# Issues in Building a Cache-Coherent Distributed Shared Memory Machine using Commercial SMPs

XXXXXX

December 8, 1994

**Abstract**

**XO**[1] is a parallel machine that supports both fine-grained user-level message passing and cache-coherent distributed shared memory (CCDSM). A site in **XO** is an enhanced commercial PowerPC 620-based symmetric multiprocessor (SMP) connected to an MPP-class network. With the aid of a simple address capture device, a designated processor at each site executes cache-coherence protocols. Given the variety of memory models and ever growing complexity of coherence protocols, **XO** should provide a relatively cheap, low risk and flexible system to explore shared memory issues. This paper describes several difficulties we have encountered in implementing CCDSM using commercial SMPs. It offers some insights into the causes of these difficulties and possible remedies.

## 1 Introduction

This paper describes the challenges we have faced in designing a parallel machine based on commercial SMPs. We, like most other researchers[7], believe that to support a general programming model, a machine must provide both a global address space and efficient message passing for

[1] **XO** is a false name

short messages. From an implementation perspective, there is some synergy between these two requirements because creating the illusion of a global address space across a network of SMPs or workstations requires efficient message passing for cache-line size messages. However, due to the presence of caches in microprocessors, supporting shared memory abstraction on a distributed memory machine is significantly harder than pure message passing.

Researchers have devised a whole spectrum of techniques to implement shared address spaces on distributed memory machines. At one extreme, Li[11, 13, 12] has implemented coherent shared memory at page-level granularity without any hardware support beyond message passing. He does so by modifying the virtual memory manager to allow demand paging from remote nodes. This scheme has been reported to work well in certain application areas.

At the other extreme are machines that provide extensive hardware support for shared memory. Some machines, such as the Cray T3D, allow the user to access global memory in a cached or uncached fashion; however, the hardware does not maintain cache coherence. Machines that support cache coherence beyond single-bus systems invariably provide some processing power to execute directory-based coherence protocols. Machines in this class include the Stanford DASH[9, 10], the Kendall Square Research KSR1[4], and the MIT Alewife[1], which are all operational, and the Stanford FLASH[8] and the University of Wisconsin Typhoon which are under design.

Our project, called **XO**, is an attempt to build an efficient message passing machine using commercial SMPs based on PowerPC 620's. Each processor directly connects to an MPP-class network via a network interface unit. The memory bus in each SMP is augmented with a simple address capture device (ACD) which can be used in conjunction with one of the processors to support various shared memory models. Each SMP *site* modified in this manner will still run a standard operating system (OS) such as an SMP version of Unix. We have chosen to run an OS image per site as opposed to a single image on the whole system for better fault tolerance.

In addition to supporting all kinds of message-passing applications, we would like to run parallel applications written for SMPs on **XO**. This requires that we implement cache coherent global memory across sites. We think that **XO** will provide an inexpensive, low-risk, and flexible experimental vehicle for exploring coherence protocols for various memory models including I-structures[2], distributed operating system issues, and application development in a hybrid message passing/coherent shared memory environment.

The main contribution of this paper is the identification of some specific problems in implementing cache-coherent distributed shared memory (CCDSM) on commercial SMPs. The problems we will discuss have origins in three different sources: the processor and its snoopy bus protocol[5], buffering within the system and the non-FIFO character of the network. Though the problems are discussed in the context of PowerPC, we think some of these problems are likely to be present in any parallel coherent shared memory system based on stock microprocessors. As we will show, there are software solutions around most of these problems. However, with a slightly different processor or bus design, some of these problems can either be completely eliminated or their solutions made much simpler. We hope that these findings may influence the design of future processors and snoopy bus protocols.

In Section 2, we will give the background information necessary to understand the challenges we faced. It will describe those aspects of processors, memory buses, buffers and the network that are relevant to implementing CCDSM. Section 3 gives a brief description of **XO** and estimates of its performance. In Section 4, we discuss the technical problems that we encountered while implementing CCDSM. It also proposes solutions. The final section is a summary of our recommendations.

## 2 CCDSM Design Concerns

Before one can undertake the implementation of a CCDSM on a network of SMPs, it is important to understand certain characteristics of the processor, memory bus, buffering strategy and network used in the system. This section briefly discusses important characteristics of each component after reviewing a simple directory based cache-coherence scheme.

### 2.1 Directory Based Cache-coherence Schemes

Cache-coherence on distributed systems is usually maintained with directory-based schemes. "Directory" information on every cache-line is kept by a "home" location. Typical directory entries are `Uncached` by any site; `Shared` among one or more sites and thus is read-only at that moment; or `Modified` (and temporarily owned) in one site and only that site can read and write that cache-line. When a memory operation misses in a processor's local cache, the request is forwarded to the home location which checks its directory. If the cache-line is in the `Uncached` state, the request is satisfied and the directory updated. The home site can also satisfy a read request when the cache-line is in `Shared` state.

A Write[2] request will, however, require all the current readers to be invalidated. Invalidation means that all processors holding that cache-line must have that cache-line removed from their cache. Regardless of the request type, if the cache-line is in the `Modified` state, the home must invalidate the cache-line at the current owner's site, forcing the up-to-date cache-line to be returned to home, which then sends it to the requester. The directory is updated appropriately during this process. Though all directory-based cache-coherence schemes follow this basic pattern, there are many variations and optimizations. For simplicity of discussion, this paper will assume a

[2]Write requests generally come out of a processor as a "read-with-intent-to-modify" request rather than a "write" because the write may just be to one word in a cache-line.

sequentially consistent implementation of memory operations when they appear on the memory bus. The rest of this section gives an overview of the building blocks of a CCDSM system and the issues related to each component.

## 2.2 Memory Model

A processor implements a specific particular memory consistency model. Most current microprocessors, including PowerPC 620, will reorder memory operations to improve performance; for example, giving loads higher priority than stores since the instruction stream will stall for want of a loaded value. To preserve the semantics of sequential execution on a processor, memory transactions from a processor to the same location proceed in order. However, memory transactions to different locations do not necessarily have any fixed ordering. This consistency model is commonly called weak consistency.

Processors that implement weak consistency invariably provide sequentializing instructions, such as the `sync` instruction in PowerPC, that provide a fence or barrier ensuring that all outstanding operations complete before subsequent operations are issued. In order to implement sequential consistency, such sequentializing instructions are inserted at appropriate places in the instruction stream.

Weak consistency is sufficient for correct execution most of the time. An instance where weak consistency is insufficient is when dealing with memory mapped I/O devices. Generally, I/O devices require that writes to their address space are performed in a specific order. It becomes important, then, that memory operations appear on the bus in the same order as their ordering in the instruction stream. Since performance of superscalar processors degrades rapidly if sequential consistency is maintained, the use of sequentializing instructions should be avoided unless absolutely necessary.

