Integrating Multiple Communication Paradigms in High Performance Multiprocessors

John Heinlein, Kourosh Gharachorloo, and Anoop Gupta

Technical Report CSL-TR-94-604

February 10, 1994

Computer Systems Laboratory
Stanford University
Stanford, CA 94305-4070

Abstract

In the design of FLASH, the successor to the Stanford DASH multiprocessor, we are exploring architectural mechanisms for efficiently supporting both the shared memory and message passing communication models in a single system. The unique feature in the FLASH (FLexible Architecture for SHared memory) system is the use of a *programmable* controller at each node that replaces the functionality of hardwired cache coherence state machines in systems like DASH. The base coherence protocol is supported by executing appropriate software handlers on the programmable controller to service memory and coherence operations. The same programmable controller is also used to support message passing. This approach is attractive because of the flexibility software provides for implementing different coherence and message passing protocols, and because of the simplification in system design and debugging that arises from the shift of complexity from hardware to software.

This paper focuses on the use of the programmable controller to support message passing. Our goal is to provide message passing performance that is comparable to an aggressive hardware implementation dedicated to this task. In FLASH, message data is transferred as a sequence of cache line sized units, thus exploiting the datapath support already present for cache coherence. In addition, we avoid costly interrupts to the main processor by having the programmable engine handle the control for message transfers. Furthermore, in contrast to most earlier work, we provide an integrated solution that handles the interaction of message data with virtual memory, protected multiprogramming, and cache coherence. Our preliminary performance studies indicate that this system can sustain message transfers at a rate of several hundred megabytes per second, efficiently utilizing the available network bandwidth.

Key Words and Phrases: message passing, shared memory multiprocessors, cache coherence, FLASH, virtual memory, protection, communication

1 Introduction

Shared memory and message passing are the two predominant communication models used by multiprocessor systems, each with its own advantages. Shared memory (also called a *shared address space*) provides the programmer with a simple memory abstraction and is well suited for programs that exhibit fine-grain or dynamic communication. In contrast, message passing leaves memory management under more explicit program control and often achieves higher performance for certain types of communication such as bulk data transfer. Since each model has its merits, an architecture which efficiently supports both models enables a programmer to use the more appropriate communication model, or even use a hybrid of the two.

Even though these models are quite different from a programmer's point of view, scalable shared memory systems and scalable message passing systems are strikingly similar in their architecture. Typically, both systems consist of high performance nodes connected by a general interconnection network. Furthermore, the memory in these systems is usually distributed among the nodes to increase memory bandwidth. Finally, nodes in both systems usually contain a controller which either maintains cache coherence (e.g., Stanford DASH [LLG+92] and Alewife [ACD+91, KA93, KJA+93]) or supports message passing (e.g., Intel Paragon). Given the architectural similarities between shared memory and message passing systems, we are investigating the mechanisms which are required to integrate these two paradigms into a single architecture.

Integrating the shared memory and message passing models efficiently is one of the major goals of the FLASH architecture currently being designed at Stanford [KOH+94]. One unique aspect of FLASH (FLexible Architecture for SHared memory) is the use of a programmable controller instead of the hardwired finite state machines that implement the cache coherence protocol in DASH. In FLASH, this programmable controller, called MAGIC (Memory And General Interconnect Controller), is used to implement *both* the cache coherence and message passing protocols. The programmability of MAGIC provides substantial flexibility in selecting protocols, and makes their design and debugging much easier. In addition, we can leverage the programmability to avoid dedicated hardware mechanisms customized for each type of protocol.

This paper focuses on the hardware and software mechanisms for supporting efficient message passing in the FLASH architecture. We have set several goals for message passing on FLASH, which guide the study of these mechanisms. First, we aim to provide message passing throughput comparable to systems with dedicated message passing hardware. We further enhance system performance by allowing message passing communication to occur in parallel with computation. Second, we want to achieve very low overhead for user level message passing. One way we reduce the overhead is by avoiding copies of the message data at both the sender and receiver side. Third, our goal is to present a seamless programming environment that integrates message passing and shared memory. To fully achieve this third goal, message passing must interact smoothly with cache coherence, virtual memory, and protection. We show that we are able to achieve these goals with minimal hardware overhead beyond that which is required for cache coherence.

The remainder of this paper is organized as follows. Section 2 provides a broad overview of the FLASH architecture, concentrating on features that are relevant to message passing. We illustrate the use of the programmable controller by discussing a basic message passing protocol in Section 3. In Section 4, we discuss mechanisms to support the interaction of message passing with cache coherence, virtual memory, and protection. Section 5 presents preliminary estimates of message passing performance on FLASH. Finally, Sections 6 and 7 provide a general discussion of our approach, including related and ongoing work, and concluding remarks.

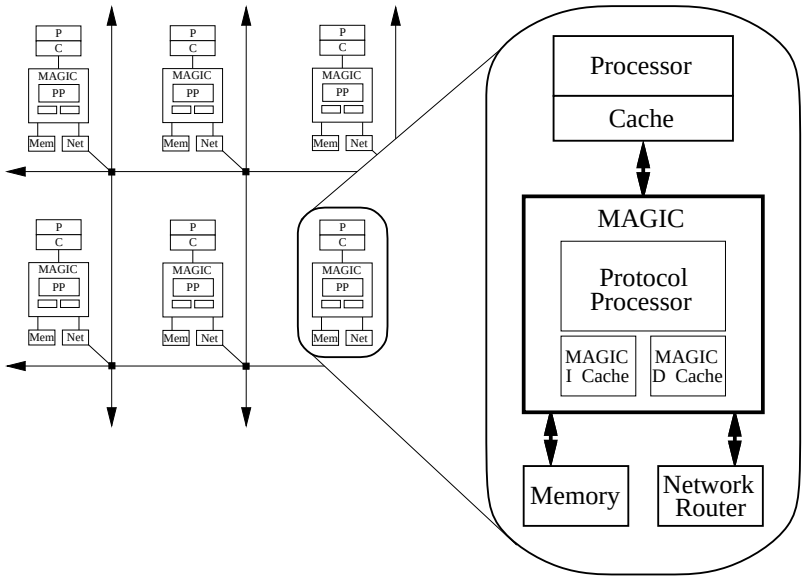


Figure 1: FLASH node architecture and interconnect.

2 FLASH Architecture

The FLASH multiprocessor [KOH+94] (FLexible Architecture for SHared memory), is a follow-on to the DASH [LLG+92] multiprocessor. Since the details of the FLASH architecture have already been published, we only present an overview of the aspects relevant to message passing. FLASH consists of a large array of processing nodes connected by a low-latency, high-bandwidth network (See Figure 1). Each node contains a high-performance processor with caches, a portion of the distributed main memory, a network port, and a programmable controller chip called MAGIC (Memory and General Interconnect Controller). The MAGIC chip contains a programmable protocol processor and dedicated hardware to tightly connect the main processor, network, and memory. By integrating dedicated hardware units and a programmable processor, FLASH can efficiently support flexible communication protocols.

FLASH nodes communicate with each other by sending inter-node commands, called *messages*.[1] Messages are split inside MAGIC into their control and data components. Dedicated hardware is used to efficiently transfer the data through MAGIC in parallel with the control processing that is mainly accomplished by the programmable protocol processor (for more details, see [KOH+94]).

