COMMUNICATION MECHANISMS IN SHARED MEMORY MULTIPROCESSORS

Gregory T. Byrd
Bruce A. Delagi
Michael J. Flynn

Technical Report CSL-TR-94-623

May 1994

This paper was presented at the *International Workshop on Support for Large-Scale Shared Memory Multiprocessors*, held in conjunction with the *8th International Parallel Processing Symposium*, Cancun, Mexico, April 26-29, 1994. Preliminary work was supported by an NSF Graduate Fellowship and a NASA/DARPA Assistantship in Parallel Processing.

Communication Mechanisms in Shared Memory Multiprocessors

by

Gregory T. Byrd

Bruce A. Delagi

Michael J. Flynn

Technical Report CSL-TR-94-623

May 1994

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305-4055

Abstract

Shared memory systems generally support *consumer-initiated* communication; when a process needs data, it is retrieved from the global memory. Systems that were designed around the message passing model, on the other hand, support *producer-initiated* communication mechanisms; the producer of data sends it directly to the other processes that require it. Parallel applications require both kinds of communication.

In this paper, we examine the performance of five shared-memory communication mechanisms—invalidate-based cache coherence, prefetch, locks, deliver, and StreamLine—to determine the effectiveness of architectural support for efficient producer-initiated communication. We find that StreamLine, a cached-based message passing mechanism, offers the best performance on our simulated benchmarks. In addition, StreamLine is much less sensitive to system parameters such as cache line size and network performance.

Contents

1 Introduction 1

2 Mechanisms 1

3 Simulation Methodology 2

4 Application Benchmarks 2

4.1 Sparse Cholesky Factorization . . . 3

Basic Performance . . . 3

Varying Private Miss Rate . . . 5

Varying Cache Line Size . . . 6

Varying Network Performance . . . 6

Discussion . . . 6

4.2 Gaussian Elimination . . . 7

Basic Performance . . . 7

Varying Cache Line Size . . . 9

Varying Network Performance . . . 9

Discussion . . . 9

5 Synthetic Benchmark 9

Basic Performance . . . 11

Varying Cache Line Size . . . 11

Varying Network Performance . . . 11

Discussion . . . 14

6 Future Work 14

7 Conclusions 14

Acknowledgements 15

List of Figures

1 Sparse Cholesky factorization, `494bps`. 3

2 Sparse Cholesky, `494bps`—Effects of cache line size. 5

3 Sparse Cholesky, `494bps`—Effects of network cycle time. 5

4 Gaussian elimination, 64×64. 7

5 Gaussian elimination, 64×64—Effects of cache line size. 8

6 Gaussian elimination, 64×64—Effects of network cycle time. 8

7 The synthetic benchmark. 10

8 Synthetic benchmark $G1$. 11

9 Synthetic benchmark $G2$. 11

10 Synthetic benchmark $G1$—Effects of cache line size. 12

11 Synthetic benchmark $G2$—Effects of cache line size. 12

12 Synthetic benchmark $G1$—Effects of network cycle time. 13

13 Synthetic benchmark $G2$—Effects of network cycle time. 13

List of Tables

1 Sparse Cholesky, `494bps`—Effects of private cache miss rates. 4

2 Description of synthetic benchmarks. 10

3 Execution time with default system parameters, relative to invalidate. 15

1 Introduction

Data communication is one of the basic mechanisms of all parallel computing systems. The performance of a parallel application often strongly depends on the latency experienced in communicating data between its threads or processes. The architectural decisions which affect communication latency, therefore, have a profound impact on the performance which can be achieved by a parallel system.

We will study several communications mechanisms in the context of scalable shared memory multiprocessor systems. These systems provide architectural support for a single global address space, which is preferred by many programmers. Multiprocessors such as KSR1 [8], DASH [16], and Hector [23, 24] have shown the feasibility of scalable shared memory systems with communication latencies orders of magnitude smaller than shared virtual memory emulations on distributed memory machines.

Shared memory systems, however, generally support only *consumer-initiated* communication; when a process needs data, it is retrieved from the global memory. Systems that were designed around the message passing model, on the other hand, support *producer-initiated* communication mechanisms; the producer of data sends it directly to the other processes that require it.

Parallel applications require both kinds of communication. Producer-initiated communication may not be possible, because the consumers of a particular piece of data may not be known when that data is produced. Consumer-initiated communication may be desirable if only parts of a complex data structure are needed by certain consumers. On the other hand, sending data close to its consumer as or before it is needed can greatly reduce the latency of communication and therefore increase effective performance.

In this paper, we examine the performance of various shared-memory communication mechanisms, to determine the effectiveness of architectural support for efficient producer-initiated communication.

2 Mechanisms

We consider five basic communication mechanisms in this study, ranging from purely consumer-initiated to aggressively producer-initiated. They also represent a range of cost and complexity, but we will focus only on performance, not implementation cost.

Invalidate-based Cache Coherence. Hardware-based cache coherence protocols are used in many scalable multiprocessor systems to move data into a processor's local cache when it is referenced. For the purposes of this study, we assume a fully-mapped directory-based invalidate protocol [3]. This is a pure consumer-initiated mechanism which is used as the baseline for comparisons. An atomic swap instruction is also assumed for synchronization purposes.

Prefetch. The prefetch operation allows the processor to ask for data to be brought into its nearest cache before it is actually needed [18]. If the data is currently in a remote cache, then some or all of the cost of fetching it may be overlapped with computation, reducing the effective latency. In our models, any number of prefetch operations may be in progress concurrently.

Lock. This approach adds states to the cache coherence protocol in order to implement atomic operations on cache-line boundaries [11]. A processor may lock a cache line, which brings the data into its local cache and prevents other processors from accessing it. Other lock requests form a queue, which is stored with the directory in main memory; when the cache line is unlocked, it is passed directly to the first local cache in the queue, if any, combining synchronization and communication in a single transaction. This mechanism is a hybrid between consumer- and producer-initiated schemes. The consumer must first ask for the data, but then it is sent by the producer as soon as it is available.

Deliver. First proposed in the DASH system [16], deliver is in a sense the opposite of prefetch. Data from one cache is transferred directly into another cache by the producer of the data, without waiting for the consumer to request it. A similar mechanism, called *post-store*, is present in the KSR1 architecture [8].

StreamLine. The StreamLine mechanism [2] allows direct cache-to-cache transfer of arbitrary blocks of data. Special regions in memory, called *streams*, are managed by hardware as circular message buffers. A producer writes data onto a local stream in its nearest cache, then directs the cache to transmit that data as a block to a remote cache, where it is written onto a different stream. The consumer then reads the data directly from its local stream.

Reading and writing stream data involves the manipulation of hardware-supported head and tail pointers. To write a message, data is written to the stream buffer and the tail is moved to indicate the end of valid data. The tail pointer on a receiving stream is automatically incremented as data arrives from the network. When a message is read from the stream buffer, the head pointer may be moved to free up space for more data.