## 2.3 Memory Instruction Semantics

We are using commercial processors to implement a CCDSM system. Because these processors are designed for the uniprocessor or SMP setting, the semantics of some of their instructions are not necessarily "scalable". As another example, the PowerPC used in **XO** has a data cache-block flush (`dcbf`) instruction which flushes a specified address out of *all* processors' caches. It may be possible to implement the exact `dcbf` semantics on a parallel machine built from multiple indirectly connected buses by essentially broadcasting every flush to all processors. However, it is certain that such a scheme would be inefficient. The basic reason to maintain directories to avoid broadcasting invalidations. An alternative semantics is to limit the scope of `dcbf` to flush only processors on the bus where the `dcbf` is issued. Using these semantics will allow us to use the `dcbf` instruction to perform invalidations within a site to implement CCDSM. These and related issues require the designers of a CCDSM system to examine the instruction set's semantics carefully.

There is a class of privileged instructions, used to support operating system activities. One such instruction is the `tlbie` which invalidates TLB entries across the snoopy bus. Whether a CCDSM system supports global semantics of such instructions can have an impact on the type of OS that can run on the system. If each site runs its own OS, these instructions do not have to be supported across sites.

## 2.4 Memory Bus Implementation

Like processor cache design, memory buses have gone through refinements to improve performance. Early memory buses permitted only one bus transaction at a time. If a processor issued a read, the memory bus would block until that read was satisfied. Later memory buses allowed pipelining which increased bus throughput. A pipelined bus, however, cannot take advantage of memory operations that are satisfied out of order. If one operation takes longer to perform, all of the operations that follow it must wait until it finishes.

A more general approach is provided in a split-phase bus, which allows replies to return in an order different from that in which their requests were issued. This approach requires some way of associating a reply to its request, making implementations more complicated but promises better performance. It appears that most microprocessors of the near future will have split-phase buses. As we shall see later in this paper, split-phase buses are needed in the CCDSM setting not only to improve performance, but also to avoid certain types of deadlocks.

It is important to note that when a processor issues a memory request on the bus, it can be rejected (retried) by the memory system. The processor has to keep retrying until the request is accepted. During this time, the processor has to perform snooping operations and react as though the request has not occurred. Once a request has been accepted by the memory system, however, the request is considered to be "committed" and the processor, in general, does not allow cancelation of the request. The need for such cancelations does not arise in SMPs.

On a split-phase bus, the time between the acceptance of a request and the arrival of a corresponding response opens up a window during which the semantics of interactions between new bus operations and the outstanding memory operation is not obvious. Generally there is no problem if the two bus operations are to different cache-lines. However, consider how a processor should respond to an Invalidate $x$ bus operation after the processor's Read $x$ request has been accepted. Should it allow the Invalidate to complete because it does not yet have a copy of $x$ in its cache, or should it prevent the Invalidate from completing until it gets back data $x$, and remove that from its cache? Both positions make sense in the SMP setting. The proper choice, however, is crucial to a CCDSM system as we will see in Section 4.

### 2.5 Buffers and Networks

Buffer management is important in a CCDSM system as the amount of buffer space for storing the state of in-progress requests in the processor, network, and coherency engine is always limited. If one is not careful in the design of the system, concurrent processing of multiple requests can lead to deadlocks due to insufficient buffers.

Another major concern is whether the network delivers messages in FIFO order. If the network is not FIFO, it is important for the coherence protocol to know that and account for it, or for some lower level software to ensure that FIFO ordering is observed by the coherence protocol. Generally speaking, non-FIFO networks offer better raw performance but add significant complexity to cache coherence protocols.

## 3 The Design of XO

This section will give a brief description of the **XO** hardware and how we plan to support global cache coherence. A block diagram of a **XO** site is shown in Figure 1. A site consists of a slightly modified commercial building block of **XO** (unshaded in the diagram) is an SMP built from PowerPC 620 processors. A network interface unit (NIU) is attached to each processor. An address capture device (ACD) is added to the coherent interconnect to help support CCDSM. When the machine is running in the mode that supports global cache-coherence, a 620 from each SMP, which we call the service processor (sP), is dedicated to this task. The rest of the processors, which are used for computation, are called data processors (dP). Parallel programs running on **XO** can communicate through both message passing and shared memory, whether they are executing on one site or across multiple sites.

Each 620 is connected to an NIU through its L2 cache interface. The NIU is being designed and

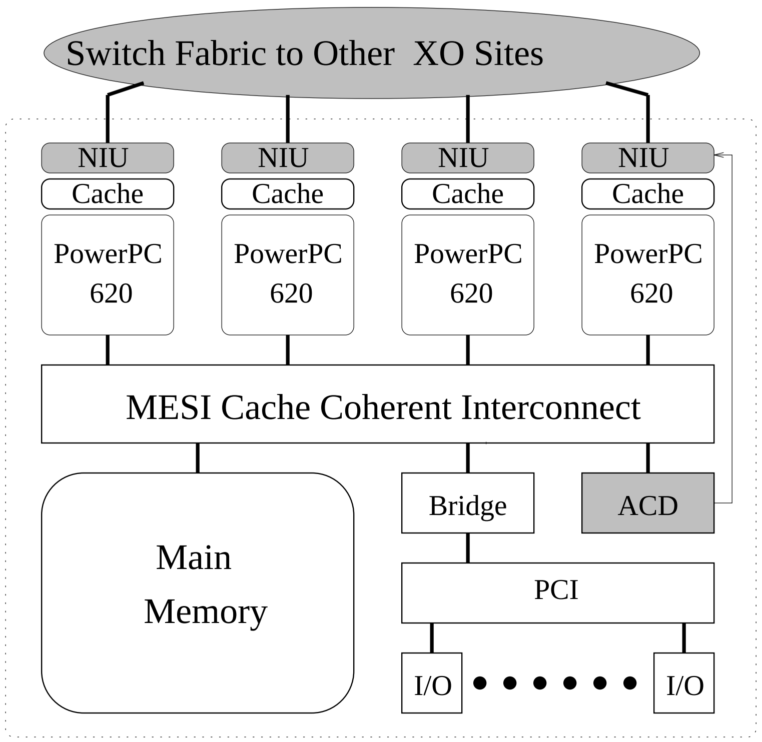


Figure 1: **XO** block diagram

implemented by Motorola and its partners and was heavily influenced by the MSU functional unit of the 88110MP[14]. The NIUs provide access to a Fat-tree network built out of Arctic[3] routing chips. Currently under development at MIT, the Arctic router is a 4x4 router that provides 200 megabytes/sec/link bandwidth.

