MODELS OF COMMUNICATION LATENCY IN SHARED MEMORY MULTIPROCESSORS

Gregory T. Byrd

Technical Report CSL-TR-93-596

December 1993

This material is based on work supported by an NSF Graduate Fellowship, a DARPA/NASA Assistantship in Parallel Processing, and MCNC.

Models of Communication Latency in Shared Memory Multiprocessors

by
Gregory T. Byrd

Technical Report CSL-TR-93-596
December 1993

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

We evaluate various mechanisms for data communication in large-scale shared memory multiprocessors. Data communication involves both data transmission and synchronization, resulting in the transfer of data between computational threads. We use simple analytical models to evaluate the communication latency for each of the mechanisms.

The models show that efficient and opportunistic synchronization is the most important determinant of latency, followed by efficient transmission. Producer-initiated mechanisms, in which data is sent by its producer as it is produced, generally achieve lower latencies than consumer-initiated mechanisms, in which data is retrieved as and when it is needed.

Contents

1 Introduction **1**

2 Characterizing Transfer Mechanisms **1**

2.1 Transmission Granularity 1

2.2 Synchronization Granularity 2

2.3 Transfer Initiation 2

2.4 Load Concurrency 2

2.5 Store Concurrency 2

2.6 Eager Transmission 3

3 Communication Mechanisms **3**

3.1 Word-Oriented 4

3.2 Line-Oriented 4

3.3 Block-Oriented 5

4 Evaluating the Mechanisms **5**

4.1 Assumptions and Parameters 5

4.2 Word-Oriented Communication 6

4.3 Line-Oriented Communication 8

4.4 Block-Oriented Communication 17

4.5 Overall Comparison 19

4.6 Summary 21

List of Figures

1 Word comm, block synch: Latency *vs.* message size. 7
2 Word comm, word synch: Latency *vs.* message size. 8
3 Line comm, block synch: Latency *vs.* message size. 9
4 Line comm, block synch: Latency *vs.* message size, 32-word cache line. 10
5 Line comm, block synch: Latency *vs.* cache line size, 8-word message. 11
6 Line comm, block synch: Latency *vs.* cache line size, 32-word message. 12
7 Line comm, word synch: Latency *vs.* message size. 13
8 Line comm, word synch: Latency *vs.* cache line size, 8-word message. 14
9 Line comm, word synch: Latency *vs.* cache line size, 32-word message. 15
10 Line comm, line synch: Latency *vs.* message size. 16
11 Block comm, block synch: Latency *vs.* message size. 17
12 Block comm, word synch: Latency *vs.* message size. 18
13 Block synchronization: Latency *vs.* message size. 19
14 Word synchronization: Latency *vs.* message size. 20

List of Tables

1 Summary of mechanisms. 3

2 Model parameters and defaults. 6

1 Introduction

In this paper, we examine several mechanisms for data communication in shared memory multiprocessors. Our goal is to select mechanisms which reduce the communication latency between two computational threads. We wish to reduce this latency, since communication is often on the critical path of an application. Other approaches, such as multithreaded processors [22], offer mechanisms to hide latency—this will increase the efficiency of the multiprocessor system but may not decrease the time required to complete a particular application.

Dally and Wills [9] cite three universal mechanisms which are required to support parallel computation: naming, communication, and synchronization. In this paper, we will only consider shared memory multiprocessor systems, i.e., systems which support a global address space. Therefore, the naming mechanism is fixed. The other two mechanisms, communication and synchronization, both affect data transfer latency and will therefore be the focus of this study.

In the following section, we identify the parameters which characterize all data transfer mechanisms. These include transmission and synchronization granularity, transfer initiation, load and store concurrency, and eager transmission.

We introduce several "canonical" mechanisms which represent tradeoffs along these dimensions. We then develop simple analytical models of data transfer latency for each mechanism. Based on the models, we will choose mechanisms to investigate further through simulation.

2 Characterizing Transfer Mechanisms

In shared memory multiprocessors, the primary (and sometimes only) mechanism for communication is load/store. A producer of data stores into some memory address, and the consumer of the data loads from that address at some later time. This mechanism, coupled with synchronization, is sufficient for data communication, but the latency associated with data transfer depends on the details of the implementation. The use of caches and the associated cache coherence protocols have a particularly significant impact.

In this section, we discuss seven parameters which characterize a data communication mechanism. These parameters affect transmission and synchronization efficiency, the overlapping of successive data transfers, and the aggressiveness with which data is transmitted as it is produced. We believe that this is a complete list of parameters that affect communication latency.

2.1 Transmission Granularity

Transmission granularity describes the amount of data that is transferred in a single network transmission. We consider three possibilities: *word*, *line*, and *block*.

Word granularity occurs when each data transmission contains a single machine word (32 or 64 bits) of data. Line granularity refers to a cache line as the unit of transfer—cache line size is determined by the architecture or its implementation and is highly variable between machines. Finally, block granularity means that the entire block of data to be transferred is handled in a single network transmission.

Increasing granularity generally implies increased communication efficiency, at least in terms of latency. Each network transmission requires some amount of header information. Increasing the amount of data associated with the transmission decreases the relative cost of transmitting the header.

On the other hand, smaller transmission packets generally lead to better network throughput. The models described in this paper do not account for network contention or throughput, so the results will be biased toward mechanisms with larger transmission granularity.

2.2 Synchronization Granularity

Data transfer between computational threads generally requires synchronization—the producer thread must somehow signal the availability of the data to the consumer thread. Synchronization granularity determines how much data is associated with each synchronization operation. As with communication, we consider three granularities: *word*, *line*, and *block*.

With word synchronization, the presence of data is signaled for each machine word. This generally implies some hardware support, such as full/empty bits [20], since implementing a software lock per word would almost always be too expensive in both time and space.[1] Line synchronization augments a system's cache coherence protocol to implement a lock per cache line [10]. Block synchronization is the simplest scheme, where a single lock signals the presence or absence of the entire block of data to be transferred. This can be implemented as a simple spin lock, with no special hardware support.

2.3 Transfer Initiation

Communications can be described as either *producer-initiated* or *consumer-initiated*, depending on whether the data is sent by the producer or fetched by the consumer.

In general, producer-initiated transfers should have lower latency, since they do not incur the network delay associated with asking for the data. Also, the data can be transmitted before it is actually needed by the consumer, which effectively hides the latency from the overall computation.

On the other hand, consumer-initiated transfers are more effective for cases in which the consumer or the data to be transferred are not known in advance; the consumer requests data as and when it is needed.

2.4 Load Concurrency

The *load concurrency* property determines whether an architecture allows for multiple load operations to be outstanding for a single processor. If there is no load concurrency, then the processor stalls after every load until the required data is placed in its register. (Even a load-concurrent system would stall when a register is read, if the data has not yet been loaded.)

Most microprocessors have a limited amount of load concurrency, allowing loads to the local cache to be pipelined. More aggressive examples include non-binding prefetch [18] and the "streaming" mechanism in the WM architecture [23]. We would consider a vector load operation in a vector architecture [13] to be an example of load concurrency, even though there might be only one outstanding load, since data is retrieved in a pipelined manner (i.e., we cannot distinguish this, from a latency standpoint, from pipelined load operations).

2.5 Store Concurrency

Analogous to load concurrency, *store concurrency* determines whether the architecture allows multiple outstanding store operations from a single processor.

If multiple outstanding stores are allowed, then stores can complete in an order different than the order in which they were issued from the processor. This violates the model of *sequential consistency* [15], which requires that stores are observed in the same order by all processors, and that stores are performed in the order prescribed by the sequential program running on a each processor. A strict adherence to sequential consistency would allow no store concurrency.

[1] Another approach would be to used a reserved value to denote an empty memory location.

Mechanism	*Trans. Gran.*	*Synch. Gran.*	*Initiation*	*Load Conc.*	*Store Conc.*	*Eager*	*Ref.*
remote	word	block/word	consumer	no	yes	no	[11]
remote-pipelined	word	block/word	consumer	yes	yes	no	[23]
reader-local	word	block/word	producer	yes	yes	yes	[21]
update	word	block/word	producer	no	yes	yes	[1]
invalidate	line	block/line/word	consumer	no	yes	no	[1, 2]
invalidate-prefetch	line	block/line/word	consumer	yes	yes	no	[18]
deliver	line	block/line/word	producer	yes	yes	yes	[17]
reader-copy	block	block	consumer	no	yes	no	[16, 19]
writer-copy	block	block/word	producer	no	yes	no	[16, 19]
message	block	word	producer	no	yes	no	[5]
message-eager	block	word	producer	no	yes	yes	[6, 4]

Table 1: Summary of mechanisms.

Weaker models of consistency have been proposed, however, which do allow store concurrency [2, 12]. These models recognize that a series of writes may proceed in any order, if they do not affect the global view of the computation, as long as they all complete before some global event, such as a synchronization point. A *fence* operation [19] is provided, which stalls the processor until all outstanding stores have completed (i.e., have been acknowledged). Fence operations are inserted by the compiler or programmer as needed to insure correct execution of the program.

In the context of this paper, which is concerned with the latency of data transfer, a weak ordering of stores is much preferred over a model in which each store must wait on the previous one to complete. For this reason, all the models considered here will allow store concurrency. (The reader should be aware, however, that some commercial systems still enforce sequential consistency in hardware.)

