nodes assigned to a particular task remained tuned to the same barrier channel. This channel has three states: IDLE, ACTIVE, and CLEAR. The time slices for the channel are allotted as illustrated in Figure 2. During the initiator block of the TDM cycle, the initiator sends a packet over the channel that indicates why the barrier

![Figure 2. Time Slices on the Time-Division Multiplexed Barrier Channel.](image-url)

*For an Extended Invalidation Broadcast, the initiator sends a packet that includes the address of the modified block. P processors are assigned to the task, so P time slices are allocated for the acknowledgment portion of the cycle.*
channel is being used. For cache coherency notifications, a memory node transmits the address of a block, which lets each processor know that the block must be invalidated if it has been cached. This transaction is called an Extended Invalidation Broadcast, or EIB, and the commencing of an EIB changes the state of the channel from IDLE to ACTIVE. The remaining time slices in the TDM cycle allow each processor to acknowledge that the message was received. If one or more processors fail to acknowledge the event, the initiator retransmits the block and again waits for acknowledgments. This cycle continues until all processors successfully acknowledge the EIB, at which point the state of the channel transitions to CLEAR and the initiator transmits the “clear” signal indicating the process has succeeded. Once all processors acknowledge this clear signal, the channel transitions back to the IDLE state and it becomes available once again.

4: The MCA simulator

In order to evaluate the performance of the proposed architecture, a simulator designed to faithfully emulate an MCA was built. This simulator is event-driven, not trace-driven. The memory controllers, buses (channels), I/O ports, and processors all have associated finite state machines aimed at ensuring the simulation remains true. At run-time, an MCA configuration is passed to the simulator. This configuration includes:

- The number of processor nodes (CPUs).
- The number of shared memory nodes.
- The number of private memory nodes.
- The number of channels allocated to processor nodes.
- The number of channels allocated to shared memory nodes.
- The number of channels allocated to private memory nodes.
- The overlapping of channels (a single channel can be used by multiple processor nodes, multiple shared memory nodes, and/or multiple private memory nodes).
- The number of tasks to be run during the simulation (each task has its own configuration of processor nodes, memory nodes, and channels, although channels may be shared between tasks).
- Cache parameters, including overall size, block size, and associativity.

A brief example will illustrate the inner workings of the simulator and the attempt to accurately model an MCA system. Suppose CPU 0 needs to fetch an instruction. First, the cache controller for processor 0 must be free. Once it becomes free, it checks to see if the address of the instruction is contained in a valid block in the cache. If the block is not cached, then the CPU must form a request packet. This packet contains the information required by the IEEE 802.3 (Ethernet) Standard, including a preamble, start, destination, source, length, type, data field, and cyclic redundancy check (CRC). A built-in delay is associated with packet formation; this delay varies according to the length of the packet. After the delay, the status of the transmit channel is checked. If it is busy, the packet must wait until the channel is free. Once the channel is free, an attempt is made to transmit. If there is a collision with another packet, the collision is resolved using the CSMA/CD protocol with a binary exponential truncation limit of 10. Once the packet is in transmission, the channel remains busy for a set time, based on the propagation delay and modulation rate for a five-meter cable length. When the memory packet arrives, it is put into the memory controller’s receive buffer. It will be serviced from there once the memory controller is free. After an appropriate service time and after the reply packet has been formed, the memory reply packet is transmitted over the proper transmit channel using the same rules and arbitration scheme as was used at the processor node. Again, the transmit time is dependent upon the length of the packet. After the processor has received the packet, the line can be put into the cache only after the cache controller is in the free state. The CPU sends an acknowledgment packet back to memory each time it receives a packet. These acknowledgments also have delay times and must undergo channel arbitration in the event of a collision. After the block has been loaded into the cache, the CPU may fetch the instruction. If the instruction uses data that is not already cached, a data block will be retrieved from memory (via the same process used to retrieve the instruction), and subsequently cached. An LRU replacement scheme is used when a private data block is ejected from a full cache, and a write-back packet is formed and transmitted if the data was dirty. The write-back packet will invoke an acknowledgment packet from main memory after it is received. If an EIB is performed, the amount of time required to complete the acknowledgment portion of the TDM cycle is dependent upon the number of processors that are assigned to the task. CPU stalls occur whenever (a) an instruction miss occurs, and (b) whenever a second miss to the same data address occurs. Condition (b) prevents out-of-order execution of the same variables. The stall conditions are intentionally conservative.