There is no existing system which implements the StreamLine mechanism directly, but the general idea of providing an application-level direct interface to the network is found in the J-Machine [4], Alewife [13], and T3D[12].

3 Simulation Methodology

We evaluate the mechanisms described above through simulation of model implementations running benchmark programs. The simulator used for this study is an extension of the SIMPLE/CARE system [6], which provides instrumented execution-driven simulation of multiprocessor systems. We also use a variation of the ALOG [1] system, for generating event traces.

The base simulated hardware is a shared memory multiprocessor system, with cache coherence implemented by a fully-mapped directory-based invalidate protocol [3]. Each processing node consists of a processor, a first-level cache for non-shared data, a second-level coherent cache, a portion of the globally shared memory, and a network interface. Benchmark programs are run by the processor modules and are annotated with cycle counts for serial operations. A first-level private cache is modeled statistically, using a user-supplied miss rate. Shared data in the memory and second-level cache is modeled explicitly, while private data is modeled statistically. By default, caches are infinitely large and there are no private data or instruction misses.

A weak consistency model [9] is provided for memory operations; programmer-inserted *fence* operations are used to wait for all outstanding write operations to complete as needed.

The simulated system for the results reported in this paper is a 64-node system, connected as a 2-dimensional mesh with oblivious wormhole routing [5]. The processor cycle time is set at 10 ns. The second-level cache also cycles at 10 ns, and data is available to the processor in two cycles. The default cache line is 4 words (16 bytes). The memory cycle time is 60 ns, with page-mode access available at 10 ns per word. Network channels are 32 bits wide and are cycled at 20 ns, resulting in 200 MB/s per channel in each direction; latency across each network switch is 20 ns.

4 Application Benchmarks

For this study we use two synthetic benchmarks and two benchmarks based on application kernels. The kernels are Cholesky factorization of a sparse symmetric matrix and LU factorization of a dense matrix using Gaussian elimination with partial pivoting.

These benchmark programs were chosen to illustrate the performance of the various communication mechanisms under different communication patterns; they are not intended as optimal parallel algorithms, although a best effort has been made to optimize the use of each mechanism.

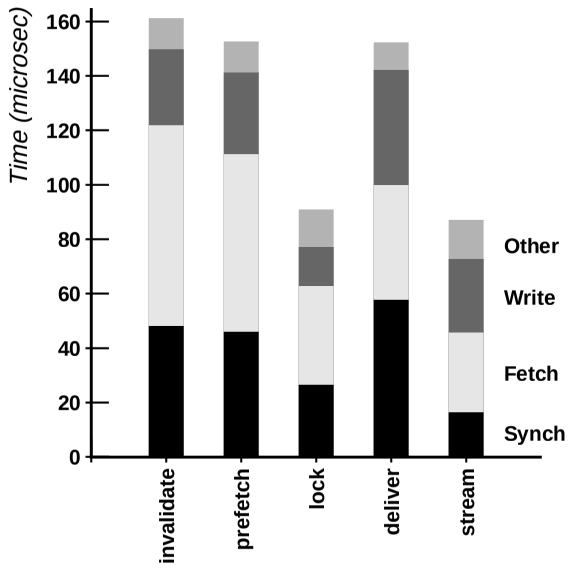

Figure 1: Sparse Cholesky factorization, `494bps`.

4.1 Sparse Cholesky Factorization

In sparse Cholesky factorization, the inter-processor communication patterns are highly data-dependent. Message[1] sizes are typically small and fan-in and fan-out is highly variable. Only the numerical factorization phase of the algorithm is considered, not the symbolic factorization or any setup of the data structures.

The algorithm chosen is a simple column-oriented scheme, where the columns are statically assigned to processors. Each column gathers contributions from factored columns to its left, completes its own factorization, and then provides the new data to the appropriate columns to the right. Columns only interact if they share non-zero elements in the same row.

A production-quality algorithm would likely use techniques like supernodes[20] to reduce communication requirements and to more effectively assign work to processors. Our simple implementation, then, is probably more sensitive to communications performance than a production-quality code.

Basic Performance

Figure 1 shows the performance of the code on the `494bps` matrix from the Harwell-Boeing collection [7], using the five mechanisms described above. Also shown are the times spent in various parts of the application, accumulated over the *critical path* of the execution. The critical path is the sequence of computation and communication events that determine the overall execution time of the program—it is determined by examining event traces written during simulated execution. The time categories are as follows:

- *Synch* is the amount of time spent synchronizing the producer and consumer threads. Synchronization time includes *wait* time, in which the consumer is stalled waiting for some event, and *signal* time, which is the delay from the signalling of an event until the recognition of the event by the consumer. For a given synchronization point, either the wait time or the signal time may be on the execution's critical path, but not both.
- *Fetch* is the time spent by a consumer reading the data needed for the computation.

[1] In this context, "message" is used to denote the amount of data that is communicated in a producer-consumer transaction. Its use does not imply a message-passing style of computation.

	time (μsecs)		
	0% miss	*10% miss*	Δ
invalidate	161.29	167.45	+4%
prefetch	152.73	165.27	+8%
lock	91.01	102.33	+12%
deliver	152.40	160.07	+5%
StreamLine	87.18	103.98	+19%

Table 1: Sparse Cholesky, `494bps`—Effects of private cache miss rates.

- *Write* is the time spent storing the computed data for consumption by other threads.

- *Other* is everything else, including the time spent actually computing. There may also be idle time during the execution, caused by some processors finishing before others, but it is not part of the critical path.

The lock and StreamLine mechanisms are clearly the best performers for sparse Cholesky, mostly because of efficient synchronization. The algorithm is quite imbalanced—some processors finish their work early and sit idle for a long time, while other processors spend a long time waiting for other columns to supply them with data. A reduction in synchronization time has a significant impact on the critical path of this application.

For the other mechanisms, synchronization is implemented by spinning on a shared variable, called the *synchronization variable*. While a consumer is spinning, a copy of the variable is resident in its cache. (In general, there may be many consumers spinning on a single variable, but that does not happen in this program.) In order to write the synchronization variable to signal an event, the producer must first get a copy of the cache line, then write to it. The consumer's copy is then invalidated, which means the consumer must fetch a new copy. Only then does the consumer see the new value, which allows it to begin fetching useful data, which is most likely resident in the producer's cache.

Because the fan-in and fan-out patterns for each column are known at the beginning of the program, separate data buffers can be allocated for each communication. In the lock-based implementation, producers initially lock the buffers, and consumers issue non-blocking lock operations long before the data is actually needed. If the line is still locked when the consumer actually reads the data, the processor will block at that point. Because both the number of consumers and the amount of data to be communicated is typically small, it is efficient for the producer to write and unlock each buffer. Since the consumer's lock request is already pending, the data is transmitted directly to the consumer as soon as it is unlocked.