2.6 Eager Transmission

The final property of data transfer mechanisms is *eagerness* of transmission—that is, whether transmission of the data occurs before all of the data is produced. This is closely associated with transmission and synchronization granularity, but is a separate issue. The distinction is especially important for block-oriented communication mechanisms.

We refer to a mechanism as *eager* if transmission is started before the entire block of data is written. Otherwise, the mechanism is called *non-eager.*[2]

3 Communication Mechanisms

Table 1 shows a collection of mechanisms which explores the space of parameters described in the previous section. Each table entry also gives at least one reference to a proposed or existing system which implements the model. The table groups models together according to transmission granularity. Also, multiple synchronization granularities may be associated with a model.[3]

[2] The term "lazy" is often used in contrast to "eager." Lazy, however, usually implies that an operation is delayed as long as possible. This is not what is implied by non-eager in this case.

[3] This was done in lieu of giving a different name to each combination of transmission and synchronization granularity.

3.1 Word-Oriented

We first consider *word-oriented* mechanisms, in which data is communicated one word at a time. These occur most often in systems which do not use local cache for transferring shared data.

The **remote** mechanism assumes that all data transfer takes place through global memory, and that the memory is remote from both the producer and the consumer. This corresponds to the case where all processors are equally distant from memory (e.g., the NYU Ultracomputer [11]), or the case in which the memory location was allocated in a random memory module, unrelated to the location of either producer or consumer.

The **remote-pipelined** mechanism is an optimization of **remote** in which we allow load concurrency through pipelined load operations.

If the location of the consumer thread is known, we can allocate the shared memory at the memory module closest to the consumer. Though this is not a different mechanism from **remote** in the sense of requiring different hardware support, it represents different assumptions about the communication. We call this approach **reader-local**. Since the consumer load operations are local, we assume pipelined loads are available.

Finally, one cache-based mechanism should be classified as word-oriented. The **update** mechanism relies on an update-based cache coherence protocol [1]. Assuming that copies of the data exist in both the consumer's and producer's caches, then each data word written by the consumer is sent to the consumer's cache, where the cached copy is updated.

The first two mechanisms, **remote** and **remote-pipelined**, are consumer-initiated. Data is not delivered to the consumer until it is fetched. The other two mechanisms, **reader-local** and **update**, are producer-initiated. In these cases, the data is placed close to the consumer thread as it is written (and potentially before it is needed by the consumer).

3.2 Line-Oriented

Line-oriented communications mechanisms are ones which transmit data one cache line at a time. The mechanisms we consider are all based on caches which use an invalidate-based coherence protocol [1]. If a cache line is written which is present in more than one cache, the writer initiates a request that invalidates all copies in other caches. Therefore, when a thread on another processor reads the data, it will miss in its own cache and retrieve the recently-written data from the writer's cache.

We refer to communication implemented strictly with an invalid-based coherent cache system as the **invalidate** mechanism.

Since the consumer relies on the cache miss mechanism to retrieve data from the consumer, there can be a considerable delay between the time the data is requested and the time it is actually available to the processor. One way to hide this latency is to use *prefetch* [18], in which the data is requested before it is needed. An invalidate-based cache which employs prefetch is called the **invalidate-prefetch** mechanism.

A mechanism which is more directly oriented toward efficient producer-consumer communication is the **deliver** mechanism, as proposed in the DASH project [17]. With this mechanism, the producer may tell the cache to send a cache line to a particular cache, whether it has been requested or not. This should greatly reduce the latency seen by the consumer, since the data may already be in its cache when it starts to read the message.

The **deliver** mechanism is obviously producer-initiated, since the producer tells when and where to send the data (one line at a time). The **invalidate** and **invalidate-prefetch** mechanisms, on the other hand, are consumer- initiated, since data does not move until the consumer reads it.

3.3 Block-Oriented

Finally, we consider mechanisms in which the entire message is transmitted as a whole from producer to consumer. These mechanisms try to make efficient use of network resources, minimizing the overhead associated with addresses and other packet header information.

Two such mechanisms are based on a memory-to-memory copy operation, as in the BBN Butterfly system [3]. In the first mechanism, **reader-copy**, the reader (consumer) copies the message from the writer's memory into its own as soon as it is available, e.g., immediately after synchronization. In the second, **writer-copy**, the writer copies the data to the reader's memory before synchronization.

A third mechanism, which we will call **message**, hardware support for cache-to-cache delivery of messages is provided. An example of such a mechanism is StreamLine [5], in which special memory regions are managed by hardware as FIFO message queues, or streams. The producer writes data to a local stream, and then issues an instruction which transmits the entire message to a stream in the consumer's cache.

Finally, we consider the **message-eager** mechanism, which is **message** with eager transmission. In this model, data is written to the network as soon as it is produced, as in the J-Machine[7]. Intel's iWarp [4] architecture has a similar mechanism, in which logical channels can be set up between processes in advance.

The **message**, **message-eager**, and **writer-copy** mechanisms are producer-initiated; the **reader-copy** mechanism is consumer-initiated.

4 Evaluating the Mechanisms

Now that we have enumerated the various classes of communication and synchronization mechanisms, we wish to evaluate them with respect to their effectiveness in supporting low-latency data transfer. In this section, we develop latency models for various combinations of communication and synchronization under rather ideal circumstances. We do not claim that these models will accurately predict performance in a real multiprocessor system. Instead, we view the models as a way to quickly examine alternatives, so that we may eliminate mechanisms which are likely to result in poor performance and identify the most attractive mechanisms for further study.

4.1 Assumptions and Parameters

An N-word message is passed from process A to process B. A word is defined as 32 bits. Process A is charged for writing all N words, and process B is charged for reading all N words once. Latency is measured from the first write by A to the last read by B. This is not necessarily a direct measure of the communications cost experienced by A and B, since computation may sometimes occur simultaneously with communication—e.g., A may be computing while B is reading.

We assume a cut-through [8, 14] communications network, which is consistent with current research and production machines. The network has average distance $\bar{D}$, and we assume that *every* packet that is sent through the network travels $\bar{D}$ hops. The network is pipelined and is clocked at the same rate as the processor—a pipeline depth parameter, p, denotes the number of clock cycles per hop. Each network channel is W bits wide. We model no contention in the network.

A memory request or acknowledge contains a target (t bits) and zero or more (32-bit) words of data. The origin is not included, since some networks can mutate the target *en route* to obtain the origin.

We assume that the memory at each processing site is interleaved, so that, after an initial delay (M cycles), data with consecutive memory addresses can be accessed at a rate of one word per processor cycle. Concurrent fetches to local memory are allowed in every model. Store concurrency is also assumed in every model.

parameter	*default*	*description*
N	8	message size (words)
$\bar{D}$	10	average network distance (hops)
p	4	network pipeline depth
t	32	size of target address (bits)
W	32	network channel width (bits)
M	4	memory access time, in processor cycles
B	4	cache line size (words)
$\mathcal{T}(k)$	—	cycles to transmit k words $= \lceil \frac{t}{W} \rceil p\bar{D} + \lceil \frac{32k}{W} \rceil$
$\mathcal{S}(k)$	—	cycles to inject k words into the network $= \lceil \frac{t}{W} \rceil + \lceil \frac{32k}{W} \rceil$

Table 2: Model parameters and defaults.

In systems with cache, we do not charge for any directory lookup or modification operations—each cache line "knows" about all the other caches which contain copies of its data. The details of directory lookup vary considerably among directory-based protocols; by ignoring this overhead, the models reflect a protocol-independent lower bound on the latency of cache-based transfers. Further, we assume infinite, fully associative caches, and we assume that only the caches associated with processors A and B may contain the message data.

We also assume a *load-through* cache, in which a cache line is always returned in such a way that the word within the line that satisfies the miss will be returned first; the other words of the line follow in wrap-around fashion. The first word is presented to the processor immediately, so the processor may continue execution while the remaining words are being loaded into cache. This assumption is favorable toward synchronization latency, since the synchronization variable is accessed as quickly as possible.

Finally, we assume a *dual-ported* cache, in the sense that the cache allows simultaneous access (to different memory addresses) from the network and the processor. This would be an aggressive implementation strategy, probably requiring duplication or interleaving of cache memory, but it is feasible. This and many of the other assumptions are biased toward an optimistic model of performance; by looking at best-case performance, we believe that we can better distinguish among different models. Simulations and more implementation-driven models will be needed to accurately predict performance.

Table 2 shows the parameters used in the models, most of which have been described above. The equations for communications latency are also shown. The transit time, $\mathcal{T}(k)$, is the number of cycles required to transmit k words of data (plus a target address) through the network. Injection time, $\mathcal{S}(k)$, is the number of cycles used by the cache to place k words into the network; this specifies the rate at which the cache can transmit data.

4.2 Word-Oriented Communication

First, we consider the word-oriented communication mechanisms: **remote**, **remote-pipelined**, **reader-local**, and **update**. The latency models for these mechanisms, with block synchronization, are shown in figure 1. Also shown in the figure is the latency experience by a PRAM model, also using block synchronization. The non-varying parameters are set according to the defaults in table 2.

