Techniques and Tools for Distributed Shared Memory Performance Improvement

Mark D. Callaghan and Ariel M. Tamches
Department of Computer Sciences
University of Wisconsin, Madison

Abstract

Distributed shared memory (DSM) systems hide the details of communication for parallel applications by providing a shared virtual memory space kept coherent via message passing. Although easy to program, performance is poor when memory contention is high. Performance can be improved by detecting and predicting future memory access patterns, such as migratory, producer-consumer, and grouped reads. These patterns are interesting because mechanisms exist to improve their performance—provided the patterns can be detected. We have investigated run-time detection techniques that require no input from the programmer and no source code modifications. The performance improvement from these techniques has several fundamental limitations, which may indicate that automated techniques are not the best approach for improving DSM performance. We have also developed powerful and flexible tools to visualize the behavior of DSM programs on Stache, a fine-grained DSM system running on a 32-node CM-5. We used these tools to analyze the behavior of several large parallel programs.

1.0 Introduction

DSM systems allow distributed memory message passing machines to be programmed without worrying about communication details [Tem94]. For the purposes of this paper, a DSM system maintains shared memory in software on distributed memory message passing machines. Two consequences of hiding communication details are a simpler programming model, and a more complex performance tuning model.

The programming model for DSM applications is simpler than the model for message passing applications because communication is implicit in DSM applications. The implicit management of communication provided by a DSM system has a performance cost associated with it. There are two aspects to the performance cost. First, there is a communication overhead with respect to the amount of communication required by a message passing application for a similar exchange of information. Second, the programmer loses the opportunity to explicitly schedule long latency communication operations.

The overhead for a DSM application is the amount of communication required to exchange information, less the amount of communication required for a similar operation using a message passing system. This overhead can be viewed at a microscopic level by counting the number of messages to share a variable that has been updated. A macroscopic view can compare the total number of messages sent, the total number of bytes sent in messages, and the amount of time spent waiting for messages.

Message passing applications must explicitly schedule all exchanges of information. An application can make sure that it has all of the remote data required to execute a function by receiving the data in one message. By doing so, the application will only participate in one communication event. Additionally, it can overlap communication with computation, and overlap multiple communication operations. Naive DSM systems perform communication operations as a result of access faults due to loads and stores, which will be performed serially, rather than in parallel.

The process of improving the performance of applications for a DSM system is described in [Fal94]. The authors developed application specific protocols and were able to get the DSM applications to run as fast as message passing versions. The application specific protocols reduced the DSM overhead by using message passing codes. This optimization also scheduled the long latency operations to occur in parallel. We feel that a better way to judge the value of DSM systems is to develop application generic protocols. When such protocols allow DSM applications to approach the performance of their message passing counterparts, DSM systems will be a valuable tool for parallel programmers.

We attempted to improve the performance of DSM systems by increasing the number of communication operations that occur in parallel and by attempting to reduce the number of operations that would occur in the future, thus reducing the overhead. Our algorithms detect specific memory access patterns and predict future accesses based on the detection. We chose three types of memory access patterns that occur frequently and can be detected easily. The patterns are producer-consumer, migratory, and grouped reads. Predictions can be made about future memory accesses for these patterns. Predictions may reduce the overhead by reducing the number of communication operations and increase the number of long latency operations that occur in parallel.

Migratory sharing can be optimized by acquiring exclusive access to cache blocks while handling read misses. This would avoid an extra fault to acquire write access when the block is written. Producer-consumer sharing can be optimized by transferring the data to the consumer as soon as it is produced. Grouped reads are blocks that are likely to be accessed together. When a read miss is handled for one of these blocks, all of the blocks can be fetched. Blocks that have been grouped do not have to be contiguous.

We modified an existing cache coherence protocol to detect interesting access patterns at run time. The modified protocol attempted to reduce the DSM overhead and reduce long latency operations when patterns had been detected. Run time detection can take advantage of access patterns that do not exist for the entire execution of the application. It is limited by the amount of state it can maintain for each cache block, a lack of global knowledge, and a lack of processing time.

We wrote two powerful and flexible visualization tools to assist the user in tracking the DSM sharing patterns of its programs and used them to carefully analyze the behavior of two large DSM benchmarks, appbt and em3d. When used in conjunction, these tools enable the user to track all five possible axes of a program's DSM sharing patterns: among nodes, among event types, over time, among DSM blocks, and among values.

2.0 Related Work

Achieving performance comparable to message passing applications for DSM applications is an open research topic that has produced and will continue to produce schemes for classifying memory access patterns, tools that analyze these patterns, and protocols that detect and take advantage of these patterns. There are many constraints that make this an interesting research problem. Classification schemes need to be verified by collecting data for many applications. The patterns detected will be dependent on the granularity of coherence. Collecting data is difficult on systems that maintain cache coherence in hardware. Analysis tools must deal with an information overload, and be able to present results for hundreds of thousands of blocks simultaneously. New protocols are difficult to simulate without a DSM system that is maintained in software.

Munin provides cache coherence protocols optimized for specific memory access patterns [Ben90]. Munin supports granularity at the size of a virtual memory page. The large size used for coherence may obsure memory access patterns. Migratory and producer-consumer patterns were among the types described.

SM-prof graphically presents summaries of cache coherence events [Smp95]. Some types of access patterns can be inferred from the visualizations. However, per cache block information is not provided, and access patterns that apply to a subset of cache blocks may be obscured.

StormWatch displays event traces, and makes the trace display manageable by filtering events that a user is not interested in [Sto95]. However, patterns may only be interesting when they occur thousands of times and such a pattern may be difficult to see. Counts of events may be more suitable for some problems.

Two protocols that avoid the write penalty for blocks that are accessed in a migratory pattern have been researched [Cox93][Ste93]. These schemes detect at run time whether a block should be fetched with exclusive access to avoid an additional fault when the block is written. These schemes are very effective at reducing the number of write faults when migratory sharing is present. The schemes do not avoid the write penalty for blocks when the migratory pattern exists on a subset of the processors that access the block.

Protocol extensions that adapt to migratory, producer-consumer, and sequential memory access patterns are described in [DDS94]. The applications used were more amenable to optimization than the applications that we used. The prefetching scheme is limited to prefetching contiguous cache blocks. A scheme capable of prefetching non-contiguous blocks can improve performance more, but is difficult to implement without hints from the application programmer.

