

Coherent Block Data Transfer in the FLASH Multiprocessor

John Heinlein, Robert P. Bosch, Jr., Kourosh Gharachorloo[†],
Mendel Rosenblum, and Anoop Gupta

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

[†]DEC Western Research Laboratory
250 University Avenue
Palo Alto, CA 94301

Abstract

A key goal of the Stanford FLASH project is to explore the integration of multiple communication protocols in a single multiprocessor architecture. To achieve this goal, FLASH includes a programmable node controller called MAGIC, which contains an embedded protocol processor capable of implementing multiple protocols. In this paper we present a specialized protocol for block data transfer integrated with a conventional cache coherence protocol. Block transfer forms the basis for message passing implementations on top of shared memory, occurs in important workloads such as databases, and is frequently used by the operating system. We discuss the issues that arise in designing a fully integrated protocol and its interactions with cache coherence. Using microbenchmarks, MPI communication primitives, and an application running on the operating system, we compare our protocol with standard bcopy and bcopy augmented with prefetches. Our results show that integrated block transfer can accelerate communication between nodes while off-loading the task from the main processor, utilizing the network more efficiently, and reducing the associated cache pollution. Given the aggressive support for prefetching in FLASH, prefetched bcopy is able to achieve competitive performance in many cases but lacks the other three advantages of our protocol.

1 Introduction

The Stanford FLASH Multiprocessor has been designed to support multiple data communication protocols by providing each node with a programmable protocol processor in its node controller. This paper explores the design and performance of an integrated cache coherence and block data transfer protocol using the hardware in FLASH. Off-loading block transfer operations to the node's controller can result in three main benefits: (i) reduction in the time taken for the transfer, since the controller is closer to the main memory and can access it more efficiently; (ii) reduction in the pollution introduced in the processors' caches; and (iii) overlap between computation done by the main processor and the block transfer done by the controller.

Accelerating block transfer is important because large data transfers are common in many classes of applications. For example, commercial applications such as databases

move large chunks of data between the processors' memories [RBH⁺95]. In fact, the operating system itself may spend a significant fraction of its time doing memory transfer [CHR⁺95]. For these applications, a block transfer protocol must provide full support for virtual memory and full integration with cache coherent shared memory. While block transfer protocols have been proposed for shared memory machines [ABC⁺95, Cray93], the lack of virtual memory support and restrictions on caching of data make these protocols unsuitable for broad use within a shared memory operating system.

Our previous work in this area described techniques to enable efficient, protected integration of block transfer with virtual memory [HGD⁺94]. In this paper, we address the additional issues which arise from fully integrating the protocol with shared memory. Specifically, integration with shared memory requires the protocol to support *all* caching cases that may be encountered for block transfer data, such as lines residing in remote processor caches. This integration allows us to export a general interface to the system, essentially a drop-in replacement for the standard bcopy call.

While previous studies have focused on evaluating the benefits of block transfer in the context of isolated scientific applications [WSH94], our study uses workloads that include operating system effects. This is made possible by new simulation technology capable of simulating applications as well as the complete operating system [RHW⁺95]. We first present an analysis of the protocol in isolation, exploring in detail the factors which limit transfer performance. We then examine the performance of Message Passing Interface (MPI) communication primitives [MPI93] using our protocol to perform the data movement for application messages. Finally we study a UNIX application in which the operating system kernel uses the protocol to accelerate block transfers.

In our experiments we compare the performance of our accelerated version of bcopy with the standard UNIX implementation based on processor loads and stores. We also compare our protocol with processor bcopy augmented with prefetches. We explore a variety of differences between processor-based techniques and our implementation of block transfer on the node controller.

The paper is organized as follows: Section 2 provides an overview of FLASH and its programmable node controller.

Section 3 describes the coherent block transfer primitive and its interface to user-level applications and the operating system. Section 4 describes the detailed protocol, focusing on support for integration with shared memory, as well as the various trade-offs and complexities in the design. Section 5 describes our experimental environment and the performance of our protocol at a low level. Section 6 presents performance results for various benchmarks. Section 7 presents related work and discussion, and we conclude in Section 8.

2 FLASH Overview

FLASH [KOH⁺94] (FLEXible Architecture for SHared memory) is a cache coherent shared memory multiprocessor currently under development at Stanford University. It consists of a large array of processing nodes connected by a high-bandwidth, low-latency network. Each node contains a single MIPS R10000 processor [SGI95], its associated caches, a portion of the machine's distributed shared memory, a network port, a PCI I/O interface, and a custom node controller called MAGIC (Memory And General Interconnect Controller). MAGIC includes a programmable *protocol processor* (PP) optimized for executing protocol operations, and dedicated data paths that tightly connect the various components on the node.

FLASH accomplishes protocol communication by sending low-level network *messages* between nodes. Messages may arrive at MAGIC from different external sources: the processor, the network, and the I/O subsystems. Additionally, there is an internal source of messages, the *software queue*, which represents a list of protocol tasks that have been delayed by MAGIC under software control. The software queue is used extensively in our block data transfer protocol.

When a message arrives at the MAGIC chip, the appropriate software protocol code, called a *handler*, is invoked on the PP. The handler decides what actions to take based on the headers of the message and protocol state, which is stored in a portion of main memory. MAGIC contains its own protocol instruction and data caches to improve handler performance. Cache line-sized (128 byte) *data buffers* are used to stage data through MAGIC in a pipelined fashion, allowing MAGIC to overlap the control processing and the data transfer associated with servicing a message. The parallel handling of data and control plays an important role in efficient protocol processing.