The **remote** mechanism is clearly the worst performer of this group. The latency associated with fetching each word from across the network, without the opportunity for overlap, results in a large overall latency. The **remote-pipelined** model demonstrates how merely allowing multiple loads can greatly reduce the effective latency.

The **reader-local** and **update** mechanism are nearly equivalent in this model. Since writes are overlapped, the main difference in the two models is that the reader process loads data from cache in the **update** case, rather than its local memory. Local memory is slower than cache, but the interleaved memory assumption

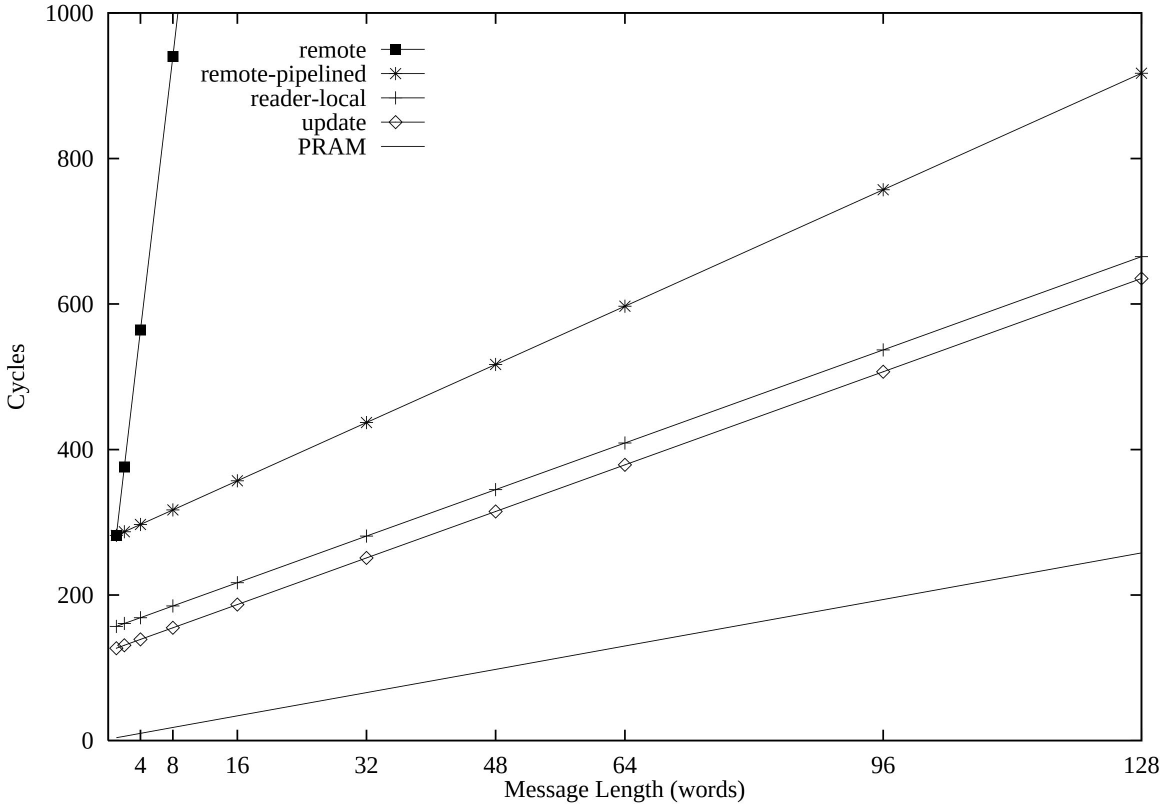

Figure 1: Word comm, block synch: Latency *vs.* message size.

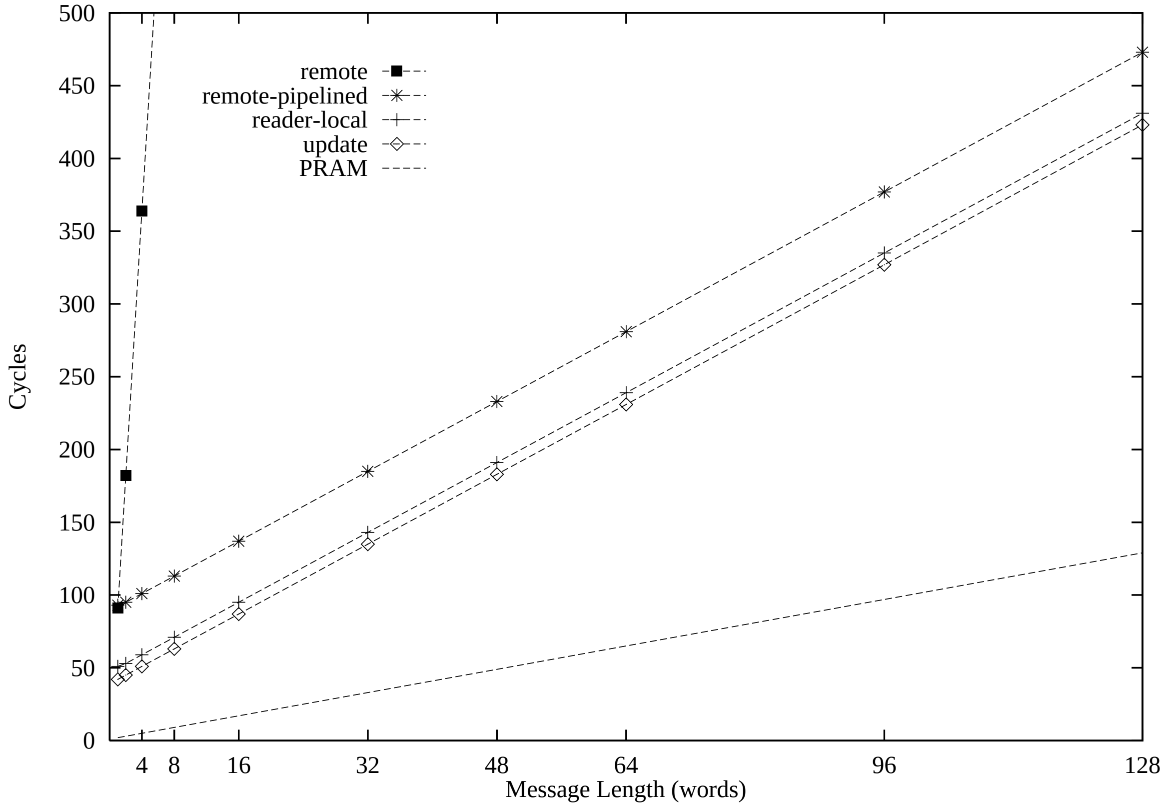

Figure 2: Word comm, word synch: Latency *vs.* message size.

means that data can be read at one work per cycle in both cases. The result is that there is little difference in performance.

Figure 2 shows the same mechanisms when word-oriented synchronization is used. All of the latencies are dramatically reduced, but their relative performance stays much the same. **Remote-pipelined** is not far behind the other two. In fact, because the memory bandwidth is the same for cache, local, and remote memories in the model, there is only a fixed offset due to the cost of accessing the first data word.

4.3 Line-Oriented Communication

The line-oriented communications mechanisms (**invalidate**, **invalidate-prefetch**, and **deliver**) are shown in figure 3, using the defaults from table 2.

The **invalidate** mechanism is the poorest performer. As the message size becomes larger than the cache line (4 words, in this case), then the network latency involved with fetching the message one line at a time dominates.

For small messages, **deliver** shows the lowest latency. When synchronization occurs, the message already resides in the reader's cache, so it does not have to be fetched. (The producer writes and transmits the data first, then waits for the invalidates caused by the writes to be acknowledged. When all acknowledges are received, the lock is written—by the time the consumer cache sees the invalidate from the synchronization write, all the deliver operations should have been completed.)

There is, however, some overhead associated with issuing the deliver command after each cache line is written.