3.0 Memory Access Patterns

3.1 Pattern Types

We focus on memory access patterns that we can detect and predict. Regular expressions are used to specify the patterns. In the following regular expressions, superscripts refer to the cache block and subscripts refer to the processor. The optimizations will be described in terms of extra coherence events that occur in a sequentially consistent write-invalidate protocol that would not occur in a message passing system.

A cache block exhibits migratory sharing when each processor that accesses the block, writes the block, and the block is accessed by no more than one block at a time [Ste93][Cox93]. A message passing system can transfer a block with one message. A DSM system requires four messages (request/reply to acquire read access, request/reply to acquire write access). Migratory access can be optimized by fetching the block with exclusive access during read misses. This reduces the number of messages to two, and reduces the overhead of a DSM system.

Migratory sharing: $R_i{}^* W_i \left(R_i \middle| W_i\right)^* R_j{}^* W_j \left(W_j \middle| R_j\right)^*$

A cache block exhibits producer-consumer access when a block is only written by one processor, and then read by a group of processors. A message passing application requires one message for each application. A naive DSM system requires at least two messages for each consumer to acquire the data, and at least two messages for the producer to invalidate the consumer's copies. Producer-consumer access patterns are optimized by delivering the data to the consumers as soon as it is produced. Delivering the data to the consumer before the consumer requires the data overlaps several communication events. There are two technical problems that must be solved to update the consumers. First, there must be a way to determine when the value has been produced. The value may be written by the producer several times before it should be sent to the consumers. The second problem is determining the location of the consumers. If the set of consumers is not stable, there must be a way to detect when a block should not be sent to a stale consumer. While the latency of delivering the data to the consumers can be avoided with this technique, avoiding the invalidations by the producer requires a special purpose protocol. We did not attempt to avoid the invalidations.

Producer-consumer: $R_i^+ W_i \left(R_i \middle| W_i\right)^* R_j{}^+$

Blocks are grouped when they are likely to be read at the same time. A message passing system can group all data into one message. The cache blocks that provide the data for the message may not be contiguous. A DSM system may incur a read miss (and two messages) for each non-contiguous block, compared to one message for a message passing application. Prefetching blocks in this manner overlaps multiple communication operations and avoids a sequence of long latency operations.

Grouped reads: $R^i R^j$

Prefetching blocks also provides some of the benefit that a producer-consumer optimization may provide. If an application has producer-consumer sharing in which each processor produces one value using the values of four neighbors, prefetching can request the values of three of the neighbors on a read miss that had been generated from accessing the value of the other neighbor. This will not reduce the number of messages, but will reduce or eliminate the latency for all but one of the read misses.

3.2 Fundamental Limitations

The limits to performance improvement that can be achieved using our techniques can be determined by comparing the number of messages required for an optimized DSM system versus the number of messages for a message passing system. Migratory sharing patterns, producer-consumer sharing patterns, and grouped reads require at least twice as much message traffic as a message passing applications needs. An application using our optimizations would still perform twice as slow as a message passing application with respect to communication.

4.0 Methodology

We used Stache to provide sequentially-consistent shared memory [Tem94]. It uses a write-invalidate protocol and supports cache coherency for blocks of 32 bytes or larger. Stache is implemented using Blizzard-E, an implementation of the Tempest interface for a CM-5. Access control is enforced by modifying the ECC bits of the memory in which the blocks of shared memory reside. A 32 node CM-5 served as the distributed memory message passing machine.

Two applications, appbt [NAS91] and em3d [Cul93], were used to test run time memory access pattern detection and to generate traces. Appbt and em3d were interesting in that application specific protocols [Fal94] have been written that significantly improve their performance compared to a naive DSM version. The optimized applications provide a much better standard for performance comparison than the unoptimized versions. The applications were also interesting because they displayed different memory access patterns which will be described.

5.0 Run Time Detection

An existing cache coherence protocol, Stache [Tem94], was extended to detect migratory access patterns, producer-consumer access patterns, and grouped reads. It was also modified to issue prefetch requests and to support the acquisition of blocks with exclusive access while handling read misses. The schemes used to detect memory access patterns are constrained by a lack of knowledge about the source program. The schemes have no knowledge of the synchronization used by the application, and do not violate sequential consistency.

5.1 Memory Access Pattern Schemes

The results of running appbt and em3d with the various detection schemes are presented in Tables 1 and 2. The cache block sizes were 32 for em3d and 128 for appbt. The results and schemes are described below. Original is the behavior for an unmodified version of Stache.

5.1.1 Migratory-s

The scheme used to detect migratory access is described in [Ste93]. Block access patterns were classified as migratory when a write access fault occurred on a block for which the faulting processor had read access, the faulting processor was not the last writer for the block, and only one other process had read access to the block at the time of the write fault. When a block is marked as migratory, read access faults will acquire exclusive access to the block, rather than shared access. The modifications to Stache detected this type of sharing, and counted the number of write access faults that the optimization would have prevented. The information was not used to acquire blocks with exclusive access on read misses. Very little migratory access patterns were detected. The scheme would have eliminated 4% of the write access faults for appbt and less than 1% of the write access faults for em3d. This scheme requires memory for each block to record the previous writer.

5.1.2 Migratory-d

An alternative scheme used to detect blocks for which exclusive access should be acquired on read misses. A count was kept at each processor of the number of times the block had read faults and the number of times the block had write faults for read-only blocks. If the number of write faults was greater than a fixed percentage (90%) of read access faults, the block was acquired with exclusive access. Intuitively, a block exhibiting this access is frequently written by the processor, and read by other processors each time it is written. This scheme eliminated 6% of the write access faults to read-only blocks for appbt and less than 1% for em3d. This scheme requires memory for each block to count the number of read access faults and write access faults. We modified Stache to detect sharing and issue requests to acquire cache blocks with exclusive access.

5.1.3 Update

Update attempted to optimize producer-consumer memory access patterns while using a write-invalidate protocol. The scheme is not aggressive. Blocks that have been marked as update blocks have a read set which is the set of processors that will receive the update. An update is delivered to all processors in the read set when one of the processors in the read set has a read miss for the block. It is assumed that the value has been produced by this point in time.