3 Coherent Block Transfer

A fast block transfer primitive can be important for both operating systems and applications that exhibit large data movement. In addition, it can serve as a building block for message passing protocols such as the Message Passing Interface (MPI) standard. Integrating this protocol with cache coherent shared memory is critical to enable applications to seamlessly utilize block transfer in shared memory programs. Our application programming interface for

block data transfer is a natural one for this environment—it is an accelerated version of the well-known *bcopy* call:

```
void fbcopy(char* source, char* dest,
           int length, uint64* flag);
```

Fbcopy copies the memory specified by the source parameter (called the source buffer) to the address specified by the destination parameter (called the destination buffer). Length specifies the number of data bytes to transfer. A completion flag is also supplied, which is updated by MAGIC when the transfer is complete. This allows the main processor to perform other calculations if desired, then later test the flag for completion.

Although the programming interface for the block transfer is simple, an efficient implementation of this interface is nontrivial. First, shared memory allows the source and destination buffers to be cached in any node of the system. For the source buffer this means that the current value of the data might be in another node's cache; when the transfer is carried out the protocol must retrieve the data from that node. Similarly, nodes caching the destination buffer must be notified when the block transfer modifies the buffer so they can invalidate their old copies.

In addition, the source and the destination buffers may span multiple virtual memory pages, resulting in non-contiguous physical pages. Furthermore, the physical pages may be scattered across different nodes of the machine for many reasons: process and page migration for load balancing, sharing of data among cooperating processes running on different nodes, and operating systems which can allocate pages on remote nodes [CRD⁺95]. As a result, it is quite possible in practice for the node requesting the transfer, called the *initiator*, to be distinct from the home nodes of the source buffer pages. We refer to this issue as *remote home* support. The added complexity for handling this case arises because coherent operations on a line must consult its directory state (located at its home node).

4 Accelerated Block Transfer Implementation

This section describes the basic implementation issues for the accelerated block transfer protocol. Sections 4.1–4.3 present the fundamentals of how we initiate and perform block transfers while maintaining coherence of the transfer data. Section 4.4 describes how the protocol handles scenarios in which the source node is different from the initiator node. Section 4.5 presents other details which arise in the design of the protocol.

4.1 Initiation

Programmers invoke *fbcopy* using the library call defined in Section 3, which describes the transfer to the MAGIC chip. The initiation sequence is similar to communicating with MAGIC as a memory-mapped device. Initiation consists of a series of uncached writes indicating the command type and its arguments, followed by an uncached read that allows MAGIC to inform the processor whether the sequence was accepted. An *fbcopy* call transferring one

page requires four uncached writes to MAGIC. These writes contain the length of the transfer and the physical addresses of the source buffer, destination buffer, and completion flag. Longer transfers require more writes for the additional pages involved.

The way physical addresses are described to MAGIC depends on whether the message is initiated by the user or the kernel. For operating system `fbcopy` requests, the physical addresses can be computed and passed directly to MAGIC. For initiation from user level, where only virtual addresses are known, the operating system establishes an uncached *shadow mapping* of the address space. Shadow mapping addresses are translated the same as normal ones, but carry special attributes indicating to MAGIC that the address is part of the initiation sequence. We use this mapping to communicate authentic physical addresses without system calls. In [HGD+94] we describe in detail this initiation sequence as well as several techniques for interacting with virtual memory in a protected manner.

4.2 Block Transfer Fundamentals

The block transfer begins after the initiation sequence is accepted by MAGIC. The *transfer handler* at the source node sends the data to the destination node in a series of cache line-sized *messages*. For each message, the PP loads a line from the source buffer into a MAGIC data buffer, adds header information, and sends it into the network. MAGIC labels each line with its destination address, which is provided during initiation. At the destination node, each message invokes a handler to store the accompanying data and maintain a count of lines received.

Transferring data as a series of cache line-sized messages instead of a single large message has several key advantages in FLASH: (i) it allows the service of other cache coherence and block transfer operations to be interleaved with a block transfer; (ii) MAGIC is optimized for the transfer of cache lines; (iii) it leverages the deadlock avoidance mechanisms provided by MAGIC; and (iv) the chances of contention within the network are reduced. Furthermore, the long cache lines in FLASH (128 bytes) amortize the network and processing overheads of dealing with a single cache line at a time.

Each invocation of the transfer handler sends several messages, updates the transfer state, and reschedules itself on the software queue to continue the transfer. Sending multiple lines in each invocation, an optimization we refer to as *chunking*, amortizes the overhead of starting the transfer handler and saving and restoring the transfer state [HGD+94]. For a 4 KB transfer, sending four lines in each handler invocation improves transfer performance 25–40% over sending a single line with each handler.

4.3 Cache Coherence for Block Transfer Data

Providing efficient and transparent cache coherence is a key aspect of making primitives such as accelerated block transfer easy to use in a shared memory environment.

Since the source and destination data may be cached anywhere in the system, the block transfer protocol must incorporate a subset of the cache coherence protocol to efficiently obtain the latest data and maintain coherence of the source and destination buffers. This section describes the mechanisms used to achieve this functionality and details the performance trade-offs for various implementation alternatives. The description assumes that the data transfer occurs directly from the home node of the source buffer to the home node of the destination buffer; Section 4.4 describes the general case.

Coherence for Source Buffer Data

There are three cases to consider for coherence of the source buffer: the data is clean, the data is dirty in the local processor's cache, or the data is dirty at a remote node. We distinguish between the first two cases because the FLASH design requires the protocol to explicitly request a dirty line from the local processor, in contrast to snoopy bus-based designs where the same request checks both the memory and the processors' caches.