Since MAGIC processes all messages from the processor and network, its architecture was designed specifically to provide flexible and efficient support for processing messages. MAGIC contains a statically-scheduled dual-issue processor, the *protocol processor* (PP), that executes the protocol code (called *handlers*). The PP stores protocol state (including directory state) in a portion of main memory rather than in a special memory like DASH. MAGIC also contains its own instruction and data caches which provide efficient access to protocol code and state.[2] To further

[1] To distinguish between these low-level protocol messages and messages in the sense of message passing, we will refer to the latter as *user messages*. Thus in this paper, as in [KOH+94], *message* refers to a single low-level command issued to another FLASH node.

[2] MAGIC has access to all regions of memory, but not all data is required to go through its caches. Therefore, MAGIC can transfer data between the memory and processor or network without caching such data, and without interfering with processor cache coherence.

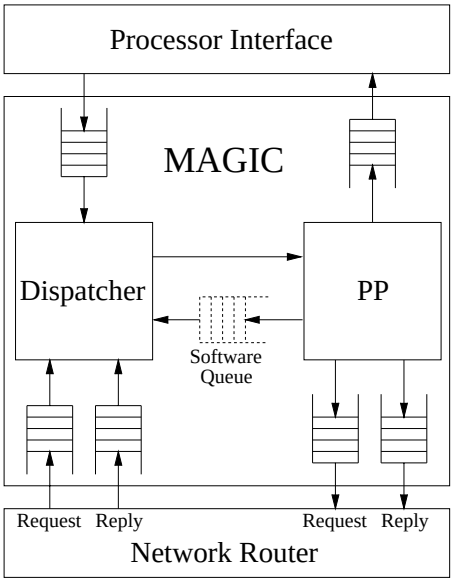


Figure 2: Simplified diagram of protocol processor queues

enhance handler efficiency, the PP also has several special instructions which provide powerful bit manipulation and other support for common protocol operations. In addition to the PP itself, MAGIC contains several dedicated hardware units that provide support functions.

To increase message throughput, MAGIC allows multiple messages to be processed at once in a pipelined fashion. The PP and other dedicated hardware units form a three-stage macropipeline. This macropipeline is separate from the conventional processor pipeline inside the PP. In the first stage, the *Inbox* prepares a request for processing by the PP, selects the handler to execute, and in some cases initiates a speculative memory operation. In the second stage, the PP executes the handler code to process the message. In the third stage, the *Outbox* assumes control of any messages sent by the PP to the processor or network. *Data buffers*, an array of cache line sized registers, allow data to be staged through MAGIC in a pipelined fashion.

MAGIC contains hardware queues for processor and network messages and a software queue which is used by the PP to store tasks for future service. Figure 2 provides a logical view of these queues (the "Dispatcher" shown is part of the Inbox stage in the macropipeline). The processor and network queues are implemented in hardware on the MAGIC chip. Similar to DASH, the network port consists of two independent channels to allow for a deadlock-free cache coherence protocol. In contrast to the hardware queues, only the request at the front of the software queue is stored on the MAGIC chip, while the rest of the queue is stored in memory. Section 3.3.2 describes the implementation of the software queue in more detail. As we will see, the software queue is particularly useful for message passing on FLASH, though it is sometimes employed for cache coherence as well.

To keep the design simple, the protocol processor in MAGIC excludes many features found in modern RISC microprocessors. For example, it does not have support for a TLB, interrupts, or floating point since the cache coherence protocol does not require them. Though hardware support for address translation would be useful for message passing, we will discuss efficient techniques to achieve the functionality in software.

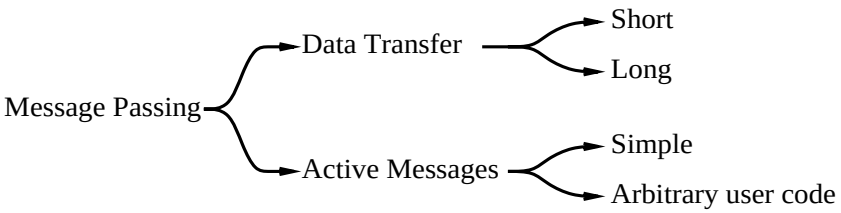


Figure 3: Uses of message passing.

3 Message Passing Implementation

Figure 3 shows a high-level view of the two main uses for message passing primitives. The first type of use is for *data transfer*. Data transfer can be divided into short messages and long messages (i.e., bulk data transfer). The second type of use is *active messages*, which invoke computation at the destination node along with the data transfer [DCF+89, vECGS92]. Active messages can also be divided into two categories, those that invoke simple computation (e.g., a Fetch-and-Op) and those that invoke more complex computation approaching the functionality of remote procedure calls (i.e., arbitrary user code). In this paper, we focus on a subset of this wide range of uses. For data transfer, we concentrate on supporting coarse grain transfers since this complements the fine grain transfer capability that is already available through coherent reads and writes in a shared-memory architecture such as FLASH. For active messages, we focus on uses that invoke relatively simple computation at the destination node.

The majority of this section describes a basic send/receive model which is used as an example to convey the core mechanisms in FLASH to support message passing and data transfer protocols. In this model, referred to as *base message passing*, the sending node provides the message data and designates the receiving node for the data. The receiving node in turn decides where in its memory the user message data should be stored. Later in the section, we describe an alternate model referred to as *memory copy* in which the sender directly specifies the address where the data should be stored in the receiver's memory. We would like to emphasize that the example protocols presented in this section mainly serve the purpose of motivating the mechanisms used in FLASH for supporting data transfer. Other protocols can be implemented on FLASH by simply changing the MAGIC handler software.

In this section, we will refer to three distinct phases in transferring user data:

Initiation The sending processor initiates the user message through a set of commands to MAGIC that describe the user message's parameters.

Transfer The MAGIC chip on the source node then transfers the user message data to the destination node as a series of cache line sized blocks.

Reception When the first message block arrives, the MAGIC chip on the receiver node reserves buffer space for the entire user message in the receiver process's address space. As each of the subsequent lines of the user message arrives, the receiving MAGIC chip accumulates the data into the previously reserved buffer. When complete, the receiving MAGIC chip notifies the receiver that a user message has arrived through setting a flag or sending an interrupt, and notifies the sender that the transfer was successful through an acknowledgement message.

The remainder of this section focuses on the base mechanisms used for data transfer. Section 3.1 describes the programmer's view of the base message passing model. Sections 3.2–3.4 describe the implementation of the

Example Message Send:

```
MID *mid;                                          /* Message ID: used for monitoring status */

mid = send(Type, DestPID, SourceBuf, Length);      /* Non-blocking message send */
/* Do other work – May not modify SourceBuf yet */
while (!senddone(mid));                            /* Spin until send completes */
/* Free to modify or deallocate SourceBuf */
```

Example Message Receive:

```
char *msg;                                         /* Pointer to message text */
short SenderPID;                                   /* Sender's PID */
unsigned long Length;                              /* Message Length */

bufalloc(BufferSize, Type);                        /* Preallocate buffer space */
⋮
msg = receive(Type,&SenderPID,&Length);            /* Non-blocking Receive of msg type Type*/

if (SenderPID != -1) {                             /* Received a message... handle it */
⋮
}
```

Figure 4: Examples of code using base message passing primitives.

model on FLASH according to the three phases described above. Section 3.5 discusses the modifications to the implementation for supporting the memory copy model. We conclude by describing the implementation of simple active messages, such as Fetch-and-Op, in Section 3.6. The interaction of data transfer with cache coherence, virtual memory, and protection will be discussed in Section 4.