Memory access patterns are classified as update when the same processor writes the block two times in succession. The read set is the set of processors that had read access to the block at the time of the second write. This scheme can adapt to short sequences of producer-consumer behavior. This scheme avoids the problem of detecting when the block has been written by the producer. The scheme is successful when the processors in the read set do not have read misses simultaneously. A read miss by one processor is the trigger that causes the block to be pushed to all processors in the read set.

Instrumentation was added to Stache to detect when this type of sharing is present, and to count the number of read misses that this scheme would have avoided. Delivery of the updated blocks was not implemented. Memory is required for each block to record the last writer and to record the read set. This scheme was successful for appbt. It eliminated 27% of the access faults for appbt. However, it only removed 1.3% of the access faults for em3d.

5.1.4 Prefetch

Prefetch attempts to issue prefetch requests for non-contiguous cache blocks during read misses. Stache was modified to support prefetch states and to issue prefetch requests. Each block could cause a prefetch request to be issued for one other block. When a read miss occurred for a block, the id for that block was recorded in the memory area of the block for which the previous read miss occurred if no invalidation had occurred between the two read misses.

Associated with each cache block is a field that stores a block id. This field stores the relation *Will Miss After(A, B)*. If block B has a read miss after block A, block A stores the id of block B in its field and the relation is valid. This relation forms grouped sets of size two. We assume the relation is transitive to construct larger grouped sets. If *Will Miss After(A, B)* and *Will Miss After(B, C)*, a read miss for block A causes blocks B and C to be prefetched.

Prefetch reduced the total number of access faults by 4.2% for appbt and by 22% for em3d. Prefetch requests can take a block away from a processor that is writing the block. Issuing requests for blocks only because the blocks are contiguous can cause this. However, this scheme only issues prefetch requests for blocks that are likely to have read misses one after another, and the effectiveness is seen by the lack of an increase in the number of write access faults.

Prefetching non-contiguous blocks may provide some of the benefit of an update protocol. Em3d executes a loop in which a processor updates values of graph nodes by getting values for the node from the node's neighbors. When a processor has a read miss at the start of a loop iteration, it can issue prefetch requests for all or most of the blocks required for that iteration. This can reduce the number of long latency operations per iteration to one.

TABLE 1. Appbt, 128 byte cache block, 32 node CM-5, event counts are multiples of 1000

	Stache			**Home**			
Method	**Read miss**	**Write to RO**	**Write to invalid**	**Read miss**	**Write to RO**	**Write to invalid**	**Total**
Migratory-d	802	331	25	14	5	<1	1178
Migratory-s	796	339	25	13	2	<1	1178
Update	475	352	24	13	5	<1	872
Prefetch	723	374	26	13	5	<1	1144
Original	796	352	25	13	5	<1	1193

TABLE 2. Em3d, 32 byte cache blocks, 32 node CM-5, event counts are multiples of 1000

	Stache			**Home**			
Method	**Read miss**	**Write to RO**	**Write to invalid**	**Read miss**	**Write to RO**	**Write to invalid**	**Total**
Migratory-d	6204	<1	246	44	4303	165	10964
Migratory-s	6204	<1	246	44	4302	165	10962
Update	6069	<1	246	44	4303	165	10829
Prefetch	3824	<1	246	43	4303	164	8583
Original	6204	<1	246	43	4303	165	10963

6.0 Performance

The decrease in performance due to the overhead of executing the detection algorithms, and bringing extra coherence state into the cache would reduce the performance improvement using the run time detection techniques. Yet even without adjusting for this overhead, the performance of appbt and em3d have not significantly improved using these techniques. A 20% to 30% performance improvement is insignificant compared to the performance of message passing implementations that improved the performance of appbt by a factor of six and em3d by more than a factor of ten.

Prefetching hints were added to the main loop in em3d. The results for a version of em3d with and without prefetching hints is presented in Table 3. The run time prefetching scheme seemed to be very effective, and a comparison to a naive implementation of prefetching may establish the effectiveness of the run time scheme. The prefetching hints were much more effective at reducing the number of read misses for em3d, and they removed 80% of the read misses, without increasing the number of write invalidations. The run time prefetch scheme was half as effective as the hint scheme. The run time scheme also has a much higher overhead due to detection. The performance of the run time detection scheme for prefetching does not seem to support its use.

An interesting point in the performance of the prefetching scheme can be seen in Table 3. Even though the number of read misses is reduced by 80%, the performance improvement measured in number of cycles per loop iteration is less than 8%. We have not spent sufficient effort to determine the cause of the disparity between the large reduction in communication operations and the small improvement in performance. Besides the overhead of issuing the prefetch requests, prefetching adds an additional overhead. When prefetching is performed, the processor will be interrupted when each prefetched block arrives. This overhead, increased network contention, and cache pollution may reduce the effectiveness of prefetching for this system.

The producer-consumer detection scheme, update, was also effective, relative to the other run time schemes. It has a much lower overhead in terms of execution and memory requirements at run time. However, there are several details that will limit its utility. If all of the consumers have simultaneous read misses, the scheme will have no performance benefit. The scheme also does not reduce or remove the number of invalidations that are issued when the producer writes the block. Again, the performance gained from this scheme does not seem to support its use.