If the source node has a valid copy of the data, it can be retrieved from the cache or memory by the transfer handler and sent to the destination node. However, when the data is dirty at a remote node, the task of retrieving the valid data requires remote communication. The simplest solution is to pause the transfer, retrieve the line from the remote node, send it to the destination, and then proceed with the transfer. However, this approach fails to exploit any parallelism in retrieving the data for multiple lines.

A more efficient solution is to accomplish the transfer in two distinct phases, one that collects the dirty data of the source buffer from remote nodes and one that sends the data to the destination [KA93]. This *collect-and-send* technique allows the collect phase to retrieve multiple lines simultaneously. For a machine such as FLASH that performs transfers a cache line at a time, it is also possible to *pipeline* the collect and send phases instead of keeping the phases distinct. When the transfer handler detects that a line is dirty on a remote node, it requests the line and proceeds with the transfer. The response eventually arrives back at the source node and triggers a handler that sends the line to the destination and writes the data back to memory. We refer to this as *pipelined collect-and-send*.

An alternative to the pipelined collect-and-send approach is to forward the data directly from the dirty remote node to the destination node (similar in philosophy to the optimization in DASH for the three-hop dirty remote case [LLG+90]). We refer to this as the *forwarding* approach. The request from the source node to the dirty remote node carries a single word of data that specifies the destination address. The dirty node forwards the data to the destination node and sends a writeback to the source node. Figure 4.1 illustrates these two alternatives.

Compared to pipelined collect-and-send, forwarding reduces protocol overhead since the source node can perform an ordinary writeback without remapping the message for sending to the destination node. On the other hand,

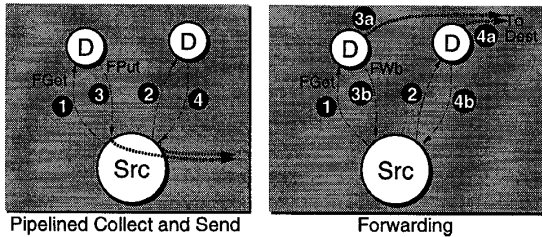


FIGURE 4.1. Source side coherence alternatives. In both cases the source node (Src) sends fbcopy Get messages (FGet) to fetch dirty cache lines from dirty remote nodes (D).

there may be higher occupancy at the network interface for the dirty node since forwarding leads to two outgoing messages (the writeback to the source node plus the forwarded message to the destination node), compared to just one message for the collect-and-send case. Therefore, the choice between the two approaches depends on the relative speeds of the network and the protocol processor. As we will see in Section 5.2, the forwarding approach is superior for network bandwidths of 400 MB/s per link or greater (assuming a 100 MHz PP).

Coherence for Destination Buffer Data

The destination node must coherently modify the data in the destination buffer, which requires invalidating all stale copies of the data. Rather than first retrieving the line from remote nodes as was done in source buffer coherence, the handler that receives the line can simply write it into the destination buffer in memory and then eliminate any copies in processor caches. Figure 4.2 shows the actions of the protocol. If there are any clean copies of the line, the handler sends an invalidation request to each copy.

Handling the coherence for a dirty line is a bit more subtle. Normally, in a cache coherent machine data is *never* modified by any node except the one that currently has exclusive ownership of the line. Since our protocol writes the contents of the entire destination buffer line *at its home*, we can violate this assumption to improve the performance of our protocol by writing the line without acquiring ownership in advance. We then send a special flavor of *dirty invalidation* to the remote node, telling its cache to discard its dirty copy of the line *without writing it back*. However, there is a possible race between the arrival of the invalidation message and a writeback of the line from the remote cache. If a writeback occurs before the invalidation arrives, the writeback will squash the block transfer data that has been written to memory.¹ To detect this race, the invalidation message checks the response from the remote cache. If

1. A similar race can occur even if the data is cached at the same node. This occurs because, as an optimization, the writeback never checks the directory state before writing memory, since only one exclusive copy should ever exist in the system and only an exclusive copy can generate a writeback. Since this race is infrequent, it is more efficient to check for the race condition than to make the writeback handler more conservative.

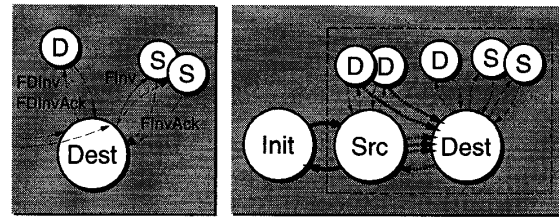


FIGURE 4.2. (Left) Destination side coherence. The destination node (Dest) sends invalidations to nodes that have a destination line cached dirty (D) or shared (S).

FIGURE 4.3. (Right) Complete fbcopy protocol. Transfers initiated at a node other than the home of the source buffer are delegated to that node. The actions inside the dashed box are present even when remote initiation is not required.

the line has already been written back, a special negative acknowledgment message is sent back to the destination node, triggering a retry of the line from the source.

4.4 Support for Remote Homes

So far we have assumed that the transfer is carried out by the initiator node. However, as described in Section 3, the initiator node may be distinct from the home nodes of the source buffer pages. Since the cache coherence protocol must consult the directory state to determine the appropriate coherence action, the initiator must communicate with the home for *each line* of the transfer. The naïve approach of requesting each line individually leads to increased traffic in the network and prevents the home node from leveraging the chunking optimization to increase performance. In addition, this implementation is substantially more complicated, since it introduces many more special cases and race conditions.