3.1 Programmer Model

Our base message passing model is a basic send/receive model that has similarities to the Intel NX protocol [Pie88]. The user invokes functions for sending, receiving, and message buffer management through a set of user-level library calls. Messages are tagged with a user-defined *type*, similar to the Intel NX protocol. To avoid extra copying, the protocol requires the application to allocate buffer space for incoming user messages in advance. Thus, the received user message can be directly transferred into the user address space and the address of the buffer where the incoming message resides can be provided as the return value of the receive call. As discussed before, we use this protocol simply for the purpose of illustrating the mechanisms used in FLASH for achieving bulk data transfer. The mechanisms described are general and can be used to implement numerous other protocols.

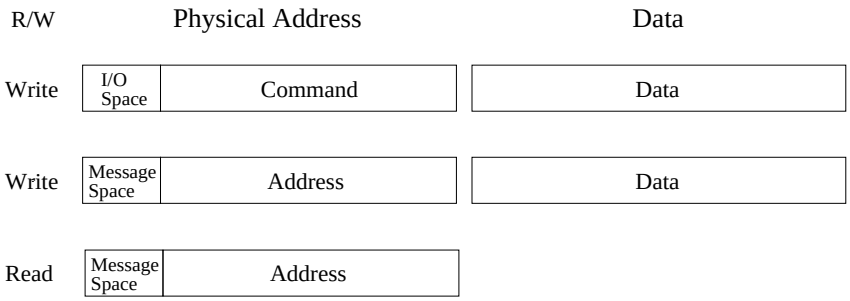


Figure 5: Processor to PP command formats.

As shown in Figure 4, a program uses the `send` library call to initiate a user message transfer. The send is asynchronous and the call returns immediately to allow the processor to continue while the transfer is in progress. However, the processor may not modify or deallocate the send buffer until the transfer is complete. The send call names the receiver by its "virtual process id" that ranges from 0 to $P-1$, where P is the total number of processes in the application. The return value from the `send` call, called a message ID or "MID", is used to identify the send as an argument to the `senddone` library call which returns the status of the user message.

Figure 4 also shows the receive side call. As discussed above, buffers for user message reception should be allocated in advance using the bufalloc routine.[3] This allows the system to avoid the overhead of storing the data in system space and then transferring it to the user space. The return value of the receive call provides the address where the user message is stored. In addition, the sender and length of the user message are returned. Buffers that are no longer in use can be returned through a special call to the pool of incoming buffers available to MAGIC.

3.2 Initiating a Message

The first of the three phases of a message transfer is *initiation*, in which the processor describes the user message to MAGIC. The various actions required by this phase are performed by the `send` library routine, which abstracts the details from the user. The main processor communicates with the MAGIC chip through a memory-mapped interface. FLASH divides the physical address space into four regions. For normal operation, programs use the base address space for accessing memory. The initiation phase uses uncached reads and writes to some of the alternate address spaces to communicate with MAGIC.

Using this memory mapped interface, we implement a simple command mechanism for communicating with MAGIC. When MAGIC sees a read or write to the alternate address space, the PP interprets the address and data differently from a regular read or write. A write can be used to provide an address and data pair to the MAGIC chip. Similarly, a read provides an address and expects a return value. Any of the information communicated to MAGIC can be interpreted as either addresses or data, and in fact, several arguments may be passed by packing bit fields in a single word.

By convention, we currently use two of the alternate address spaces defined in the FLASH architecture for sending information to MAGIC. The first command type we use, shown in Figure 5, is issued in the *I/O space*. In this type, the address the program writes specifies the command, and the data it writes specifies an argument. The second command type, issued in the *message space*, is used for communicating an address to MAGIC as well as

[3] If an incoming message does not find a preallocated buffer, it is placed in system space and will later get transferred to user space.

```
⋮
uncached enum {INIT_SUCCESS, INIT_RETRY, INIT_FAIL} InitResult;

do {                                    /* Repeat until protocol accepted */
    /* Begin Initiation Protocol */
    Issue Transfer Command
    Issue First Page Command
    ⋮
    Issue Last Page Command
    InitResult = Read Status
    /* End Initiation Protocol */
} while (InitResult == INIT_RETRY);

    /* Protocol complete */

if (InitResult == INIT_FAIL) {
    /* Message Description Invalid */
} else {
    /* Message Accepted */
}
⋮
```

Figure 6: Conceptual library routine implementation for a message send.

a word of data. In fact, the processor writes to a virtual address, and MAGIC sees a translated physical address. This allows the library to send authentic physical addresses at the user level using this command type (we discuss this in more detail in Section 4.2). The third kind of command, uncached reads, are issued in the I/O space, and allow MAGIC to return a value to the processor.

The use of an alternate address space and memory-mapped commands has two key performance advantages. First, the dispatcher on the MAGIC chip can distinguish alternate address space accesses from normal memory accesses and select a special handler. Thus the handlers for regular coherent reads and writes are unaffected and don't need to spend time checking for alternate accesses. Second, once the OS has installed mappings for the alternate address spaces, message initiation occurs entirely at the user-level, thus eliminating the overhead of system calls that would otherwise be required to guarantee protection and address authenticity.

The `send` library routine issues a series of alternate address space accesses to describe the user message. There is a simple protocol by which these accesses are issued by the processor and accepted by MAGIC; we refer to this as the *initiation protocol*.[4] MAGIC maintains structures in memory called *sender records* on a per-process basis that are used to accumulate the information in the commands issued by each user process. The operating

[4]The same grouping of commands to describe a complex operation is applicable to other uses such as synchronization, etc., however these other uses are beyond the scope of this report.

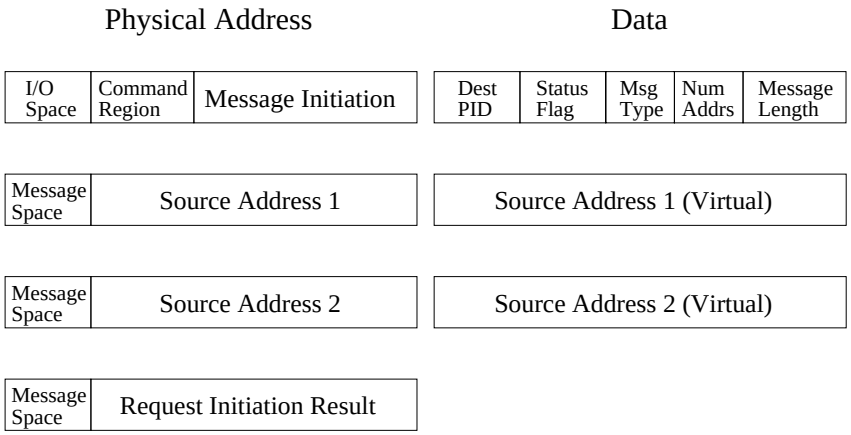


Figure 7: Sample commands in a message initiation protocol.

system is required to inform MAGIC at context switches of the new PID that is running. This allows MAGIC to determine which record should be used to accumulate the command information. Figure 6 shows the pseudo-code for implementing the `send` library routine. Below we describe the commands that this pseudo-code generates, which are illustrated in Figure 7.