### 3.1 ACD

The ACD has two basic functions: (i) "capture" global bus transactions and pass it to the sP and (ii) allow the sP to return data to the requesting processors. Capturing means to recognize a global memory transaction, store away a complete copy of the transaction (including the address, transaction type and bus tag to return a value if necessary) and respond accordingly to the processor that initiated that transaction. If the ACD buffers are full, *i.e.*, have not been emptied by the sP, it forces the processor to retry the transaction. Because of limited ACD buffering, the sP should remove captured transactions as quickly as possible.

ACD can be instructed by the sP to selectively flow-control (retry) certain types of transactions. This is needed for buffer management because the ACD hardware buffers are backed by software buffers under the sP's control.

The ACD also performs Invalidations (flush bus operation). It has hardware support that allows it to interleave a Flush operation with the other functions it performs. We will examine the reasons for having the ACD do the Flush bus operation in Section 4.5.

It is possible to build the sP into the ACD or provide some dedicated path between the two. Either design requires substantially more hardware than our design wherein the sP communicates with and commands the ACD over the SMP bus. The ACD also has a one bit line going to the sP's NIU, allowing the sP to poll its NIU for ACD events.

## 3.2 Supporting CCDSM with the ACD and sP

The address space in **XO** is divided into four disjoint parts: (i) dP local; (ii) dP global; (iii) sP local; and (iv) ACD command spaces. The dP local space is private to each site, directly accessible by load/store operations only from the dPs of that site. The dP global space is the address space supported by CCDSM. Bus operations to this address space are captured by the ACD. The global physical address that is seen by the ACD is not really a physical address but encodes information that will allow the sP to get to the real physical address of the cache-line. We call this address a *virsical* address. Part of the virsical address indicates the current home site and the rest of the virsical address is a virtual address in the sP local address space of the home site. Finally, the ACD command space is used by the sP to communicate with the ACD.

The ACD and the sP support CCDSM as follows. When a processor accesses a global location and misses in its cache, the request propagates to the memory bus and is captured by the ACD.

The ACD then signals the sP by setting a bit in its status word as well as asserting a line connected to the sP's NIU. The sP either polls the ACD directly or through the NIU.

The sP then reads out the captured transactions, clearing the transactions from the buffers. The sP will take appropriate action to satisfy the request by checking an inclusive, software L3 cache which it maintains, and sending messages to the home site if necessary. The sP will also serve as a protocol processor, receiving messages from other sites requesting cache-lines whose home is its site. It is important, therefore, for an sP to continue to serve incoming requests even though it has outstanding requests. When the response to the remote request returns, the sP updates whatever state it needs to, including L3 cache state, and returns the data through the ACD to the requesting processor.

### 3.3 Planned Features

We plan to try the following ideas in the implementation of CCDSM on **XO**.

**Software L3 Cache:** An essential idea is a large software L3 cache, implemented in main memory. The sP can implement an inclusive software cache which stores all remote data brought to the site. Any sort of caching strategy can be implemented at this level, including automatic prefetching, presending and update protocols.

**Split-phase Cache Accesses:** During a *split-phase* cache access, a processor switches threads when an access to global memory misses in its cache. The tough part is how to indicate that there was a cache-miss. We plan to implement this idea by having every load of global locations checked against a "miss pattern". The miss pattern would be returned by the sP to the requesting processor after not finding the desired cache-line in its L3 cache. If a load returns a miss pattern, a cache-miss is assumed and the current task is swapped out. Part of the swap-out would be a request to fetch the cache-line. A microthread descriptor (an instruction pointer/frame pointer pair), which would allow the task to restart, would be included in that request. The cache-line could be delivered to the requesting processor's SP for insertion into the L3 cache, while the desired value sent directly to the requesting processor along with the microthread descriptor. When the desired value returns to the requesting processor, the thread wakes up, and continues with the value it requested. If another processor requests the same cache-line in the near future, the SP can satisfy the request directly from the L3 cache.

**Supporting Coherence from Page Level to Cache-line Level:** During the time when a global page is accessed by only one site, "localizing" the page temporarily allows cache miss processing to by-pass the ACD and SP, resulting in a lower cache miss penalty. "Localization" is achieved by incorporating Virtual Shared Memory[11] techniques. A global page can either have its coherence managed at the page-level or cache-line level, with dynamic, and possibly automatic, switching between the two schemes based on access patterns.

**Integrating I-structure Semantics into Cache Coherence Protocols:** Our research language relies heavily on I-structures[2], data-structures with synchronizing semantics. These are write-once, read-many-times memory locations that allows reads to be issued before the locations are written. Since they never need to be invalidated unless storage is reused, coherence protocols for these locations may become significantly easier. We plan on investigating these and other custom protocols tailored for specific applications.

### 3.4 Penalty of a Cache Miss

Since **XO**'s support for CCDSM is mostly in software, a cache miss on global data incurs a significant penalty. The penalties of cache-misses are shown in Figure 2. The times are given in processor cycles and are our current best estimates. The corresponding numbers for FLASH, as reported in [6] are given for comparison.

| Type of Miss | | | | | XO (proc cycles) | FLASH (proc cycles) |
|---|---|---|---|---|---|---|
| Add Space | Home | L3 hit? | Dir State | Owner | | |
| Local | – | – | – | – | 35 | 54 |
| Global | Local | – | Clean | – | 199 | 54 |
| Global | Local | – | Dirty | Remote | 692 | 286 |
| Global | Remote | Hit | – | – | 157 | 54 |
| Global | Remote | Miss | Clean | – | 699 | 222 |
| Global | Remote | Miss | Dirty | Home | 786 | 290 |
| Global | Remote | Miss | Dirty | Remote | 1219 | 382 |

Figure 2: Miss penalties in processor cycles. The FLASH numbers[6] have been normalized to processor cycles to allow easier comparison with **XO**. The **XO** numbers assume an 8-site machine where 6 network hops are needed to get to a remote home and 6 network hops to get to a dirty site from the home. (see text)

To the first order, the global memory miss penalties for **XO** are roughly three times those of FLASH, with the penalty for one case approaching four times. Our miss penalty for local address space appears to be faster than FLASH's partly because our machine will run at a slower clock rate, making the memory latency a fewer number of processor cycles. Though our miss penalties are significantly higher than those of FLASH, it is important to remember that we only pay these penalties when we miss in the L1 and L2 caches. We can express the relative efficiency of a particular implementation by average clocks-per-instruction (CPI) which is computed using the following equation.

$$CPI_{global\_mem\_ops} = CPI_{mem\_op\_cache\_hit} + miss\_rate \times miss\_penalty.$$

$$CPI_{average} = (1 - \%global\_mem\_ops) \times CPI_{non\_global\_ops} + \%global\_mem\_ops \times CPI_{global\_memory\_ops}$$

A low miss-rate and/or a low percentage of global memory operations will reduce the impact of **XO**'s disadvantage. A recent FLASH paper[6], which measured the performance of FLASH using a simulator, gives some ideas about the kind of miss rates that one may expect. For the seven programs (mostly SPLASH programs) that they ran on the simulator, three have miss rates below 0.1%, three have miss rates between 0.1-1.0%, and only one had the rather high miss rate of 6.0%.