A more efficient approach is to delegate the transfer to the home node and have it carry out the transfer on the initiator's behalf. After delegating the transfer to the home node, the protocol behaves exactly as if the block transfer were initiated there. The only difference is that a completion signal is sent from the home node back to the initiator node when the transfer is complete. Even though there is an overhead of approximately 2.5 μ s to delegate the transfer from the initiator node to the home node, this approach leads to better overall performance for transfers longer than a few lines. Figure 4.3 illustrates a remote home scenario, showing how it can be achieved without significant additional complexity.

4.5 Implementation Details

Race Conditions: Since the processors continue to issue requests while the transfer occurs, cache coherence events may affect lines involved in the transfer. The case where the writeback of a dirty line overwrites the transfer data

(described in Section 4.3) is a prime example. We implement an efficient retry mechanism to keep track of lines which experience problems during transfer. MAGIC uses this mechanism to regenerate requests for these lines.

Deadlock: The block transfer protocol leverages deadlock avoidance mechanisms already present in FLASH for cache coherence. For example, deadlock could occur if outgoing network queues were allowed to fill and the protocol processor blocked waiting for them to empty. The chunked transfer handler could cause such a scenario, but by monitoring the available space in outgoing queues it prevents deadlock from occurring. The cache coherence protocol has similar potential deadlock cases (such as the handler which sends invalidations for widely shared lines) but prevents deadlocks in the same manner. These issues are covered in more detail in [KOH⁺94].

Copies within a node: Our goal in providing block transfer is to accelerate copies between two nodes, so we do not provide support for copies in which both buffers reside on the same node. Since local memory access is highly optimized, our experience suggests that the processor is more appropriate for these *intranode* copies. During transfer initiation, the protocol processor checks the pages involved in the transfer to see if they are located on the same node. If so, initiation returns a result code indicating that the processor should perform the copy. By encapsulating this functionality in the `fbcopy` library call, the user can simply invoke `fbcopy` without regard for page placement.

Protocol code size: The total amount of handler code added for the `fbcopy` protocol is about 19 KB. In contrast, the base cache coherence protocol in FLASH is about 44 KB. Although the block transfer protocol code is only about 40% of the cache coherence protocol in size, it is much larger than we had initially expected. Intuitively, since the `fbcopy` protocol also provides cache coherence, it may seem that many cache coherence handlers can be leveraged for `fbcopy` as well. Unfortunately, in many cases these handlers take almost the right action needed by `fbcopy`, but not exactly the right action. We had the option of making these handlers more general (fostering direct code sharing), but that would have slowed them down. We did not consider slowing down the common case for cache coherence to be an acceptable option.

5 Low-Level Performance Analysis

In this section we consider the performance of the `fbcopy` protocol at a low level. We begin by describing the simulation environment used in our experiments. We then provide a detailed analysis of the block transfer protocol to illustrate the various factors that determine the overall performance of this protocol on FLASH.

5.1 Simulation Environment

The simulation environment used in this study provides a complete model of the FLASH system, allowing us to accurately study its performance running a variety of workloads. We use the FlashLite memory system simulator which models the performance of MAGIC in detail. We simulate the MAGIC chip running at its target frequency of 100 MHz (in our discussion, *cycles* refer to 10 ns MAGIC cycles). The network is assumed to have a link bandwidth of 400 MB/s; header overhead leaves 320 MB/s of useful bandwidth. Simulation of the main processors and I/O system is provided by the SimOS environment [RHW⁺95].

The accelerated block transfer protocol and the base cache coherence protocol are specified in C and compiled to PP instructions. The results presented in this paper are based on a cycle-accurate emulation of the actual protocol code. Even though the automatically generated code is fairly efficient, higher performance can be achieved by hand tuning the compiled code—we address this issue in Section 6.

5.2 Transfer Bottlenecks

The performance of the block transfer protocol is determined by one of three possible bottlenecks: source side processing, destination side processing, or network bandwidth. To illustrate these factors concretely, Figures 5.1 and 5.2 show timelines of the handler occupancies for block transfers in two different caching scenarios. The results are from the simulation of a 4 KB transfer initiated at the source node, using the forwarding protocol and a maximum chunk size of 4. We have divided the handlers into four categories: (i) those that handle initiation, retry, and completion flag update (labeled *Init/Retry/Flag*); (ii) those that send and receive the data messages (*Transfer*); (iii) those that handle coherence for remotely cached copies (*Coherence*); and (iv) those that process corner case negative acknowledgments (*Nak*). We use thin vertical lines to indicate the end of each handler and show the duration in cycles for handlers that last for more than 100 cycles.

Figure 5.1 shows a transfer in which both source and destination buffers are uncached. The initiation phase consists of four uncached writes followed by an uncached read. The time line shows the initiation phase on the source node followed by the communication of the “envelope” to the destination, lasting about 3.5 μ s. Once the destination node replies, the source node repeatedly invokes the chunked transfer handler, each time sending 4 lines (thus 8 invocations are needed to send the entire 4 KB). The destination receives each line individually, acknowledging the source when the transfer is complete. The total time for the transaction is about 24 μ s, with the actual transfer occupying about 16 μ s. This corresponds to peak transfer bandwidth of about 276 MB/s (out of the 320 MB/s supported by the network) and an average bandwidth (including initiation and completion overheads) of 170 MB/s. Note the

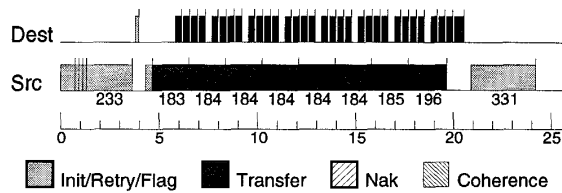


FIGURE 5.1. Time line of a two-node block transfer; buffers are uncached. The axis reads time in microseconds. Other numbers reflect long handler durations in MAGIC cycles.