The first command is a write to the I/O space that indicates to MAGIC that a user message description will follow and provides some additional information about the user message in its data word. The next several writes provide addresses for the user message data. The number of commands issued depends on the number of pages spanned by the user message data (this number is specified by the library in the data field of the I/O command), with each write providing the address of an entire page or fraction of a page. Figure 7 shows that these addresses are provided through message space commands. (The example assumes the user message data spans two pages.) The need for providing the virtual addresses in the data part of these writes will become apparent when we discuss the interaction with virtual memory in Section 4.2. MAGIC stores the information conveyed by these writes in the sender record as part of the user message description.

The last command is an uncached read to the message space that performs two functions. First, it notifies MAGIC that the user message description is complete. Second, it requests a response from MAGIC to indicate if the user message description is accepted. An acknowledgement response indicates a successful initiation protocol and thus the processor can continue. However, this acknowledgement does not indicate the completion of the transfer. The *status flag* indicated to MAGIC in the first command (see Figure 7) indicates one of several pre-designated locations used to indicate user message status. This status flag, which the sender can poll, is updated by MAGIC when the transfer is complete.

The initiation protocol may be rejected by MAGIC for one of several reasons, each signified by a different response code returned for the final read. The first class of rejection (“retry”) signals a temporary problem. The initiation protocol should be retried when this response is returned. This may occur, for example, because a process has too many user messages outstanding, or because virtual memory paging took place during the initiation protocol (described later). The second class of rejection (“failure”) is more serious. The initiation should not be retried after a failure response because it will merely fail again. This class of failure would occur if the description were invalid—because of an invalid destination process number, for example. If the `send` library call is used, the second

class of failure should not occur, since the library correctly formats the commands. It can usually only occur if the commands are directly issued by the user program.

Since the `send` library routine executes at user priority and involves multiple operations, it is possible for the initiation protocol to be interrupted before it is complete. However, our approach makes user message initiation appear to be atomic. Therefore, the initiation protocol is not necessarily corrupted by context switches. As we discussed before, the information communicated to MAGIC during an initiation protocol is accumulated in a per-process sender record. On a context switch or interrupt, the OS is required to tell MAGIC the PID of the new process. This information is used by the PP handlers to select the "active" sender record on incoming initiation protocol commands. Since there is a per-process sender record, the initiation protocol of a given process can transparently resume when it is switched back in. We further ensure the atomicity of user message initiation by delaying any irreversible actions until the final command in the protocol is issued. On the last command, MAGIC decides atomically whether to commit or abort the entire operation. Therefore, user message initiation can be aborted by the user program up until the last command in the protocol is issued.

Using the initiation framework we have described, we are able to flexibly communicate information between the processor and the MAGIC chip. The commands from the processor to MAGIC consist mainly of uncached writes, which can be efficiently pipelined by the main processor. In addition, uncached reads may be used to efficiently request information from the MAGIC chip. Finally, as we discussed, this technique interacts smoothly with interrupts and context switches and achieves atomic initiation of complex operations such as a user message send.

3.3 Sender Side

This section describes how the actual message transfer is done on the sender side once the message initiation protocol has completed successfully. The data transfer is accomplished by sending a series of cache line sized messages. As we will see, the software queue is used to allow the servicing of other requests by MAGIC to be interleaved with a large transfer that is in progress.

3.3.1 Message Transfer

The transfer of the user message begins after the initiation protocol is accepted by MAGIC. The sender record constructed during the initiation protocol holds the description of the user message and the up-to-date transfer status for the duration of the transfer. As with directory state for the cache coherence protocol, sender records are accessed through the MAGIC data cache for efficiency. The PP begins the transfer by sending a header message to the receiver that contains information about the user message. The user message data follows in the form of cache line sized messages.

The header message sent to the receiver contains information about the user message such as sender PID, destination PID, length, and type. The header also contains a *message number* that is used to uniquely identify the user message. The message number is a merge of the sender's node number and a non-decreasing count maintained by the sender. By including the message number in the remaining messages from the sender, the receiver can correctly associate the message with a given transfer. We assume the network maintains point-to-point order of messages; thus the header information arrives at the destination before the data components of the user message. However, our solution can be generalized to an environment without point-to-point order with a small additional overhead.

MAGIC sends the user message data in a series of cache line sized messages that we call *components*. To form a component, the PP reads the appropriate memory line from the source buffer and sends the data to the receiving node. This continues until all memory lines in the source buffer have been sent to the receiver. Each component is labeled with an offset in addition to the message number described above; this offset serves as a sequence number that allows the receiver to uniquely identify the component and also to detect missing components.[5] After the last component is sent, the sending MAGIC chip expects an acknowledgement message from the receiving node's MAGIC chip. The sender record and send buffer remain intact until the acknowledgement arrives; this allows for retransmission of the data in case of an error. Note that breaking a large user message into cache line sized messages is handled entirely by MAGIC without any main processor involvement. In addition, the transfer is accomplished without requiring a DMA engine, and uses only the hardware features that are provided for cache coherence (message alignment is the only exception; this will be discussed shortly).

Sending the user message as a series of cache line sized messages instead of as a single large message has several advantages and only a minor disadvantage:

- The memory system and MAGIC are already optimized for transferring cache lines and therefore cache line packets are handled efficiently.
- By sending cache lines, the message passing protocol can leverage the deadlock avoidance and recovery techniques already used for the base coherence protocol.
- Networks typically perform better (due to lower contention) when large messages are broken into pieces. Systems such as Alewife, which send all the data in a single message may not perform as well.
- One drawback is that cache line sized transfers incur the header overhead (i.e., 16 bytes) on each cache line. However, this does not affect our overall performance significantly since cache lines are relatively large in FLASH (128 bytes).

3.3.2 Scheduling the Transfer Using the Software Queue

We use the software queue for scheduling a user message transfer into multiple invocations of the transfer handler. From a fairness point of view, it is important to interleave other requests (e.g., cache coherent reads and writes) while the PP is in the process of doing a large user message transfer. In addition, attempting to complete the entire transfer can cause the outgoing queues to fill up. Unless the PP is relinquished to service other requests when the queues are full, the machine may deadlock.

Each time the message transfer handler is invoked, it sends several components of the user message, updates the transfer state in the sender record, and reschedules itself using the software queue. The software queue consists of a linked list of sender records and similar structures for other operations. Each record has two sections: a fixed-format header and a variable region that is operation dependent. The fixed-format header allows a single scheduling routine to enqueue or dequeue different types of tasks. The software queue resides in memory, and only the fixed-format header of the first task on the queue is stored in a MAGIC hardware register. Any task that is invoked from the software queue is required to run a scheduling routine to load the register with the next task on the queue before it relinquishes the PP. The scheduler can achieve back-to-back service for a given task by rescheduling it at the

[5] If a gap in the sequence numbers is detected due to message loss in the network, the protocol requests retransmission of the missing (and subsequent) lines.