The StreamLine case is similar—separate streams are allocated for each message required by the consumer. The consumer simply reads the head of the stream, blocks if data has not yet arrived, and continues as soon as it arrives.

Prefetch is ineffective for sparse Cholesky because of the small message size. The first line cannot be fetched until after synchronization, which is the same as in the non-prefetch case. Because all or most of the data is contained in that first cache line, there is little opportunity to hide the latency of additional fetches.

The deliver mechanism is used by the producer to send copies of the data to each consumer as it is copied into the buffers. This does result in a reduction of average fetch time, as shown in Figure 1, but the additional overhead of the deliver operations increases write time, and synchronization is still expensive, so the overall impact on performance is minimal.

Because of the success of the lock mechanism in reducing synchronization time, a combination of lock and deliver was tried to see if we could take advantage of an additional reduction in fetch time. However, possibly because of the small message size, deliver did not noticeably improve the fetch time, and a slight increase in write overhead made the combined approach less attractive than the lock mechanism by itself.

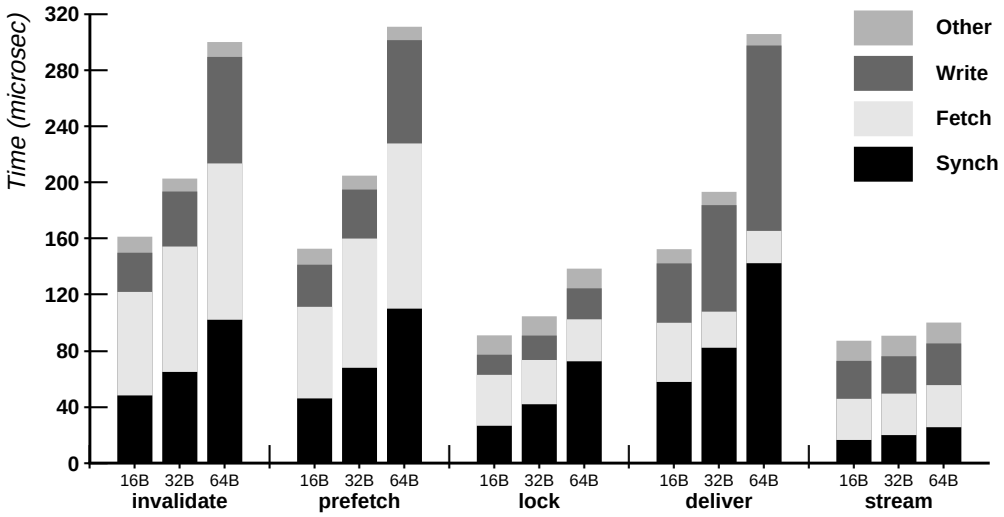

Figure 2: Sparse Cholesky, `494bps`—Effects of cache line size.

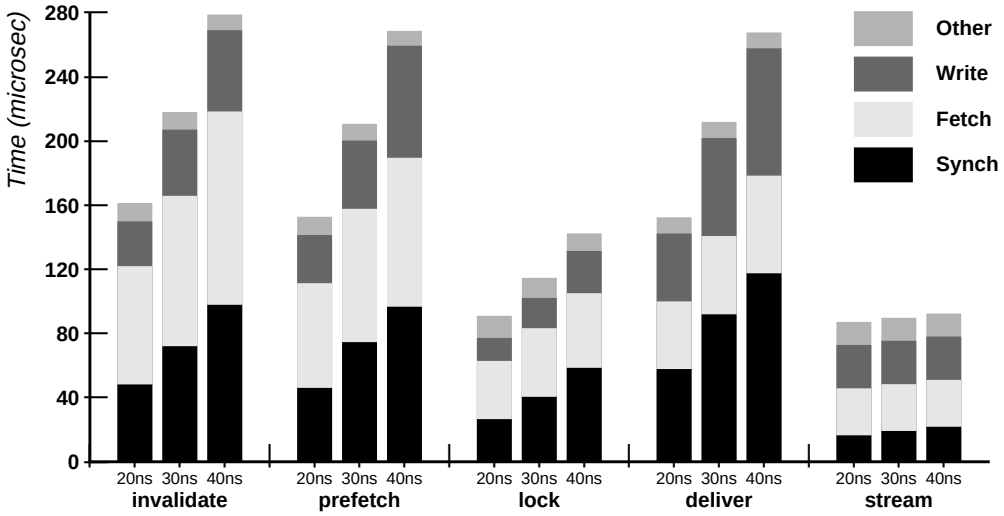

Figure 3: Sparse Cholesky, `494bps`—Effects of network cycle time.

Varying Private Miss Rate

Table 1 shows the effects of varying the miss rate for private data and instructions between 0% and 10%. The model for private (first-level) cache misses is a statistical one, which makes no distinction between instruction and data references. Cache misses are randomly inserted into the processor's instruction stream, based on the user-supplied miss rate.

The StreamLine mechanism is more sensitive to this parameter because of the extra instructions needed for manipulating the stream head and tail pointers, as described earlier. The figure shows that StreamLine performance degrades 19% with a 10% miss rate, compared to 12% for lock and 4% for invalidate.

The effect of private misses is similar for all of the benchmarks, so this variation will not be considered further.

Varying Cache Line Size

Figure 2 illustrates the effect of changing the line size of the second-level cache. Normally, one would expect an increase in line size to result in a decrease in the cache miss rate [21], which usually translates into a reduction in overall execution time. This is not always true for cache-based multiprocessors, however. In this case, there are two overheads which are working against the cache coherence mechanism:

1. Since the message size is small, much of the data transferred with a large cache line is unused. This increases the cost of cache operations and increases network load, without providing any benefit in the form of decreased memory reference time.

2. Synchronization variables are allocated in their own cache lines, in order to avoid false sharing with the message data. The `swap` operation results in the invalidation and transfer of an entire line. Therefore, the cost of synchronization goes up with cache line size in the invalidate, prefetch, and deliver mechanisms.

For the invalidate, prefetch, and deliver mechanisms, the above effects translate into an increase in synchronization time with cache line size. Write time is also increased—this is probably due to increased load on the network and the memory modules. In the deliver case, write time increases dramatically because data is delivered a line at a time. The apparent decrease in fetch time is due to a shift in the critical path toward the more write-intensive parts of the program.

Lock and StreamLine, which do not depend on a simple synchronization variable, are not as sensitive to the change in line size. The decrease in performance for these mechanisms is due to the first effect—the overhead of transferring unused data. The StreamLine mechanism is the least affected, because it transfers variable-sized messages rather than cache lines, so only useful information is transmitted. Only the few non-stream accesses are impacted by line size.

Varying Network Performance

Finally, we show the effect of degrading the network performance relative to the processor cycle in Figure 3. In the current simulation model, both bandwidth and latency are affected by the network cycle time. For example, a 40-ns cycle time means that one data word can enter the network every four processor cycles, and the latency across each node is also four processor cycles.