To get some idea of what these miss rates mean for performance, we did some quick back-of-the-envelope calculation for the Average CPI, using some reasonable estimated numbers for the missing components of the Average CPI formula. The numbers we used are: $CPI_{non\_global\_ops}$ is 0.8; $CPI_{mem\_op\_cache\_hit} = 0.8$; % $global\_mem\_ops = 10\%$; $miss\_penalty = 800$ for **XO** and 266 for FLASH. This exercise shows that for miss rate of under 0.1%, the difference in miss penalty between **XO** and FLASH does not make any significant difference. At 1.0% miss rate, the difference becomes significant, with FLASH about 50% faster. However, it is not clear that both machines will have the same miss rate, and any difference in miss rate around the 1% neighborhood have great impact on overall performance. **XO** should encounter lower miss rates than FLASH since each site is an SMP, making the home site of more global data the local site. The use of a large L3 cache should also increase the sharing of remote data on a site. Miss rates that are in the range of 6% is not of very great interest because both machines will be suffering from such high "parallel slowdown" that neither can handle the program with any efficiency.

Thus, for programs that run well on an aggressive implementation like FLASH, **XO** has the potential of running reasonably. Clearly "embarrassingly local" programs will run equally well on either machine. Programs that do not run as well on **XO** as on FLASH do exist, but even FLASH has its limits. It will be interesting to see just what kind of performance we can get out of **XO**'s minimal cost CCDSM.

## 4 Issues in Designing CCDSM

In designing **XO**, we encountered a surprisingly large number of complications from using a processor in the SMP as the sP. In many cases, the exact nature of the problems is dependent on the microprocessor bus interface and the design of the snoopy bus itself. Other problems deal with buffer management and allocation. In the following section, we outline the major problems we encountered and their solutions. We believe that these problems or the suggested solutions to them are encountered in most CCDSM implementations today.

Before we discuss these examples, it should be noted that cache coherence protocols tend to be quite complex in their actual implementations. That complexity is a compelling reason for doing them in software instead of hardwiring them. In this paper, we are not presenting any specific protocols. However, what the following examples will illustrate is that without some cooperation from the underlying hardware, implementing CCDSM will be either impossible, or so inefficient as to be practically uninteresting. The following examples will not describe any complete protocols, but will just show protocol fragments to illustrate the problems encountered. Except for Section 4.7, all the sections assume a FIFO network.

## 4.1 Need for a Split-phase Bus in SMP

A blocking bus, which locks up the bus until an operation is complete, can cause deadlocks in a CCDSM system. This problem is due to concurrent events that have no immediate knowledge of each other. For example, consider processors $P_A$ and $P_B$ at sites $A$ and $B$ respectively.

1. $P_A$ has $x$ in `Modified` state and now wants to read $y$. The Read $y$ request goes on its bus and is accepted.
2. At the same time, $P_B$, which has $y$ in `Modified` state wants to read $x$. It issues a Read $x$ request which is accepted on its bus.
3. Neither can proceed any further because each request needs to get the cache-line from the other bus.

A snoopy bus used in a CCDSM system should therefore be split-phase allowing processors to snoop the bus and service bus initiated operations while waiting for its own requests to be fulfilled.

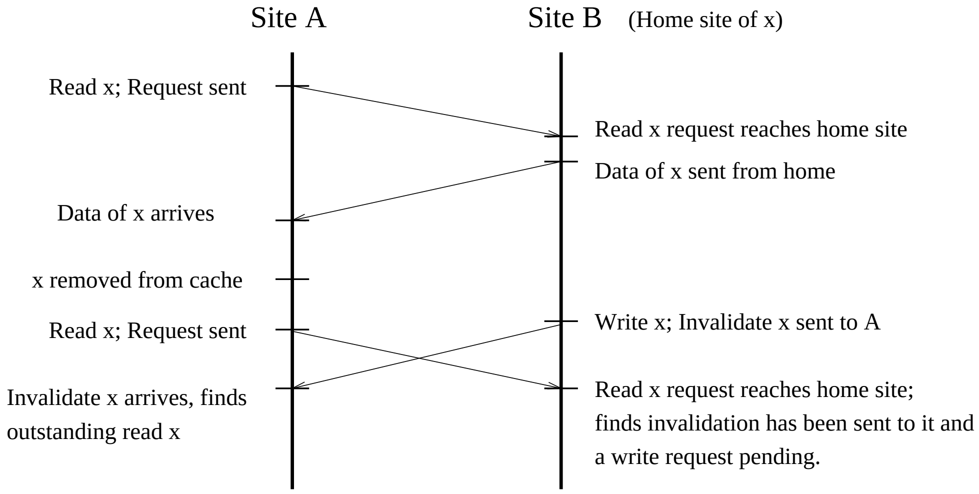


Figure 3: This time-line diagram illustrates the scenario described in the example of Section 4.2. Because of the concurrent initiation of memory requests, the system must handle an invalidation request which arrives at a site to find an outstanding Read request to the same cache-line.

It is true that as long as it is possible to retry bus operations, it is possible to implement CCDSM using either a blocking or pipelined snoopy bus protocol. However, the implementation complexity and the performance penalty makes either an unreasonable engineering choice. This was the main reason for us to select an SMP based on PowerPC 620 rather than PowerPC 601.

### 4.2 Interaction of Invalidates and Outstanding Reads

Suppose a processor retries an Invalidation to $x$ if there is an outstanding bus operation to $x$. This processor could deadlock a CCDSM implementation that uses it. For an example of the problem, consider a simple scenario, shown in Figure 3, where an Invalidate $x$ request arrives at $A$ when there is an outstanding Read $x$.

1. $P_A$ has $x$ in `Shared` state but replaces it in its cache.

2. $P_B$ wants to write $x$. Invalidation sent to $A$.

3. $P_A$ reads $x$ again.

4. $P_A$ receives Invalidation.

Since the processor does not allow the Invalidation to complete until the Read completes, the Invalidation of $x$, if put on the bus, will deadlock as it has to wait for the outstanding Read $x$ to complete. This depends on $B$'s Write $x$, which in turn depends on the Invalidation completing. In this case, the CCDSM implementation *must* track outstanding operations, intercept invalidation requests and send back a message indicating that the invalidation is completed. Skipping the actual invalidation is legal, since we know that the cache-line does not exist at the requesting site.