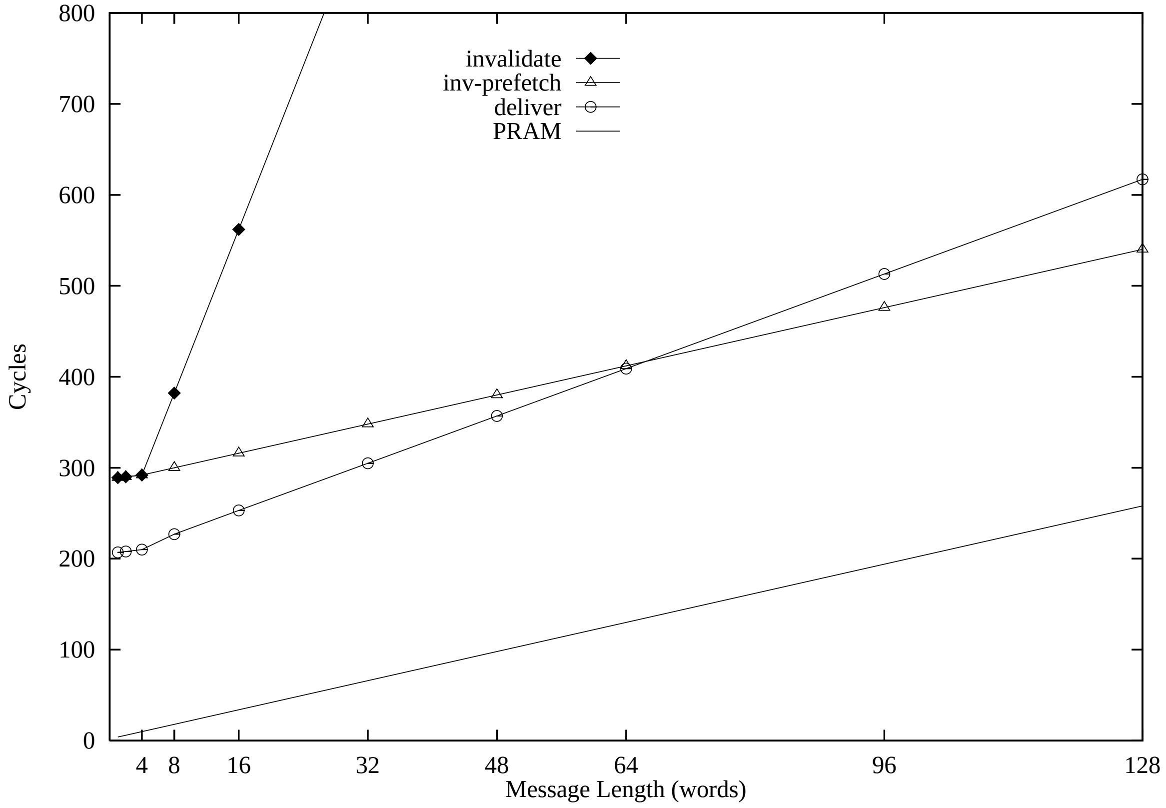

Figure 3: Line comm, block synch: Latency *vs.* message size.

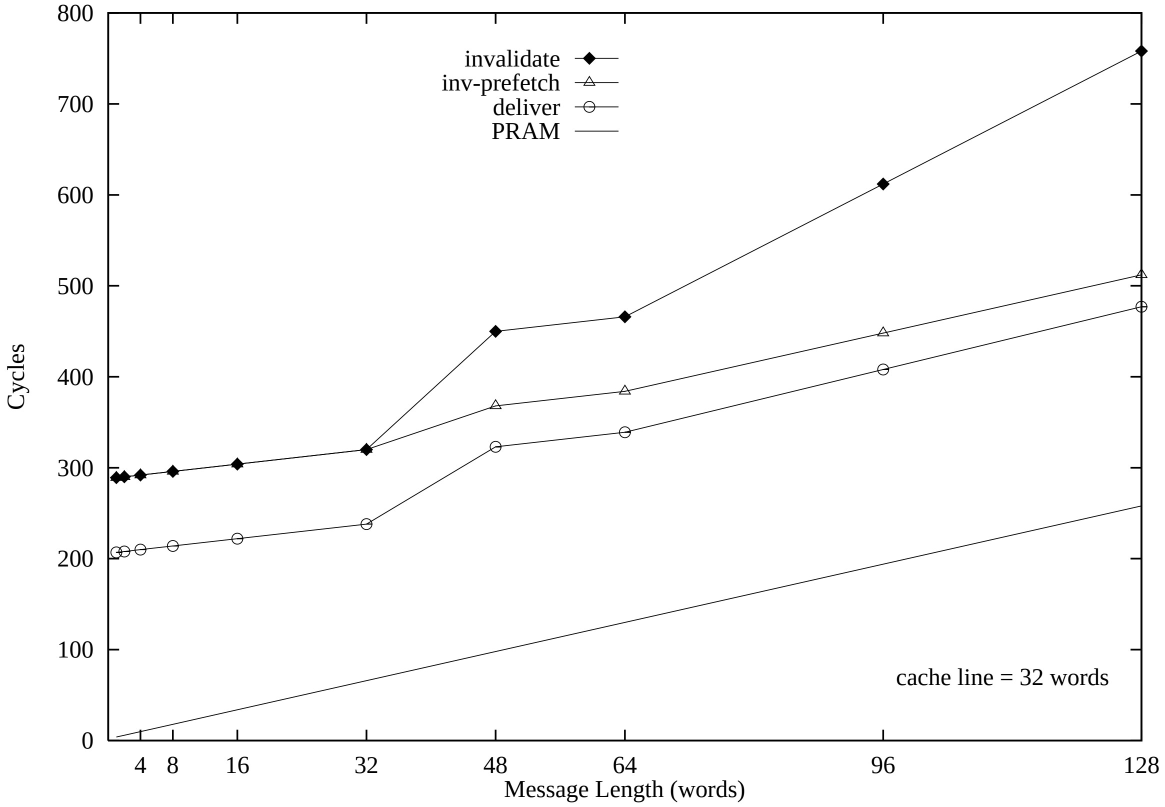

Figure 4: Line comm, block synch: Latency *vs.* message size, 32-word cache line.

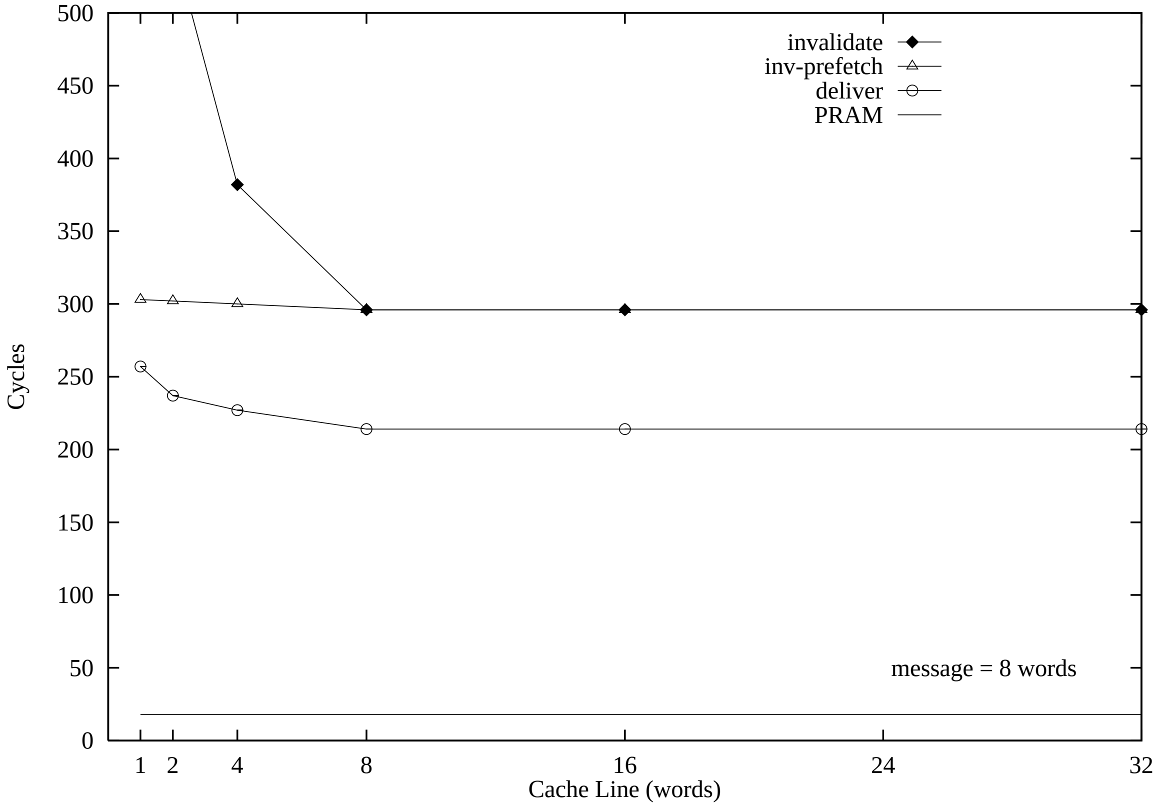

Figure 5: Line comm, block synch: Latency *vs.* cache line size, 8-word message.

These instructions are issued by the writing process, before synchronization, and are therefore visible in the overall latency. The **invalidate-prefetch** mechanism, on the other hand, overlaps prefetch instructions with the fetching of data – the latency of all but the first line fetch is hidden. This difference in overhead becomes significant for messages greater then 64 words (16 cache lines).

The latencies shown in figure 3 assume a cache line size of four words (16 bytes). Larger cache lines tend to improve performance, since data is transmitted more efficiently in larger blocks. Figure 4 shows the latency for various message sizes, assuming a 32-word (128-byte) cache line. There is no longer a crossover between **invalidate-prefetch** and **deliver**, because the **deliver** overhead has been reduced by a factor of four.

Figures 5 and 6 show the performance for different line sizes, assuming a message size of 8 and 32 words, respectively. As in the earlier figures, the performance of **invalidate** and **invalidate-prefetch** are identical when the message fits within a cache line. The **deliver** mechanism favors larger cache lines, since an extra instruction is required to initiate the deliver after every cache line. If the line size is too large, however, performance would degrade, because the entire cache line must be written to the network before the invalidate associated with writing the lock can be sent. This degradation is not observed here, because the time to receive the acknowledgements is larger than the time to write the line to the network.

