Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor

John Heinlein, Kourosh Gharachorloo*, Scott Dresser†, and Anoop Gupta

Computer Systems Laboratory
Stanford University
Stanford, CA 94305-4070

Abstract

The advantages of using message passing over shared memory for certain types of communication and synchronization have provided an incentive to integrate both models within a single architecture. A key goal of the FLASH (FLexible Architecture for SHared memory) project at Stanford is to achieve this integration while maintaining a simple and efficient design. This paper presents the hardware and software mechanisms in FLASH to support various message passing protocols. We achieve low overhead message passing by delegating protocol functionality to the programmable node controllers in FLASH and by providing direct user-level access to this messaging subsystem. In contrast to most earlier work, we provide an integrated solution that handles the interaction of the messaging protocols with virtual memory, protected multiprogramming, and cache coherence. Detailed simulation studies indicate that this system can sustain message-transfers rates of several hundred megabytes per second, effectively utilizing projected network bandwidths for next generation multiprocessors.

1 Introduction

Shared memory and message passing are the two predominant communication models supported by multiprocessor systems. Shared memory provides the programmer with a simple abstraction and is particularly well suited for programs that exhibit dynamic communication behavior or fine grain sharing. However, certain types of communication, such as the transfer of coarse grain data, can sometimes be achieved more efficiently using an explicit message passing model. Active messages [5, 23], which provide the extra ability to invoke simple computation at destination nodes along side the data transfer, further extend the scope of communication behaviors that may be optimized through the use of explicit messages (e.g., specialized synchronization such as barrier trees). The complementary nature of the shared memory and message passing communication styles, in conjunction with the convergence of underlying hardware mechanisms used to implement each model, has led to a surge of interest in hybrid architectures that support both models efficiently [4, 12, 14, 19].

Realizing performance gains from the use of explicit messages in a shared memory system requires an efficient message passing implementation. By far the most important performance optimization for improving message latency and bandwidth is to alleviate expensive system calls in the common path of sending and receiving messages. This can be achieved by providing *direct user-level access* to the messaging subsystem. The challenge is to provide user-level access without compromising protection or correctness in environments that support general *multiprogramming* and *virtual memory*. Performance can be further enhanced by avoiding message copying through direct transfer of data between user address spaces. Finally, it is important to allow overlap of computation with ongoing message communication by minimizing processor overhead for handling messages and avoiding unnecessary processor interrupts. The integration of message passing with shared memory presents its own set of challenges. Providing meaningful semantics for messages requires supporting a seamless interaction of message data with cache coherence. Furthermore, keeping complexity to a minimum while supporting both modes of communication can be a challenging task.

We have been investigating efficient mechanisms for integrating message passing and cache coherent shared memory in the context of the Stanford FLASH multiprocessor [14]. One unique feature of FLASH (FLexible Architecture for SHared memory) is the use of a custom programmable node controller that connects the processor, memory, and network components at each node. The controller, called MAGIC (Memory and General Interconnect Controller), contains an embedded processor that can be programmed to implement both cache coherence and message passing protocols. The programmability of MAGIC provides substantial flexibility in selecting protocols and simplifies the design and debugging of such protocols as compared to pure hardware implementations. In addition, the inherent flexibility of the design allows us to avoid dedicated hardware mechanisms customized for each type of protocol.

This paper focuses on the hardware and software mechanisms for efficiently supporting message passing protocols on the FLASH architecture (see [8] for an evaluation of cache coherence protocols on FLASH; our earlier work on message passing in FLASH appears in [7]). Our goal has been to provide a general set of mechanisms that apply to a wide variety of messaging protocols, ranging from simple memory copy and simple active message (e.g., fetch-and-increment) to more complex protocols such as Intel NX [18] and MPI (Message Passing Interface) [15]. By delegating much of the protocol functionality to the programmable node controllers, most messaging protocols can be supported on

*John Heinlein may be reached at heinlein@leland.stanford.edu. Kourosh Gharachorloo is currently at DEC Western Research Laboratory, 250 University Ave., Palo Alto, CA 94301 (kourosh@wrl.dec.com).

†Scott Dresser is currently at Hewlett-Packard, Workstation Systems Division, 300 Apollo Drive, Chelmsford, MA 01824 (sad@apollo.hp.com).

To appear in the Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 1994.

FLASH while keeping the overhead on the compute processors low. Latency and overhead are further reduced by providing protected user-level access to the programmable controllers through specially mapped read and write operations. As part of the support for direct user-level messages, we propose novel software techniques that allow us to efficiently deal with protection and correctness issues related to virtual memory and multiprogramming. Finally, we support efficient data movement by programming MAGIC to transfer message data as a sequence of cache line sized units. MAGIC also manages the coherence for such data, thereby leveraging many of the mechanisms already present in FLASH for supporting cache coherence protocols.

Our preliminary evaluation of message passing performance in FLASH indicates that we can achieve peak data transfer rates of over 300 megabytes per second. We analyze the various factors that contribute to the performance and discuss several key optimizations that highlight the importance of flexibility and programmability in the design. Compared to previously proposed systems that support message passing, FLASH is unique in the combination of functionality and performance it provides within a single architecture. The flexibility of the design allows us to achieve this with virtually no extra hardware mechanisms (except for data alignment) beyond those already provided for cache coherence.

The rest of the paper is organized as follows. The next section provides a brief overview of the FLASH architecture. Section 3 presents the basic mechanisms for supporting protected user-level message passing and achieving coherent data transfer. Section 4 describes our evaluation methodology, followed by a preliminary analysis of message passing performance in Section 5. A general discussion of our approach, along with comparison with other related research, is provided in Section 6. We conclude in Section 7.

2 FLASH Architecture

This section provides a brief overview of the FLASH architecture. We mainly emphasize aspects that are relevant to message passing; a full description is provided in an earlier publication [14]. As shown in Figure 1(a), FLASH consists of a large array of processing nodes connected by a low-latency, high-bandwidth network. Each node contains a high-performance processor and cache, a portion of the distributed main memory, a network port, and a custom node controller called MAGIC (Memory and General Interconnect Controller). The MAGIC component consists of a programmable processor optimized for executing protocol operations and dedicated data paths that tightly connect the main processor, memory, and network port at each node.

FLASH accomplishes protocol communication through low-level network messages that are exchanged among the nodes (these are referred to as protocol messages to distinguish them from user-level messages). The MAGIC component at each node is responsible for processing these messages. MAGIC splits an incoming message into its control and data components. This allows MAGIC to use its dedicated data paths to efficiently transfer data while the control processing is accomplished in parallel by the programmable protocol processor. The parallel handling of data and control plays an important role in providing efficient protocol processing. One of the key components in the data path is a set of *data buffers*. These buffers form an array of cache line sized registers that allow data to be staged through MAGIC in a pipelined fashion.

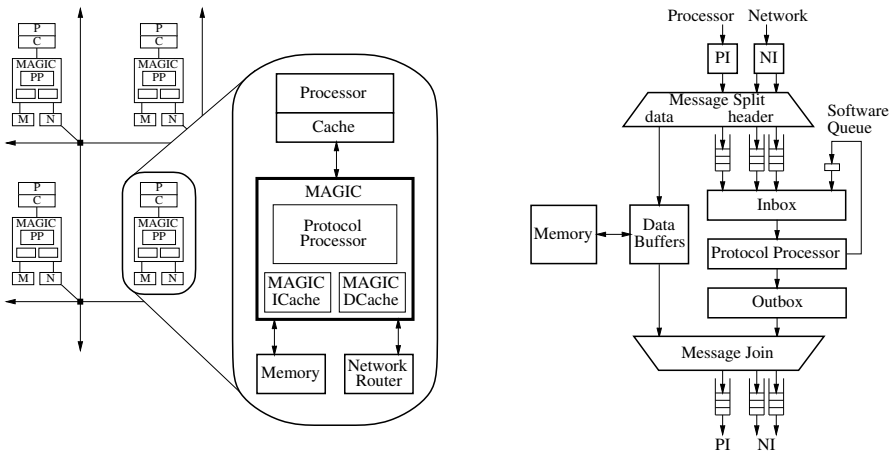
Since all protocol messages in the system are serviced by MAGIC, its architecture is designed to efficiently process such messages. A key component of MAGIC is the statically-scheduled dual-issue *protocol processor* (PP) that executes the protocol code (called *handlers*). To increase message throughput, MAGIC provides a macropipeline that allows multiple messages to be processed in a pipelined fashion. Figure 1(b) provides an overview of the three stages in the macropipeline. In the first stage, the *Inbox* prepares an incoming protocol message for processing by the PP. This involves selecting the address for the handler to be executed by the PP, and in some cases, initiating a speculative memory operation on behalf of the incoming message. The second stage involves executing the handler code on the *protocol processor*. The PP executes handlers to completion with no preemption. During the third stage, the *Outbox* assumes control for any messages sent by the PP handlers to the processor or network port. All protocol state (including directory state for cache coherence) is stored in a portion of main memory. MAGIC contains its own instruction and data caches which provide efficient access to protocol code and state. Even though MAGIC can access all of the local main memory through its cache interface, data movement between external units is typically supported using the separate dedicated data paths in MAGIC.