This problem is also solved if the snoopy bus protocol allows an Invalidate $x$ bus operation to complete even if a processor on the bus has an outstanding Read $x$. The read request is satisfied as normal when the cache-line is returned. This solution is preferred as it also solves the problem described in the next section, but has to be implemented by the snoopy bus protocol.

### 4.3 Complications from Multi-processor Sites

Multiple processors at a site create problem situations in CCDSM systems which an inclusive cache shared by all site processors eliminates. A variation of the example in Section 4.2 with two processors $P_{A1}$ and $P_{A2}$ at site $A$ explains why.

1. $P_{A1}$ reads $x$, $B$ is the home site of $x$. $P_{A1}$ obtains $x$ and caches it.
2. Processor $P_B$ tries to write $x$. The directory indicates site $A$ has a copy. An invalidation is sent to site $A$.
3. Meanwhile, $P_{A2}$ tries to read $x$, misses in its cache, and sends a Read $x$ request to the home site.
4. When the Invalidate $x$ request arrives at $A$, there is an outstanding read request to cache-line $x$.

Unlike the example in Section 4.2, there is a cache copy of $x$ in $P_{A1}$ which *must* be invalidated by an invalidation bus operation. If the snoopy bus protocol allows the invalidation to complete despite $P_{A2}$'s outstanding operation, there is no problem. Otherwise, an inclusive shared cache avoids the problem altogether by eliminating the situation where there is an outstanding Read $x$ from a site and $x$ in some cache of that site.

An extra cache is not necessary if the snoopy bus protocol supports cache-to-cache transfer of all "non-local" data. Cache-to-cache transfer when a Read $x$ occurs while $x$ is in the `Modified` state in some processor's cache is currently supported on many snoopy buses. Extending it to happen when $x$ is in the `Shared` state will allow all the caches to function as a combined per-site cache.

### 4.4 Need to Separate Buffers by Functionality

Assuming our protocol is logically correct, another source of deadlock is a finite number of buffers. The sharing of hardware resources can introduce dependencies between operations that are logically unrelated. There is no system that requires unbounded buffering; it is possible to estimate the number of buffers needed given the size of the machine and the protocol. This limit may even be acceptable in certain cases. The limit is not satisfactory, however, if buffer size is directly proportional to the number processors per site or the number of sites per machine or the number of read/write buffers in the microprocessors because the smallest change in machine configuration can make the system deadlock prone. We will show that separating buffers by functionality is a better idea.

Consider an implementation with four types of buffers: (i) Capture Buffers, used for storing captured requests until they can be forwarded to the home site; (ii) Home Buffers[3], which hold requests at a home site until they are processed; (iii) Invalidation Buffers, which hold invalidation requests until they complete; and (iv) Reply Buffers which allow the protocol to return a cache-line to the requester. Our example will deal with three sites, $A$, $B$ and $C$ with processors $P_A$ and $P_B$ on sites $A$ and $B$ respectively. Assume each site has two Capture Buffers, four Home Request Buffers,

[3]Entries do not have to be serviced in FIFO order. In particular a Home Request Buffer containing a currently blocked transaction does not block transactions in other Home Buffers on the same site.

one Invalidation Buffers, and one Reply Buffer.

1. $P_A$ writes cache-lines $x_1$, $x_2$, ..., $x_{10}$, (all of whose home is site $C$), caching them in the `Modified` state.

2. Similarly, $P_B$ writes cache-lines $y_1$, $y_2$, ..., $y_{10}$, (whose home is also site $C$), caching them in the `Modified` state in $P_B$.

3. $P_A$ and $P_B$ "exchange" the data that they work on, *i.e.*, $P_A$ reads cache-lines $y_1$, $y_2$, ..., $y_{10}$, while $P_B$ reads cache-lines $x_1$, $x_2$, ..., $x_{10}$. If the bus is split-phase or if there are many processors per site, many requests can be outstanding from sites $A$ and $B$.

4. All the read requests are buffered in Capture Buffers at the respective requesting sites, before they are buffered in site $C$'s Home Buffers. Invalidation requests are issued to sites $A$ and $B$.

5. When the invalidation requests, Inval $x_1$ and Inval $y_1$ arrive at sites $A$ and $B$ respectively, invalidation bus operations are performed. However, in order for the `Modified` data to be invalidated and written back, there must be Capture Buffers to accept them. Unfortunately, all the Capture Buffers have been used up by outstanding read requests, so the invalidations cannot complete.

6. Worse yet, no Capture Buffers will free up at $A$ or $B$ until there is space in the Home Buffers of $C$. But all the Home Buffers in $C$ are in use, and will not free up until the invalidations are done. A deadlock has now occurred. This situation is illustrated by Figure 4, where the arcs indicate dependence.

The above deadlock scenario arises purely from the constraints of finite buffering. If the home site, $C$, or the capture buffers has infinite buffering, this problem will not occur.

This deadlock is easy to see with a static dependence graph, which, for the example, is shown in Figure 5[4]. Each node of this graph is a pool of hardware resources used by the CCDSM implementation in a blocking fashion. An arc, such as from the Capture Buffers node to the Home Buffers node indicates that a Capture Buffer in use is not released until a Home Buffer is obtained.

[4]We have developed this technique for illuminating deadlock possibilities. It is very similar to techniques used to show networks are deadlock free. An article will be written on the technique in the near future.