TABLE 3. Em3d, 32 byte blocks, thousands of events, millions of cycles

Method	Read mess	Write to RO	Write to invalid	Total faults	Cycles per iteration
Prefetch	3132	1592	1226	5950	45.1
No Prefetch	15375	1592	1225	18193	48.3

7.0 Generic Mechanisms for Reporting DSM Access Patterns

7.1 Producing Data

We modified Stache to record, within each event handler of interest, a record containing the event type (4 bytes), the virtual memory address that was the subject of the event (4 bytes), and the event time (8 bytes). These records are buffered locally on each CM-5 node and get periodically flushed over a Unix socket to an external tool, who will process the trace of records as desired. We presently report 8 Stache events; they are:

TABLE 4. Stache Event Types

Stache Read Miss	Read miss on a block owned (i.e., managed) by another node. The miss is handled by sending a message to the owner node, who then obtains a copy of the block (in the process, downgrading anyone who has a write copy of it) and returns it to us. The home node also obtains a read copy of this block, if it did not have one, because it can be done at no extra cost.
Home Read Miss	Read miss on a block owned (i.e., managed) by the local node. Home misses can occur, in general, because block ownership by a given node does not imply anything about its current location(s). Stache uses a *fixed-distribution* management scheme; the owning node of a given block never changes and is trivial to compute. Home read misses are quicker to process than stache read misses, because we (the home node) immediately know which nodes have copies of the block.
Stache Write RO	Write access fault (specifically, write to a read-only block) on a block owned by another node. We need to “upgrade” our access from read to write; but first, we contact the home node who will invalidate other existing read copies before giving us the OK to upgrade.
Home Write RO	Write to a read-only block owned locally. Home Write RO faults are quicker to process than Stache Write RO faults, because we immediately have access to existing read copies that need invalidation before we may perform our “upgrade.”
Stache Write Inv	Write to an invalid (as opposed to read-only) page owned by another node.
Home Write Inv	Write to an invalid page locally owned.
Inv RO	Not a fault, *per se*; this operation is performed on a block in response to an invalidation order from its home node. Can result from any of the above events.
Inv RW	Same as above, but a block with write access (as opposed to read access) is being invalidated.

Trace records do not include a node identifier. Since each node sends its trace records over a distinct socket, the consumer implicitly knows the node each event occurred on. Also note that nodes do not synchronize during tracing. Traces record timestamps are guaranteed to be reported in order only within each node.

Because CM-5 nodes do not support sockets, we modified the Blizzard-E library to simulate them. Familiar-looking socket routines are provided to the processing nodes; each is actually a stub which transmits the request to the control processor, who carries out the operation on the node’s behalf, and returns applicable results.

Any Tempest application linked with our modified Stache and Blizzard-E libraries automatically turns into one that sends trace records over sockets. No application source code changes are required.

7.2 Scalability Issues

Program perturbation (percentage increase in runtime) varied greatly among the applications we studied. We were pleasantly surprised to find only a 25% perturbation in the NAS appbt application; however, em3d with extremely fine-grained (32-byte) blocks was slowed by a factor of three. We were actually surprised that it wasn't worse; after all, 32 nodes are sending trace information over a socket to a common (i.e., bottleneck) process. Network congestion collapse did not occur because we traced DSM events, all of which are expensive; hence, the time between two trace records on a given node is bounded below by the cost of processing the event of the first, which (as for all Stache events) is fairly high. We currently buffer 400 trace records (6400 bytes) on each node, so 400 records are sent over a socket for 0.25% of events, and nothing is sent for the other 99.75% of events.

Nevertheless, tracing event records will not scale. We feel that scalable reporting inherently requires a combination of sampling and tracing. That is, we take local action on seeing an event of interest, and periodically (say, once per second) send a *summary* of those events. The summary information—whatever its form—must be constant in size to be scalable. The challenge is making it as semantically useful to the consumer as traces. Most tools have no need for a trace record per Stache event (with an event code, timestamp, and virtual address). For example, updating a display of misses (by type) over fixed intervals requires just 8 integers (one per event type) to be sent each interval by each node. On the other hand, updating a display of misses *per block* over a fixed interval requires 8 integers to be sent by each node per block that experienced an event in the most recent interval. Although more expensive than the prior example, this is still cheaper than tracing.

Since we did not experience large perturbations, we used tracing in our initial implementation. Future versions will eliminate tracing by allowing data consumers to inform Stache that it only needs to be sent a particular function of the trace records. For example, the Barchart tool described below can get all the information it needs with just one record—of 8 integers—giving the total number of events (by type) occurring on this node during the last interval of time.

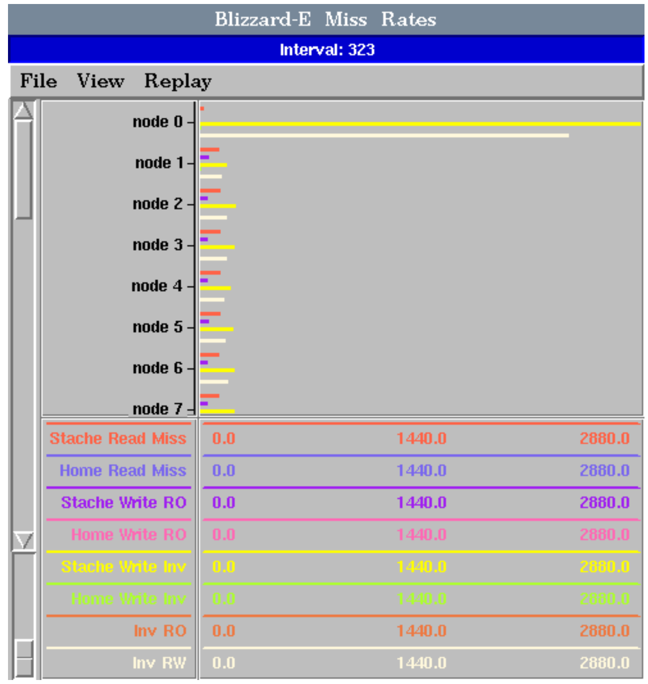
8.0 Barchart Analysis Tool

8.1 Introduction

The roots of many DSM application bottlenecks can be found by visualizing the frequency of cache coherence events on each node over discrete time intervals. For example, we might find that on average, node 15 has an unusually high proportion of “Write RO” to “Read Invalid” block events, suggesting that some of its blocks should be fetched in read-write mode rather than in read-only mode on Read Invalid events. As another example, noting a high rate of invalidations of read-write blocks on some node and high values of read misses on another might lead to speculation that a “checkin” operation [Hill93] should be performed when done writing certain blocks. (We would also like to know *which* blocks are the ones we should concentrate our study on; such a tool is discussed in the next section.)

While CM-5 application is running, Barchart conceptually updates values for each node every time a new interval begins. An interval is the discrete time granularity at which it displays data; presently, it is 10,000,000 CM-5 cycles, or about one-third of one second. A typical Barchart display is shown below. Note that only one time interval can be displayed at a time; each new interval redraws the entire screen.