Figures 7–9 show the latencies of the line-oriented mechanisms when word-oriented synchronization is used.

The **invalidate-prefetch** model shown here assumes *perfect* prefetching. With word synchronization, it can be detrimental to prefetch the entire message, since the latter part of the message may not have been written before it is fetched. If empty or partially-empty lines are prefetched, then they will have to be fetched again when the consuming process tries to read the empty words. If, however, a programmer (or compiler) knows about the network latency between the consumer and producer, prefetches can be staggered

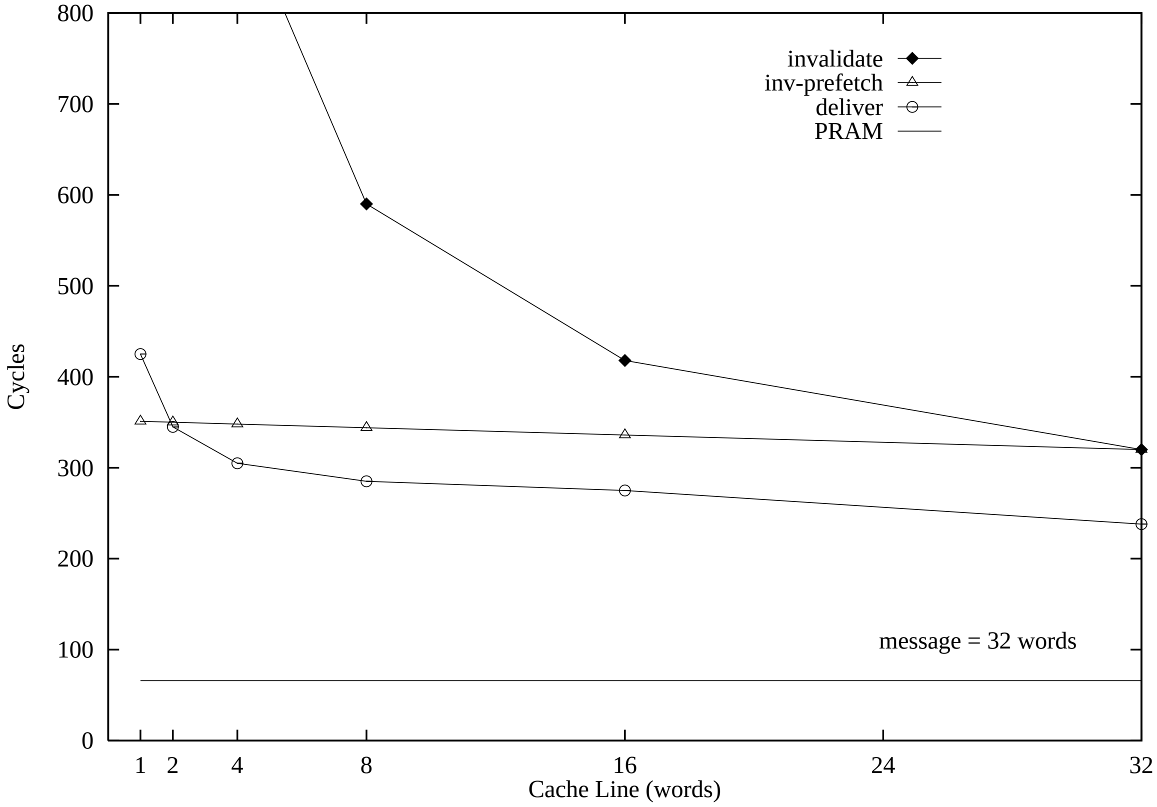

Figure 6: Line comm, block synch: Latency *vs.* cache line size, 32-word message.

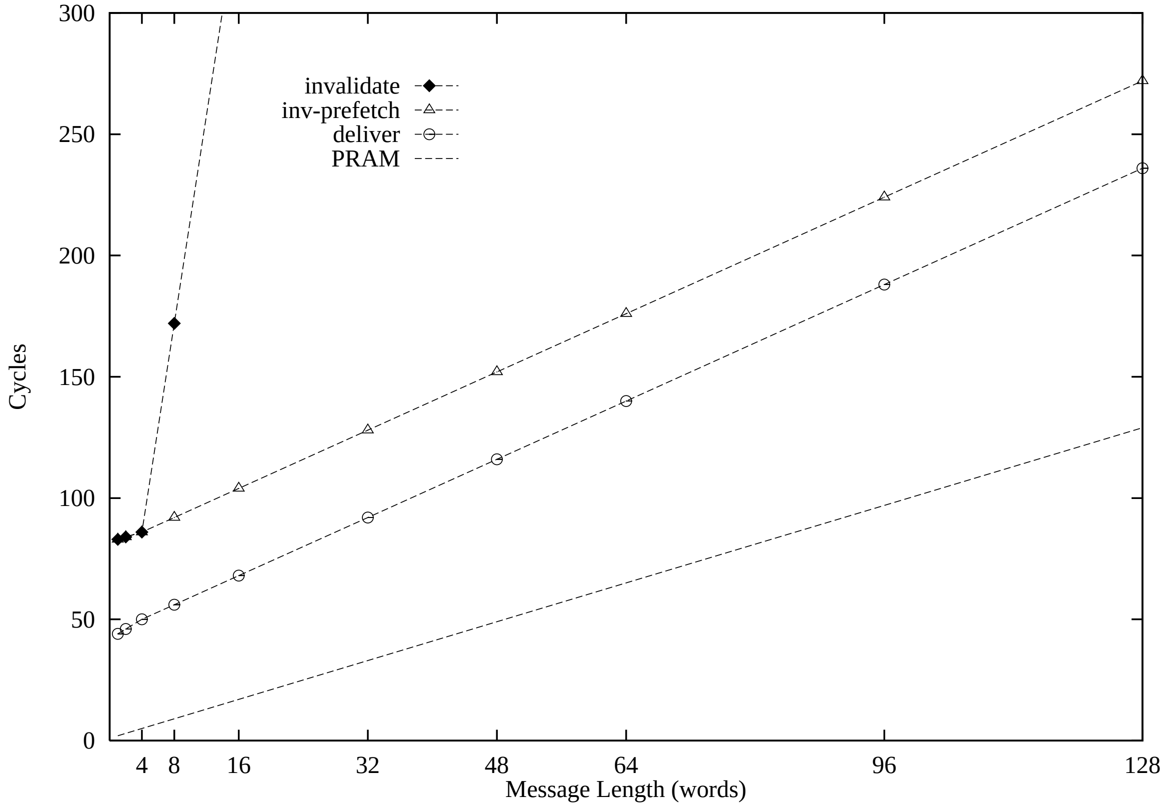

Figure 7: Line comm, word synch: Latency *vs.* message size.

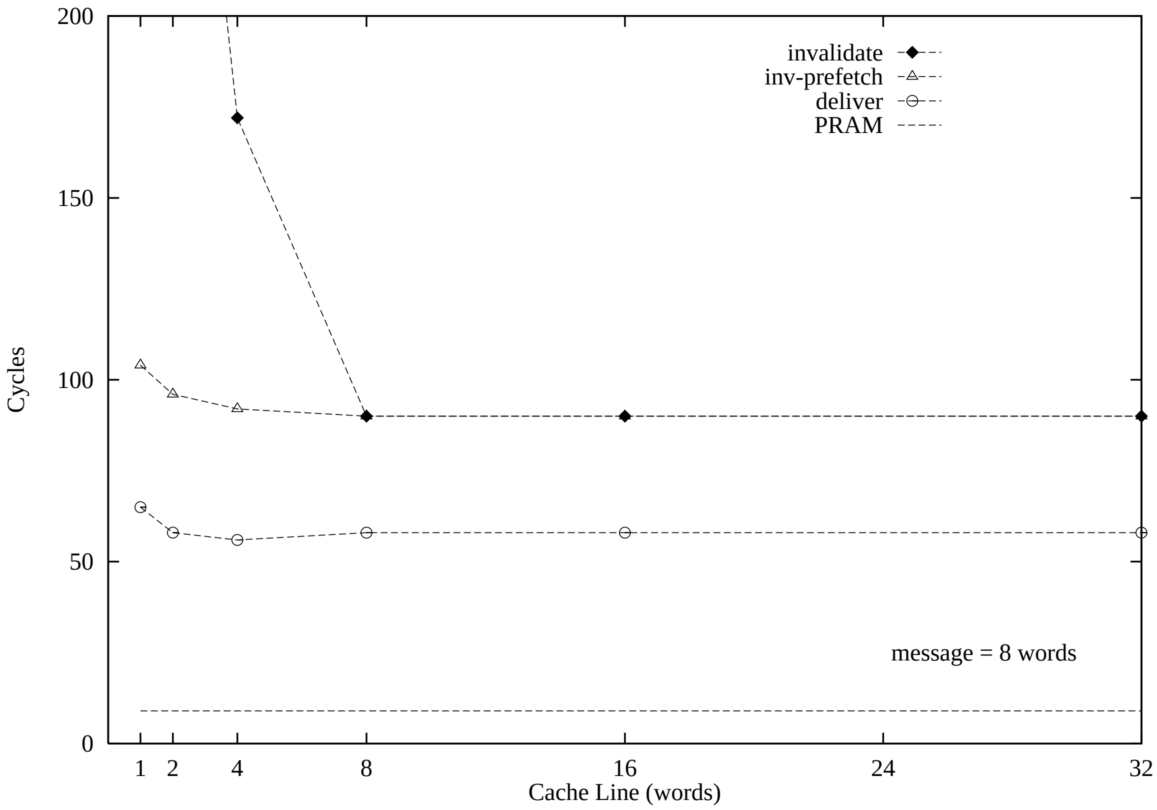

Figure 8: Line comm, word synch: Latency *vs.* cache line size, 8-word message.