In addition to support for dual-issue, the PP also provides several special instructions that enhance the performance of common protocol operations (e.g., flexible bit manipulation instructions). To keep the design simple, the PP excludes many of the features found in general purpose RISC microprocessors. For example, there is no support for floating point operations or interrupts. Furthermore, the PP provides no hardware support for address translation (i.e., hardware TLB). Though hardware support for address translation can be useful for message passing, such support is not required by cache coherence protocols and could significantly complicate the PP implementation. The next section discusses this tradeoff briefly and proposes efficient techniques that achieve this functionality in software. Finally, since the PP lacks sufficient protection mechanisms to support user-level code, we currently require protocol handlers that execute on the PP to be system level code.

Figure 1(b) also provides a logical view of the queues in MAGIC that are used to buffer incoming and outgoing protocol messages. The processor and network queues are implemented in hardware. We exploit the virtual lane capability of the network to provide a pair of incoming and outgoing network queues. This simplifies the solution for deadlock in most protocols since separate queues can be used for request and reply messages. In contrast to the processor and network queues, the software queue is implemented as a single entry that represents the head element in the queue. Remaining elements are stored in memory. Each handler that is invoked from the software queue is required to schedule the next operation by loading the hardware register with the next element from the queue in memory. The next section describes the use of the software queue for supporting message passing protocols.

3 Mechanisms for Supporting Message Passing Protocols

Message passing protocols may be used for a variety of different communication purposes. The traditional use for message passing has been to achieve *data transfer*. Messages used for this purpose may vary in size depending on whether they are used for synchronization and coordination, which typically requires a small amount of data, or for actual transfer of large data regions. Similarly, protocols for achieving data transfer range from simple protocols that essentially provide memory copy capabilities to more complex protocols, such as Intel NX [18] and MPI [15], that provide sophisticated buffer management. More recently, *active messages* have been proposed as a generalization of message



(a) FLASH Node Architecture

(b) MAGIC Control Macropipeline

Figure 1: The FLASH architecture.

passing to incorporate the capability of invoking computation at remote nodes along with the traditional data transfer functionality [5, 23]. Uses for active messages also range in complexity from performing simple computation at destination nodes (e.g., a fetch-and-op) to more complex computation approaching the functionality of remote procedure calls.

Our primary goal for FLASH has been to provide a common set of mechanisms that efficiently support a wide range of message passing protocols. Even though the large variety of possible protocols may make this goal appear elusive, we find that these protocols all share a small number of basic requirements for achieving their efficiency and functionality. This section describes the base hardware and software mechanisms used to support various message passing protocols in FLASH. The first three sections describe the support for protected user-level access to MAGIC. Section 3.1 describes the mechanisms for communicating commands from the compute processor to MAGIC to initiate and monitor protocol operations. In Section 3.2, we address the issue of communicating authentic addresses to MAGIC and ensuring their validity in the presence of virtual memory. Section 3.3 briefly describes how MAGIC takes actions on behalf of a user process while ensuring protection in the presence of general multiprogramming. We then discuss the support for coherent data transfer and for invoking computation at remote nodes (e.g., to support active message) in Sections 3.4 and 3.5.

3.1 Communication between Processor and MAGIC

We use a general memory-mapped interface to communicate information from the compute processor to MAGIC. This interface allows direct user-level access for initiating protocol operations and monitoring their status. The processor specifies commands to MAGIC by issuing uncached read and write operations to specially designated regions in the address space. Below, we discuss the details of address space mappings and describe the support for command descriptions that require multiple memory operations.

The physical address space in FLASH is divided into four distinct regions: normal space, I/O space, message space, and a reserved space (currently unused). Regular read and write memory operations use the *normal space*. Operations to this space trigger the cache coherence protocol handlers on MAGIC. Message passing additionally uses the *message space* and part of the *I/O space*. References to these alternate address spaces trigger special handlers. The Inbox accomplishes this efficiently by providing the PP with a different software handler during the dispatch stage. Writes to the alternate spaces provide MAGIC with information encoded within the physical address field and the data field of the memory operation. Similarly, reads provide information through the address field and expect a response from MAGIC. Since the operating system controls the mapping of virtual to physical addresses in the alternate spaces, it is possible to use the memory mapped interface directly from user programs without compromising protection. As discussed in Section 3.2, this interface can also be used to convey authentic physical addresses at the user level.

We support information exchange between the compute processor and MAGIC through a simple *command sequence protocol*. This sequence typically consists of a series of writes followed by a read that returns status to the compute processor. The handlers invoked on MAGIC by the uncached writes accumulate command information in *protocol operation records* (these are maintained on a per-process basis in MAGIC's portion of the main memory). To allow MAGIC to use the appropriate protocol operation records for a given process, we require the operating system at each node to inform its local MAGIC of the newly activated process id (PID) at every context switch. Operation records maintain the status of protocol operations for the duration of the operation and are accessed through the MAGIC data cache for efficiency.

The last memory mapped operation in a command sequence is usually an uncached read. This read performs two functions: it notifies MAGIC that the command sequence is complete and should be committed, and it requests a response to indicate if the command sequence is accepted. An acknowledgement response indicates the commands are accepted by MAGIC.[1] A negative acknowledgement, on the other hand, indicates that MAGIC has rejected the sequence (e.g., due to temporary resource limitations or errors in the commands). Command sequences that fail may be retried by the user program.

Since the command sequence interface executes at user priority and involves multiple operations, it is possible for the sequence to be interrupted before it completes. Because operation records are selected based on PID, however, intervening processes cannot corrupt the interrupted process's record. Instead, the interrupted process can simply resume its sequence when it is rescheduled to execute. Compared to the solution used in CM-5 [22] and Alewife [13], our approach does not require saving and restoring of the commands across interrupts. This is because MAGIC accu-

[1] This does not imply the completion of the requested operation. Completion is checked through predesignated memory locations in most protocols.

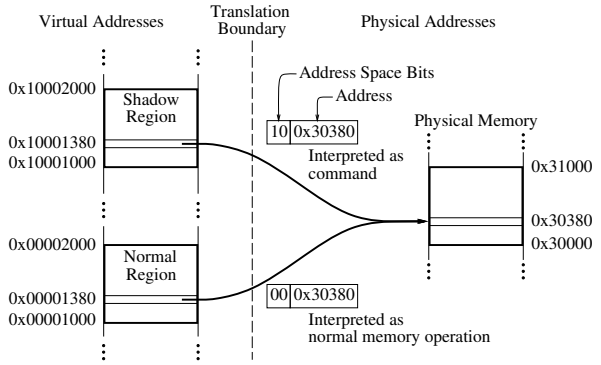


Figure 2: Example of double mapping illustrating the use of alternate physical address spaces.

mulates the memory mapped operations within software records as opposed to a single hardware queue. In addition, the programmability of MAGIC allows us to customize the command sequence protocol for various uses as opposed to providing a single hardwired protocol [13].

3.2 Virtual Memory

User-level messaging requires support for virtual to physical address translations since the user process can only specify virtual addresses and MAGIC needs to directly access memory with physical addresses. We begin this section by briefly describing our approach for providing authentic physical addresses to MAGIC and then present three different techniques that may be used to ensure the validity of addresses used by MAGIC. The underlying assumption in this section is that virtual to physical translation changes will be infrequent. Therefore, our general philosophy is to provide the necessary mechanisms to handle translation changes without slowing down the frequent case.

As discussed in Section 2, one of the simplifications in the design of the protocol processor (PP) as compared to a general microprocessor is the removal of the TLB. We made this simplification because (i) cache coherent memory operations do not require address translation since the processor already presents MAGIC with physical addresses, (ii) handling TLB and page faults on the PP would add significant complexity, and (iii) a small hardware TLB may not be an effective structure for caching translations, especially since PP reference patterns are different from that of code executed on the compute processor. Instead, we support the required address translation for protocol operations fully in software.

To communicate authentic physical addresses from a user program to MAGIC, we use the memory mapped interface discussed in the previous section along with a double mapping scheme that is supported by the operating system. Figure 2 shows an example of how this works. The OS maps a shadow region which has a translation only differing in its address space bits. References to the normal region are thus interpreted as ordinary memory operations while references to the shadow region are interpreted as commands and provide MAGIC with authentic physical addresses. Note that a single user-level reference to the shadow region provides MAGIC with an authentic translation for a whole physical page.