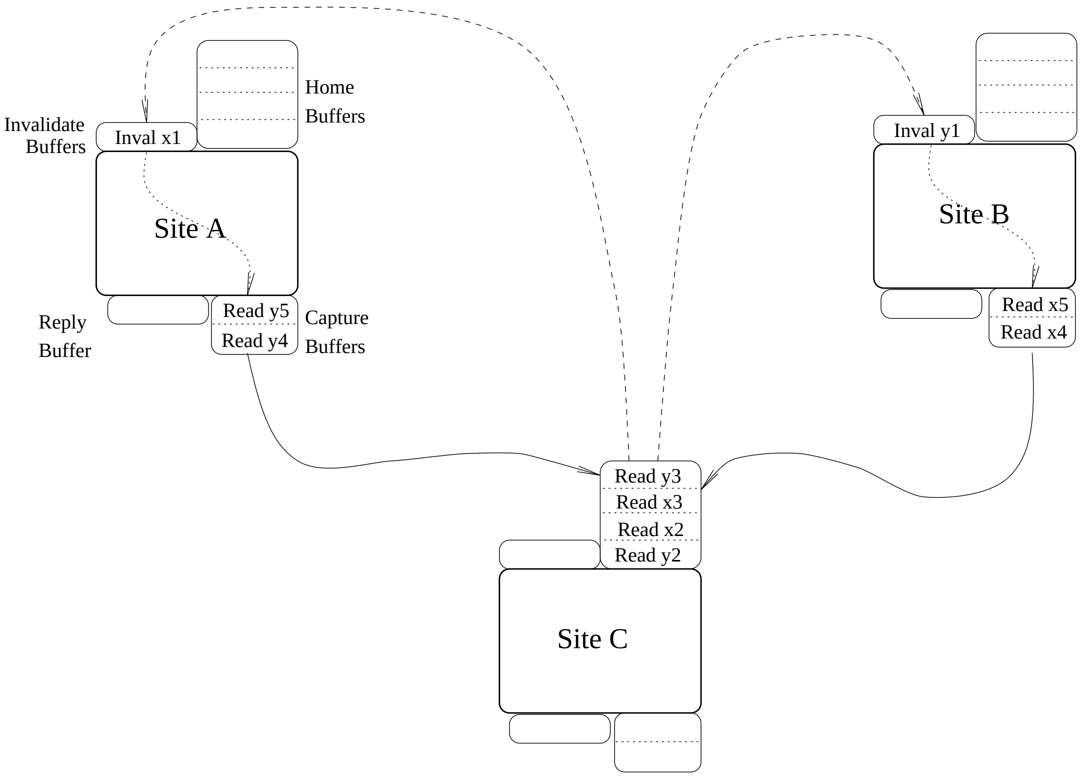


Figure 4: A deadlock caused by limited buffering. Both sites $A$ and $B$ have initiated many read requests to site $C$, and in so doing filled up their own Capture Buffers and $C$'s Home Buffers. These reads require invalidation operations to be performed at both sites $A$ and $B$. Unfortunately, the invalidations cannot complete until Capture Buffers are available for write-backs triggered by the invalidations. In the mean time, the invalidation buffers at sites $A$ and $B$ fill up, and deadlock occurs.

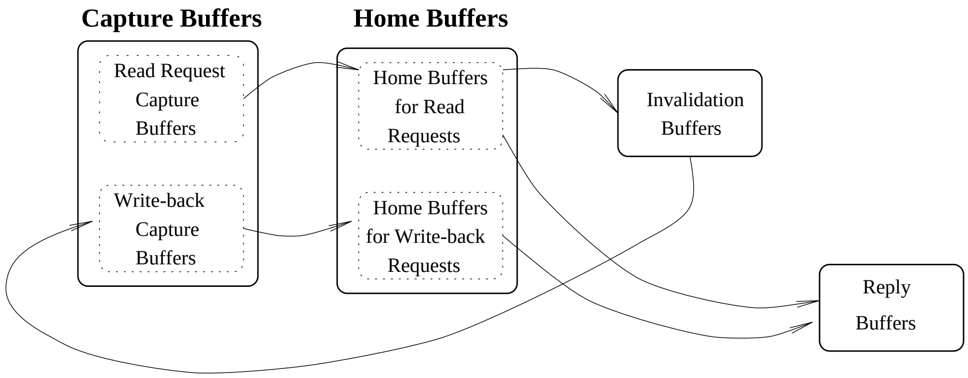


Figure 5: Static dependence graph for the system described in Section 4.7. Each solid box represents a pool of resource, while the arcs indicate potential dependences. The presence of a cycle between Capture Buffers, Home Buffers and Invalidation Buffers, indicates that deadlocks are possible.

If this graph does not contain any cycles, no deadlock due to sharing of hardware resource can occur dynamically. Unfortunately, a cycle exists for the example. This indicates that deadlock can occur if too many concurrent events are outstanding[5].

A solution for the example requires not only separating write-back capture buffers from read capture buffers, but also the home read request buffers from the home write-back request buffers. The deadlock-free dependency graph is shown in Figure 6. Intuitively, this means that write-backs of cache-lines have a dedicated path to memory which read requests cannot block. Reads are only blocked if write-backs triggered by invalidations are blocked, so avoiding write-back blocking avoids deadlock altogether. Splitting buffers into separate categories requires the capability to retry selected types of bus accesses that use buffers that are temporarily unavailable while allowing others to proceed. This selective retry capability is provided in our ACD.

[5]It is possible for a particular implementation with cycles to avoid deadlock by constraining the number of concurrent actions.

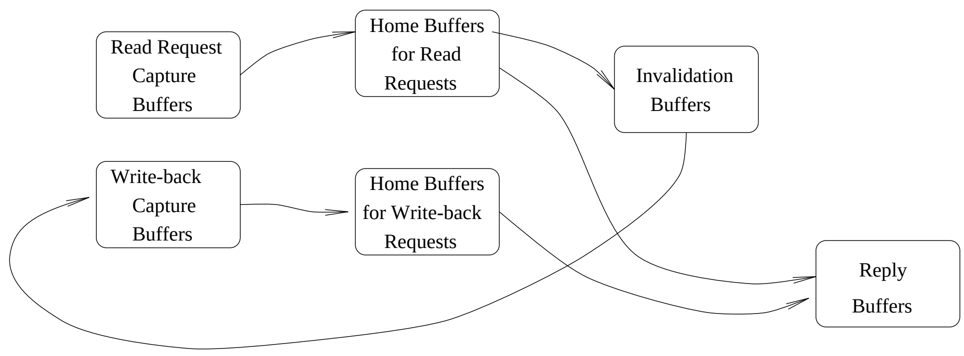


Figure 6: By dividing the Capture Buffers into Read Capture Buffers and Write-back Capture buffers and Home Buffers into Read Request Home Buffers and Write-back Request Home Buffers, we are able to remove the dependence cycle.

### 4.5 Why Our sP Cannot Issue Invalidates

Blocking shared resources include more than just buffers. In **XO**, the sP is capable of executing invalidations (using a Flush command). Unfortunately, an Invalidation is not allowed to complete until any write-back it triggers completes. An sP performing an Invalidation operation can thus be blocked. This would make the sP a blocking resource, leading to deadlocks because of the sP's other roles.

We can see why Flushes could cause a deadlock in the following scenario.

1. Write-back capture buffers on a site are full. Store buffers in the dP are also full.
2. sP initiates Invalidate $x$ which dP has in `Modified` state.
3. dP cannot push-out $x$ because all store buffers are full waiting to go to ACD.

Deadlock occurs because completing the sP's Invalidation depends on the write-back occuring. The write-back cannot occur until the sP services the ACD. The sP cannot service the ACD until its Invalidation is unblocked – Deadlock! This problem can be seen with the static dependence graph of Figure 7, with the sP modeled in the graph because it is used in a blocking fashion to perform the invalidation.