By updating a node's bars to reflect the latest reported time interval, data for all prior intervals is lost. In other words, the visual display at any given time is a function of only the most recent time interval; the user has no feel of how the various miss rates are changing over time. This limitation, if left unaddressed, would cripple the Barchart's usefulness. We solved this problem in an elegant manner using the concept of *replay logs.* Every time the CM-5 program feeds us data for a new interval, we not only display it, we also append it to a log structure in memory. Conceptually, a Barchart replay structure stores 8 integers (one for each event type) for each interval on each node. When the program ends, the replay structure is saved to disk.



At any later time, Barchart can be started in "replay" mode instead of the usual "data collection" mode. Replay mode loads the replay log from disk and provides two special menus. The "replay" menu provides Play, Reverse Play, and Stop items to control a replay. Implementing replay simply requires page-flipping replay log entries (which are at the granularity of conventional time intervals) at a user-specified rate. Further control is provided by a slider widget at the bottom-left of the screen; as the user drags it to the desired interval, the screen is updated to reflect the widget's position. Replay can optionally run in *cumulative mode,* where a display at interval *n* shows the *sum* of values for intervals 0 through *n.*

As presently defined, replay files will not scale; we cannot afford to continue appending entries for each time interval, because long-running programs would create arbitrarily growing replay logs. We borrowed a trick from the Paradyn system [Hol94] to solve this problem. A fixed-size (say, 5000-entry) replay log is allocated for each node. When a node's replay log fills up, we *fold* it. A fold operation turns *n* log entries of time granularity *g* into *n/2* entries of time granularity *2g* by storing the sum of two consecutive entries into a single entry. This leaves *n/2* entries available for further appending; folding can repeat *ad infinitum.* The increase in granularity will cause only sporadic events to be lost; this is entirely acceptable because, by definition, sporadic events don't last long enough to be performance bottlenecks. A fixed-sized replay log of 5000 entries is guaranteed to show the effects of every interval taking at least 0.02% of execution time (regardless of how many minutes of time that is). Interestingly, a full log entry of 5000 entries is also guaranteed to *not* reflect any execution phase taking less than 0.02% of total execution time; we consider this a major (if initially unanticipated) feature.

As the CM-5 program runs, Barchart updates the display of miss rates per node in real-time. Unfortunately, the real-time display is not very accurate. Although we would like to see every bar of every node updated once per interval, this won't happen for two reasons. First, even if each node sends trace data every interval, the simultaneous updates would require explicit node synchronization once per interval—an unacceptable cost. Second, to reduce perturbation, each node sends trace data only when its 400-entry buffer fills up, not each new time interval. Fortunately, choosing to sacrifice real-time accuracy for lower perturbation was the right choice; assuming we have the luxury of deferring program analysis until the program finishes[1], we can view a replay of the run. Replay mode has none of the disadvantages of real-time mode; the bars of all nodes are updated simultaneously with correct values, regardless of how out-of-sync any two nodes were during real-time mode (which would occur if one node had more frequent events, requiring it to pause more while sending its trace buffer). This is where the trace record timestamps come into play; while gathering the replay log in memory during program execution, timestamps let us easily determine which interval a trace record belongs in; and hence, to which replay log entry should its values be added. Assuming synchronized clocks among the nodes during data collection mode[2], displaying data during replay for any interval will show the nodes in perfect sync.

We anticipate using Barchart to look for particular memory-reference patterns among the various nodes. For example, seeing a high ratio of (Home Write RO + Stache Read RO) to (Home Read Inv + Stache Read Inv) on a given node should tip user that the program may have migratory sharing. Better yet, since we have the luxury of replay, we can see if this holds true over time. In addition, we can see if it happens only for certain nodes. In general, think of a program's DSM sharing patterns as a 5-dimensional entity; dimensions are nodes, event types, event values, event times, and blocks[3]. Barchart's replay mode provides four of the axes: nodes and event types are clearly shown along the vertical axis; values are clearly shown along the x-axis. Event times are available via replay. The only dimension remaining is isolating the activity of particular blocks; a block-based visualization is discussed in the next section.

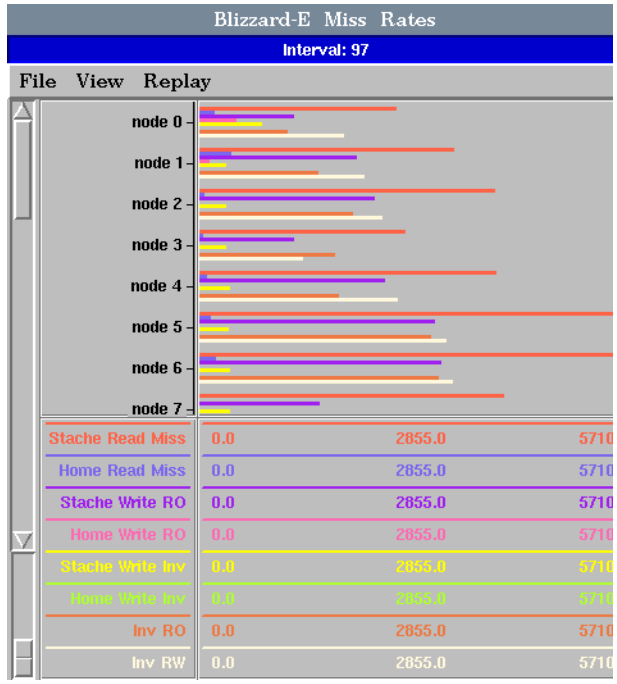
---

1. Presumably a burden only for extremely long-running programs.
2. Clock nodes are not perfectly synchronized; we empirically observe the clocks of any two nodes to be within 10,000 cycles (about 300 microseconds) of each other. Since this value is orders of magnitude less than our interval length of ten million cycles, the slightly out-of-sync clocks are not a concern.
3. Just thought of a sixth dimension—the program instruction causing the event. (-ari 11 May 95)