Given that MAGIC receives authentic translations, the remaining problem is to guarantee the validity of an address throughout its use by a protocol operation. The simplest solution is to lock the appropriate virtual pages to prevent the operating system from changing the mapping. Locking pages in memory for the duration of the program is clearly undesirable since it defeats the flexibility of demand paging. Alternatively, the pages may be locked for the duration of a protocol operation. This is common in systems with DMA where address changes are disallowed for the duration of DMA access. However, this latter approach is also undesirable because it requires expensive interaction with the operating system every time a user program invokes a protocol operation.

In what follows, we propose three alternatives for maintaining valid translations for physical addresses used by MAGIC. The first two techniques assume that MAGIC receives authentic physical addresses as part of the user-level command sequence that describes a protocol operation. The third technique removes this requirement by supporting a software TLB that allows MAGIC to perform translations itself.

3.2.1 Hold-Off Technique

The *hold-off* technique is a simple extension of an existing mechanism used in most multiprocessors to protect pending read and write memory operations against translation changes. To protect these operations, a processor can delay translation change requests (e.g. a TLB entry invalidate request) until all pending memory operations complete. We extend this idea to also include certain message protocol operations in the set of operations that need to complete before translations changes are allowed to proceed.

Hold-off is enabled by incrementing a software counter maintained by MAGIC that indicates the number of outstanding protocol operations currently using hold-off. When an operation completes, it releases its use of hold-off by decrementing the count. Before proceeding with a translation change, the operating system is required to notify MAGIC and to wait for a response. If the hold-off count is zero, MAGIC responds immediately. However, if the count is non-zero, MAGIC delays the response to the processor until the count reaches zero. This also has the side effect of preventing the initiation of any new operations that require hold-off.[2] Since the processor is prevented from initiating new protocol operations after a translation change arrives, the counter is guaranteed to return to zero *as long as* the previous operations eventually complete.

Preventing translation changes from occurring during the hold-off period can potentially lead to *deadlock* for certain types of operations. For this reason, the hold-off technique is only applicable to operations that are guaranteed to complete *on their own* without requiring any interaction with compute processes. Otherwise, a deadlock cycle could occur if the compute process required a translation change that was in turn prevented by the hold-off.

Even though the hold-off technique does not apply to all types of protocol operations, it is an extremely simple and efficient mechanism for providing correct virtual memory semantics to a class of useful operations. Example operations that can safely use hold-off include remote fetch-and-op or simple memory copy. On the other hand, operations such a barrier tree or a traditional synchronous message send can not use the hold-off technique since they require interactions with other processes before completion.

3.2.2 Invalidation Technique

Unlike hold-off, which temporarily prevents translation changes from taking place, the *invalidation technique* allows the change and invalidates obsolete physical addresses that may be stored by MAGIC. When the physical address is next used by MAGIC, the software handler detects the invalid state and requests a mapping

[2]Operations in the middle of their command sequence protocol are conservatively forced to retry when they attempt to commit.

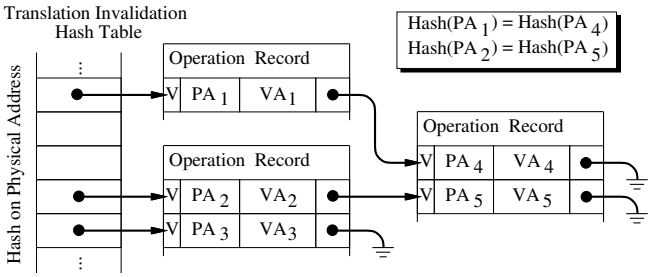


Figure 3: Translation invalidation data structure.

for the page from the main processor. Below, we briefly describe the required support for this technique.

Just as in hold-off, the user program provides MAGIC with authentic physical addresses through uncached writes. Unlike hold-off, the program also provides the virtual address in the data field of the command. Though the data field is not authentic, incorrect virtual addresses can at worst corrupt memory belonging to the requesting process and do not compromise protection for other processes.[3]

To support the invalidation technique, MAGIC keeps track of the physical addresses it is using. Physical addresses provided to MAGIC are stored in the protocol operation record and also linked into a hash table of in-use physical pages, as shown in Figure 3. Using this data structure, the PP can efficiently invalidate uses of a physical translation on a translation change notification from the local compute processor. An entry in the hash table is removed when the corresponding operation completes (we use doubly linked lists to do this efficiently).

By convention, handlers for operations that use the invalidation technique check the validity of a physical address within the operation record on each use of the address. If an invalid physical page address is detected, MAGIC interrupts the processor and communicates the corresponding virtual page address and the process id (also stored in the operation record). MAGIC can resume the operation once the processor responds with the new translation. In the meantime, other handlers are allowed to run on the PP.

The invalidation technique is quite efficient in the common case, when translations do not change. For each protocol operation, the overhead involves adding the page addresses in an operation record into the appropriate linked lists and removing them when the operation completes. The hand coded PP instruction sequence to add a translation to a list only takes 8–9 cycles to execute. The time required to check the address on each use is minimal, adding only a single branch instruction to the handler.

One of the limitations of the invalidation technique is that it can not protect remote physical addresses. For example, consider a memory copy operation from a local to a remote buffer that is implemented by sending destination physical addresses along with the data. Since the invalidation technique allows translation changes to occur immediately, destination physical addresses that are in transit can become stale. Note that this example would work correctly with the hold-off technique since translation changes are delayed until the memory copy operation completes.

[3]If the virtual address is incorrect, then one of two things will occur. Either the retranslation request will fail because the application does not have the necessary rights to access the region of memory, or a new translation is returned even though it does not match the same location that was originally specified. Neither case compromises the protection of other processes' memory.

3.2.3 Optimized Software TLB

The previous two techniques do not provide the ability to translate virtual addresses directly on MAGIC. The third technique we describe provides this functionality by supporting a software TLB on MAGIC. For operations that use a translation many times (e.g., message send of a long buffer), we plan to use the TLB in conjunction with ideas used in the invalidation technique. In such a scheme, the TLB is used initially to translate the addresses for an operation. These translations are then stored in the operation record (similar to the invalidation technique) to avoid a TLB lookup on every use, and the translation invalidation structure (shown in Figure 3) is used to protect against translation changes.

Providing translations on MAGIC via a software TLB has several advantages. The TLB allows us to specify protocol operations without the need to communicate authentic physical addresses to MAGIC during the protocol command sequence. For example, a message send that transfers multiple pages of data can be specified as a single virtual address and length pair instead of multiple physical page addresses. The ability to translate addresses efficiently also allows us to send messages to remote nodes using virtual addresses, which is useful for operations such as memory copy. Finally, since the TLB is in software, we can access it through the PP data cache for efficient lookups and can easily adjust its size and policies to increase its hit rate.

Similar to the previous two techniques, we require the operating system on the local node to inform MAGIC about translation changes in order to keep the software TLB consistent. Like the invalidation technique, the software TLB relies on interrupting the processor if it fails to resolve a translation request.

3.2.4 Summary of Virtual Memory Techniques

The techniques described above differ in the way they trade off efficiency for functionality. While hold-off is by far the most efficient technique, it is not applicable to all types of operations. The invalidation and software TLB techniques are more widely applicable, but entail higher overheads. The flexibility of MAGIC allows us to use the most appropriate technique for each type of protocol operation, in addition to experimenting with variations on the above techniques.

3.3 Protection

The mechanisms described above allow a user process to specify protocol operations that are executed by MAGIC on the process' behalf. Our goal is to provide these operations without compromising protection in the presence of multiprogramming. Below, we describe the three types of protection that we provide.

The *first* type of protection is memory protection. To achieve this, we leverage the access rights that are provided along with virtual memory address translation; our mechanisms for supporting virtual memory can easily be extended to communicate such access right information along with the translation. The *second* form of protection, preventing user messages from being forged or received by an incorrect process, requires MAGIC to have access to authentic process id's. As mentioned before, the operating system notifies MAGIC about the PID of the locally running process at context switch points. Therefore, forging can be disallowed by tagging all messages that are generated on behalf of a requesting process with the authentic PID. Furthermore, MAGIC is always aware of the identity of the running process which can be used to restrict its operations. Finally, the *third* type of protection is to restrict the effect of protocol operations to a specific group of

processes (e.g., messages may be restricted to processes within a single application). This type of protection can be supported by translating a "virtual PID" to a physical node number and PID pair (analogous to achieving memory protection through virtual translation).