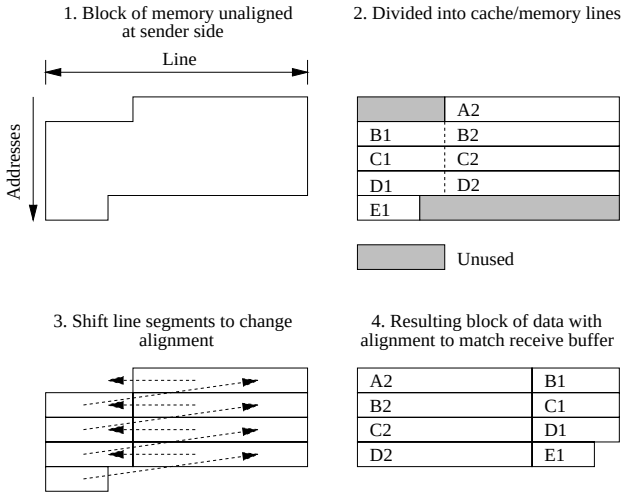


Figure 8: Support for arbitrary user message alignment.

head of the queue; the PP can run the task immediately if there are no requests on the other incoming queues (i.e., processor and network queues).

By using the software queue to schedule large transfers across multiple PP invocations, other requests can be serviced by the PP until the next time the transfer handler is invoked. Though each component of the user message is sent separately, the PP can send multiple components during each invocation of the transfer handler. 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 plan to experiment with a variety of chunking granularities once the system is fully implemented.

3.3.3 Message Alignment

So far, we have implicitly assumed that data sent in a user message is aligned at cache line boundaries in the sender and receiver memory. To provide a general solution for message passing, we must also consider messages that are not aligned in this way. For example, transfers may begin in the middle of a cache line or may be destined for a buffer that has a different cache line alignment than the source memory. Figure 8 shows an example message transfer from a source buffer that begins and ends in the middle of cache line boundaries (panels 1 and 2). The same figure (in panels 3 and 4) also shows how the alignment needs to be changed if, for example, the destination buffer is cache line aligned. Shifting the data during transfer without any hardware support can result in a potentially unacceptable loss in performance. In addition, since we perform the data transfer a cache line at a time, the adjustment for alignment has to be done for all the lines sent.[6] Therefore, we have included a simple hardware mechanism in FLASH to make alignment adjustments more efficient.

The hardware alignment mechanism we have adopted in FLASH allows the MAGIC chip to flexibly load cache line unaligned data into the data buffers. As mentioned before, the data buffers are cache line sized registers used to transfer data in a pipelined fashion among the processor, network, and memory. Data buffers are normally

[6] Systems that send all the data in a single message (or a small number of messages) are less affected by alignment issues since the adjustments can be confined to only the beginning and end of the message. Thus, the cost of performing the adjustment is negligible for large messages. As we discussed earlier, however, there are several advantages to dividing the transfer into smaller (cache line sized) messages.

loaded with data that is cache line aligned (this is always true in the cache coherence protocol). The alignment mechanism extends this ability by allowing a line of memory to be loaded in a data buffer starting at an arbitrary 64-bit doubleword (the starting point is determined based on the relative offset between the sender and receiver memory buffers). We choose a 64 bit alignment granularity because we believe it will cover the frequent cases and is simpler to support as compared to arbitrary byte granularity. The portion of the data line that extends past the end of the data buffer "wraps" into a second buffer specified with the memory load. A load of the succeeding memory line can then fill the second buffer, forming a complete component with the desired alignment. To assure 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.[7]

The alignment mechanism described above requires the sending node to perform the required adjustment for unaligned data. To utilize the mechanism the sender needs to know the alignment of the receiving buffer. In the memory copy model that we discuss later, the sender knows the destination address and can therefore immediately begin transferring lines with the correct alignment. Unfortunately, this is not true for the base message passing protocol. Therefore, our approach for base message passing is to assume the receiver buffer is aligned. Once the receiver selects the receiver buffer, the sender can be notified to transmit the remaining lines with the appropriate alignment.

3.4 Receiver side

At the destination node, there are three stages in user message reception: the preallocation and selection of receive buffers, the reception of user message components, and the detection and notification for completion.

3.4.1 Allocation of Receiver Buffers and Data Reception

Our goal in message reception is to directly store the incoming message into the user's address space to avoid extraneous copying between system and user buffers. To achieve this, we require the program to preallocate a set of buffers that are designated for arriving messages. As discussed previously, the `bufalloc` call allocates buffer memory and communicates the appropriate addresses to MAGIC through commands similar to those described for message initiation.

On arrival of a header message, the PP reserves a portion of the preallocated buffer space for the incoming user message data. Buffer space is selected based on the destination process and user message type. The PP then constructs a *receiver record* for the message that contains the user message description and pointers to the buffer space that is reserved. This structure is also used to keep track of the number of components that are received.

When each component arrives, the handler on MAGIC first looks up the appropriate receiver record in a hash table indexed by the message number. Once the record is located, the base address of the receive buffer is combined with the message offset to generate the physical address where the data should be stored. The handler also checks that the message is in sequence (e.g., to detect message loss). Furthermore, the handler checks the directory state, possibly taking action to maintain cache coherence. Finally, it updates the count of remaining components and stores the data.

[7] We relinquish data buffers across invocations of the transfer handler since these buffers are an important resource used by other handlers. Therefore, any buffer that is partially filled with unsent data needs to be refilled with that data by the next invocation of the transfer handler.

Example Memory Copy:

```
MID *mid;                                          /* Message ID: used for monitoring status */

mid = fastmemcpy(SourceBuf, Length, DestBuf);      /* Send Message */
/* Do other work – May not modify SourceBuf yet */
while (!senddone(mid));                            /* Spin until send completes */
/* Free to modify or deallocate SourceBuf */
```

Figure 9: Example of sender side using memory copy primitives.

3.4.2 Completion Notification

When the destination MAGIC receives the last component of a user message, the PP detects that the user message is complete because the remaining line count reaches zero. The PP then notifies the receiving process of the user message by storing information about the user message (sender PID, address, length) in a structure called the *receive table* in the receiver process's address space. Then, it composes an acknowledgment to the sender, which tells the sender to consider the user message delivered.

The `receive` library routine uses the receive table to find user messages that have arrived for the process. A non-blocking `receive` call polls this table for the arrival of a new user message. Since the receive table is in user space, no system call is required to poll for user messages. A blocking `receive` call might be implemented to suspend the process if no user messages are present. In this case, MAGIC can interrupt the processor when a user message arrives.

The completion scheme described here does not address the issue of ordering among different user messages (i.e., preserving the delivery order for successive user messages). Under the protocol we described, a short user message sent after a long one could actually finish its transfer and be delivered to the receiver first. In case ordering is required, we merely prevent the receiver from observing a later user message until prior user messages are complete.