The simulator tracks many performance statistics. Tallies are made of the number of packets requested and sent by each node, the number of collisions on each channel, channel utilization percentage, CPU utilization, cache hit rates, cache ejections, and literally dozens of
other items. Additionally, periodic histograms for most of these statistics are available so that performance trends may be analyzed. For example, a lower cache hit rate during the first 50,000 clocks would be expected, since the caches are “cold” at program initialization. Similarly, more bus collisions might also be expected, because the CPUs are sending more requests to the memory nodes in order to cache needed data.

5: Separate cache for shared data

Private data may be accessed by one and only one processor in the system. Shared data may be accessed by multiple processors. Thus, the cache coherency problem only pertains to shared data. One significant design decision for the MCA protocol was to cache shared data in a separate cache. Using two caches per processor has some potential advantages over a “unicache” system that caches private and shared data together. Some potential benefits include:

1) Copy-back vs. Write-through. When cached data is private, a copy-back scheme is more efficient. Updates are not sent back to main memory except when a cache block is ejected, because there is no need to forward multiple updates to memory if only the last modification is significant. However, a copy-back scheme may complicate the coherency problem when caching shared data. Depending on the protocol, it may be convenient for main memory to always contain an up-to-date value of the block. Separate caches allow a copy-back scheme to be used for private data and a write-through scheme for shared data. The MCA protocol indeed uses copy-back for its private cache and write-through for the shared cache.

2) Cache Parameters can be customized. How does varying cache parameters affect overall system performance? This issue has been studied in the past [7, 13, 14, 15]. Parameters of interest include block size, overall cache size, set associativity, and replacement policies. Block size is an especially noteworthy parameter when comparing shared and private data. Although a larger block size might improve performance when caching private data, a smaller block size might be more suitable for shared data. As stated in [12], “Finer granularity is needed to reduce the number of cache misses caused by the invalidation of a large block because of a write to a small portion of it.” By using separate shared and private caches, parameters for each cache can be optimized according to the behavior of shared and private data, potentially yielding increased performance.

The MCA simulator uses separate cache controllers for the shared and private cache at each processing node. The private caches use copy-back and the shared caches use write-through. Additionally, separate parameters may be specified at run-time for both shared and private caches.

6: Additional coherency considerations

As discussed thus far, the MCA cache coherency protocol requires a separate shared cache, memory controllers that maintain overflow directories, and a Time Division Multiplexed optical channel for secondary notifications. In order to maintain coherency, two problems need to be addressed. Both of these problems involve trying to comply with the definition of coherency (that is, a read to any address returns the last value written to that address) in a parallel environment.

Because an EIB transpires on a special-purpose TDM channel, it requires less overhead than other communication traffic on the network. Because of this, an invalidation might “beat” a packet back to a particular processor. Consider the following scenario:

1) Processor A requests shared block Z from memory node M.

2) Memory node M receives the request, assembles a packet with block Z, and puts the packet in its transmit buffer.

3) While attempting to transmit to processor A, a collision occurs. Through the CSMA/CD arbitration scheme, memory node M’s packet waits.

4) Processor B writes to block Z, and sends the new line to memory node M.

5) Memory node M receives the new line from processor B. Assume that the directory has overflowed. An EIB occurs.

6) Processor A receives the EIB. Since the line is not in its cache, it sets its acknowledgment bit but ignores the invalidation.

7) Processor A finally receives the requested packet from memory node M. This packet contains stale information. The data is no longer coherent.

To prevent this scenario from occurring, the MCA uses a “reservation required” scheme. Whenever a shared block is requested, the block’s address is put into the cache before the request is sent to main memory. This is called the reservation.

As shown in Figure 3, the initial state of a reserved block is VALID but NOT READABLE due to LOAD-WAIT. A valid reservation is required in order to load a
block of memory into the shared cache. That is, when a requested block is received from memory, it can only be loaded into a VALID location within the cache that has the correct block address. If such a reservation is found, the block is loaded into the cache, and the state is changed to VALID and READABLE. If this VALID spot is not found, it is assumed the reservation was “canceled" by an EIB.