3.4 Achieving Data Transfer

The flexibility of MAGIC allows us to transfer data efficiently without the need for specialized hardware support such as a DMA engine. We begin with the basic data transfer mechanism, including the use of the software queue and the rationale behind transferring data as a series of cache lines. The subsequent parts of the section address the interaction of message data with cache coherence and the transfer of data between unaligned regions. We evaluate the performance of these mechanisms in Section 5.

3.4.1 Basic Data Transfer

Data transfer operations can be specified from user programs through the memory mapped interface described earlier. The operation record is used to store the transfer parameters and also maintains the operation status during the transfer. Once a message send command is accepted by MAGIC, the transfer handler at the sending node sends the data in a series of cache line sized messages, referred to as *components*. To form a component, the protocol processor reads the appropriate line from memory into a data buffer in MAGIC, adds header information, and sends it to the receiving node. At the receiving node, each component runs a handler that stores the accompanying data into the receiver's memory.

From a fairness point of view, it is important to interleave other requests (e.g., cache coherent reads and writes) while the protocol processor (PP) is in the process of doing a large transfer. In addition, attempting to complete the entire transfer can cause the outgoing queues to fill which can cause the machine to deadlock unless the PP is relinquished to service other requests. Therefore, we use the software queue for scheduling a data transfer operation into multiple invocations of the transfer handler at the sending node. Each time the message transfer handler is invoked, it sends one or more components of the message data, updates the transfer state in the operation record, and reschedules itself using the software queue. Though each component of the user message is sent as a separate packet, the handler has the option of sending multiple components during each invocation. This technique, referred to as *chunking*, is effective because it amortizes the overhead of starting the transfer handler and saving and restoring the transfer state over multiple lines. We will evaluate the effect of this optimization in Section 5.

Transferring data as a series of cache line sized messages instead of a single large message has several advantages. The memory system and MAGIC are already optimized for transferring cache lines and therefore cache line packets are handled efficiently. In addition, allowing the service of cache coherent operations to be finely interleaved with large transfers is beneficial for achieving efficient overlap of computation and communication. Furthermore, by sending cache lines, the message passing protocol can leverage the deadlock avoidance and recovery techniques already used for the base coherence protocol. Finally, networks typically perform better (due to lower contention) when large messages are broken into pieces. The disadvantages are the network overhead for the headers on each component and the processing overhead on the PP to handle each line. However, these overheads are not as significant in FLASH due to the large cache line size (128 bytes).

One of the important issues in achieving high performance data transfer is avoiding extra message copying. The techniques to accomplish this are very protocol and implementation dependent, though they mainly center around buffer management issues. For example, sending the message directly from the application's address space avoids copying on the sender, and depositing an arriving message directly into the receiver's data structures can avoid copying at the receiver. The programmability of MAGIC allows us to experiment with various protocols and implementations in order to take full advantage of these optimizations.

3.4.2 Coherence

The choice of the coherence model for message data is a key issue in implementing data transfer on cache coherent shared memory systems. Below, we discuss the various options and their corresponding tradeoffs. Kubiatowicz et al. present a similar categorization in the context of message passing in Alewife [13].

The simplest option is to provide *no coherence* for user message data. This corresponds to reading the data directly from the sender's memory and storing the data directly into the receiver's memory without taking any coherence actions. However, this option is not acceptable since it precludes caching of message data at both the sender and the receiver. The second option, called *local coherence*, provides coherence if the message data is uncached or cached only at the home node. This closely matches the functionality provided by most message passing architectures where each processor can only cache the data that resides in its local memory. The third and most general option is to provide *global coherence* which imposes no restrictions on the caching of data. The additional functionality provided by global coherence can be beneficial for some applications (e.g., memory copy used to achieve page migration in a cache coherent system). In cases when the lines are clean in local memory or are cached only locally, global coherence performs the same as local coherence. However, global coherence requires extra coherence transactions when message lines are cached remotely or have remote homes.

The mechanism to support local coherence is a straightforward extension of the data transfer implementation described in the previous section. At both the sender and the receiver side, the directory information for each memory line is used to determine whether any coherence transactions are required for reading or writing data. Implementing global coherence can be more complex, especially when the homes of the buffers are not at the sender or receiver nodes. In contrast to designs such as Alewife [13] that provide hardware support for only local coherence, the flexibility of FLASH allows us to efficiently support either local or global coherence without the need to commit to a single choice while designing the hardware. Section 5.3 provides some preliminary results that compare the performance of local and global coherence on FLASH.

3.4.3 Alignment

The data transfer mechanism discussed so far is optimized for sender and receiver buffers that have the same cache line alignment. However, to provide a general solution, we must also consider large transfers between buffers that have a different alignment. Figure 4 shows an example data transfer from a sender buffer that begins and ends in the middle of cache line boundaries (panels 1 and 2). The same figure shows (in panels 3 and 4) how the alignment needs to be changed during the transfer assuming the receiver buffer is cache line aligned. Since transferring data a cache line at a time requires the adjustment on every line, FLASH includes a simple hardware mechanism to make such adjustments more efficient.

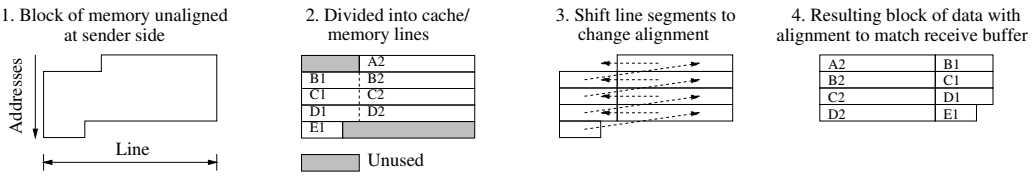


Figure 4: Support for arbitrary user message alignment.

The hardware alignment mechanism FLASH adopts allows the MAGIC chip to flexibly load cache-line unaligned data into its data buffers starting at any doubleword (64-bit) boundary. This allows the sending MAGIC to efficiently perform any adjustments required for alignment. Memory loads using this mechanism specify two buffers. The portion of the data line that extends past the end of the first data buffer "wraps" into the second buffer. Loading the next memory line into the second buffer where the first left off will form a component with the desired alignment that can be sent to the receiver. To ensure that most memory lines are only read once at the sender, it is important to send multiple components (i.e., use chunking) during each invocation of the transfer handler. The only restriction for using this mechanism is that the sending MAGIC must know the alignment of the receive buffer. For operations such as memory copy, this information is naturally known at the sender. For other operations, the protocol can be modified to communicate this information during the early stages of a transfer.

3.5 Invoking Computation at Remote Nodes

Aside from supporting data transfer, the MAGIC chip can also be used to support active message style operations that require computation to be invoked at remote nodes. Ideally, we would like to support most such computation on the protocol processor (PP). However, since the PP lacks sufficient protection for executing user-level protocol handlers (e.g., lack of translation hardware and general preemption), we also provide a reasonably efficient interrupt path that can be used to delegate the handler to the compute processor at the node. To avoid the cost of interrupting the processor, operations that are frequently used can be executed directly on MAGIC if implemented as system-level handlers. The data transfer protocols described in the previous section fall into the latter category. Similarly, operations such as fetch-and-op, barrier trees, scatter gather, or multicast can be directly supported on MAGIC. In addition, we are currently exploring software techniques that provide the required protection to allow a restricted set of user-level handlers to execute on the PP as well.

4 Experimental Methodology

This section describes the simulation environment and architectural parameters that we use to evaluate the performance of message passing in FLASH. The latter part of this section describes a connection-oriented data transfer protocol that we have implemented within our detailed simulation environment. We use this protocol as a basis for our performance evaluation of data transfer in the next section.

4.1 Simulation Environment and Architectural Parameters

Our experimental results are based on a detailed thread based simulator that has been developed as part of the FLASH system design effort. The simulator, called FlashLite, is based on the Tango Lite simulation system [6] and provides a detailed model of the various independent hardware units on a cycle-accurate basis. FlashLite has been used to develop and verify the base coherence protocol and to evaluate many of the architectural features in FLASH. The software environment also includes a port of gcc, called PPgcc, that generates PP assembly code, and a code scheduler called PPtwine which is a port of the twine scheduler originally written for the Stanford TORCH project [20]. This simulation environment enables us to accurately evaluate the performance of different protocols by implementing the required handler codes for the protocol processor in C, compiling them using PPgcc and PPtwine, and finally running them through the simulator.

Our architectural parameters and assumptions are largely based on the design parameters for FLASH. The compute processor is assumed to be a 200 MHz dual issue RISC microprocessor. We assume two levels of caching, an on-chip primary cache and a secondary cache, both with a line size of 128 bytes. Except for the processor core, the rest of the system, including MAGIC, is assumed to run at a 10 ns cycle time. The protocol processor thus corresponds to a 100 MHz dual issue integer core. All cycle numbers discussed in the following two sections are based on this 10 ns system cycle time.