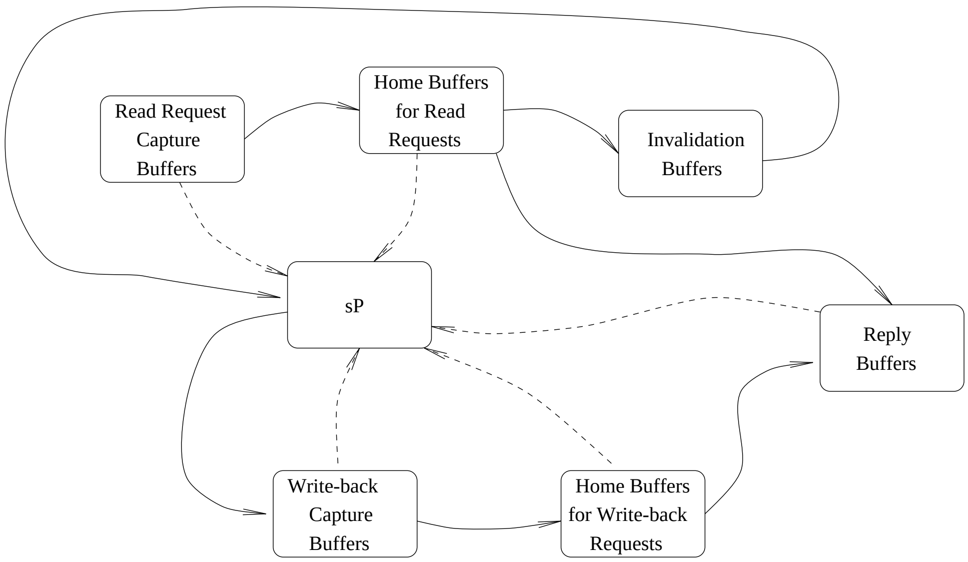


Figure 7: When the sP is used to perform invalidation operations, it becomes a blocking resource that has to be modeled in the static dependence graph. In doing so, many cycles are immediately introduced. The dash arcs represent dependences on sP that utilize the sP in a non-blocking fashion. The addition of the arc from the sP to the Write-back Capture Buffers introduced by the sP performing invalidations causes cycles.

This problem is solved by providing extra hardware that performs the invalidation. This gives rise to the static dependence graph shown in Figure 8 that has no cycles. In **XO**, we have chosen to implement this within the ACD hardware, though it functions independently of all other functionalities of the ACD.

### 4.6 Difficulty in Sharing Cacheable Data between dPs and sP

The snoopy bus protocol should allow cache-to-cache transfer of `Modified` data to a processor performing a Read *without requiring a write-back to main memory.* This requires supporting a `Shared Modified` state, where the data is shared among multiple caches, with one cache which holds it in the `Shared Modified` state[16] given the responsibility of eventually updating main memory.

Without this capability, the sP cannot share cacheable, writable data with the dPs. If the

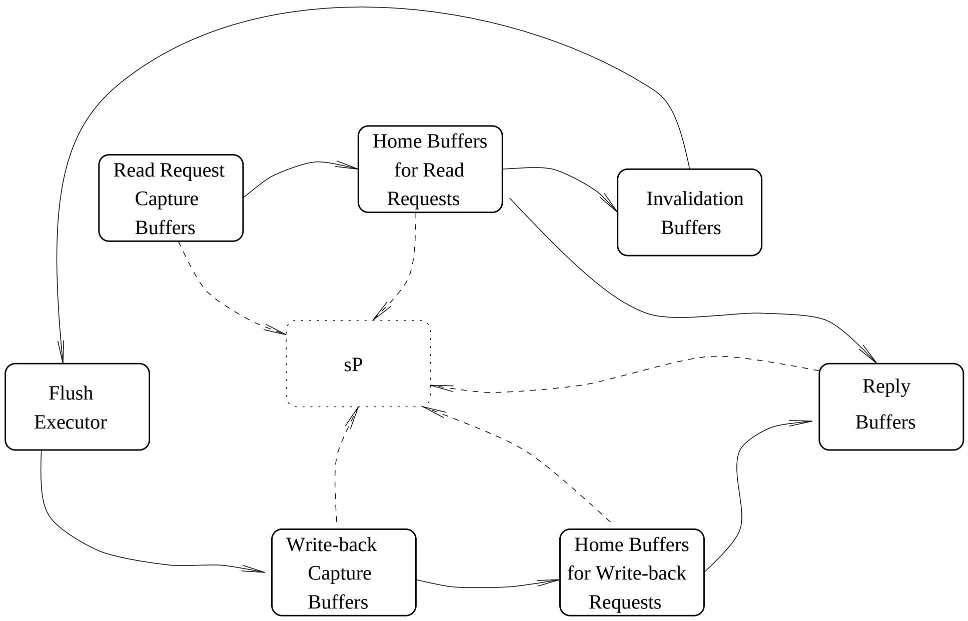


Figure 8: By providing extra hardware that performs the invalidation, the arc linking the SP to write-back capture buffers node is removed. Now no cycle is present in the static dependence graph of the system.

SP reads data that is `Modified` in a DP, it will block until the DP is able to push the `Modified` data out with a write-back. Although this write-back is to local memory, it may be waiting behind write-backs that are going to global memory. Hence, the write-back may be blocked because all the write-back capture buffer resources are in use, and can only be freed by the SP. In the static dependence graph, sharing of writable, cacheable data between the SP and the DPs introduces an arc from the SP to the write-back capture buffers resource, resulting in a cycle reminiscent of that cause by SP performing Invalidations. Thus, data sharing between the SP and DP, if desired, must be through uncached memory.

It is important to note that since operating systems data-structures on SMPs are usually cacheable and writable by any processor, this prohibition on sharing cached, writable memory between the DPs and the SP means that the SP cannot use the same operating system services used by DPs. For this reason, the SP will in most likelihood have to run firmware code without

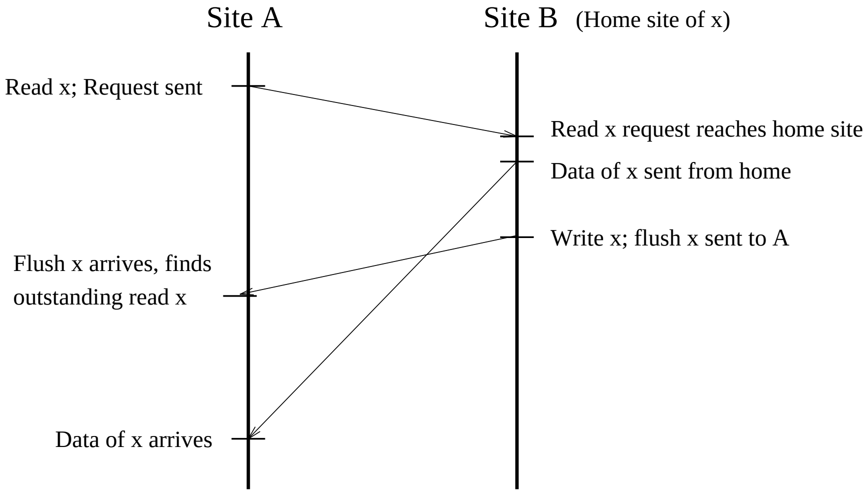


Figure 9: This time-line diagram illustrates the scenario described in Section 4.7. Owing to a non FIFO network, a invalidation request overtakes the data that it is trying to flush.

virtual memory, although it is conceivable that the SP could run a simple operating system of its own that does not share any resource with the main operating system.