When an EIB is received for any VALID block, the state is changed to INVALID, regardless of the status of the READABLE bit, thereby canceling the reservation. This prevents the aforementioned scenario from occurring. When the reservation is not found (i.e., has been canceled), a new reservation is made, and the block is requested again.

Another problem occurs because of the elapsed time that exists between the time a processor writes to a shared item and the instant the coherency actions are complete. Because any number of reads could occur within this time lapse, it is difficult to ensure that every read returns the last value written to the variable. The MCA addresses this problem with a policy whereby the writing cache is the last processor that is able to use the new data in a modified block, and that it may do so only after all other processors have received and acknowledged the update or invalidation. Thus, the write is not considered complete until the writing processor has received a “write-grant” from memory. Before the modified block is sent to memory, the writing cache sets its NOT READABLE bit, and is not allowed to use the modified block until the write-grant is received from memory and the state changes back to READABLE. This procedure defines an exact time when a write has taken place, when the write-grant is received from memory. This conservative definition ensures that no processor will use a stale value, even though it may allow a processor to access an updated value “early.” Furthermore, the memory node uses an EIB-In-Progress bit for each block, which prevents an EIB from commencing before the write-grant from a previous EIB is received.

7: Experiment

An experiment was set up on the simulator to answer questions about the protocol. Among these questions were the following:

(1) Do our simulations affirm the Gupta and Weber study [6]? That is, will a small directory suffice so that the secondary notification scheme does not have to be frequently used?

(2) What size directory yields optimum performance?

(3) What performance gains are realized by the shared coherent cache?

(4) Can a performance gain be attained by varying parameters between the shared and private caches?

In order to answer these questions, six configurations were made. These configurations used 4, 8, 16, 32, 64, and 128 processors. Channel sharing gradually increased as the number of processors grew. The specific configurations are shown in Table 1.

Two parallel applications were used in testing. The first was a matrix multiply program that multiplied two 128x128 matrices. The second application used a filtering algorithm to perform a smoothing operation on a 64x64 pixel image. A naming scheme was developed: FILTER 16 referred to the median filter problem solved on the 16-processor configuration; MATRIXMULT 128 referred to the matrix multiply performed on the 128-processors configuration.

8: Results

The first task was to run both applications on the previous version of the simulator. This version employed the simplest coherency scheme in existence: disallow the caching of shared data. These runs provided baseline figures which could be used to measure performance changes. The results of these runs are shown in Table 2.

The next task was to verify that the results presented in [6] indeed occurred in our simulations; namely, because most invalidations affect only a few processors, a small directory will usually suffice. To do this, two of the larger configurations were used, MATRIXMULT 128 and FILTER 64. These two applications were each run three times—with directory sizes of 0, 2, and 4. With a directory size of 0 (no directory available), every shared write will generate an EIB. With a directory size greater than zero, some of these will be eliminated, since an EIB is not generated unless the directory is overflowing. With a directory size of 2, 62% of the EIBs were eliminated for MATRIXMULT 128, and 77% of the EIBs were eliminated for FILTER 64. With a directory size of 4, 80% of the EIBs were eliminated for MATRIXMULT 128, and 90% of the EIBs were eliminated for FILTER 64. Indeed a small directory eliminates the need for the secondary notification scheme a majority of the time.

The simulator was also modified to track how many processors had actually cached the data each time
an EIB is performed. This was basically an extension of the research Gupta and Weber performed in [6], but we wanted to see if the trend they identified would in fact continue in configurations with 64 and 128 processors (the biggest configuration in their study was 32 processors). Again, our simulations agreed with their results; most invalidations affected less than five CPUs.

The next task was to determine an optimum directory size. Six configurations were used, and each configuration was run with directory sizes of 4, 3, 2, 1, and 0. The results of these 30 runs are shown in Table 3. Table 3 shows that the optimum directory size is 0. This means that the existence of any directory hinders performance. This was very unexpected. When the decision to design an overflow directory-based protocol was made, it was assumed that the secondary notification scheme would be more likely to cause a performance degradation, not a performance improvement. Furthermore, for several of the runs, the directory scheme performed worse than the initial baseline runs, in

Table 1. Six Configurations Used in the Experiment.