The memory system provides MAGIC with the first 64-bit word of a cache line in 14 cycles, followed by an additional word on each successive cycle (i.e., total of 15 extra cycles). Memory can be accessed with a new address during the transfer stage. Accessing data in the processor's cache from MAGIC takes longer than accessing main memory because of the required processor intervention. In this case, the cache responds with the state of the line after 23 cycles and with the first data word 5 cycles later, followed by 15 cycles of transfer time for the remaining data words. Unlike memory, accessing the next line of data from the processor cache is delayed until the previous transfer phase completes. The interconnection network is assumed to have a bandwidth of 400 MB/sec, with an average latency of 22 cycles (assuming a small system with average of three hops). The maximum network packet size contains 128 bytes of data with 32 bytes of header information. Therefore, the true data transfer bandwidth is reduced to approximately 320 MB/sec (i.e., one cache line every 40 cycles). As a point of comparison to data transfer latencies, the latency for cache coherent reads in FLASH is approximately 27 cycles for local misses and 111–191 cycles for remote misses (larger latency corresponds to the data being dirty at a third node).

4.2 Example Data Transfer Protocol

Data transfer protocols vary in the level of buffer management and selection they support, ranging from a simple memory copy primitive that relegates all buffer management to the application to more robust send/receive models such as Intel NX [18] or MPI [15]. To provide concrete results, we have chosen a connection-oriented send/receive protocol that lies somewhere in between these two extremes and have carefully implemented it in our simulation environment. Our choice of protocol was influenced by discussions with designers at Intel SSD who expressed interest in the evalua-

tion of such lightweight protocols in the context of FLASH [17]. Below, we describe the basic protocol along with its implementation and modeling in FlashLite. The next section provides the actual simulation results for this protocol.

The protocol we have chosen to study is based on sends and receives and requires two processes to establish a bidirectional connection prior to transferring data with one another. The connection supports a byte stream model where sends and receives need not be of the same length. For example, the sender can send a 4 KB block of data and the receiver can receive it into four 1 KB buffers that are non-contiguous (using four distinct receive commands).

The data transfer in our implementation works as follows. When a receive command is issued, the receiver sends a notification to the connection partner indicating the amount of space available on the receiver end. The connection partner keeps track of this information and sends any waiting data to the receiver (up to the amount of space specified by the receiver). We use several of the mechanisms discussed in Section 3 for supporting direct user-level access and for achieving data transfer. The invalidation technique discussed in Section 3.2 is used to maintain the validity of the physical addresses for the buffers at the sender and receiver. Our base implementation of the protocol also supports local coherence and unaligned buffers as described in Section 3.4.

We have implemented the above protocol in detail by writing the necessary protocol processor (PP) handlers in C. For some handlers, we implemented several versions to experiment with various optimizations (these will be discussed in the next section). The handlers fully support the protocol, including checks for exceptions and corner cases (e.g., translation changes, partial line sends, coherence races, and network queue checks). We used the PPgcc compiler to generate assembly instructions from these handlers, which were then scheduled for dual issue by our superscalar scheduler. For the commonly executed handlers, we hand optimized and hand scheduled the assembly code to further improve handler performance. The resulting handler lengths were fed back into the C version of the handler codes as timing annotations. Therefore, even though the simulator executes the C code, it can charge the accurate times for the various handlers. In the future, we plan to emulate the scheduled assembly code directly using Flashlite's PP emulator, called PPsim. To verify the protocol, we simulated several micro-benchmarks that tested individual features. In addition, we ported a Gaussian elimination application to use this protocol for its communication. We plan to implement and evaluate other message passing protocols (such as NX and MPI) in the near future.

5 Performance Evaluation

In this section, we begin with a qualitative description of the various factors that determine the performance of message passing in FLASH. We then present detailed performance results for our connection-oriented protocol.

5.1 Qualitative Analysis

We divide the time to do a data transfer into two parts: (i) a fixed component consisting of startup latency plus overhead, and (ii) a variable transfer time proportional to the size of the message. Figure 5 provides a high-level illustration of these components. The startup latency plus overhead encompasses sender and receiver library calls, initiation time, network latency, and transfer set up and clean up. For the most part, the complexity of these operations is directly related to the level of functionality provided

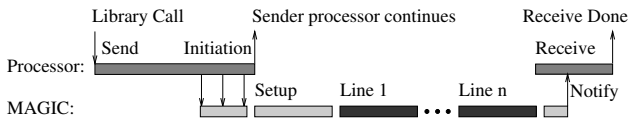


Figure 5: Simplified timeline of message transfer including library overheads.

Table 1: Breakdown of startup latency and overhead components (10 ns cycles).

Overhead component	Cycles
Initiation command processing	41
Transfer setup and command verification	116
Initial line delay	24
Network and queuing latency	45
Receiver cleanup at completion	58
Total overhead	284

by a given protocol, especially relating to protection and buffer management. Thus, the amount of startup overhead can vary substantially from protocol to protocol. In contrast, the transfer time is largely independent of protocol choice since the actual transfer involves nearly the same functionality in any protocol (e.g., checking directory state, taking coherence actions, storing message data, updating transfer state). Since the performance of the transfer is the common factor among most protocols, we mainly concentrate on evaluating this phase and focus less on such issues as startup overhead.

In a system like FLASH, data transfer performance can be limited by a number of factors: transfer handler occupancies, network bandwidth, and latency and bandwidth to access data in the memory or the processor cache from MAGIC. In FLASH, the memory latency and bandwidth are rarely the limiting factors. However, the total latency of retrieving a dirty line from a processor cache, 43 cycles, is substantially larger than accessing memory and indeed limits performance in some cases. The next section quantifies the effect of these factors on the performance of data transfer.

5.2 Performance of the Send/Receive Protocol

We begin by describing the startup overheads associated with our implementation of this protocol. We then evaluate three implementations of the protocol with varying levels of optimizations. As we will see, the flexibility of MAGIC plays a key role in enabling such optimizations.

Table 1 shows the various components that make up the fixed cost for a data transfer operation. The total overhead is 284 cycles (2.84 μs), which is approximately twice the average remote read latency in FLASH (note that the sending compute processor is stalled only until the initiation command sequence completes). This represents the overhead from when the initiation command sequence begins at the sender until the receiver MAGIC has processed the last line. This fixed overhead gets amortized over the number of cache lines that are sent during a transfer. As we will show in Section 5.4, simpler operations such as a fetch-and-op have a considerably lower startup overhead.

Figure 6 shows the net transfer time per cache line as a function of lines sent. The vertical axis on the right hand side shows the corresponding bandwidth that is achieved. The data shown is from the simulation of the most optimized implementation of the protocol. In these simulations, the receives were posted in advance of the sends (this is the case for the remainder of the results unless stated otherwise). The performance of data transfer in FLASH de-

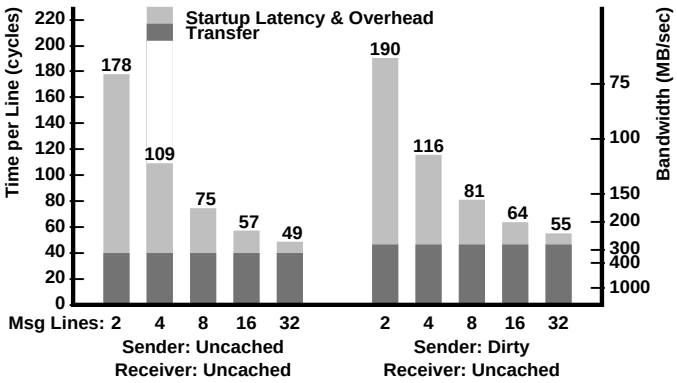


Figure 6: Transfer time per cache line as a function of number of lines (128 byte lines, 10 ns cycles).

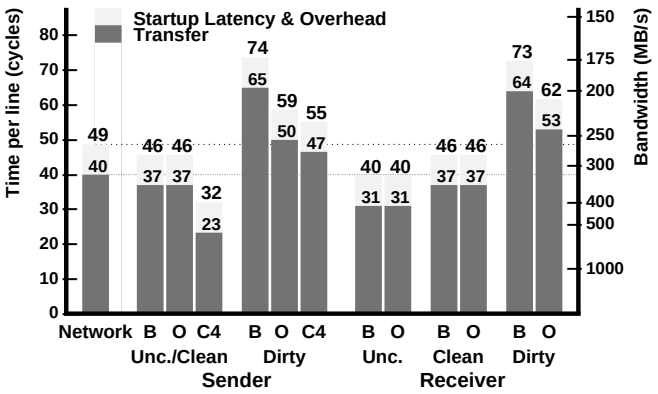


Figure 7: Message transfer performance under various optimizations (32 lines, 10 ns cycles).

pends on whether the send and receive buffers are cached. In the case on the left of the figure, all buffers are uncached; in the case on the right, the entire send buffer is dirty in the local cache. The time per line is divided into two components: the marginal time to transfer a line (labeled Transfer) and the amortized fixed cost. As shown in the figure, the impact of the fixed cost diminishes significantly for larger transfers. Transferring 32 lines of data (4 KB) takes a total of 15.6–17.7 μs (dirty line case takes longer), which corresponds to a sustained transfer bandwidth of 231–263 MB/sec. This corresponds to a peak bandwidth of 275–320 MB/sec during the transfer phase. In what follows, we analyze the various factors that determine this performance.