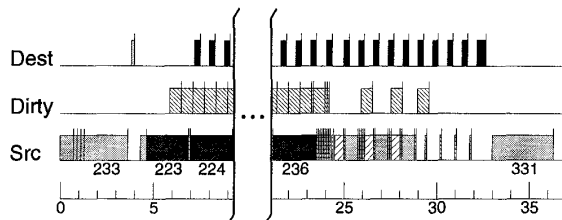


FIGURE 5.2. Time line of a block transfer with the source buffer dirty on a remote node. Queue limitations cause negative acknowledgments forcing three lines to be retried.

gaps between invocations of the destination handler due to the faster processing of lines at the destination.

The time line in Figure 5.2 shows a transfer using the forwarding protocol described in Section 4.3 in which the source buffer is cached dirty at a remote node and the destination buffer is uncached. For each line, the transfer handler sends a request to the dirty node to forward the line directly to the destination. The dirty node also sends a writeback to the source node (visible as the short handlers on the source). The network is the bottleneck in this case: the dirty node's protocol processor sends messages faster (two messages every 62 cycles) than the network can absorb them (one message every 40 cycles). This causes the outgoing network queue to fill three times during the transfer. Each time, the handler detects the full outgoing queue and sends a NAK to the sender (visible on the dirty node in Figure 5.2 as the three short handlers between 23–25 μ s). MAGIC retries these requests successfully (between 26–30 μ s on the dirty node).

As discussed in Section 4.3, the decision to use forwarding instead of pipelined collect-and-send (PC&S) depends on the relative cost of protocol processing and network bandwidth. Figure 5.3 shows the simulation results for these two alternatives in FLASH, varying only the network bandwidth and source caching state. For a slow network, the network overhead of the extra messages in the forwarding approach is greater than the protocol processing overhead incurred by PC&S. At higher bandwidth, these extra messages become less expensive and forwarding becomes more efficient. In this simulation, the cross-over point is between 200 and 400 MB/s. Since we model a 400 MB/s network, we use the forwarding approach.

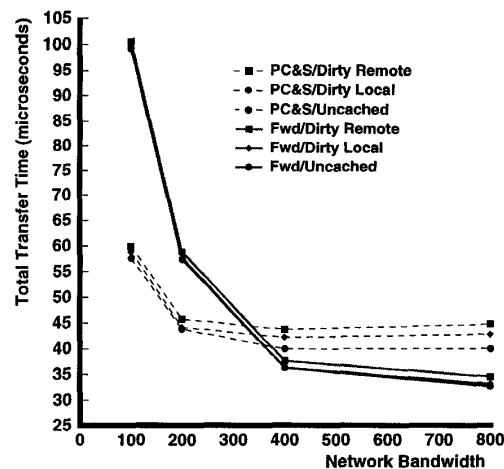


FIGURE 5.3. Forwarding (Fwd) vs. pipelined collect-and-send (PC&S) for different source caching states.

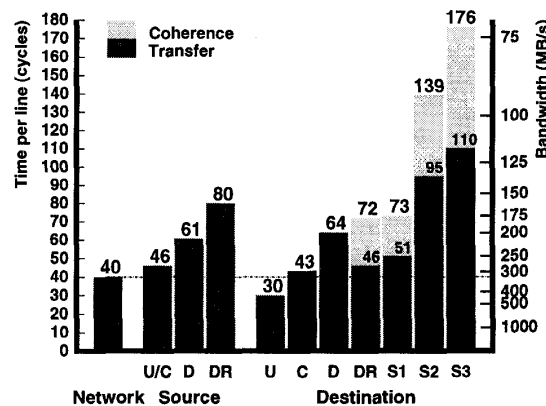


FIGURE 5.4. Fbcopy transfer source, destination, and network bottlenecks. Buffer caching states: uncached (U); cached (C); dirty (D); dirty remote (DR); clean with 1-3 sharers (S1-3).

So far we have discussed two specific caching scenarios for source and destination side buffers. Figure 5.4 shows the performance for other possible situations. This figure indicates whether the performance limit in each case is network throughput, source side handlers, or destination side handlers. As an example, 46 cycles for the uncached/clean case on the source side corresponds to one fourth of the chunked handler occupancy of 184 cycles (shown in Figure 5.1). For a given caching state, the peak transfer rate is limited by the slowest of these three components. For the destination side, the coherence portion of the bar shows the time to handle invalidation acknowledgments from remote copies. One interesting effect is the sudden jump in destination node transfer handler cost for two or

more remote sharers (between the S1 and S2 bars in Figure 5.4). This is an artifact of our cache coherence directory format, which stores the first sharer in the main entry but places subsequent sharers on a linked list. The remote sharer cases are also impacted by the increasing coherence overhead of 22 cycles per invalidation acknowledgment.

6 Benchmarks and Performance Comparison

In this section we extend the results in the previous section by studying the accelerated block transfer protocol in the context of microbenchmarks and applications. In these simulations we compare *fbcopy* with *processor copy*—we consider both regular *bcopy* and *bcopy* augmented with prefetch instructions [MG91]. Prefetched *bcopy* can be very effective in FLASH due to its low latency network and highly optimized memory system, including a processor with an aggressive prefetching implementation. FLASH also contains long 128 byte cache lines, which can achieve many of the benefits of block transfer [WSH94].

In the first section, we study the block transfer implementations without operating system effects. In subsequent sections we show experiments using the IRIX Version 5.3 operating system from Silicon Graphics, Inc., which we modified to use the accelerated block transfer protocol.