<table>
<thead>
<tr>
<th>Processors</th>
<th>Shared Memory Nodes</th>
<th>Private Memory Nodes</th>
<th>Processors per Channel</th>
<th>Shared Memory Nodes per Channel</th>
<th>Private Memory Nodes per Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>16</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>32</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>64</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>128</td>
<td>64</td>
<td>128</td>
<td>8</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
which no shared caching was performed. This is most evident in the FILTER 64 run, where all directory sizes greater than 0 performed far worse than the baseline run, yet a significant performance gain occurred when the directory did not exist.

Analysis was performed to find why the directory scheme fared so poorly. The fault was found to lie in the receipt of acknowledgment packets. Consider MATRIXMULT 8, which has two CPUs tuned to each processor channel. Assume the directory size is four, and there are three entries in the directory when the shared write occurs. Memory sends out three packets, one on each valid channel in the directory. This process is done rather easily. However, in order to assure the memory node that all six CPUs have successfully received their packets, the processor nodes send acknowledgments back to memory. Since these six packets must all be sent on the same channel, and all six packets were formed close to the same time, collisions occur. Until all the arbitration is performed and all six packets have been sent, the bus used by that memory node is kept busy. So even though the CPUs receive their updates quickly, the acknowledgment process clogs the interconnection network. Close analysis of simulation data found that sometimes over 300 CPU clocks transpired during the arbitration process for acknowledgment packets.

In contrast, the EIB occurs on a separate channel, so normal transactions are not hindered. Additionally, the special-purpose nature of the EIB channel allows the entire acknowledgment process to transpire rapidly via time-division multiplexing. Therefore, what was originally designed as a secondary scheme turned out to be an effective primary coherency scheme. In spite of the accurate assumptions derived from Gupta and Weber [6], the acknowledgment process for a directory scheme poses a severe scalability problem. Some efficient way of collecting acknowledgments must be developed in order for a directory scheme to be scaleable. Simply eliminating acknowledgments altogether is a very questionable alternative, since one lost or damaged packet could render cached data incoherent.

The next task was to measure performance gains realized by the MCA’s optical TDM coherency protocol. The MCA performed better using the protocol for all six configurations of both applications. The results of this part of the experiment are depicted in Figures 4 and 5. The shaded region in these two graphs represent the extra clock cycles required to complete the program when shared caching is not used.

Perhaps the most significant aspect of the performance improvement is found in Figure 5. Without shared caching, the performance for MATRIXMULT 128 was worse than MATRIXMULT 64. This meant the old system was not scaleable for the matrix multiply program past 64 processors. However, with shared caching, performance continued to improve in the 128-processor configuration, thereby improving the scalability of the system.

The last task was to determine whether or not a performance gain could be realized by varying parameters between the shared and private caches. In particular, we were most interested in varying line size. FILTER 8 was selected as a proof-of-concept testbed. The first step was to determine what size cache was sufficiently large for this application. When a cache is sufficiently large for an application, the cache hit rate does not improve when the overall size of the cache is