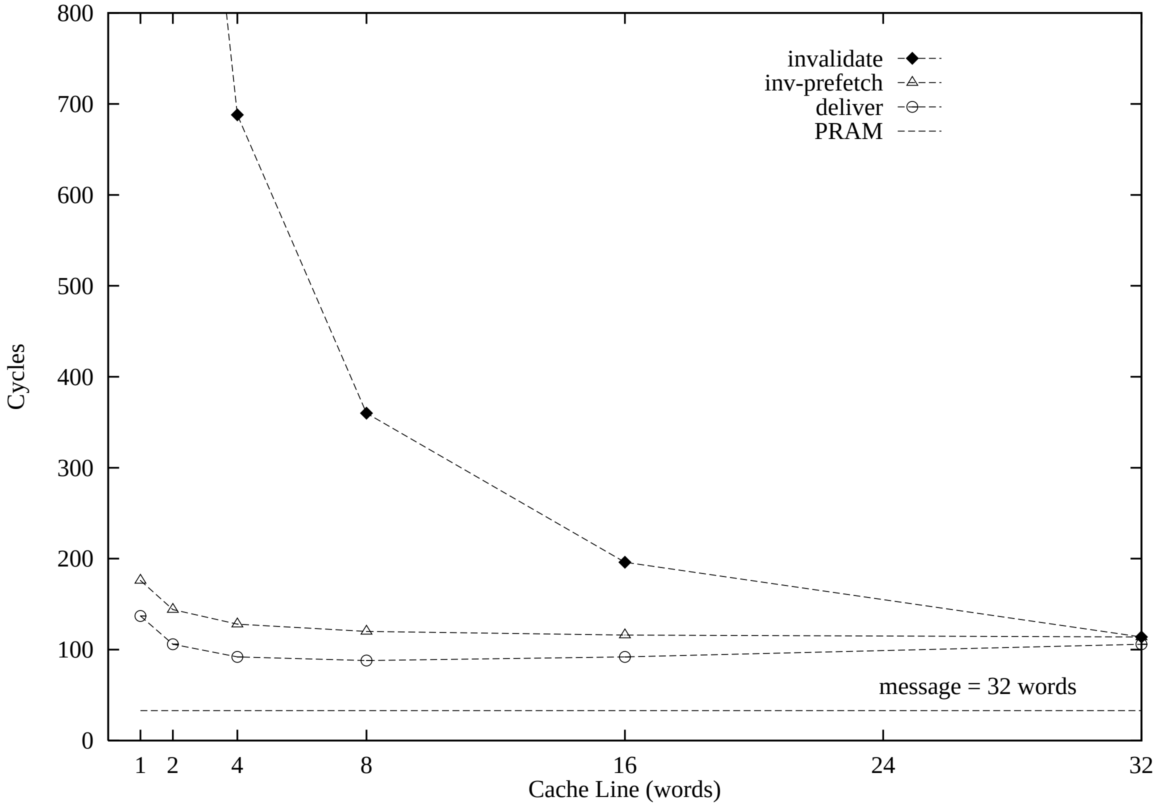

Figure 9: Line comm, word synch: Latency *vs.* cache line size, 32-word message.

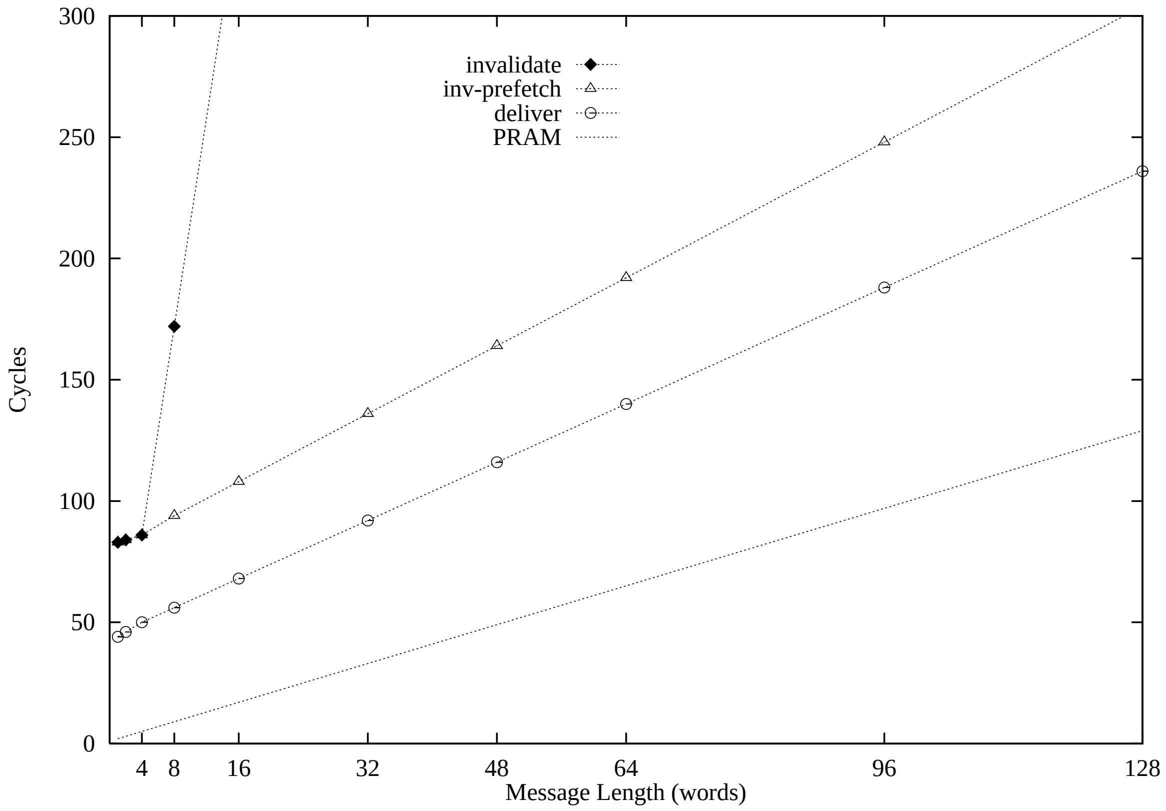

Figure 10: Line comm, line synch: Latency *vs.* message size.

in time to increase the likelihood of fetching data that has, in fact, been written. The model assumes that this strategy is used perfectly, and that no line has to be re-fetched. This would be difficult in practice—the most conservative strategy is to prefetch only a line at a time, which degenerates into the **invalidate** strategy.

The **invalidate** mechanism also has a risk of fetching empty or partially empty lines. With the default parameters, however, the latency of the fetch request is larger than the time required to write the entire line, so the producer is able to keep ahead of the consumer. Using word synchronization dramatically reduces the **invalidate** latency, but the cost of fetching only a single line at a time dominates for larger messages, making this mechanism much worse than the other two.

The **deliver** mechanism performs very well, especially for small messages. It out-performs **invalidate-prefetch** because of its producer-driven nature—the data is transmitted as soon as it is written and is consumed as soon as it arrives. For large cache lines, **invalidate-prefetch** and **deliver** have essentially the same performance, except for messages smaller than a cache line.

Figures 8-9 show the effects of various line sizes on latency. The **deliver** retains is advantage across varying line sizes, but the curve now shows a pronounced local minimum. There is a tradeoff between the overhead associated with smaller cache lines and the benefit of transmitting the data earlier (and thus being able to consume the data earlier). The optimal line size appears to be around four or eight words, depending on the message size.

For the line-oriented mechanisms, we can also consider line-oriented synchronization, shown in figure 10. (The latency for word-oriented synchronization is used for the **para** model, since the effective "line size" for a PRAM is one.)