3.5 Supporting Memory Copy Semantics

We now briefly consider the memory copy model for bulk transfer of data. The main difference for the programmer when using memory copy versus base message passing is that both the source and destination addresses are specified by the sender. Thus the receiver PP is no longer required to select a buffer for the user message. Figure 9 shows one possible abstraction for providing a memory copy send in a program.

Memory copy uses nearly the same mechanisms for data transfer as the base message passing implementation we described. The sender side has a similar sender record which utilizes scheduling in MAGIC to send data in bursts. The main difference is that MAGIC is able to explicitly label the line with the destination physical address instead of merely using an offset as in base message passing. As a result, the header message requires less work at the receiver since the receiving PP isn't required to compute the destination buffer addresses for the component messages. In the common case, the receive handler is not much more complex than a basic shared memory write.

The main addition is that it must maintain a count of the number of components that have arrived. Since the receive handler for memory copy has lower PP occupancy than the handler for base message passing, memory copy can potentially achieve higher bandwidth. However, the sender must do additional work in memory copy to calculate the destination address for the component, which may offset the gain from a simpler receive handler. We hope to evaluate this tradeoff in the future.

Some implementations of message passing utilize memory copy to synthesize all other primitives. The memory copy operation we describe is sufficient to synthesize base message passing. However, an implementation of message passing using memory copy would incur the latency of an extra round-trip to request the address of the destination buffer. The pre-arranged space would then be used as the explicit destination for a memory copy which transfers the actual data.

3.6 Example of Supporting Active Messages: Fetch-and-Op

Aside from supporting data transfer, the MAGIC chip can also be used to support active messages that require computation to be invoked at remote nodes. Ideally, we would like to support all active message handlers on the PP. However, since the programmable controller lacks support for virtual translation and general preemption, the PP is unsuitable for executing arbitrary user code. To deal with arbitrary active messages, we provide the PP with an efficient interrupt path that can be used to hand off the computation to the main processor. As an optimization, however, active messages which are frequently used can be written as system code and can be executed directly on the PP to avoid interrupting the processor. A simple active message such as Fetch-and-Op is one example of an operation that can be easily integrated into the MAGIC code in this way. Below we briefly describe how this can be done.

Similar to the message passing operations we described, Fetch-and-Op can be viewed as consisting of three main phases: initiation, operation, and completion. The program initiates the operation through a user-level uncached write which MAGIC interprets as a command. This command indicates the type of operation, the address to operate on, and a constant value (e.g., in the case of Fetch-and-Add). The initiating processor then issues a second command, an uncached read, to accept the result value from the operation. MAGIC begins the operation by sending a message to the home node of the address. Once the remote MAGIC chip receives the request, it performs the operation on the value in memory and stores the result back. It also replies to the initiating node with the result. When the initiating MAGIC chip receives this message, it replies to the processor with the value (as a response to the uncached read), and the operation completes. Though Fetch-and-Op only requires two commands, more complex active messages may require a longer initiation protocol.

4 Cache Coherence, Virtual Memory, and Protection

In this section, we describe the additional mechanisms we use to manage the interaction of message passing with cache coherence, virtual memory, and protection. We also describe the various support we require from the OS to implement these features. Finally, we describe some restrictions on PP handlers to guarantee system correctness (such as avoiding deadlock).

4.1 Cache Coherence

One issue that arises when implementing message passing on a cache coherent shared memory system is the choice of the coherence model for user message data. The base cache coherence mechanism allows memory locations to be cached by any of the processor caches in the system. In general, this implies that to send a user message component from one node to another, the source node may need to retrieve the data from a cache on another node. Similarly, the destination node may need to maintain coherence among cached copies of the destination memory at other nodes. The flexibility in caching user message data and the level of coherence that is maintained for such data can directly influence the performance of the message passing implementation. Below, we discuss the various options and their corresponding tradeoffs. Kubiatowicz et al. present a similar categorization in the description of the message passing support in Alewife [KA93]. As part of our description of ongoing work, we describe a more seamless integration with shared memory in Section 6.3.

The simplest option is to provide *no coherence* for user message data. This corresponds to reading the user message data directly from the sender's memory and storing the data directly into the receiver's memory without taking any coherence actions. This option leads to ambiguous semantics if the corresponding memory locations at either the sender or the receiver are cached. However, this is not an acceptable option since we would like to allow user message data to be cached at the sender and receiver.

The second option, called *local coherence*, provides coherence only if the user message data is clean or cached at the home node. Therefore, as long as the data is in the sender's main memory or the cache on that node, the data transfer obtains the latest copy of the data. Similarly, as long as the receive buffer is uncached, or is cached only at the receiving node, the receive buffer will be kept coherent. This allows the sender and receiver processors to cache user message data and closely matches the functionality provided by most message passing architectures where each processor can only cache the data that resides in its local memory. It is possible to enforce correctness in this model by disallowing remote caching of user message data through address mapping support from the operating system.

The third and most general option is to provide *global coherence* which imposes no restrictions on the caching of data used in message transfer. In this model, the sending MAGIC chip must retrieve the latest copy of the data from wherever it is cached in the system. Similarly, the receiving MAGIC chip must maintain the coherence of the receive buffer by potentially updating or invalidating multiple cache copies. In the case where the line is clean in local memory or cached only locally, global coherence is essentially the same as local coherence and therefore performs similarly. However, extra coherence transactions are needed before sending and/or receiving the data when a line is cached remotely. These extra coherence transactions can make it more difficult to pipeline the transfer efficiently under global coherence. As a result, the performance of global coherence may be lower than that of local coherence.[8]

The mechanisms to support local or global coherence are straightforward extensions of the message passing implementation described in the previous section. At both the sender and the receiver side, the directory information for the memory line can be used to determine whether any coherence transactions are required to read or write the data. However, whether there is need for global coherence depends on the way applications use user message data. Applications that are written for message passing architectures only need the functionality offered by local coherence. Alewife assumes this model in their current implementation, and only provides hardware support for local coherence. However, the extra functionality provided by global coherence may be beneficial for applications

[8]The performance degradation of global coherence is related to the number of lines which require extra remote coherence transactions.

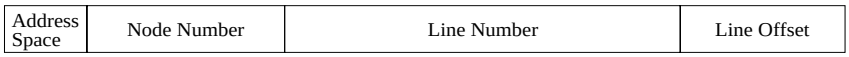


Figure 10: Physical address format on FLASH

which utilize both shared memory and message passing on a hybrid architecture like FLASH. While we plan to implement local coherence as the initial mechanism, the software flexibility in FLASH allows us to efficiently support either local or global coherence without the need to commit to a single choice while designing the hardware.

4.2 Virtual Memory

One of the challenging issues in supporting a message passing protocol on FLASH is dealing with virtual address translation. In this section, we describe several efficient techniques that can be used for address translation and preserve correct behavior in an environment with virtual memory.

As discussed in Section 2, one of the simplifications made to the PP as compared to a general processor is the removal of hardware support for memory management and address translation. This simplification makes sense because (i) cache coherence transactions do not require address translation at the PP since the processor already presents MAGIC with physical addresses, (ii) handling TLB and page faults on the PP would add significant complexity, and (iii) it is not clear that a hardware TLB would be an effective structure for caching translations because of the unconventional reference patterns of the PP.