The invalidate, prefetch, and deliver mechanisms are more sensitive to this system parameter than the other mechanisms. The synchronization variable scheme requires more network transactions than the more integrated lock and StreamLine mechanisms.

StreamLine performs better than lock as network performance degrades. The lock mechanism requires multiple round trips through the network (request and response) to retrieve data, while StreamLine only requires one network transaction (send).

Discussion

Because of small message size and load imbalance, the mechanisms which support efficient producer-consumer synchronization perform best for this implementation of sparse Cholesky factorization. Locks and streams both offer good performance, but StreamLine is less sensitive to cache line size and network performance.

We have also shown StreamLine to be more sensitive to private cache misses, because of the increased instruction overhead required to manipulate the streams. Because this is only a statistical model, however, we cannot be sure how often instruction misses will actually occur in the those critical areas of the code.

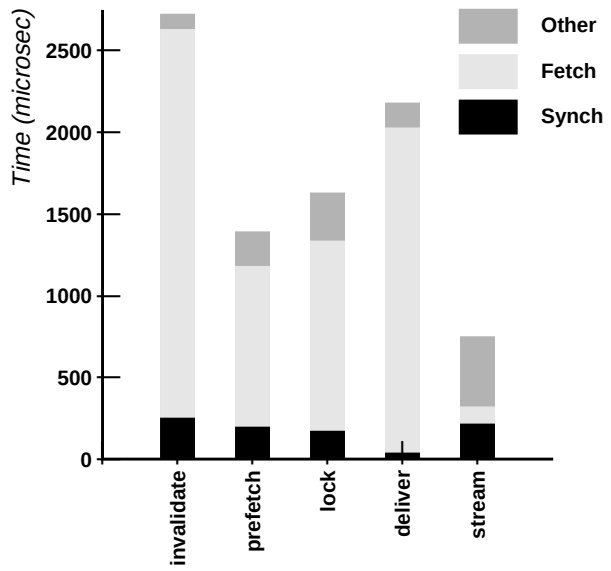

Figure 4: Gaussian elimination, 64×64.

4.2 Gaussian Elimination

The second kernel benchmark is LU factorization of a dense matrix using Gaussian elimination with partial pivoting. This is a column-oriented algorithm: each column reads the pivot and scaled data from the columns to its left, performs its own pivot and scaling, then communicates that result with the columns to the right. Unlike the sparse matrix application above, every column must communicate with every other column. The message size is much larger on average, since all data below the diagonal of the pivot column must be seen by every column to the right.

In the shared memory versions, the scaled column is read directly from the column-based matrix data structure. The pivot index is written to a shared vector; writing a legal value for the pivot signals the columns to the right that the scaled data is ready for consumption. Separate vectors are used to signal the next pivot column first.

For the StreamLine version, the pivot column sends a message containing the pivot and the data to each column to the right, and then writes the computed data into the shared matrix data structures. The matrix data below the diagonal is stored and updated directly on the stream, so that when the final calculation is done, the data is ready to be transmitted. After the data is transmitted, it is copied into the shared matrix.

Basic Performance

The performance of Gaussian elimination for a 64×64 dense matrix (one column per processor) is shown in Figure 4. As before, we show the breakdown of times along the program's critical path. (No write time statistics were taken for these runs.)

In all cases except StreamLine, the majority of time is spent fetching data from the other columns. Because this is a dense matrix, prefetch is able to appreciably reduce the effective fetch time—multiple lines are often required and most or all of the data in the line is actually used. There is a small overhead with issuing the prefetches, shown as an increase in "other," relative to invalidate.

The lock mechanism is only used to control access to the variable which communicates the pivot row for each column during the factorization. It does not try to combine transfer of the column data with the synchronization, as was done with the Cholesky code. Nevertheless, fetch time is greatly reduced. This

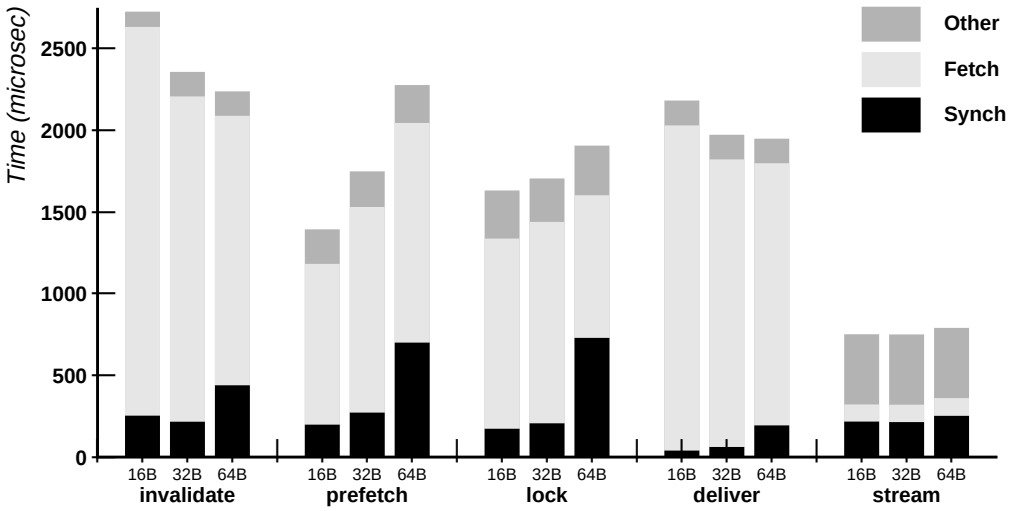

Figure 5: Gaussian elimination, 64×64—Effects of cache line size.

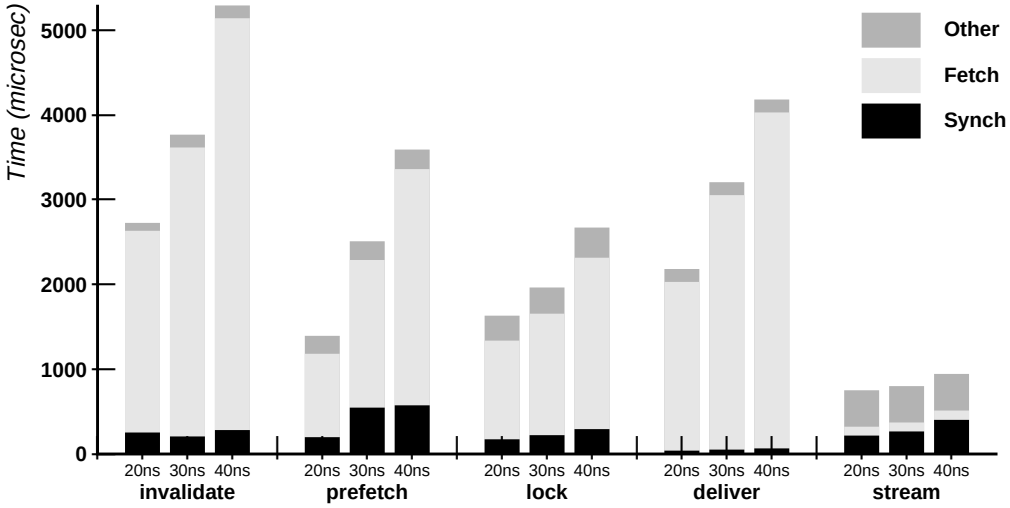

Figure 6: Gaussian elimination, 64×64—Effects of network cycle time.

suggests that a significant part of the cost associates with fetching in the invalidate case is due to contention at memories and caches; the synchronization method results in many columns trying to read the pivot column at roughly the same time. The lock mechanism, on the other hand, serializes the synchronization through lock and unlock operations, with a resulting serialization of access to the pivot column.