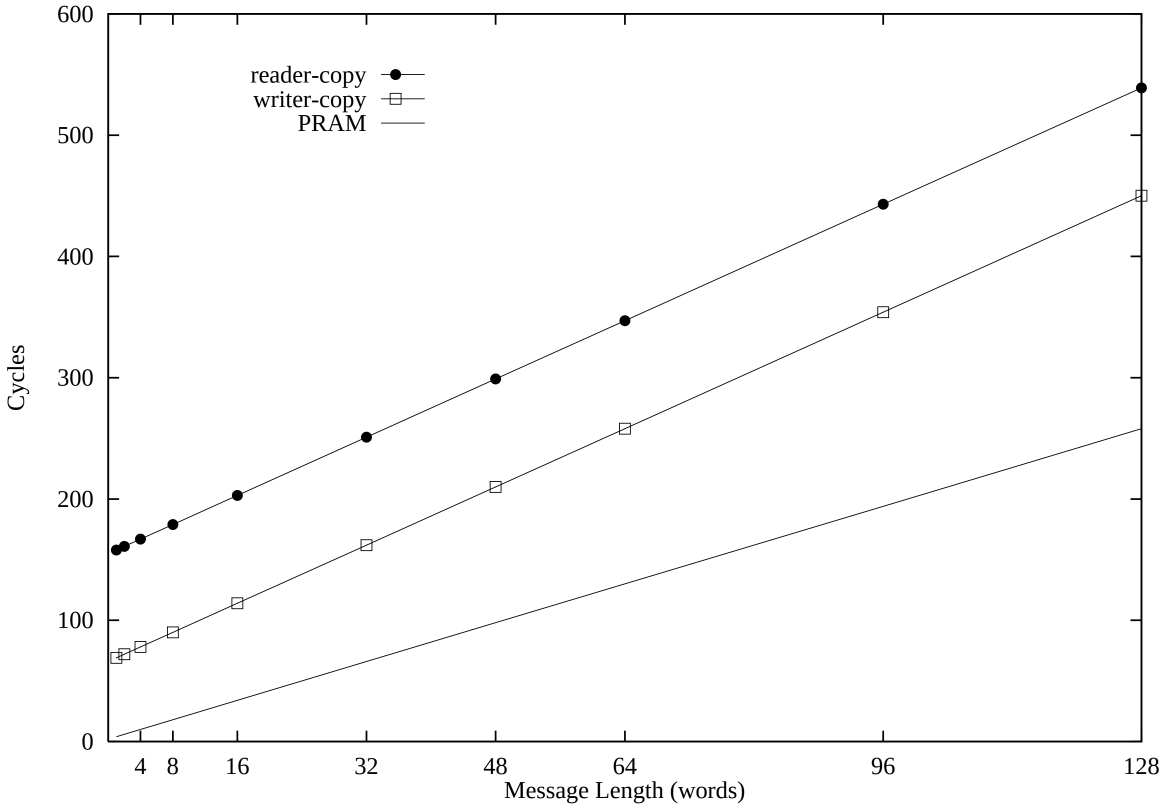

Figure 11: Block comm, block synch: Latency *vs.* message size.

The **deliver** performance with line synchronization is the same as for word synchronization. In both cases, the line is transmitted when the producer has finished writing it, and the data may be consumed as soon as it arrives.

Invalidate-prefetch is similar to **deliver** if a `prefetch_with_lock` instruction can be used to prefetch all lines at the beginning of the transaction. The prefetch requests are queued at the writing processor; when each lock is released by the writer, the data and the lock are sent to the consumer. The difference between the two models is that **invalidate-prefetch** requires the consumer to ask for the data before it can be transmitted.

The **invalidate** model shows the same performance as with word synchronization. One advantage of line synchronization, however, for both **invalidate** and **invalidate-prefetch** is that a line will never be delivered empty or partially full. With word synchronization, an empty or partially-full line may need to be re-fetched, which greatly increases latency. Therefore, the *worst-case* performance of word synchronization is worse than line synchronization for the consumer-initiated protocols.

4.4 Block-Oriented Communication

Finally, we consider the block-oriented mechanisms: **reader-copy**, **writer-copy**, **message**, and **message-eager**.

The performance of the first two using block synchronization is shown in figure 11; the other two are inherently word-synchronized. The **writer-copy** model performs better than **reader-copy**, because it eliminates the

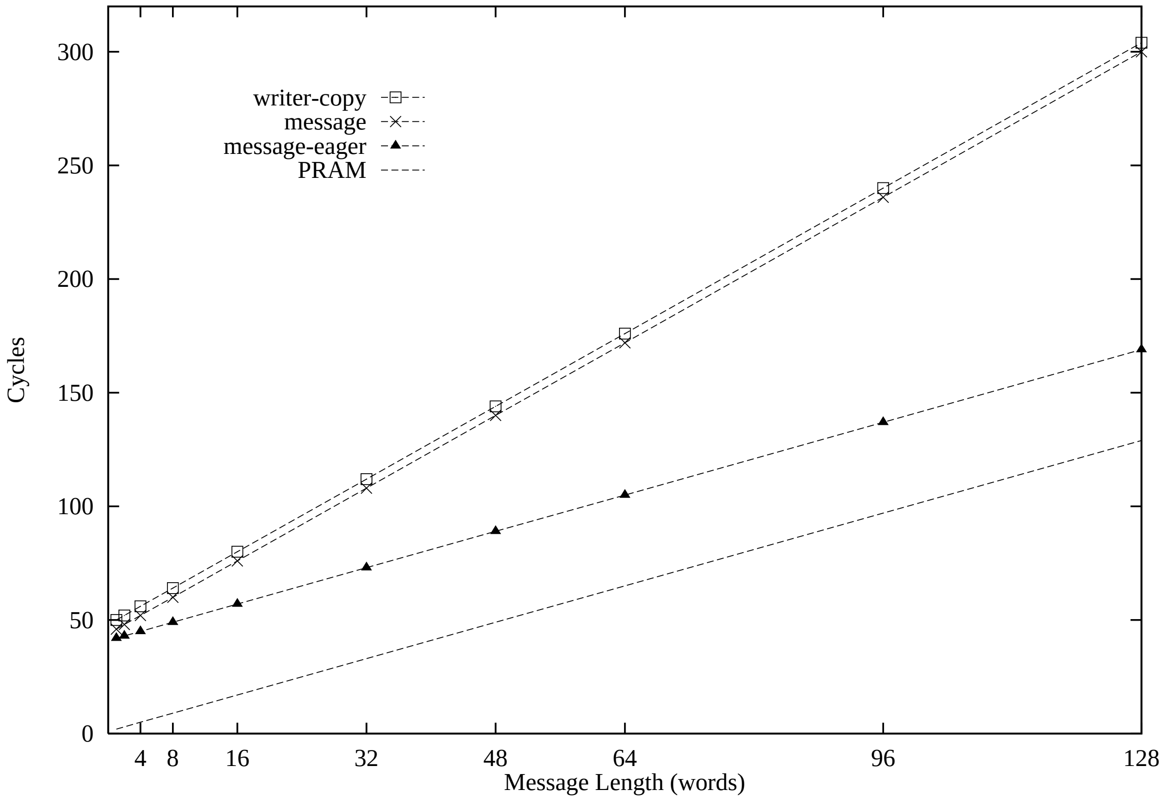

Figure 12: Block comm, word synch: Latency *vs.* message size.

network latency required to fetch the message from the writer's memory. In the **writer-copy** case, once the reader observes that the synchronization lock has been written, it reads the message directly from its local memory.

Figure 12 shows the effect of word-oriented synchronization on the block-oriented mechanisms. The **message-eager** mechanism shows a dramatic advantage over the other two. It takes full advantage of word synchronization by transmitting each word as soon as it is written; the other two must wait for the entire message to be written before transmitting.

It is also interesting to note how close **message** and **writer-copy** are. The StreamLine mechanism [5], a proposed implementation of the **message** model, is much more complex than a block copy, yet there is little apparent advantage in latency. There are other issues, however, such as buffer management and synchronization, which would benefit from StreamLine in a full application.

The **reader-copy** mechanism is not shown, because the synchronization model does not make sense in that situation. The reader cannot request that the message be sent until it knows that the message has been written. Therefore, unless a mechanism is set up to queue read requests, **reader-copy** implies block synchronization.

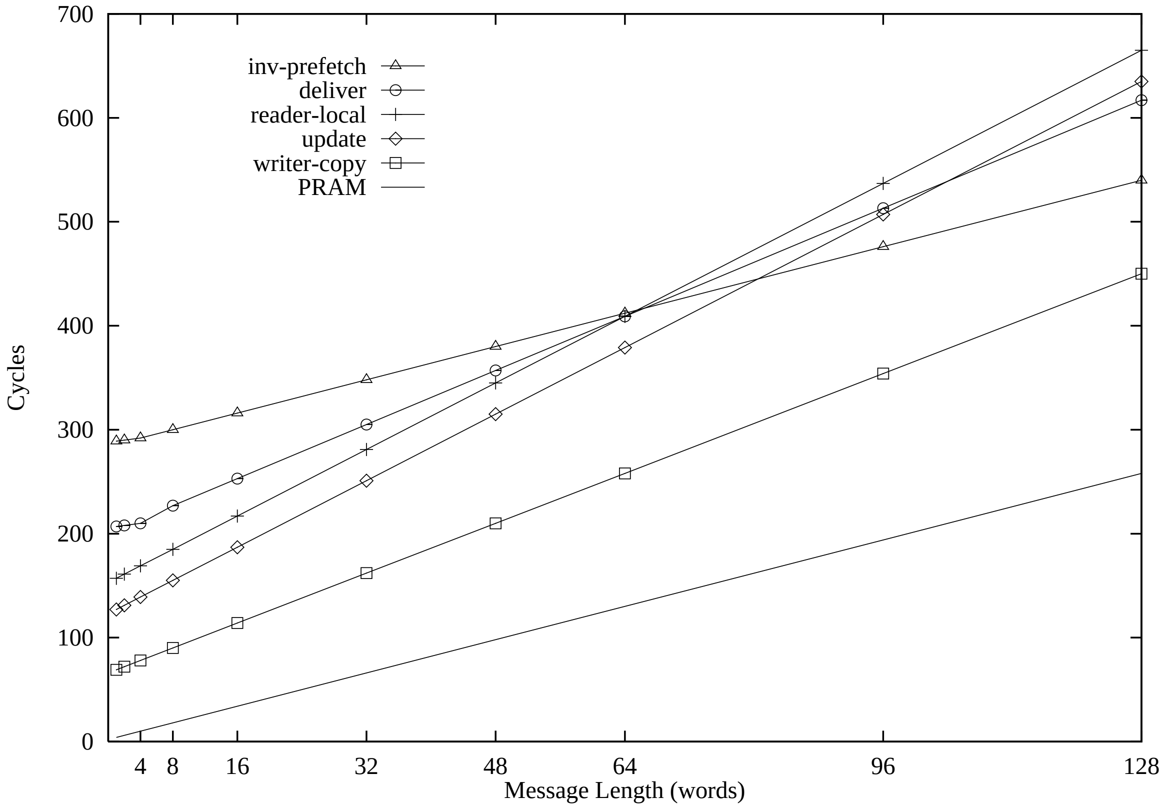

Figure 13: Block synchronization: Latency *vs.* message size.