We use the main processor's translation mechanism for performing the necessary virtual to physical address translation during the initiation protocol (described briefly in Section 3.2). Thus, the physical addresses provided by the processor to MAGIC are guaranteed to be authentic and valid. However, changes to the virtual to physical mapping of these pages while MAGIC is using the originally supplied physical addresses can lead to incorrect behavior. Below we explain how physical addresses are communicated from the processor to MAGIC and discuss several methods for ensuring the validity of the physical addresses for the duration of a data transfer.

The first issue is how to communicate physical addresses to MAGIC initially. As we discussed before, we reference memory using alternate physical address spaces to allow MAGIC to distinguish special commands from ordinary reads and writes. Figure 10 illustrates the interpretation of the physical address bits in the FLASH system. As shown, the top two bits of the address are used to distinguish four different physical address spaces.

To communicate an address during the initiation protocol, the message passing library writes to a special virtual address range, which the processor translates to an alternate physical address space. Using this technique, MAGIC obtains authentic physical addresses corresponding to the memory regions used for message transfer. To achieve this, we use a double mapping scheme. Figure 11 shows an example of how double mapping works. The alternate range of virtual addresses is labeled "Dual Mapping" in the figure. Note that the result of translating a virtual address in the alternate range is identical to that of the "Base Mapping" with the exception of the address space bits.

Given that MAGIC receives authentic translations, the remaining problem is to guarantee the validity of an address throughout the transfer. The simplest solution is to lock the appropriate virtual pages to prevent the operating system from changing the mapping. One way to do this is to lock the involved pages of memory for the duration of the program. However, this in effect defeats the flexibility of demand paging and is therefore undesirable. Alternatively, the region can be locked before each transfer and unlocked once the transfer is complete. This is common in systems with DMA, in which the processor provides the DMA unit with physical addresses

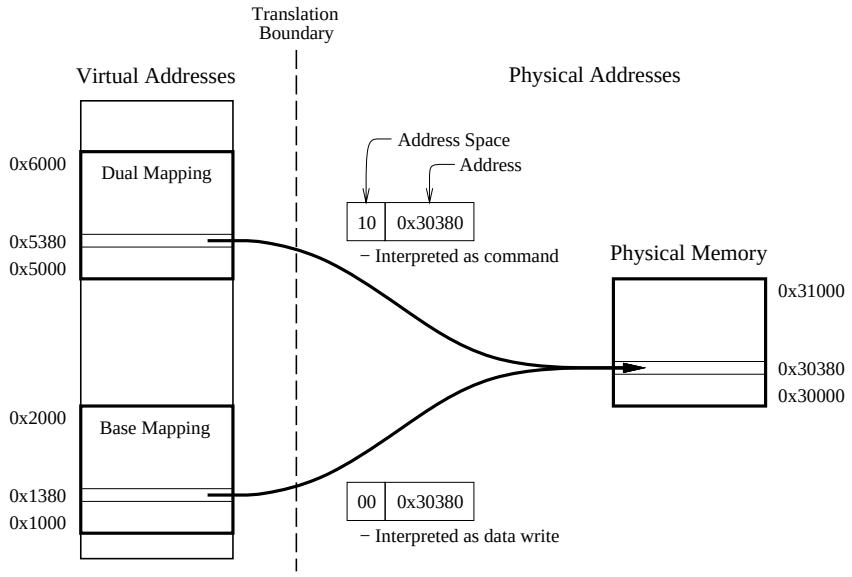


Figure 11: Example of double mapping illustrating use of alternate physical address spaces.

and disallows address changes for the duration of the access. This second approach is also undesirable because it requires expensive operating system calls on a per-transfer basis. Therefore, we don't consider the page locking technique to be a viable option for use with our message passing implementation.

Below we present three alternative solutions that are more efficient since they neither defeat demand paging nor require operating system interaction on a per-transfer basis. The first method is a simple extension of the mechanism that is already used for the base cache coherence protocol. In cache coherence, translation changes on a given processor are delayed until all the outstanding reads and writes that have already been issued are complete. We have extended this technique to work with more complex transactions such as message transfer and refer to it as the *hold-off* technique. In contrast to hold-off, our second method allows translation changes to occur, but it marks the obsolete virtual-to-physical translations stored by MAGIC as invalid to avoid incorrect results. When the physical address is next used by MAGIC, the handler software detects the invalid state and requests a mapping for the page from the main processor. We refer to this as the *invalidation* technique. The third technique we consider involves implementing a *software TLB* to maintain the translations. This allows the PP to perform authentic virtual to physical translations. The following sections discuss these alternatives in greater detail, followed by a comparison of them.

4.2.1 Hold-Off Technique

The hold-off technique extends an already existing mechanism in the cache coherence protocol for dealing with translation changes. Even with simple read and write memory operations, the processor must prevent translation changes until all outstanding operations complete. In cache coherence, this is implemented by performing a sync (or fence) operation before changing the translation. We extend this idea to include message transfer or other more complex operations in the set of operations that need to complete before translation changes are allowed.

The hold-off technique relies on the fact that physical addresses that are used by MAGIC are originally translated

by the main processor at the same node. We assume that the main processor is always informed of possible changes to these translations (e.g., through the TLB consistency mechanism). The processor can in turn notify MAGIC of such translation changes, and is required to wait for a response before proceeding. If the processor notifies MAGIC of a translation change when hold-off is enabled, MAGIC waits until the protected operations complete before responding to the main processor. Therefore, the hold-off technique effectively disallows translation changes for the duration of such operations. A key benefit of this technique as compared to locking specific memory ranges for the duration of an operation is that it incurs no overhead (e.g., operating system intervention) in the common case where no translation changes occur during the operation.

To protect the addresses communicated to MAGIC, hold-off must be enabled at the beginning of the initiation protocol. If the processor changes the translation of an address sent as part of the initiation protocol, or if a context switch occurs (while hold-off is enabled), we cause the initiation protocol to retry. This is a conservative technique to guarantee the correctness of the physical addresses. Also, since the addresses must all be valid, all the pages used by an operation using hold-off must fit in memory at the same time, otherwise the initiation protocol cannot possibly succeed.

Translation hold-off is enabled by incrementing a counter whose value indicates the number of outstanding operations that are currently using hold-off. When an operation completes, it releases its use of hold-off by decrementing the hold-off count. If a translation change arrives when hold-off is enabled, MAGIC notes this event and then prevents the initiation of new operations which require hold-off. As described before, the main processor waits for a response before allowing the change to complete. MAGIC does not grant this permission until the hold-off count reaches zero. Since the processor is prevented from initiating new operations that use hold-off after a translation change arrives, it is guaranteed that the counter will eventually reach zero as long as the previous operations eventually complete.