The deliver mechanism succeeds in reducing fetch time, but not as much as the prefetch and lock mechanisms. The approach taken with this mechanism was to just deliver the column to the next pivot column; this resulted in better performance than delivering to all of the consumers. The critical path for deliver is almost entirely in the rightmost column, which explains the small synchronization time—when the last column is finished with one pivot, the next pivot is ready to be consumed.

In the StreamLine case, fetch time is very small because fetches are satisfied locally. Synchronization time represents a larger fraction of the execution time. The non-pivot columns are able to quickly consume the pivot message, so the delay in sending the next message becomes more critical. The increase in the "other" category represents the overhead of stream manipulation and the fact that more of the pivot calculation is along the critical path.

Varying Cache Line Size

In contrast to the Cholesky case, increasing the cache line size improves performance for invalidate and deliver, as shown in Figure 5. Because of the larger message sizes, these mechanisms benefit from fetching a larger amount of data at once. Synchronization time increases slightly but is still a small part of overall performance.

Prefetch, on the other hand, has already substantially decreased fetch time and does not benefit from larger cache lines. Instead, the increased cost of synchronization decreases performance dramatically.

The lock mechanism shows an interesting change with the 64-byte cache line—an increase in synchronization time is almost completely offset by a decrease in fetch time. This represents a shift in the critical path caused by an increase in synchronization overhead. Some fetches that were previously on the critical path are now overlapped with synchronization.

As before, StreamLine shows very little change, since neither synchronization nor data transfer depend on the cache line as a unit of transfer.

Varying Network Performance

Figure 6 shows the effects of degrading network performance for this application. As with sparse Cholesky factorization, the memory-oriented mechanisms degrade much more severely than StreamLine. The memory mechanisms require multiple network transactions to transfer data between producer and consumer, while StreamLine requires only one. As a result, increasing the network cycle time by a factor of two only increases StreamLine execution time by 26%, compared to 94% for invalidate and 158% for prefetch.

Discussion

The StreamLine mechanism is shown to perform much better for this kernel than the memory-based alternatives, over a range of system parameters. This is consistent with earlier studies [14] which show that a row-based Gaussian elimination algorithm performs better when written in a message-passing style. The improvement shown by the lock mechanism in reducing memory contention also illustrates the benefits of imposing some order on data transfer. We plan to experiment with more message-oriented Gaussian elimination algorithms for the non-StreamLine mechanisms to try to improve their performance.

5 Synthetic Benchmark

In order to have more direct control over computation and communication parameters, we have developed a synthetic message-passing benchmark. The benchmark program is expressed as a graph, as shown in Figure 7, in which nodes represent serial work and weighted edges represent messages. The weights on the edges indicate the size of the message being sent. Work nodes are organized into *ranks*. A rank consists of a number of nodes equal to the number of processors in the system. Communication only happens between ranks, never between nodes within a given rank.

By adjusting the amount of work, message size, and communication patterns, this general framework can be used to model specific message-passing algorithms, especially those with regular communication patterns, such as explicit PDE solvers. For this study, we consider two randomly generated graphs, whose characteristics are shown in Table 2. The $G1$ benchmark uses medium-sized messages (between 80 and 160 bytes), with a small degree of fanout; the $G2$ benchmark, uses small messages (20 bytes), but with a higher degree of fanout. In the first case, we would expect data transmission latency to be most important; in the second, synchronization will play a more significant role.

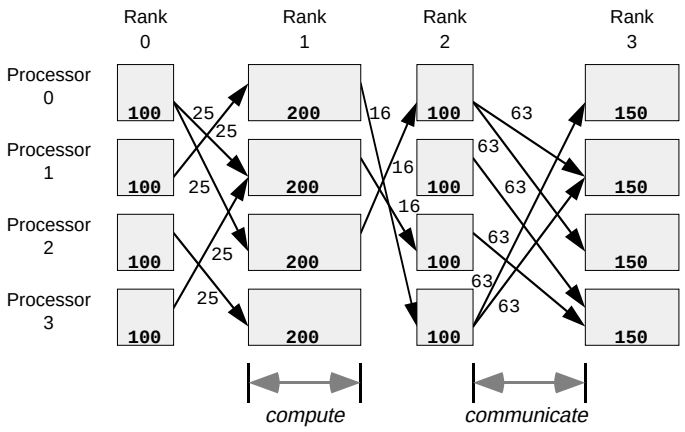

Figure 7: The synthetic benchmark.

The locality of the message transmissions are determined by the clustering shown in table 2. The cluster notation

$$\{(h_1, p_1), (h_2, p_2), \ldots, (h_n, p_n)\}$$

means that there are $n+1$ "neighborhoods" defined for each node. A message destination will be within h_1 hops of the origin with probability p_1, within h_2 hops with probability $p_1 + p_2$, and so on. The last cluster contains everything further than h_n hops.

In generating the random graphs, we wanted to keep the execution graph reasonably balanced, in order to avoid pathological critical paths, such as a series of extremely large messages or computationally-intensive nodes. For this reason, we constrained our graphs with the following restrictions:

1. All nodes in a given rank have the same amount of serial work. This work time is chosen from a user-supplied distribution for each rank.
2. All messages leaving a given rank are the same size, again chosen by a user-supplied distribution.
3. Fanouts and message destinations are randomly determined, but are subject to a maximum per-node fanin limit.
4. Messages will not be sent out of rank order. In other words, processor A will not send a message from rank n to processor B until B has received all of its messages from rank $n-1$. A shared variable approach is used for this synchronization for all mechanisms, even StreamLine.

In any message-passing algorithm, a decision must be made concerning how to deal with out-of-order messages. By not allowing them in this case, we ensure that roughly the same execution path is followed for every mechanism, and that only the effective communications performance is being measured. If some critical paths included the overhead of dealing with out-of-order messages and others didn't, the comparison would be less than fair.