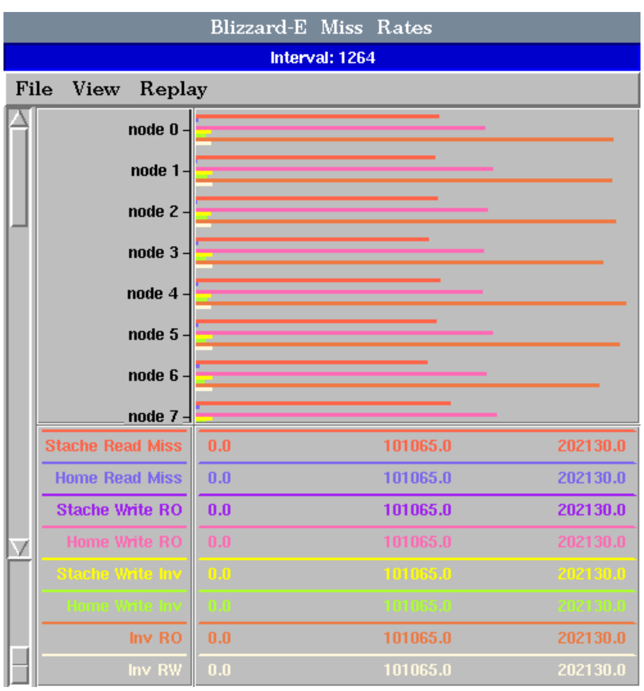
8.2 Experiences with Barchart

8.2.1 appbt

A quick glance at the Barchart to the right reveals an optimization opportunity. There are many Stache Write RO events on each node. Acquiring exclusive block access while handling Stache Read Misses could avoid these faults. There are also a high number of invalidations for blocks—both shared-mode and exclusive-mode. We cannot avoid these events without an update protocol or source code hints such as checkin/checkout [Hill93].



8.2.2 em3d



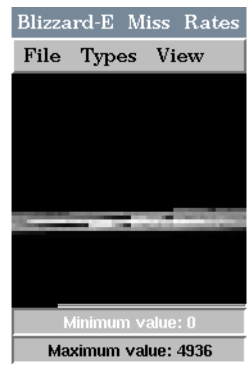
Barchart (left) clearly reveals a producer-consumer sharing pattern in em3d. The home nodes for producer blocks are the only writers to them. The high number of coherence events is due to three factors. First, Stache Read Misses occur when the consumer reads the block after having been updated by the home node. Second, Home Write RO faults occur when the home node attempts and fails to write the block after having been read by the consumers. Third, there are a high number of Inv RO requests sent to the consumers as a result of the previous fault.

9.0 Hotblocks Analysis Tool

9.1 Introduction

Many DSM applications have performance bottlenecks which can be isolated to Stache event activity on a particular set of blocks. High block miss rates can occur for several reasons; typical cases include contention for synchronization variables, migratory data sharing patterns, and producer/consumer data sharing patterns. DSM assistance can take the form of prefetching (in read-only or exclusive mode), acquiring exclusive access on a Read Miss, and checkin [Hill93]. The Hotblocks visualization tool provides a framework for identifying which blocks have high miss rates. This contrasts with the Barchart visualization tool of the previous section, which provides a framework for identifying which *nodes* have high miss rates. As with Barchart, we can distinguish between the several event types (write miss, write to a read-only block, etc.). Unlike Barchart, we have not yet implemented the concept of time (i.e., replay mode) into Hotblocks; information the user sees corresponds only the entire program duration.

A sample Hotblocks display is shown to the right. Every pixel corresponds to one Stache block, in increasing virtual memory order. To save screen real estate, we have invoked a program option to exclude blocks which were “unused”—those with absolutely no Stache events on them for the entire program duration. Hotblocks provides menu options to display any combination of the 8 Stache event types, or the special metric of (Stache Write RO + Home Write RO) / (Stache Read Invalid + Home Read Invalid). The screen display to the right represents the metric {Stache Read Invalid + Home Read Invalid} for a 20-minute run of a parallel benchmark.



When the set of types to be displayed is changed, Hotblocks loops through its internal structures (which hold, for each Stache block, 8 integers representing values of the 8 Stache event types that block experienced throughput the entire program duration) to calculate minimum and maximum values. Blocks having the maximum value are displayed in bright white; blocks having the minimum value are displayed in black; 100 shades of gray represent middle ground. To enable useful comparisons of two Hotblocks displays, the minimum and maximum values are printed out. (After all, a bright white pixel of one Hotblocks display is saying something quite different from the bright white of another display—unless they coincidentally have the same minimum and maximum values.) We note that a black pixel does not necessarily imply that none of the selected events happened on that block during the program’s execution; it might simply be black because the maximum value (bright white) was so high as to make moderate values relatively close to 0.

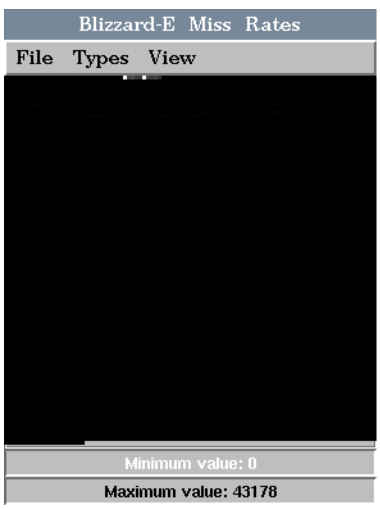
We anticipate using Hotblocks in an analogous way to Barchart; the former helps isolate Stache events and values to particular blocks, and the latter helps isolate Stache events and values to particular nodes. The two complement each other very well; we feel that both are necessary. Recall that we consider a program’s DSM sharing patterns to be a 5-dimensional entity; dimensions are nodes, event types, event times, blocks, and values. Hotblocks—in its current implementation—can give the user information on three of the five axes. To view how a Hotblocks display changes with respect to event types, the user selects from the possible permutations under the “Types” menu. Values are represented by the shades of gray. Implementing replay in Hotblocks would add the time axis, bringing the total to four of the five dimensions (like Barchart). What Barchart sorely lacks (the ability to distinguish individual blocks) is easily handled by Hotblocks; what Hotblocks presently lacks (time and the ability to distinguish individual nodes) is gracefully handled by Barchart.

Implementing replay in Hotblocks will be difficult because collecting and storing the necessary logs are more expensive than in Barchart. Presently—without replay—Hotblocks requires 8 integers per block. The applications we have seen use between 10,000 and 500,000 blocks, thus requiring from 320K to 16MB of memory. Adding replay with 5000 total buckets (the default currently in Barchart) would require gigabytes of storage.