### 4.7 Complications due to Non-FIFO Network

A non-FIFO network introduces considerable complexity since messages can get arbitrarily out of order. Although the protocol can use acknowledgments to serialize its actions and hence avoid the effects of a non-FIFO network, such a solution is unattractive because of the additional overhead and latency. Better solutions use more states in the protocol to handle possible out-of-order situations, sending extra messages only when necessary. Keeping track of outstanding operations, which is required in the example of Section 4.2, is also necessary here. Let us consider the scenario below which is related to the example in Section 4.2 (see also Figure 9).

1. $P_A$ sends a Read $x$ to site $B$, home of $x$.

2. $B$ sends back data $x$.

3. $B$ writes $x$, sending $A$ Invalidate $x$.

4. Invalidation arrives before data $x$!!!

5. Data $x$ arrives.

Because the arrivals of Data $x$ and Invalidate $x$ at $A$ are reversed, it is not correct to process them in their arrival order, since the home processor might then believe that site $A$ does not have a copy of $x$. Simply having the invalidation wait while there is an outstanding operation will potentially deadlock since it is possible that the Invalidate *should* come before the Data and the Data won't come until the Invalidate completes as in the example of Section 4.2.

If there is an outstanding operation to the same cache-line on site $A$, an invalidate can wait at site $A$. The home site $B$, upon receiving a Read $x$ request from a site to which it has already sent an invalidation *must* reply with a Retry Read $x$. If this retry gets back to site $A$ and finds a pending invalidation, the invalidation can then be allowed to complete without being put on the bus. Should the sequence of events be that described in Section 4.2, the reply data returns to find a waiting invalidation. The outstanding read is first completed, then the Invalidation is done on the bus. Proving this solution works is involved because other messages can overtake each other. For example, the Retry Read $x$ can overtake the Invalidate $x$. Another document[15] describes cache coherence protocols we intend to implement on **XO** and proofs of their logical correctness.

# 5 Conclusion

In this paper, we have presented the design of a CCDSM system layered on top of a message passing machine. The design relies on a simple address capture device and uses one of the processors in each SMP for protocol processing. Based on the problems we have encountered, we draw the following conclusions:

**Processors and their Snoopy Buses:**

By necessity, the implementation of a CCDSM system is layered on top of a particular snoopy bus protocol. The CCDSM implementations can benefit from the snoopy bus having the following properties.

- Invalidations should not be blocked by outstanding operations to the same cache-line. (Section 4.2 and Section 4.3)
- Cache-to-cache transfer of `Shared` as well as `Modified` data should be supported. (Section 4.3)
- `Shared Modified` state[16], which allows cache-to-cache transfers for Reads of modified cache-lines without write-backs to memory should be supported. (Section 4.6)
- The snoopy bus should be a split-phase bus (Section 4.1).

Though it is possible to work around snoopy bus protocol and processor oversights with software running on the sP, it would be preferable if we did not have to waste sP cycles to do so.

**Buffering:**

To avoid deadlock, buffers should be partitioned into different classes according to functionality. For example, for invalidation based protocol, Read Requests and Write-back Requests should not share a common buffer pool. (Section 4.4). To avoid deadlocks, the service processor should not issue Invalidations. (Section 4.5) Static dependence graphs are a good way to detect potential deadlocks due to buffer sharing.

**Network:**

Protocols are generally much more complicated for a non-FIFO network because messages can get out of order. The increase number of messages may negate the performance advantage of non-FIFO networks. It is also possible to build a layer of abstraction on top of non-FIFO networks so that they appear to be FIFO. While this simplifies the protocol, it may take even more messages than dealing with the non-FIFO problems directly in the protocol.

Our preliminary estimates indicate that **XO**'s CCDSM will perform well for programs with good hit rates. Like all CCDSM machines, the performance of coherent shared memory on **XO** depends mostly on the global memory miss ratios. Thus, a miss penalty that is three or four times worse than that of an aggressive implementation can still give good performance for a range of programs. This is very encouraging, considering the cost and experimental nature of our implementation. Of course the actual performance can only be determined for sure after we get the machine built, and up and running. We are looking forward to a real **XO** machine on which to experiment and measure performance.

## 6 Acknowledgments

Supressed

## References

[1] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocesors*. Kluwer Academic Publishers, 1991.

[2] Supressed Supressed Supressed

[3] Supressed Supressed Suppressed

[4] H. Burkhardt III, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR1 computer system. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, February 1992.

[5] J. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of The 10th Annual International Symposium on Computer Architecture*, pages 124–131, June 1983.

[6] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J. P. Singh, R. Simoni, K. Charachorloo, D. Nakashira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architecture Support for Programming Languages and Operating Systems, San Jose, CA*, pages 274 – 285, October 1994.

[7] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.-H. Lim. Intergrating message-passing and shared-memory: Early experience. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego*, pages 54–63, 1993.

[8] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture, Chicago, Il*, pages 302–313, April 1994.

[9] D. Lenoski, J. Laudon, K. Charachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocesor. In *Proceedings of The 17th Annual International Symposium on Computer Architecture, Seattle, WA*, pages 148–159, 1990.

[10] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of The 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia*, pages 92–103, May 1992.

[11] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, September 86. (Also as YALE/DCS/RR-492).

[12] K. Li and R. Schaefer. SHIVA: An operating system transforming a hypercube into a shared-memory machine. CS-TR 217-89, Princeton University, Department of Computer Science, April 1989.

[13] K. Li, M. Stumm, D. Wortman, and S. Zhou. Shared virtual memory accommodating heterogeneity. CS-TR 210-89, Princeton University, Department of Computer Science, February 1989.

[14] suppressed suppressed suppressed

[15] supressed supressed supressed

[16] C. P. Thacker, L. Stewart, and J. E.H. Satterthwaite. Firefly: a multiprocessor workstation. DEC/SRC-TR 23, Digital Equipment Corporation, Systems Research Center, December 1987.

# Contents