	$G1$	$G2$
Work (cycles)	Exponential: $\mu = 100$	Exponential: $\mu = 100$
Msg. Size (words)	Uniform: $[20, 40]$	Fixed: 5
Fanout	$p(1) = 0.8, p(2) = 0.2$	$p(3) = 0.5, p(4) = 0.5$
Clusters	$\{(1, 0.5), (3, 0.3)\}$	$\{(1, 0.5), (3, 0.3)\}$
Max. Fanin	2	5
Ranks	10	10

Table 2: Description of synthetic benchmarks.

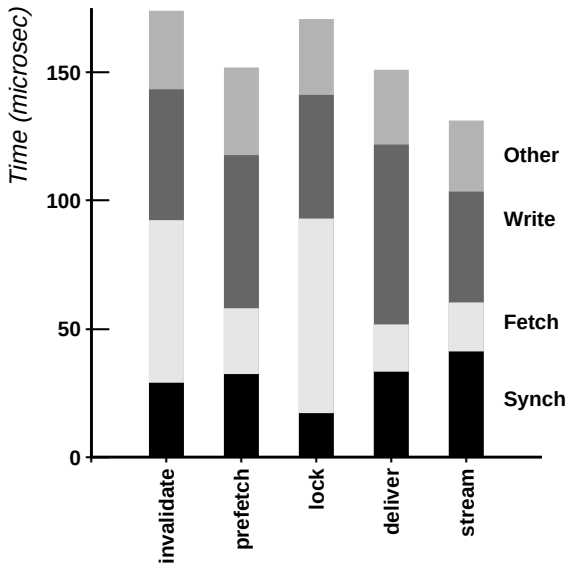

Figure 8: Synthetic benchmark $G1$.

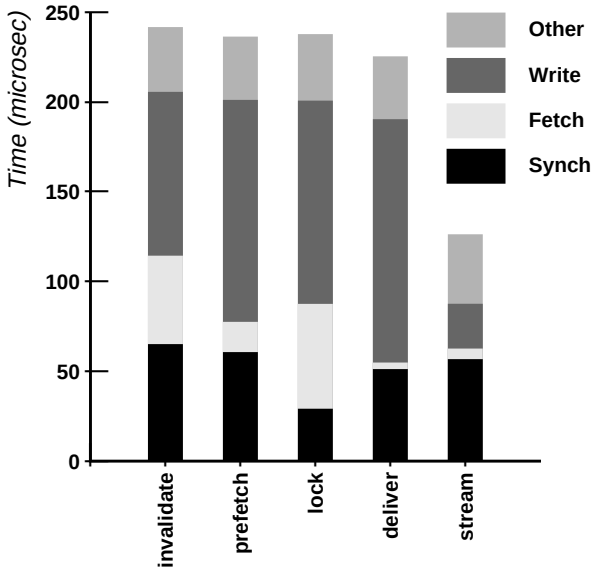

Figure 9: Synthetic benchmark $G2$.

Basic Performance

The results reported in this section are for a 16-processor 2-D mesh system, rather than the 64-processor system in the sections above. All other parameters are the same. Performance of the two synthetic benchmarks is shown in Figures 8 and 9.

The StreamLine mechanism performs well, relative to the others. StreamLine spends more time in synchronization along its critical path, because the other components are less costly. (For example, if fetches are faster, then the processor spends more time waiting for the next message, so synchronization time increases.) The shared memory enhancements work to improve some aspects of the execution, but other aspects degrade: prefetch and deliver reduce fetch time, but write time increases; lock reduces synchronization time, but fetch time increases.

Unlike the previous two benchmarks, nothing is known about communication patterns ahead of time, so there is little opportunity to fetch or write ahead of when the data is really needed. This is especially true in the $G2$ benchmark, which features high fanout/fanin and small messages. This combination stresses the ability of the mechanisms to deal with message queues, and the advantage of StreamLine is more evident.

Varying Cache Line Size

Figures 10 and 11 show the effects of increasing the second-level cache line size. For $G1$, the larger line size decreases fetch time for invalidate and lock, but increasing synchronization and writing costs degrade overall performance.

The synchronization cost is especially noticeable in the $G2$ benchmark, which uses smaller messages. Almost all of the degradation of StreamLine is due to the synchronization variable used to prevent out-of-order messages, discussed above.

Varying Network Performance

The performance impact of a slower network, shown in Figures 12 and 13, is consistent with the earlier benchmarks. StreamLine shows less degradation than the other mechanisms, due to its efficient use of the

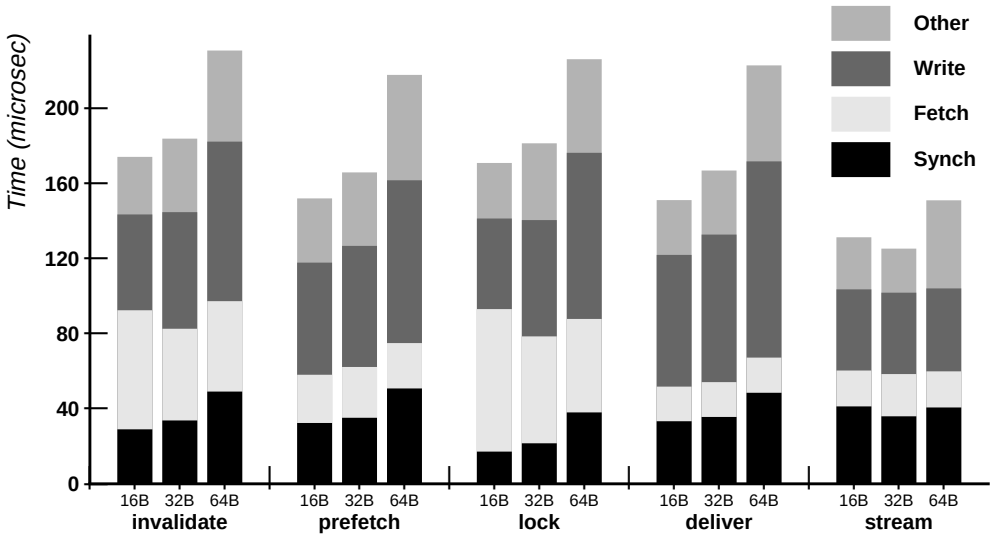

Figure 10: Synthetic benchmark $G1$—Effects of cache line size.