9.2 Experiences with Hotblocks

9.2.1 appbt

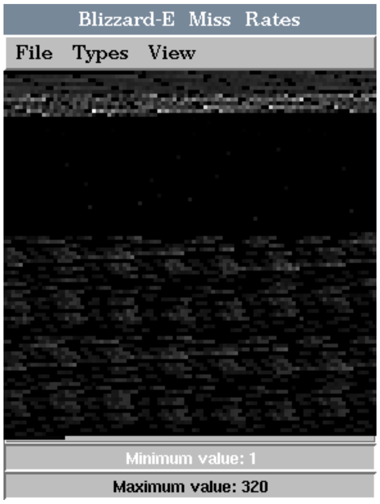
A Hotblocks display of Stache Read Misses for appbt+busy (which ran for about 2.5 minutes) immediately pinpoints a tremendous performance bottleneck to a set of two blocks.



Notice two striking aspects to this display. First, the maximum value (43,178 Stache read misses) is extremely high for a program running for just 2.5 minutes. That explains why the vast majority of pixels are black; their values are probably not 0, just less than 431 (the scale spread from maximum downto minimum values currently involves 100 grays). Although we don't have room to show every picture permutation, adding Home Read Misses to the mix only increases the maximum value to 43,666 total events. Adding Stache Write RO events increases it to 51749; each time, the same two blocks are drawn in bright white.

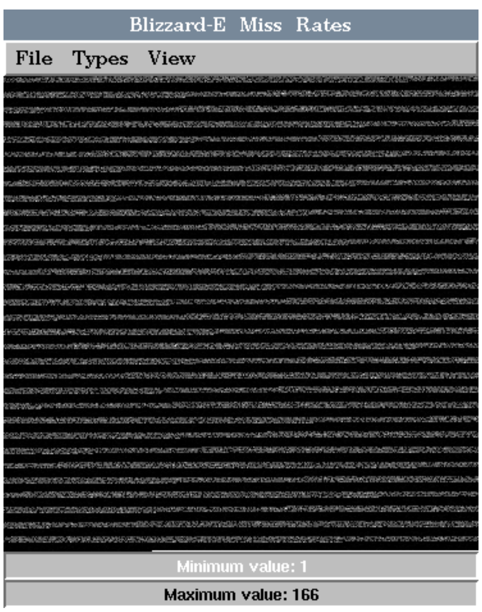
What is happening here? appbt+busy is a naive DSM implementation of a program with tremendously high contention for an array of shared busy-wait lock variables. Clearly, the sequence of six pixels represents that array.

appbt+sigwait was developed by [Fal94] to alleviate the DSM thrashing of appbt+busy using "signal" and "wait" queue primitives; a node successfully completing an update signals the node waiting at the head of the queue (using Tempest active messages), who then proceeds with its calculations.

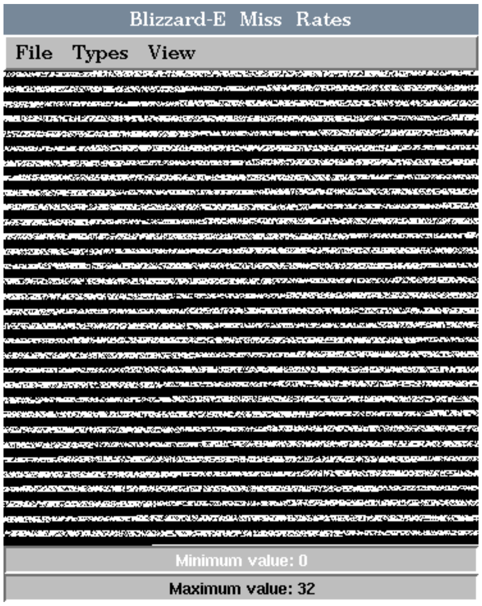


A Hotblocks display of appbt+sigwait (with a chosen Stache event type set of {Stache Read Miss + Home Read Miss + Stache Write RO + Home Write RO + Stache Write Inv + Home Write Inv}) shows that DSM performance has improved significantly over appbt+busy. Although there are many more shaded blocks (including more bright white pixels), the maximum value is now just 320, down from over 50,000! However, if taken alone, the change in Hotblocks graphs may lead one to wrongly conclude that appbt+sigwait is orders of magnitude faster than appbt+busy. Since our tracing system cannot (yet) report application-specific types; signal and wait operations are not seen; even if we could count them, non-block-based types have no place in a Hotblocks display; they belong in Barchart.

9.2.2 em3d



A Hotblocks display of em3d+32, a naive DSM version using 32-byte Stache blocks, reveals that no block immediately stands out as a singular performance bottleneck, as in appbt+busy. The picture has been set to include the Stache event type set of {Stache Read Miss + Home Read Miss + Stache Write RO + Home Write RO + Stache Write Inv + Home Write Inv}. No single block has more than 166 of these events over the program's running time, which was roughly 8 minutes; however, a rough glance indicates that about half the blocks have medium gray values, indicating they incurred around 50% of the events of the most contentious block.



Recall that our Barchart analysis of em3d concluded that it exhibits update sharing. We wanted to use Hotblocks to help determine if certain blocks have a combined "migratory update" sharing pattern. The Hotblocks display to the left has event types set to {Home Write RO + Stache RO}; it shows that no block has more than 32 of these misses (although about half of the blocks approach this maximum). Since these values are significantly less than those of the previous slide (166), we conclude that em3d does not exhibit migratory sharing for any significant fraction of program execution time.

9.3 Future Hotblocks Work

Hiding unused DSM blocks (the default in Hotblocks) makes it impossible to deduce the virtual DSM address of a given pixel being displayed. Furthermore, this collapsing together of all used blocks may artificially give the appearance of interesting patterns between consecutive blocks—artificial because the blocks might not be truly contiguous in virtual DSM. We have recently augmented Hotblocks to display a virtual address for any desired pixel (by moving the mouse over a given pixel). But there are times when we want a feel for the spatial locality; removing unused blocks makes it impossible to distinguish "runs" of consecutively allocated virtual DSM. We have addressed this concern with an option to explicitly allocate pixel space for displaying unused blocks (they are drawn in a distinct color). But even this display does not help the user distinguish between separately-allocated chunks of virtual DSM which happen to be contiguous.