Figure 7 shows the results of our simulations for 32 line transfers (4 KB). We have presented the results in a way that shows whether the network, sender side handler, or receiver side handler limit the performance. The left most bar in the graph depicts the effective bandwidth due to network bandwidth limitations. This corresponds to a lower bound of 40 cycles per line (48.7 cycles including the amortized startup overhead). The remaining bars show the limitation due to handler occupancies at either the sender or receiver side. For both the sender and receiver side, we consider the performance based on the caching state of the buffer (our simulations assume lines on each side all have the same state). In the figure, uncached is abbreviated as "Unc.". Within each caching state, we show results for the three implementations of the protocol as discussed in Sections 5.2.1–5.2.3. The transfer rate in cycles per line can be determined by taking the maximum of (i) the network time, (ii) the appropriate sender handler occupancy, and (iii) the appropriate receiver handler occupancy.

The results in the figure show that maintaining coherence (e.g., when message data is dirty in the processor cache) in our base implementation (labeled "B") significantly degrades performance. The optimized implementation (labeled "O") decreases the impact of coherence by hiding the cache access latency. The chunked implementation (labeled "C4") further increases performance on the sender side by transferring multiple lines on each invocation of the handler. We describe these implementations in more detail below.

5.2.1 Base Transfer

We begin by briefly describing the base implementation and its performance, shown as "O" in Figure 7. On the *sender side*, uncached or clean lines are read from memory, so the performance of these cases is the same and is faster than the network limitation. As shown in the figure, the sender side handler takes 37 cycles per line. When lines are dirty at the sender, the latency of cache access to retrieve the data (i.e., 43 cycles) makes the performance far worse than the clean case. On the *receiver side*, if the receiver buffer is uncached, the message data can be directly written to memory. If the receiver buffer is cached but clean, a small additional overhead is required for sending an invalidation to the processor cache. Finally, if the line is dirty in the receiver's cache, the PP needs to query the cache and wait for a response.[4] While the performance of the dirty case at the receiver is much lower, in practice this scenario should be uncommon. Overall, we see that even with our base implementation, the network is the limiting factor in all cases except when the send or receive buffers are dirty in the processor's cache.

5.2.2 Optimized Transfer

The main drawback to the base implementation is the dirty case performance, hindered mainly by the long latency for cache responses. A natural optimization is to perform speculative processing during the cache access, such as updating the directory and transfer state, under the assumption that the cache will respond with success. In the infrequent case of a negative response (because the line is being written back), the handler must "roll back" the state to undo the effects of speculation. We refer to this as the *optimized* implementation, shown as "O" in Figure 7. For the sender side, this optimization saves 15 cycles over the Base dirty case; the savings is 11 cycles for the receiver side. Table 2 shows the static instruction counts for the PP send transfer handler itemized by the various subtasks for both the clean and dirty path (recall that the PP is dual issue so these do not correspond to cycle counts). We also show the number of loads and stores to protocol state; these numbers convey the importance of accessing the state through the MAGIC data cache. Appendix A shows the actual PP code for the clean case in the send transfer handler.

To graphically illustrate the optimized transfer implementation, Figure 8 shows a timeline of the resource occupancy on the sender side for uncached/clean state. The transfer handler issues the read to memory 18 cycles into its execution. As soon as the memory responds (14 cycles after the access), the network can begin the transfer while the remaining words are being supplied. As shown by the timeline, the network is the bottleneck in this case. Figure 9 shows a similar diagram for dirty lines at the sender. In this case, the PP requests the dirty data from the processor's cache 17 cycles into the handler execution. However, instead of stalling, the handler continues by doing speculative work. Eventually, at cycle 32,

[4] Even though the full line is written by the transfer, the wait is necessary to avoid a race involving pending writebacks of the dirty data.

Table 2: Breakdown of instructions for the optimized send handler.

Transfer Handler Task	Instructions Clean	Dirty
Initialization and state load	10	10
Load directory, check	6	6
Send get to cache	–	4
Update directory	–	2
Wait for cache, write to memory	–	4
Read memory	3	–
Form header, update state	15	16
Send, reschedule	12	14
Total Instructions	46	56
Loads	8	10
Stores	4	5

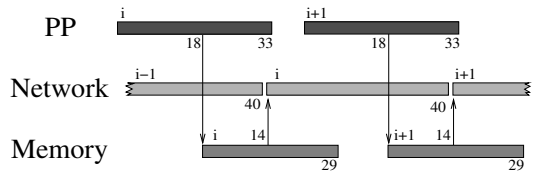


Figure 8: Timeline of optimized transfer of clean lines at the sender.

the handler exhausts the work it can do speculatively and stalls for the cache result. Once the response returns, the PP can issue the writeback to memory and the send to the network. Unlike the clean case where the network limits performance, here the occupancy of the PP (due to the cache access) is the bottleneck.[5] This timeline points out three important effects: (i) though we use speculation in the PP, the transfer handler is still stalling because the cache access latency is long, (ii) a significant amount of time is spent initializing the handler before the cache access is started, and (iii) even though the cache access time affects the performance limit, it is not the bottleneck itself. The next section describes how *chunking* (i.e., sending multiple lines in a single handler invocation) can further enhance performance by addressing these problems.

5.2.3 Chunked Transfer

To implement chunking, we software pipelined the optimized handler to send multiple lines within a loop.[6] This allows the PP to perform speculative processing on the current line as well as the initial processing for the *next* line while waiting for the cache response. An additional benefit is that some key transfer state can be kept in registers across multiple line transfers. Note that this optimization is not applicable to the receiver side because the receiver still observes transfers as single cache line components.

Figure 7 shows the effect of sending four lines at a time using this technique (labeled "C4"). The reduction in handler latency per line compared to the optimized handlers is 13.7 cycles for the clean case and 3.5 cycles for the dirty case. Figure 10 shows how the performance of the sender side transfer handler varies with the chunking factor (bars labeled "Aligned"). The latency per line in this figure represents the marginal time per line and therefore excludes the amortized overhead time. As shown, a chunking factor of 4 achieves most of the gain. Note that in the dirty case, a chunking factor of 8 leads to processing times per line that approach the cache access latency. Therefore, pipelined access to the cache would be required to further improve performance. Chunking is also important for achieving efficient unaligned transfers. We implemented a version of the chunked handler that uses the double buffer wrap-around feature in FLASH to realign data at the sender side (see Section 3.4.3). As shown in Figure 10, unaligned transfers incur between 10% to 37% performance degradation (at chunking factor of 4) as compared to aligned transfers, mainly due to the extra memory load and additional data buffer allocation on each invocation.

[5] In both caching cases, a four cycle gap between successive executions of PP handlers arises from software queue rescheduling. We can successfully eliminate this delay through hand optimizations by inlining the rescheduling code earlier in the handler. However, our results include the delay to be conservative.

[6] To avoid overflowing outgoing queues, the handler must dynamically adjust the number of lines it can send.

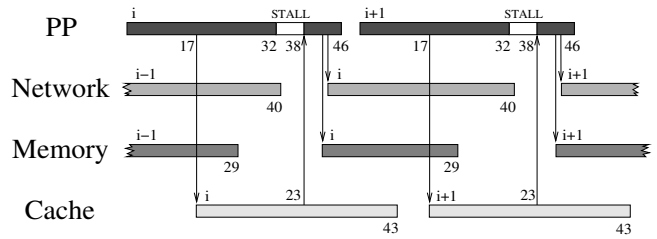


Figure 9: Timeline of optimized transfer of dirty lines at the sender.