6.1 Microbenchmark Analysis

We first compare the performance of *fbcopy* against the other transfer primitives by simulating the transfer of a 4 KB page between two nodes. As described in the previous section (and as shown in Figure 5.4) the performance of coherent block transfer depends on the caching state of the involved lines; we focus on several cases which tend to be more common in applications using block transfer. For simplicity, we assume all the lines of a particular buffer are in the same caching state.

There is an additional caching effect which impacts processor copy since it must bring the buffers into the cache. If this causes spills—which is likely—writebacks or replacement hints are generated, which may increase the latency of the block transfer operation. To model this effect, we first warm the caches with an equal mix of clean and dirty lines then cache the transfer buffers as appropriate for the experiment. Since *fbcopy* does not bring the buffers into the processor cache, it is immune to this effect.

Normal Protocol Processing

Table 6.1 shows the time required by the source node to perform the transfer in the scenarios described above. *Fbcopy* significantly outperforms standard *bcopy*, executing the transfers more than twice as fast in most cases. *Bcopy* incurs expensive cache misses to copy the data, which poorly utilize the processor during the transfer. Prefetched *bcopy* largely avoids the miss penalties of *bcopy* by acquiring the lines in advance of their use. This technique is quite effective in improving transfer perfor-

	Source caching state	Destination caching state				Net. Traffic
		Normal PP		Faster PP		
		U/L	R	U/L	R	
B	U	65.8	67.6	62.2	62.5	8 KB
	DL	47.7	52.0	45.0	45.4	8 KB
	DR	105.0	106.8	98.1	98.3	12 KB
P	U	28.8	30.0	26.5	27.3	8 KB
	DL	20.5	24.0	19.5	21.0	8 KB
	DR	37.3	38.6	35.2	36.1	12 KB
F	U	23.3	33.4	19.0	21.1	4 KB
	DL	28.7	33.7	19.2	21.3	4 KB
	DR	36.7	37.7	33.7	34.6	8 KB

TABLE 6.1. Latency in microseconds of a 4 KB transfer for *bcopy* (B), prefetched *bcopy* (P) and *fbcopy* (F). Caching states: uncached (U), shared local/remote (L/R), dirty local/remote (DL/DR).

mance, so much so that in some cases prefetched *bcopy* finishes the transfer faster than *fbcopy*.

However, prefetching ties up the processor for the entire duration of the transfer in order to achieve this performance, rendering it unable to perform other work. By delegating the block transfer to MAGIC, *fbcopy* frees the processor to continue working while the transfer proceeds.

Fbcopy also uses the network more efficiently. Processor copy fetches the entire destination buffer even though it will immediately be overwritten. In contrast, since *fbcopy* does not acquire ownership of the destination buffer at the source, these lines never cross the network. Instead, because *fbcopy* communicates directly between memory systems, network traffic is only incurred for the data actually being transferred. The rightmost column of Table 6.1 shows the total traffic caused by the different implementations for a 4 KB transfer.

Consumption of Transfer Data

Though transfer time is important, the performance of block transfer is not based solely on that factor. It may also include the time needed to “consume” the transfer data at the destination node. To quantify this effect, we read the values in the buffer at the destination node.

In processor copy, the destination buffer ends up in the source node’s cache. When the buffer is consumed, the destination node experiences dirty remote cache misses. Reading the buffer in this case takes 70 μ s. Because *fbcopy* deposits the data in the memory of the destination node, only local misses are taken. As a result, the consumption phase after *fbcopy* lasts only 20 μ s. Furthermore, in processor copy the data involved in the transfer may replace unrelated lines still in use by the processor. This effect is particularly significant in the primary cache, which can be completely filled by a mild-sized block transfer. The impact of this *cache pollution* is application dependent, based on the amount of subsequent reuse of the replaced

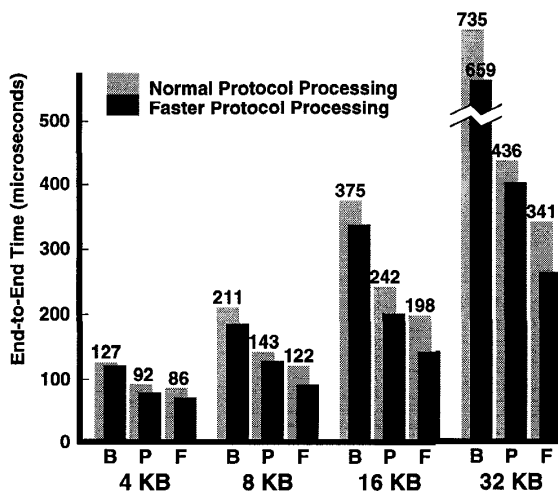


FIGURE 6.1. MPI transfer performance for a range of message sizes and transfer techniques.

lines. As an indication of the potential cost, it takes approximately 32 μ s to refill the 8 KB of data displaced during a 4 KB processor copy (longer if the memory is remote).

Faster Protocol Processing

We also considered the effect of faster protocol processing to determine if the performance of fbcopy is fundamental to the architecture. We altered our simulator to execute protocol code twice as fast, allowing us to quantify the potential for increased performance due to protocol code optimizations or a more aggressive protocol processor implementation.