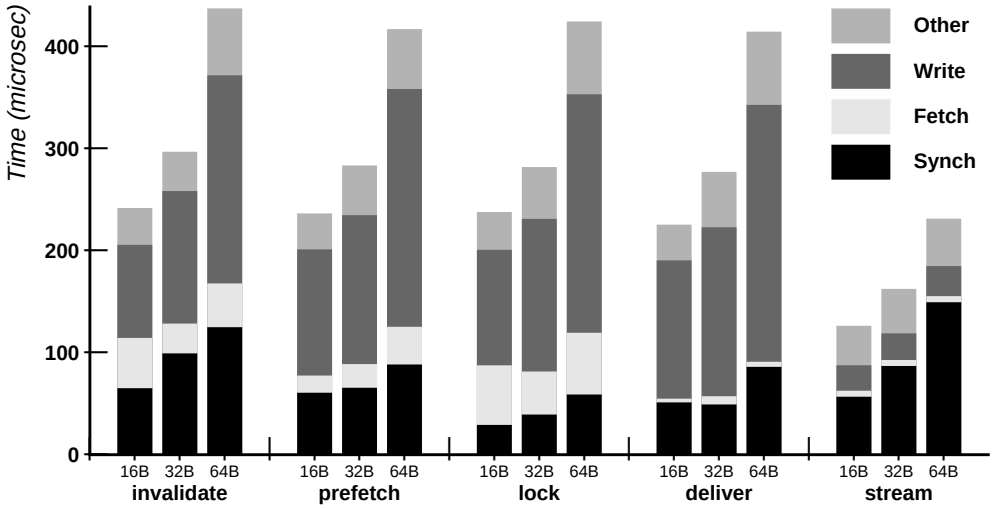

Figure 11: Synthetic benchmark $G2$—Effects of cache line size.

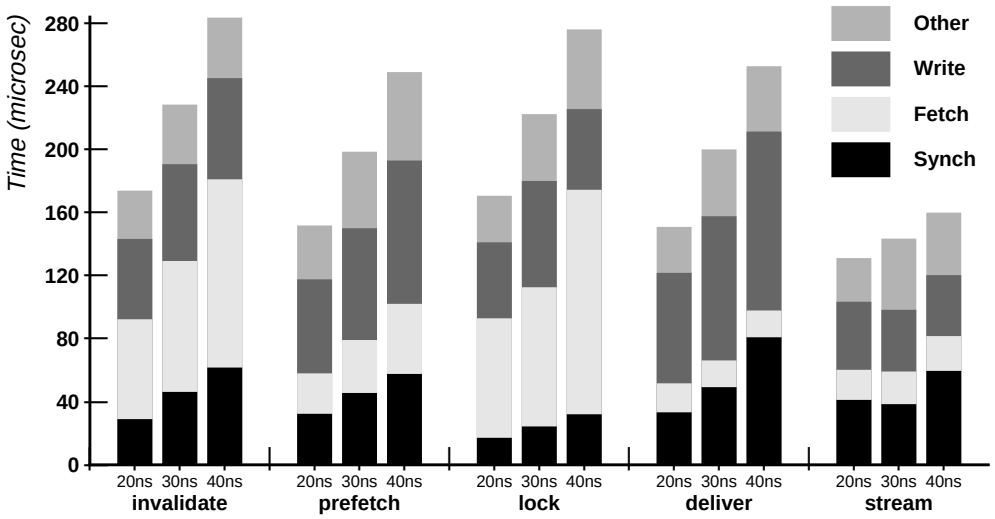

Figure 12: Synthetic benchmark $G1$—Effects of network cycle time.

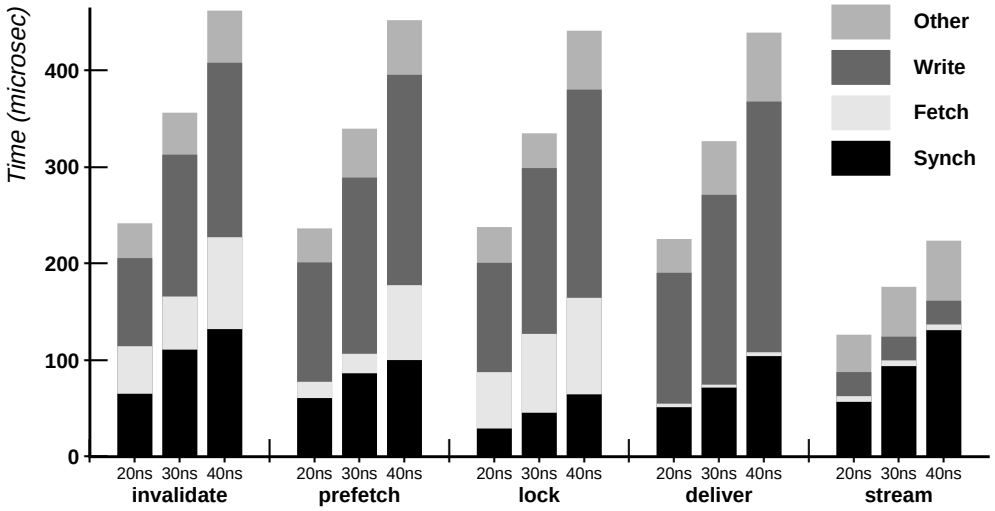

Figure 13: Synthetic benchmark $G2$—Effects of network cycle time.

network. Again, the increase in synchronization time for $G2$ reflects the cost of the synchronization variable. The other mechanisms also show this increase in synchronization time, as well as degrading performance in other aspects, such as fetch or write time.

Discussion

In a synthetic benchmark designed to emulate a message passing style of producer-consumer communication, one would expect the StreamLine mechanism, designed to support messages, to perform best, and indeed it does. The interesting question is whether any of the other mechanisms, which may be easier to implement, approach the performance of StreamLine.

For the default set of system parameters, prefetch and deliver get within 15% or so of StreamLine performance for the $G1$ benchmark, with moderate-sized messages and small fanout. As the design point changes, however, the memory-based mechanisms degrade due to their greater dependence on cache and network behavior. For the $G2$ benchmark, which emphasizes fine-grained messages, none of the enhancements to invalidate-based cache coherence are able to improve performance, while StreamLine cuts execution time in half.

6 Future Work

While the results presented thus far illustrate the potential benefits of providing direct support for messages in shared memory systems, there are several areas that warrant further exploration:

- In every performance study above, we pointed out the cost of synchronization variables, especially in the case of *event synchronization*, in which a producer wishes to signal some event (such as the availability of data) to a group of consumers. An invalidate-based scheme is clearly not optimal, since all the consumer copies must first be invalidated and then all the consumers must re-fetch exactly the same data.

  A more efficient scheme would be to allow a limited update facility, in which the producer can send the new value to all the consumers, rather than just telling them that something has changed. This sort of facility is provided by DASH [16], and we are interested in understanding the impact this would have on the results presented here.

  Future study will involve a full update-based protocol, as proposed by Glasco [10]. The Galactica Net architecture [25] proposes a hybrid protocol, with update-based coherence selectable at a page level.

- One of the potential problems with sending data to a consumer's cache before it is requested is *cache pollution*, i.e., the new data pushes older but still useful data out of the cache. Since the results presented here all assumed infinite second-level cache, this was not a consideration. Some preliminary experiments with finite caches have not shown cache pollution to be a significant problem, but we plan to do more investigation of this area.