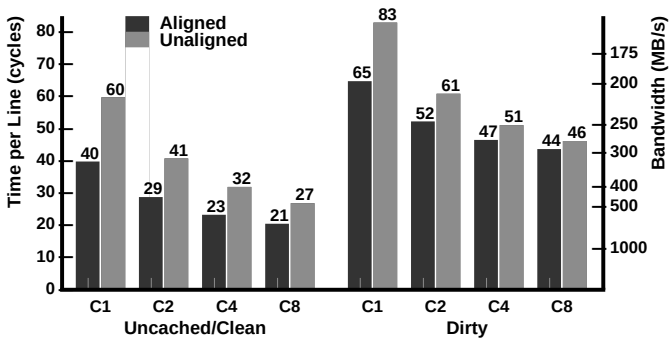


Figure 10: Sender side performance under varying degrees of chunking (32 line transfers).

As mentioned above, chunking only benefits the sender side. Our results show that with faster networks, the receiver side may become the bottleneck. Possible techniques that can speed up the receiver include a faster protocol processor, sending more data with each network message, or reducing the processing requirements at the receive side (e.g., by pushing some of the work to the sender side).

The optimizations described in the above two sections highlight the usefulness of programmability in the controller for optimizing protocol performance. Achieving similar optimizations would be more complex in pure hardware solutions.

5.2.4 Bidirectional Transfers

The simulation results we have shown up to now correspond to a single transfer in isolation. To consider contention effects, we also did an experiment with simultaneous transfers. For this purpose, we set up a bidirectional transfer between two nodes (i.e., both nodes are sending and receiving) using the chunked handlers with a chunking factor of 4. We also changed our test program so that the sends are posted before the receives are issued. This approximates the performance of receiver-initiated memory copy that may, for example, be used by the operating system to achieve page migration. Table 3 shows the total time to achieve a sin-

Table 3: Total time for bidirectional transfers (32 line transfers).

Cache States			
Sender	Receiver	Unidirectional	Bidirectional
Uncached	Uncached	16.2 μs	23.6 μs
Uncached	Clean	16.3 μs	27.0 μs
Dirty	Uncached	18.3 μs	27.8 μs
Dirty	Clean	18.4 μs	29.7 μs

gle transfer and a bidirectional transfer of 32 lines for the various caching states. The total time to achieve the bidirectional transfer is substantially less than twice the unidirectional transfer time in all cases, but higher than the single unidirectional time due to resource contention. Note that in the unidirectional case, the transfer time is essentially independent of receiver cache state (limited by network when uncached and by sender when dirty). However, in the bidirectional case, the performance is sensitive to the receiver state because the extra processing required on receives (to invalidate the cache) delays the sending on both sides.

5.3 Local versus Global Coherence

We ran a simple experiment to evaluate the performance of global coherence using the flexible controllers in FLASH as compared to an approach like Alewife [13] that uses a DMA engine. This experiment considers a transfer in which the local send buffer is cached dirty in a remote node. The Alewife approach requires global coherence to be implemented in two phases, the first phase *collect*s the dirty data and the second phase *send*s the data. Alewife performs these as distinct phases because the DMA unit has to send the message as a single stream and can not interleave the transfer with requests for dirty data. In FLASH, however, the programmability of the controller allows us to overlap and pipeline these two phases. 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 sending node and triggers a handler that forwards the line to the destination as well as writing it back to memory. Doing transfers as cache line sized components is critical for achieving this type of overlap and pipelining during the transfer.

To evaluate the two approaches, we implemented the Alewife-style collect and send in FlashLite using the mechanisms in FLASH. For the collect phase, we implemented a handler that sends requests for the dirty lines and collects the responses in a pipelined fashion. After all the lines are collected, we perform a chunked transfer (factor of 4) of the message to the receiver. We compared this to the pipelined implementation described above that is more appropriate given the mechanisms in FLASH. The Alewife-style collect and send transfer took approximately 31.2 μs to transfer 32 lines (including all startup latency and overhead), with the collect and send phase spending nearly equal times. Our pipelined implementation achieved the same transfer in approximately 20.6 μs, as compared to 17.7 μs if the lines were dirty in the local cache. Therefore, even though global coherence can be more costly than local coherence, the optimizations possible in FLASH can narrow the gap between these two models.

5.4 Simple Active Messages

The above evaluation has concentrated on the performance of data transfer in FLASH. Below, we briefly discuss the performance of a simple active message, remote fetch-and-add, implemented using the mechanisms in FLASH. This operation is initiated with a single uncached write to the alternate address space that specifies the command, address, and operand. This write is followed by a read to accept the result. The hold-off mechanism is used to ensure the validity of addresses provided to MAGIC. Approximately 25 cycles after the write is received by MAGIC, a request is sent to the home. The message spends 42 cycles at the home's PP, which includes time to read and modify the data in memory. On the return path, the originating PP takes 12 cycles to respond to the processor. After adding in network time and queuing delays through both MAGIC chips, the total latency of a remote fetch-and-add is 175 cycles (or 1.75 μs). Note that the overhead for implementing simple operations such as a fetch-and-add can be significantly smaller than the overhead of the example transfer protocol.

6 Discussion and Related Work

Though numerous systems have been proposed for supporting message passing protocols efficiently, most of them do not address the issues of integrating message passing with shared memory and cache coherence. Below we provide a comparison of our design with some of these systems.

Many recent systems and proposals advocate provisions for direct user-level access to message protocols. The messaging interface is typically either *memory mapped* or *register based*. The Connection Machine CM-5 provides access to the network through a memory mapped interface [21]. Register based approaches provide tighter coupling by moving the network interface into the processor and providing direct access to the interface through special registers [5, 9]. One of the problems with the above systems is that they are typically optimized for short messages, thus limiting the achievable bandwidth for large transfers. Another drawback is that the compute processor handles the complete transfer, thus taking cycles away from the main computation.

Several systems have proposed delegating message protocol handling to a second processor on the node to alleviate overheads and allow for overlap of computation and communication. The Intel Paragon [11] and *T [2, 16] designs advocate the use of a second processor on the same node. The key issues in these designs are the level of integration with the network, the protocol handling performance of the second processor, and the additional cost and complexity of providing this processor. Paragon uses a conventional processor that is not well integrated with the network. *T provides tighter network integration, but requires the use of custom processors as compute engines. Neither design has the ability to support shared-memory cache coherence. The approach in the Meiko CS-2 [10] is more similar to FLASH since protocol processing is delegated to a programmable network controller. However, the CS-2 controller contains a specialized DMA unit and is also not capable of supporting cache coherence protocols. In FLASH, the controller is optimized for efficient protocol handling and provides parallel support for data movement. In addition, the cost and complexity of the controller is amortized by handling both cache coherence and message passing in a single flexible unit.

SHRIMP [3] is a recent project at Princeton that advocates the use of simple network controllers for supporting message passing style communication. SHRIMP's philosophy is to separate protection and buffer management issues from the data transfer functionality and to only support the latter in hardware. SHRIMP provides support for communication by mapping local writes to remote addresses. Handling protection and buffering in advance is effective for communications with static buffering requirements. In fact, the flexibility of MAGIC allows us to leverage this same technique. However, for protocols with more dynamic buffering

requirements, the SHRIMP approach would require operating system calls or message copying on each transfer. SHRIMP provides two modes of data transfer, an explicit DMA transfer and an implicit transfer mode that gathers uncached processor writes and sends them to other nodes at a word or block granularity. The implicit mode inherently involves substantial processor and bus bandwidth overhead. The DMA mode partly alleviates this problem; however the processor is still involved for large transfers since the DMA must be reprogrammed for each page.

The Cray T3D [4] supports message passing within a single address space, but without cache coherence. The T3D supports two modes of transfer: small messages (32 bytes) that interrupt the destination processor, and large block transfer through a DMA engine. Both mechanisms incur high overhead: small messages incur an interrupt cost on every message, and large transfers must be initiated by an operating system call.

Alewife [1, 12, 13] attempts to integrate message passing and cache coherent shared memory within a single system. Each Alewife node has a hardware controller to handle the common cases of cache coherence, and a DMA unit (in the controller) to facilitate message passing. In addition, the main processor has an efficient memory-mapped interface to the controller that is used for controlling message sends. Though most coherence transactions are handled by the hardware controller, all user messages interrupt the processor for service. Thus, Alewife relies on hardware support for fast processor interrupts. The drawback to this approach is that interrupting the processor can take time away from other computation. In addition, Alewife does not provide support for virtual memory, and provides protection only between the kernel and user processes, leaving user processes unprotected from one another. Furthermore, while the Alewife research has addressed the issue of coherence of message data, only local coherence is supported in hardware.

Typhoon [19], a recent design that has been proposed by Wisconsin, shares a lot of the same philosophies as FLASH. The design uses a SPARC processor within the network controller to allow execution of software handlers. Therefore, many of the mechanisms we have discussed for efficiently supporting message protocols are directly applicable to Typhoon.