It would be better if Hotblocks explicitly distinguishes each allocated chunk of DSM. We can think of two ways to implement this. In the first, a distinct color pixel is inserted between every Stache-allocated chunk of DSM. In the second, each allocated chunk of blocks is drawn on a separate line; the window width is now the maximum number of fine-grained blocks per allocated chunk, and the window height is now the number of allocated DSM chunks. Implementation might take the form of sending a special kind of trace record whose type indicates a Stache heap allocation and whose address field gives the beginning of the allocated chunk. The chunk length (in blocks) can be implicitly calculated, assuming the beginning-of-chunk information is available for every chunk. This solution, by distinguishing chunks of DSM, would be extremely useful. However, we can do even better. We *really* want separators between DSM-allocated program variables; we've been assuming that each consists of one chunk. That is often true, but many complex data structures (such as trees) are made of several separately-allocated heap blocks. Conversely, some heap blocks contain several logical variables (e.g., arrays). A generic solution to these problems requires the application to explicitly send type-specific hints to help Hotblocks map the set of heap chunks to a set of logical variable chunks. Although such a scheme is complex, the potential utility is so high that we plan to implement it.

10.0 Variable Tracker Tool

Variable tracker is a proposed visualization tool that gives Barchart the ability to reveal data sharing among blocks—the only dimension (among nodes, event types, event times, blocks, and values) of DSM sharing pattern behavior that it presently cannot track. We plan to implement this visualization in early Summer, 1995. In this section, we briefly outline the intuition behind the Variable Tracker, and how it can be implemented.

The Variable Tracker graphical interface looks like Barchart. As in Barchart, each variable tracker bar displays the value for a given event type on a given node, during the current interval. Also as in Barchart, if we are replay mode, the values displayed are cumulative for this and all previous intervals. Unlike Barchart, Variable Tracker bar values are shown not for events experienced by *every* block on a given node, but rather by a user-specified subset of distributed virtual memory locations created with the new "stache_malloc_trackme()" instead of the usual "stache_malloc()"[1]. The former version augments the latter by sending a special record to the Variable Tracker informing it of the range of distributed shared memory addresses the user wishes to track. The Variable Tracker shows Barchart values for *just these blocks*, enabling the user to effectively visualize the sharing patterns of variables it has declared an interest in. For example, it might see thrashing behavior as many nodes try to update a shared block simultaneously, and conclude that an explicit barrier with data forwarding is called for. Or, it might see a high "Home Write RO" rate on one node, and high "Stache Read Inv" rates on others, vividly revealing an update sharing pattern on this variable.

Variable tracking will not be difficult to implement on top of Barchart. The key is that the *virtual* address of a given block of Stache DSM memory does not change once allocated. Our trace records have always included virtual block addresses. Hence, it is simple to decide whether or not a trace record refers to given variable(s) of interest.

11.0 Conclusion

Automated detection and feedback from that detection does not provide significant performance improvement. There are fundamental limits to the performance improvement which can be achieved using our techniques. The number of messages for a DSM using our hints will be at least two times that of a message-passing version. Improving the performance through explicit source code hints and application-generic protocols may provide the best combination of performance and utility. Unfortunately, performance improvements do not necessarily follow reductions in the number of communication and long latency operations which are not overlapped. We

1. There is no routine actually called stache_malloc(); we use it only for clarity. In addition, Stache actually provides two allocation routines, one for performing a home allocation (i.e., on the node making the request), and one for performing a Stache (i.e. remote) allocation.

found that network contention and the overhead of handling interrupts can nullify the savings achieved from prefetching.

Although we have only used our Barchart and Hotblocks visualization tools a short time, they have been extremely useful in helping us understand the memory access patterns of the DSM programs we have run. We identified the five dimensions (nodes, event types, event values, event times, and blocks) of memory reference patterns. Barchart can show four of them (all except blocks); Hotblocks can show three (all except nodes and time). Using both tools on a program will obtain an excellent understanding of all five. The proposed Variable Tracer visualization tool will help users track the memory reference patterns of desired variables; in effect, it changes the "block" memory reference dimension into a more meaningful "variable" memory reference dimension.

12.0 References

[Ben90] J. K. Bennett, J. B. Carter and W. Zwaenepoel, "Adaptive Software Cache Management for Distributed Shared Memory Architectures," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, June 1990.

[Cox93] A. L. Cox and R. J. Fowler, "Adaptive Cache Coherency for Detecting Migratory Shared Data," *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[Cul93] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, "Parallel Programming in Split-C," *Proceedings of Supercomputing '93,* November 1993.

[DDS94] F. Dahlgren, M. Dubois, and P. Stenström, "Combined Performance Gains of Simple Cache Protocol Extensions," *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.

[Egg90] S. J. Eggers and R. H. Katz, "A characterization of sharing in parallel programs and its application to coherency protocol evaluation," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, May 1988.

[Fal94] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood, "Application-Specific Protocols for User-Level Shared Memory," *Proceedings of Supercomputing '94*, November, 1994.

[Hill93] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D.A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems,* 11(4):300-318, November, 1993.

[Hol94] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic Program Instrumentation for Scalable Performance Tools. *Proceedings of the 1994 Scalable High Performance Computing Conference,* May, 1994.

[NAS91] D. Bailey, J. Barton, T. Lasinski and H. Simon, "The NAS Parallel Benchmarks," Technical Report RNR-91-002 Revision 2, Ames Research Center, August 1991.

[Smp95] M. Brorsson, "SM-prof: A Tool to Visualise and Find Cache Coherence Performance Bottlenecks in Multiprocessor Programs," To appear in *ACM Sigmetrics and Performance '95*.

[Ste93] P. Stenström, M. Brorsson, and L. Sandberg, “An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing,” *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[Sto95] T. M. Chilimbi, T. Ball, S. G. Eick, J. R. Larus, “StormWatch: A Tool for Visualizing Memory System Protocols,” Submitted for publication, *Supercomputing ‘95*.

[Tem94] S. K. Reinhardt, J. R. Larus, and D. A. Wood, “Tempest and Typhoon: User-Level Shared Memory,” *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.