- The synchronization mechanisms studied here have been either simple shared variables or line-based synchronization primitives. We have not yet considered word-level synchronization primitives, such as full/empty bits [22]. Such mechanisms allow early synchronization between producers and consumers and may allow more overlap between writing and reading data.

7 Conclusions

We introduced this paper by claiming that data communication is an important factor in the performance of many parallel applications. In cache-based shared memory systems, communication is optimized through

	Sparse Cholesky	Gaussian elimination	Synthetic	
			$G1$	$G2$
invalidate	1.00	1.00	1.00	1.00
prefetch	0.95	0.51	0.87	0.98
lock	0.56	0.60	0.98	0.98
deliver	0.94	0.80	0.87	0.93
StreamLine	0.54	0.28	0.75	0.52

Table 3: Execution time with default system parameters, relative to invalidate.

the management of data locality. Other researchers have shown that message passing algorithms, which explicitly manage locality, sometimes perform better than shared memory algorithms, even on shared memory multiprocessors [14, 15, 17, 19]. LeBlanc and Markatos [15] claim that, except for load balancing considerations, message passing models always perform better than shared memory models on modern multiprocessor architectures because of the explicit attention paid to data locality.

We argue that further increases in performance can be achieved providing a user-level interface to the network, an argument that is also reflected in the development of the Alewife machine [13]. Table 3 shows a comparison of execution times, relative to the base invalidate implementation, for all of the benchmarks presented above. The direct network interface, represented by StreamLine, achieves a 30 to 70% reduction in execution time. The performance increase is substantial in every case, and is not equaled by any other mechanism.

We understand that the benchmarks used in this study are small, compared to real-world applications. In order to study differences in communications mechanisms, we chose benchmarks in which improvements in communication latency could clearly be seen. Applications which can completely hide or eliminate communication will perform well on any parallel system; a system which provides efficient fine-grained communication, however, will be useful for a wider range of applications.

Compared to other proposed messaging mechanisms [12, 13], StreamLine is more tightly integrated with the memory system. Communication buffers are implemented in main memory; stream pages can be added or removed according to dynamic system requirements. Each thread can have its own communication buffer, so that contents of the message need not be saved if one thread is interrupted or time-sliced with another. By integrating the transmission mechanism in the cache, StreamLine avoids the local coherence problem that occurs with DMA-based mechanisms, such as in Alewife [13]. Issues involved in implementing StreamLine will be discussed in a future paper.

Independent of the implementation details, the two architectural trends of larger cache lines and faster processor clocks should make a messaging interface more and more attractive. The 16-byte cache line assumed by this study is small compared to modern workstations, which rely on good cache performance to overcome the increasing disparity between processor and memory speeds. Also, processor clock rates are increasing faster than off-chip communication rates, so we are likely to see networks get slower in terms of processor clocks. Both of these conditions favor the use of proactive, efficient data transfers between processors.

These trends, along with an increasing recognition that no one programming style is best for all programs or programmers, should lead to more systems which truly integrate fine-grained messages and shared memory.

Acknowledgements

The authors would like to acknowledge the continuing support of MCNC, especially the management of the Information Technologies Division. Preliminary work was supported by an NSF Graduate Fellowship and a NASA/DARPA Assistantship in Parallel Processing.

References

[1] Argonne National Laboratory. Alog. Developed at ANL and available through anonymous ftp from `anagram.mcs.anl.gov` in the `pub/upshot` directory.

[2] G. T. Byrd and B. A. Delagi. StreamLine: cache-based message passing in scalable multiprocessors. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume I, pages 251–254, Aug. 1991.

[3] L. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Trans. Comput.*, C-27(12):1112–1118, Dec. 1978.

[4] W. J. Dally, J. S. Keen, and M. D. Noakes. The J-Machine architecture and evaluation. In *Compcon Spring '93*, pages 183–188, Feb. 1993.

[5] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Comput.*, C-36(5):547–553, May 1987.

[6] B. A. Delagi, N. Saraiya, S. Nishimura, and G. Byrd. Instrumented architectural simulation. In *Proceedings of the Third International Conference on Supercomputing*, volume 1, pages 8–11, May 1988.

[7] I. Duff, R. Grimes, and J. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software*, 15(1):1–14, 1989.

[8] S. Frank, H. Burkhardt III, and J. Rothnie. The KSR1: Bridging the gap between shared memory and MPPs. In *Compcon Spring '93*, pages 285–294, Feb. 1993.

[9] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[10] D. B. Glasco, B. A. Delagi, and M. J. Flynn. Update-based cache coherence protocols for scalable shared-memory multiprocessors. Technical Report CSL-TR-93-588, Computer Systems Laboratory, Stanford University, Nov. 1993.

[11] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS III)*, pages 64–75, Apr. 1989.

[12] R. E. Kessler and J. L. Schwarzmeier. CRAY T3D: A new direction for Cray Research. In *Compcon Spring '93*, pages 176–182, Feb. 1993.

[13] D. Kranz et al. Integrating message-passing and shared-memory: Early experience. In *Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '93)*, 1993.

[14] T. J. LeBlanc. Problem decomposition and communication tradeoffs in a shared-memory multiprocessor. In M. Schultz, editor, *Numerical Algorithms for Modern Parallel Computer Architectures*, pages 145–162. Springer-Verlag, New York, 1988.

[15] T. J. LeBlanc and E. P. Markatos. Shared memory vs. message passing in shared-memory multiprocessors. In *Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing*, pages 254–263, 1992.

[16] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH multiprocessor. *Computer*, 25(3):63–79, Mar. 1992.

[17] C. Lin and L. Snyder. A comparison of programming models for shared memory multiprocessors. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages 163–170, Aug. 1990.

[18] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12:87–106, 1991.

[19] T. Ngo and L. Syder. On the influence of programming models on shared memory computer performance. In *Scalable High Performance Computing Conference (SHPCC '92)*, Apr. 1992.

[20] E. Rothberg and A. Gupta. A comparative evaluation of nodal and supernodal parallel sparse matrix factorization: Detailed simulation results. Technical Report STAN-CS-90-1305, Department of Computer Science, Stanford University, Feb. 1990.

[21] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, Sept. 1982.

[22] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Real-Time Signal Processing IV*, pages 241–247, May 1981.

[23] M. Stumm, Z. Vranesic, R. White, R. Unrau, and K. Farkas. Experiences with the Hector multiprocessor. In *Proceedings of the Parallel Systems Fair*, pages 10–17, Apr. 1993. Held in conjunction with the 7th International Parallel Processing Symposium.

[24] Z. Vranesic, M. Stumm, D. Lewis, and R. White. Hector—a hierarchically structured shared memory multiprocessor. *Computer*, 24(1):72–80, Jan. 1991.

[25] A. W. Wilson, Jr. and R. P. LaRowe, Jr. Hiding shared memory reference latency on the Galactica Net distributed shared memory architecture. *Journal of Parallel and Distributed Computing*, 15:351–367, 1992.