7 Concluding Remarks

FLASH attempts to integrate shared memory and message passing in a single architecture. We achieve this by using a programmable node controller, called MAGIC, to efficiently support both communication paradigms. MAGIC provides a tight coupling with the network and memory and allows for efficient protocol handling through parallel support for control processing and data movement. By delegating protocol handling to the controller, we achieve effective overlap between computation and communication. We provide a memory mapped interface that allows direct user-level access to MAGIC without compromising virtual memory and multiprogramming protection. To manage the cost and complexity of the design, message passing leverages virtually the same set of hardware mechanisms used for supporting cache coherence protocols. Using the same controller for both protocols also enables us to efficiently maintain coherence during message transfers.

We implement data transfer as a series of cache line sized messages. Our preliminary studies indicate that we can achieve peak data transfer rates of approximately 320 MB/sec, which is limited by the effective network bandwidth given the header overheads per cache line. Transferring data as cache lines uses the same set of mechanisms in MAGIC that have been optimized for supporting cache coherence protocols. Furthermore, it allows cache coherence operations to be interleaved with large transfers. The possible drawbacks of this approach are the network overhead to send the headers with each message and the time to process each message individually. Fortunately, the large cache lines in FLASH amortize the header overhead, and optimizations such as chunking can be used reduce the processing overhead. Another observation from our experiments is that providing coherence can be costly due to the latency of external accesses to modern processor caches. However, we showed that the flexibility of MAGIC, in combination with sending messages as cache lines, make implementing local and global coherence simpler and more efficient.

Overall, the use of programmable node controllers for supporting communication protocols appears to be a promising design approach. In particular, our experience shows that the flexibility and programmability of FLASH is extremely useful for efficiently supporting a wide variety of protocols and for experimenting with optimizations that would likely be considered too complex to implement in hardware-based systems. Finally, the techniques we presented are general and can be applied to other architectures that attempt to support efficient message passing or integrate message passing with shared memory.

8 Acknowledgements

We gratefully acknowledge the effort of the entire Stanford FLASH team, the authors of [14], who designed the FLASH architecture. We owe special thanks to Mark Heinrich for developing the FlashLite simulator, and to Joel Baxter and Supratik Chakraborty for developing PPgcc. We also thank John Chapin, John Hennessy, Mark Horowitz, Mendel Rosenblum, Jaswinder Pal Singh, Steven Woo, and the anonymous reviewers for valuable comments, and David Ofelt for working with us in the early stages of this research. We are also grateful to designers at Intel Supercomputer Systems Division for discussions that led to the protocol used in our study.

The FLASH project is funded by DARPA Contract N00039-91-C-0138. John Heinlein is supported by an Air Force Laboratory Graduate Fellowship (AFOSR). Kourosh Gharachorloo is supported by Digital Equipment Corporation's Western Research Laboratory. Scott Dresser is supported by a Hewlett-Packard Resident Fellowship. Anoop Gupta is partly supported by an NSF Presidential Young Investigator Award.

References

[1] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife machine: A large scale distributed-memory multiprocessor. In *Proceedings of the Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. This paper also appears as MIT/LCS Memo TM-454, 1991.

[2] Michael J. Beckerle. An overview of the START(*T) computer system. Motorola Technical Report MCRC-TR-28, Motorola, Inc., One Kendall Square, Building 200, Cambridge, MA 02139, July 1992.

[3] Matthias Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward Felten, and Jonathan Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 142–153, April 1994.

[4] Cray Research, Inc. *Cray T3D System Architecture*, 1993.

[5] W. Dally, J. Fiske, J. Keen, R. Lethin, M. Noakes, P. Nuth, R. Davison, and G. Fyler. The message-driven processor: A multicomputer

processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–39, 1992.

[6] Stephen Goldschmidt. *Simulation of Multiprocessors: Accuracy and Performance*. PhD thesis, Stanford University, June 1993.

[7] John Heinlein, Kourosh Gharachorloo, and Anoop Gupta. Integrating multiple communication paradigms in high performance multiprocessors. Technical Report CSL-TR-94-604, Stanford University, Computer Systems Laboratory, February 1994.

[8] 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.

[9] Dana S. Henry and Christopher F. Joerg. A tightly coupled processor-network interface. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 111–122, September 1992.

[10] Mark Homewood and Moray McLaren. Meiko CS-2 interconnect Elan-Elite design. In *Proceedings of Hot Interconnects 93*, August 1993.

[11] Intel Corporation. *Paragon XP/S Product Overview*, 1991.

[12] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating message passing and shared-memory: Early experience. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 54–63, May 1993.

[13] John Kubiatowicz and Anant Agarwal. Anatomy of a message in the Alewife multiprocessor. In *Proceedings of the 7th ACM International Conference on Supercomputing*, July 1993.

[14] 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*, pages 302–313, April 1994.

[15] Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report No. CS-93-214, University of Tennessee, November 1993.

[16] Rishiyur Nikhil, Gregory M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 156–167, May 1992.

[17] Paul Pierce, Intel Supercomputer Systems Division, November 1993. Personal Communication.

[18] Paul Pierce. The NX/2 operating system. In G. Fox, editor, *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, volume 1 of 2, pages 384–390, 1988.

[19] 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*, pages 325–336, April 1994.

[20] Michael David Smith. *Support for Speculative Execution in High-Performance Processors*. PhD thesis, Stanford University, November 1992. Tech. Report CSL-TR-93-556.

[21] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*, 1991.

[22] Thinking Machines Corporation. *Programming the NI*, March 1992.

[23] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

Appendix A Sample Handler Code

To illustrate the type of operations performed by transfer handlers, we have included the code for the clean case in the transfer handler discussed in Section 5.2.2 below (note that the speculative optimization discussed in Section 5.2.2 only affects the dirty case which is not shown here due to space constraints). Horizontal lines denote cycle boundaries in the statically scheduled code for the dual-issue protocol processor. The assembly mnemonics are those used by the MIPS/DLX architecture, with a few extensions (mainly for bit manipulation). The handler begins by loading the relevant transfer state, followed by checking the directory state for the line. If the line is clean, the handler fills the data buffer from memory, forms the message headers (fields are shifted and inserted into previously prepared header templates), and updates the relevant transfer state (e.g., sequence number, remaining bytes to send). Finally, the handler sends the message component to the network and reschedules itself on the software queue. Note that instructions for the various subtasks are intermixed due to code scheduling.

```
TransferHandler:
    add     $4,$17,$0           # Move operation record ptr to $4
    ls      $14,-24576($0)      # Allocate data buffer
    addi    $sp,$sp,65488       # Allocate stack space
    ld      $11,224($4)         # Load current transfer address
    bbs     $14,0,AllocFail     # Did data buffer allocation fail?
    nop
    andfi   $5,$11,7,23         # Extract local address, no offset
    nop
    srl     $8,$5,4             # Generate offset into directory
    nop
    beq     $11,$0,AddrFault    # Check for stale translation
    add     $27,$21,$8          # Add offset to base of directory
    nop
    ld      $7,0($27)           # Load directory state
    nop
    sd      $11,32($sp)         # Save transfer address
    bbs32   $7,16,LineBusy      # Check for busy directory state
    nop
    bbc32   $7,18,Clean         # Test clean bit... is line clean?
    ld      $17,176($4)         # Speculative load for dirty case
    sll32   $14,$14,27          # Shift buffer number
    add     $18,$11,$0          # Speculative op. for dirty case
    ...                         # Handle dirty case
Clean:
    andfi   $11,$11,0,25        # Extract local address (bits 0-25)
    ld      $5,112($4)          # Load current sequence number
    nop
    ld      $9,168($4)          # Load 1st header template
    sll     $2,$5,24            # Shift sequence number
    add     $27,$11,$14         # Form data buffer fill command
    insfi   $9,$2,24,55         # Insert sequence number into header
    ld      $2,232($4)          # Load remaining byte count
    addi    $5,$5,1             # Increment sequence number
    lblock  0($27)              # Issue fill command to memory
    addi    $2,$2,65408         # Subtract 128 from remaining bytes
    sd      $5,112($4)          # Store sequence number
    addi    $5,$0,127           # Load constant for comparison
    sd      $2,232($4)          # Store remaining byte count
    sgteu   $8,$5,$2            # Is there more to transfer?
    ld      $17,160($4)         # Load 2nd header template
    bltz    $8,LastLine         # Branch to special code if done
    ld      $27,32($sp)         # Recall transfer address
    add     $18,$9,$0           # Move 1st header to $18
    insfi   $17,$14,59,62       # Insert buffer number into header
    addi    $2,$27,128          # Increment transfer address
    send    $17,$18,6           # Send component to network
    j       SWQReschedule       # Schedule next transfer
    sd      $2,224($4)          # Store transfer address
    addi    $sp,$sp,48          # Free stack space
    nop
```