The results are shown in the middle of Table 6.1. They show mild gains in most cases for processor copy, but more significant gains for fbcopy. This can be explained by looking at the performance limits imposed by the different resources in the system. With faster protocol processing, cache misses are serviced more rapidly, which improves the performance of processor copy. However, in the base coherence protocol in FLASH, the PP is only a few cycles slower than the memory system in the common miss handlers so these gains are small [HKO⁺94, KOH⁺94]. Fbcopy, in contrast, generally performs more protocol processing per line. As a result, there is more potential for gains from faster processing. While processor copy sees only 4–12% benefit from faster protocol processing, fbcopy gains between 10–36% in these benchmarks.

6.2 Message Passing Interface

To study the performance of block transfer operations in a more realistic context, we examine message passing primitives from the Message Passing Interface (MPI) standard [MPI93]. We use version 1.0.11 of MPICH from Argonne National Laboratory [GL]. MPICH supports several different underlying physical transport mechanisms,

including one designed for shared memory multiprocessors such as the SGI Challenge. This code runs without modifications in the SimOS simulation environment.

In the shared memory implementation of MPICH, bcopy is used to perform the data transfer of large messages (2 KB or greater). MPICH uses IRIX's support for allocating a shared memory region between processes, while the rest of the application address spaces are private, consistent with the distributed memory model of MPI. To send a message, the sender first copies the message data from its private memory into the shared buffer. The receiver then copies the data from the buffer into its private memory. We augmented the base implementation to utilize fbcopy or prefetched bcopy instead of the standard bcopy.

We implemented a basic two-node application using standard MPI send and receive primitives in which data is communicated in a producer-consumer relationship. Figure 6.1 shows that for both normal and faster protocol processing, fbcopy outperforms both normal and prefetched processor copy over a range of message sizes. The gains from fbcopy increase with message size, due largely to the amortization of fbcopy start-up overhead.

The flexibility of the protocol processor in MAGIC enables an additional option not possible in processor copy. Since the MPI send and receive routines execute at user level, maintaining private address spaces requires two copies using the shared buffer as described above. We could avoid this limitation by implementing buffer management functionality in MAGIC, allowing the receiver to "post" its buffer to the sender in a protected fashion. With the aid of this support, the sender could use fbcopy to transfer its message directly from the sender's memory to the receiver's memory, avoiding the temporary buffer entirely. This technique has the potential to perform better than all two-copy implementations.

6.3 GNU tar Application

Since our protocol is fully integrated with cache coherence we can use it to accelerate block transfer in shared memory applications as well. In fact, one of the biggest users of block transfer is the operating system itself. There are several kernel tasks that are primarily concerned with the movement of block data, such as uiomove which is used to move I/O data between user and kernel space. In this section, we focus on accelerating uiomove through the use of fbcopy or prefetched bcopy. Modifying the IRIX kernel was straightforward since fbcopy and prefetched bcopy are essentially drop-in replacements for bcopy. We study the impact of this acceleration on the GNU tar application performing a conventional directory copy:

```
cd $OLD; tar cBf - . | (cd $NEW; tar xBf -)
```

We simulate three nodes, two executing the tar commands and the third modeling the buffer cache in memory (we assume the buffer cache contains the files involved in the transfer). The first tar process reads data from the buffer cache through a read system call, which uses uiomove to copy the buffer cache data into the tar pro-

	Normal PP			Faster PP		
	B	P	F	B	P	F
Overall:						
Execution Time (ms)	215	180	188	202	167	160
Speedup (vs. bcopy)	1.00	1.19	1.14	1.00	1.21	1.26
Block Transfer:						
% of Execution time	30%	22%	27%	29%	20%	21%
CPU Occupancy (ms)	65	39	13	59	34	12

TABLE 6.2. Simulation results for GNU tar.

cess's user address space. The tar process sends the data into the pipe buffer (in 10 KB chunks) using a write system call. The kernel again uses `uiomove`, this time to bring the data from user space into kernel space. Similar actions occur on behalf of the second tar process.

We show the results from the tar application in Table 6.2. Both prefetched bcopy and fbcopy show speedups over the standard bcopy implementation. Though fbcopy is slightly slower than prefetching with normal protocol processing, with faster protocol processing it shows gains over prefetching. Furthermore, fbcopy has some advantages over processor copy in overall system resource utilization. As shown in the table, the processor copy versions require more than three times as much CPU occupancy to perform the same amount of block transfer. In the fbcopy version, this time could be utilized for parallel computation, further increasing the performance of fbcopy relative to processor copy techniques.

7 Discussion and Related Work

While a number of systems have addressed support for block transfer protocols [ABC⁺95, BLA⁺94, FAB⁺96, KA93, RLW94], none provides full integration with cache coherence. The protocol we describe not only achieves this integration, but preserves protected virtual memory support as well. Many systems have advocated a limited block transfer model, referred to as *local coherence*, which constrains the caching of source and destination buffers to their respective local nodes. However, this model breaks down in a shared memory system running a modern operating system such as IRIX, in which processes may migrate to improve load balancing. In such an environment, assumptions about the local coherence and local allocation of pages no longer hold; only a block transfer protocol without such restrictions is widely usable.

The Alewife system [ABC⁺95], like FLASH, supports both cache coherent shared memory and message passing. While the Alewife research has addressed the issue of coherence of message data [KA93], only local coherence is supported in hardware. Typhoon [RLW94] shares many of the same philosophies as FLASH; however, protocols such as fbcopy have not been studied in the context of Typhoon.

Woo et al. studied the benefits of block transfer in scientific and engineering applications [WSH94], including the

impact of prefetching as an alternative to block transfer. Some of their conclusions were similar to ours—for instance, they showed that prefetching can in some instances achieve similar gains to block transfer, and that applications are able to effectively utilize the spatial locality afforded by longer cache lines. However, while their study found limited block transfer in those applications, the workloads we consider expose additional opportunities for block transfer, such as in the operating system.