<table>
<thead>
<tr>
<th>Program Name Configuration</th>
<th>Number of Instructions (millions)</th>
<th>Number of CPU Clocks (millions)</th>
<th>CPU Utilization Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATRIXMULT 4</td>
<td>18.4</td>
<td>3.9</td>
<td>87.2</td>
</tr>
<tr>
<td>MATRIXMULT 8</td>
<td>9.4</td>
<td>2.0</td>
<td>85.6</td>
</tr>
<tr>
<td>MATRIXMULT 16</td>
<td>4.9</td>
<td>1.0</td>
<td>82.2</td>
</tr>
<tr>
<td>MATRIXMULT 32</td>
<td>2.7</td>
<td>0.6</td>
<td>74.5</td>
</tr>
<tr>
<td>MATRIXMULT 64</td>
<td>1.5</td>
<td>0.4</td>
<td>56.2</td>
</tr>
<tr>
<td>MATRIXMULT 128</td>
<td>1.0</td>
<td>0.3</td>
<td>30.1</td>
</tr>
<tr>
<td>FILTER 4</td>
<td>2.2</td>
<td>0.8</td>
<td>81.4</td>
</tr>
<tr>
<td>FILTER 8</td>
<td>1.2</td>
<td>0.3</td>
<td>70.6</td>
</tr>
<tr>
<td>FILTER 16</td>
<td>0.7</td>
<td>0.2</td>
<td>58.9</td>
</tr>
<tr>
<td>FILTER 32</td>
<td>0.4</td>
<td>0.1</td>
<td>49.7</td>
</tr>
<tr>
<td>FILTER 64</td>
<td>0.3</td>
<td>0.1</td>
<td>41.6</td>
</tr>
<tr>
<td>FILTER 128</td>
<td>0.2</td>
<td>0.1</td>
<td>39.3</td>
</tr>
</tbody>
</table>

Table 2. Baseline Performance Results. Data shown is for the master processor, which executes the most instructions during the run. Percent utilization and instruction count for other processors are lower, since these processors spend more time waiting for the master processor. The master processor is the critical one in terms of performance.

<table>
<thead>
<tr>
<th>Application and Configuration</th>
<th>Directory Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>MATRIXMULT 8</td>
<td>21.2</td>
</tr>
<tr>
<td>MATRIXMULT 32</td>
<td>7.8</td>
</tr>
<tr>
<td>MATRIXMULT 128</td>
<td>6.2</td>
</tr>
<tr>
<td>FILTER 4</td>
<td>4.26</td>
</tr>
<tr>
<td>FILTER 16</td>
<td>4.87</td>
</tr>
<tr>
<td>FILTER 64</td>
<td>7.22</td>
</tr>
</tbody>
</table>

Table 3. Number of Clock Cycles (millions) Needed to Complete the Program, as Directory Size Varies.
increased. For FILTER 8, the cache hit ratios and number of clocks required for completion show no improvement after the cache reaches 32 K bytes (see Table 4).

The next step was to vary the line size using a sufficiently large cache and observe how the hit rate was affected (see Table 5).

Finally, a run was performed using a line size of 128 bytes for the private cache and a line size of 64 bytes for the shared cache, since the highest hit rates were attained using these sizes. The results are shown in Table 6.

As Table 6 indicates, a performance gain was indeed realized when the line sizes were unequal. The FILTER 8 completion times were nominally different when using identical line sizes; a more significant speedup was attained only after the line sizes were customized. This would imply that varying line sizes is a potential benefit of separating shared data into its own cache.

9: Summary, and future research

Our experiment affirmed the study conducted by Gupta and Weber [6]. This result was used to design a cache coherency protocol with the knowledge that a relatively small directory would suffice for most coherency notifications. However, our research also discovered that acknowledgment packets can be a bane to a directory-based protocol.

A novel protocol that uses a time-division multiplexed optical channel to enforce coherency was
explained. This protocol was shown to consistently improve the performance of the proposed Multiple Channel Architecture.

Potential benefits of a separate cache for shared data were discussed. One potential benefit is a performance improvement attained by customizing and optimizing cache parameters based on the different behavior of shared and private data.

Experiment results were based on simulations run on an event-driven simulator. However, only two applications were available for simulating the MCA. A more comprehensive test suite needs to be developed, and more data needs to be collected. The protocol has shown some promise, but more simulations with a broader range of applications would increase the degree of confidence in the protocol’s performance. Also, a previously non-existent improvement was shown as the number of processors increased from 64 to 128, demonstrating the improved scalability that occurred with the coherency protocol. It would be interesting to see if performance continues to improve with more processors, but using a 256-processor system or a 512-processor system did not seem logical with a 128x128 matrix.

Finally, many additional simulations using a broader test suit should be performed before unequivocally asserting that differing line sizes may offer an improvement over line sizes that are equal for shared and private data.

Acknowledgments
The authors would like to thank John Armitstead, who greatly assisted us during simulator debugging and protocol design. We would also like to thank William Hobart for providing valuable input during his review of our experiments.

References