4.5 Overall Comparison

Figures 13 and 14 show the lowest latency mechanisms, using block and word synchronization, respectively.

For block synchronization, **writer-copy** shows the best performance, due to its efficient block-oriented transmission. **Deliver** and **update** are both less efficient, since data is transmitted a line or word at a time, and each transmission incurs some network overhead (address, packet header, and so forth).

For small messages, the producer-initiated mechanisms perform best, but **invalidate-prefetch** eventually catches up and surpasses the others with sufficiently large messages. The producer-initiated schemes must have transmitted all of the data before they can perform the synchronization operation—with **invalidate-prefetch**, the producer need only issue invalidates before synchronizing. This allows the consumer to be signaled earlier and to begin fetching and reading the data. Eventually the early synchronization overcomes the additional fetch latency, and **invalidate-prefetch** wins.

Figure 14 shows the performance of mechanisms which use word synchronization. The range on the graph is the same as in figure 13, so that the reader may easily compare the performance of word and block synchronization mechanisms.

The difference in performance among the word-synchronized mechanisms stem from efficient communication and eager transmission. **Message** performs better than **update** because of block transmission. **Deliver** and **message-eager** perform progressively better than **message** because of eager transmission: in **deliver**, each line is transmitted as it is written; in **message-eager**, each word is transmitted as soon as possible.

Invalidate-prefetch with line synchronization is also shown on the graph. (Line synchronization is more

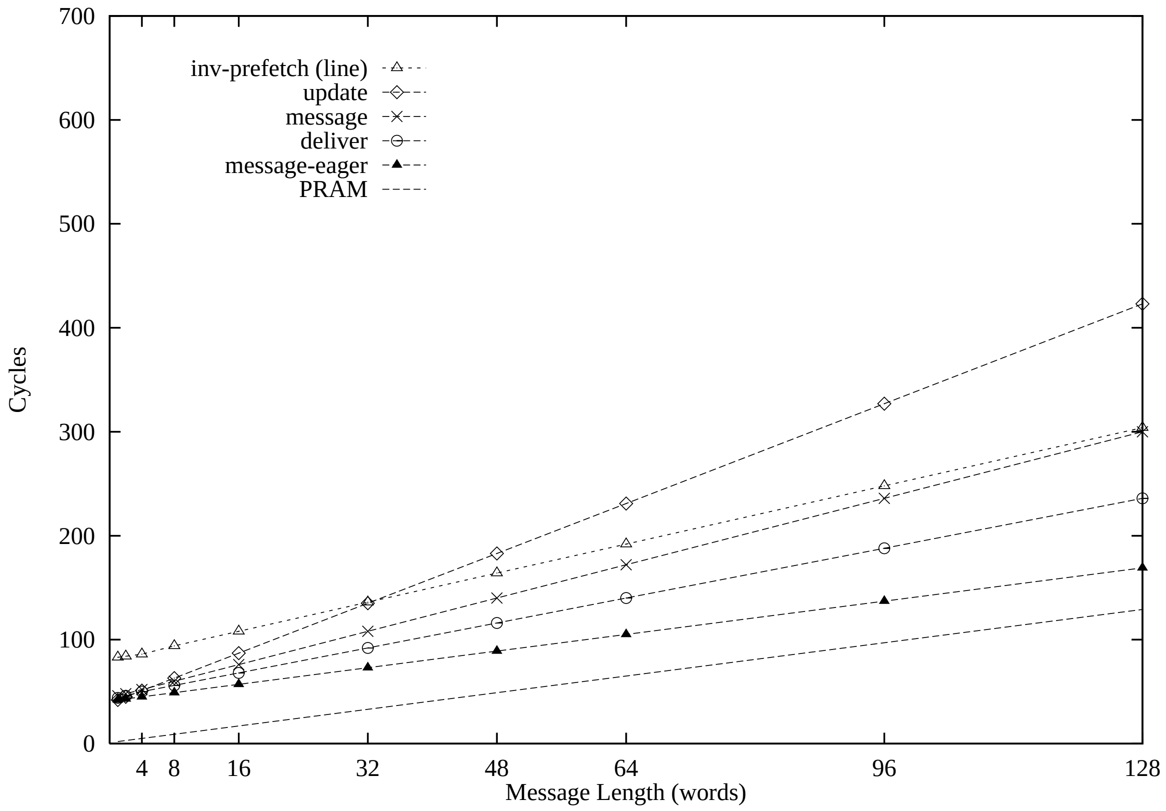

Figure 14: Word synchronization: Latency *vs.* message size.

realistic than the perfect prefetching assumption used with the word model.) As in the block synchronization case, consumer-initiated **invalidate-prefetch** starts off with a larger overhead, due to the latency associated with the first consumer fetch. For large enough messages, however, the early synchronization and transmission wins out over the non-eager **update** and **message** mechanisms.

4.6 Summary

The results of this study should not be surprising, because they illustrate concepts that are well known to computer system architects:

1. Producer-initiated communication results in low latency, because data is sent where it is needed before it is needed.
2. Efficient, opportunistic (eager) synchronization is the most important mechanism to reduce data transfer latency.
3. Efficient, block-oriented communication is important in reducing data transfer latency.[4]
4. Prefetch and pipelined loads and stores are very effective at reducing observed latency.

While all of these mechanisms have been explored before, this study provides some insight into which mechanisms are the most effective. There are many shortcomings of the simple models used here, and we do not assert that the models accurately predict performance in a real system on a real application. The models are useful, however, in pointing toward mechanisms that are most interesting to investigate. Future simulation studies will concentrate on the **deliver**, **message**, and **message-eager** mechanisms.

References

[1] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[2] James K. Archibald. *The Cache Coherence Problem in Shared-Memory Multiprocessors.* PhD thesis, University of Washington, February 1987.

[3] BBN Laboratories Incorporated. Butterfly parallel processor overview. BBN Report 6148, March 1986.

[4] Shekhar Borkar et al. iWarp: An integrated solution to high-speed parallel computing. In *Supercomputing '88*, November 1988.

[5] Gregory T. Byrd and Bruce 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. CRC Press, Inc., Boca Raton, FL, August 1991.

[6] William J. Dally et al. Architecture of a message-driven processor. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 189–196, June 1987.

[7] William J. Dally, John S. Keen, and Michael D. Noakes. The J-Machine architecture and evaluation. In *Compcon Spring '93*, pages 183–188, February 1993.

[8] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[9] William J. Dally and D. Scott Wills. Universal mechanisms for concurrency. Manuscript, obtained from author., September 1989.

[4] With the caveat that small packets tend to improve overall network throughput.

[10] James R. Goodman, Mary K. Vernon, and Philip 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, April 1989.

[11] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer—designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.

[12] Kourosh Grarachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John 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.

[13] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[14] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3:267, 1979.

[15] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[16] Thomas J. LeBlanc. Problem decomposition and communication tradeoffs in a shared-memory multiprocessor. In Martin Schultz, editor, *Numerical Algorithms for Modern Parallel Computer Architectures*, volume 13 of *The IMA Volumes in Mathematics and Its Applications*, pages 145–162. Springer-Verlag, New York, 1988.

[17] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH multiprocessor. *Computer*, 25(3):63–79, March 1992.

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

[19] G. F. Pfister et al. The IBM Research Parallel Processor Prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771. IEEE Computer Society Press, Washington, DC, August 1985.

[20] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Real-Time Signal Processing IV*, pages 241–247. SPIE—the International Society for Optical Engineering, May 1981.

[21] Hong-Men Su and Pen-Chung Yew. Efficient interprocessor communication on distributed shared-memory multprocessors. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume I, pages 45–48. CRC Press, Inc., Boca Raton, FL, August 1991.

[22] Mark R. Thistle and Burton J. Smith. A processor architecture for Horizon. Technical Report SRC-TR-88-010, Supercomputing Research Center, Institute for Defense Analyses, August 1988.

[23] William A. Wulf and Charles Hitchcock. The WM family of computer architectures. DRAFT—obtained from author, August 1989.