Compared with the aggressive network in FLASH, other systems typically have a larger disparity between local and remote access times. In such systems, the long latency of an internode block transfer cannot be completely hidden by prefetching since processor implementation concerns fundamentally limit the number of outstanding accesses. As a compelling example, if the network latency increases by a factor of four in our tar workload with faster protocol processing, fbcopy would show a speedup of 1.42 over bcopy, versus 1.22 for prefetched bcopy. In addition, as researchers consider systems in which remote access latencies are even longer, such as the proposed Distributed FLASH machine [KOH⁺94], coherent block transfer is also attractive for migration and replication of pages to improve memory locality [VDG⁺96].

Though not provided by the R10000, some modern architectures such as PowerPC and the Sun UltraSPARC include support for block loads and stores. For example, UltraSPARC can load or store blocks of floating point registers directly from or to memory without affecting some levels of the cache hierarchy [Sun95]. This support affords a unique opportunity to avoid cache pollution from processor block copy. However, implementation concerns such as restrictions on the number of simultaneous block operations may limit the applicability of these operations for achieving internode transfers.

8 Conclusions

The design of the fbcopy protocol has shown that block transfer can be fully integrated with cache coherence with manageable complexity. Though the performance of a transfer decreases when coherence actions must be taken, fbcopy achieves near-peak network bandwidth when transfer buffers are uncached or cached only locally. Since our fully integrated solution does not degrade performance in locally coherent scenarios, there is little motivation to restrict the implementation to local coherence.

While not appropriate for all situations, fbcopy offers some advantages over processor copy techniques. Our study of MPI primitives demonstrated that fbcopy enables significant performance gains over implementations of the MPI communication library using processor copy. While the microbenchmark results showed that prefetched bcopy outperforms fbcopy in some cases, through optimizations modeled by faster protocol processing we demonstrated that fbcopy can match or exceed the performance of prefetched bcopy. Fbcopy also results in lower network traffic, since it transfers only the data inherently being

communicated. The results from the tar application also showed that fbcopy achieves competitive performance with a fraction of the CPU occupancy of processor copy techniques. This parallelism can be utilized at the application level to reduce the effective cost of block transfer communication. Finally, while gains from accelerated block transfer in FLASH are modest due to its relatively low remote memory latency, the potential benefits are much greater in systems with longer remote latencies.

Acknowledgments

We gratefully acknowledge the work of the entire FLASH hardware and SimOS development teams. We owe special thanks to Joel Baxter, Mark Heinrich, and Steve Herrod, whose generous help with the simulation tools was invaluable.

This research is part of the Stanford FLASH project, funded by ARPA grant DABT63-94-C-0054. John Heinlein is supported by an Intel Foundation Fellowship. Robert Bosch is supported by an Office of Naval Research Graduate Fellowship. Mendel Rosenblum is partially supported by a NSF Young Investigator Award.

References

- [ABC+95] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiawicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.
- [BLA+94] Matthias Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, and Edward W. Felten. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.
- [CHR⁺95] John Chapin, Stephen A. Herrod, Mendel Rosenblum, and Anoop Gupta. Memory System Performance of UNIX on CC-NUMA Multiprocessors. In *SIGMETRICS/PERFORMANCE '95*, May 1995.
- [CRD⁺95] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault Containment For Shared-Memory Multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.
- [Cray93] Cray Research, Inc. Cray T3D System Architecture, 1993.
- [FAB+96] Edward W. Felten, Richard D. Alpert, Angelos Bilas, Matthias A. Blumrich, and Douglas W. Clark. Early Experience with Message-Passing on the SHRIMP Multicomputer. In *Proceedings of the 22nd International Symposium on Computer Architecture*, May 1996.
- [GL] William Gropp and Ewing Lusk. User's Guide for mpich, A Portable Implementation of MPI. Argonne National Lab Tech Report, <http://www.mcs.anl.gov/mpi/mpich>.
- [HGD+94] John Heinlein, Kourosh Gharachorloo, Scott Dresser, and Anoop Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.
- [HKO+94] Mark Heinrich, Jeffrey Kuskin, David Ofelt, John Heinlein, Joel Baxter, Jaswinder Pal Singh, Richard Simoni, Kourosh Gharachorloo, David Nakahira, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.
- [KA93] John Kubiawicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the 7th ACM International Conference on Supercomputing*, July 1993.
- [KOH+94] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.
- [LLG+90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990.
- [MG91] Todd Mowry and Anoop Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. In *Journal of Parallel and Distributed Computing*, 12(2):87–106, 1991.
- [MPI93] Message Passing Interface Forum. Document for a Standard Message-Passing Interface. Technical Report No. CS-93-214, University of Tennessee, November 1993.
- [RLW94] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level Shared Memory. In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.
- [RHW+95] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete Computer Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, Winter 1995.
- [RBH⁺95] Mendel Rosenblum, Edouard Bugnion, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Impact of Architectural Trends on Operating System Performance. In *Proceedings of the 15th ACM Symposium on Operating System Principles*. December 1995.
- [SG95] Silicon Graphics, Inc. MIPS R10000 Microprocessor User's Manual, 1995.
- [Sun95] Sun Microsystems, Inc. The UltraSPARC Processor – Technology White Paper. <http://www.sun.com:80/sparc/whitepapers/UltraSPARCtechnology/>.
- [VDG+96] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.
- [WSH94] Steven Cameron Woo, Jaswinder Pal Singh, and John Hennessy. The Performance Advantages of Integrating Block Transfer in Cache Coherent Multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.