The fact that translation changes are prevented from occurring during hold-off places a significant restriction on the use of this technique. To avoid deadlock, operations that use the hold-off technique must be guaranteed to complete *on their own* without requiring interaction with code running on any of the main processors. Consider the case where an operation requires interaction with some process before it can complete. If that process is stalled because of a translation change (e.g. page fault), and if the translation change can not be serviced until the operation is complete, then neither the process nor the operation can proceed and deadlock would occur. There are several types of operations that are indeed guaranteed to complete on their own. Obviously, base read and write memory operations complete on their own and only require interaction with other MAGIC chips. Other operations such as Fetch-and-Op can also use the hold-off mechanism. However, operations such as a barrier can lead to deadlock because other processes must make progress for the barrier to complete.

Though memory copy can use the hold-off technique, hold-off is not suitable for the base message passing protocol described in Section 3. The main difficulty arises from maintaining valid translations for the receive buffers that are allocated in advance. Protecting these addresses using hold-off would require hold-off to remain enabled until all the receive buffers are exhausted by incoming user messages. However, this situation can result in deadlock. For example, consider a sending process that experiences a page fault before sending its user message. Assume the receiving node's MAGIC chip needs to be notified of an address mapping change before the page fault can be serviced. Since hold-off is enabled, the page fault is not allowed to continue, so the user message buffers will never be exhausted. As a result, hold-off will never be released thus resulting in deadlock. To address these limitations, we provide the invalidation technique, described next.

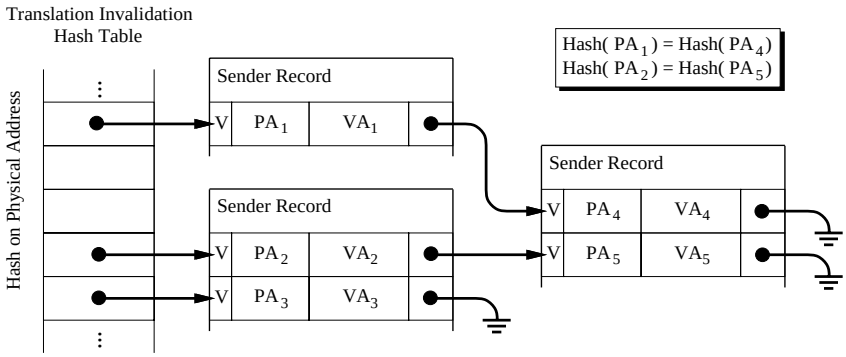


Figure 12: Translation invalidation data structure.

4.2.2 Invalidation Technique

Unlike hold-off, which temporarily prevents translation changes from taking place, the invalidation technique allows the change, then "invalidates" the obsolete physical address that may be stored by MAGIC. The convention for handlers using this technique is to check that a physical address is valid each time before using it. This check is very inexpensive, adding only a branch to the handler. On detecting an invalid address, the PP interrupts the main processor to request the new translation (or a new mapping if the page is not mapped). Below we describe the mechanisms used to implement the invalidation technique.

First, MAGIC logs the physical page that is communicated to it by the processor. Physical addresses provided to MAGIC are initially stored in the sender or receiver record along with the corresponding virtual address (provided as the data field for writes). Once an operation is accepted by MAGIC, these addresses are linked into a list of physical addresses in use by MAGIC. Figure 12 illustrates the data structure used to achieve this. Each page address in the sender record is linked into an appropriate list based on a hashing function. This data structure provides a simple way of finding all the uses of a given physical address by MAGIC handlers, and can be used to invalidate an address translation when it becomes stale. A given entry in the linked list is removed when the operation that uses the address is complete.

Second, MAGIC must mark uses of a physical address as invalid when the processor signals a translation change for the page. On notification of a translation change, MAGIC first checks for hold-off and replies to the processor if the count is zero. Before yielding the PP, however, the same handler invalidates the uses of the stale physical address in the various records. This is achieved by looking up the appropriate list of records through the hash table (see Figure 12), traversing the linked list, and marking occurrences of the physical address as invalid. While this can conceptually be done using a valid bit, it can also be implemented by setting the physical address equal to zero, for example. In addition to traversing the linked list, the handler must also check for operations that are in the middle of their initiation protocol, since addresses communicated for these operations are not yet included in the invalidation data structure.

Third, MAGIC needs a method to get a new translation for a given virtual page. When an invalid physical address is detected, MAGIC requests a new physical address corresponding to the given virtual address. The virtual address used for retranslation is provided by the initiation protocol, in the data field of the command that provides the original physical address (see Figure 7). MAGIC communicates the virtual address and the PID for which the translation is being requested to the processor along with the interrupt request. The processor determines a new

translation, (mapping the page if necessary) and responds to MAGIC with the new mapping.

Even though the virtual address is provided by the user as data, the processor's translation mechanism protects the system against invalid virtual addresses. If the program provides a virtual address that corresponds to the physical address, the translation request will return a correct new translation. If the virtual address is incorrect (i.e., it did not correspond to the physical address), 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. In either case, protection of other process's memory is not compromised.

The invalidation technique is quite efficient in the common case when a translation does not change since the only overhead incurred is adding the translation into the appropriate linked list and then later removing it.

4.2.3 Software TLB

The third alternative we are considering is to maintain a software TLB on MAGIC. The invalidation and hold-off techniques are aimed at protecting against unexpected mapping changes while physical addresses are in use by MAGIC. However, these techniques do not provide the ability to translate virtual addresses on MAGIC, and therefore handlers using virtual addresses must interrupt the processor for a translation. By adding a software TLB to MAGIC, virtual addresses which hit in the TLB can be translated without interrupting the processor. The TLB could be used to translate addresses each time they are used, or it could work in conjunction with the invalidation technique. In the latter option, the TLB would be used to translate an address initially, then the invalidation technique would protect against translation changes; this technique can be more efficient than the base TLB option since it removes the need to translate on every use of an address.

Providing translations on MAGIC via a software TLB has several advantages. A TLB allows user messages to be initiated more rapidly since the initiation protocol doesn't need to send a command for each individual page of the user message. Instead, the user message could be specified as a virtual address and length, which allows any size user message to be described with only two to three uncached writes. It also allows operations such as memory copy to specify a virtual address on the remote node as opposed to requiring the destination physical memory addresses to be known at the sending node. Since addresses in the components of the memory copy would then be virtual, hold-off is no longer required. If the mapping changes on the remote node, the receiver will detect a translation change via its TLB. Finally, since the TLB is in software, we can adjust its size and policies to increase its hit rate.

Like the invalidation technique, the software TLB relies on interrupting the processor if it fails to resolve a translation request. Even though we believe the interrupt path for such a translation request can be highly optimized, it is still important for the TLB to have a high hit rate. To achieve this, we are considering heuristically filling the TLB with translations before they are required by the PP. This can be accomplished, for example, by notifying MAGIC through an explicit command whenever the OS creates a new local page mapping. We are currently evaluating the use of a software TLB implementation on MAGIC.

4.2.4 Comparison of Virtual Memory Techniques

To efficiently support the range of operations we are interested in, the PP must at least provide a subset of the techniques described above for dealing with virtual memory. Two subsets which are sufficient are a software TLB

(in conjunction with invalidation, for efficiency), or the hold-off and invalidation techniques. Below we summarize the trade-offs between these two choices.

The invalidation and hold-off techniques are both necessary to be able to support a wide range of operations. The invalidation technique has the advantage that it never prevents translations changes from proceeding, which helps avoid deadlock. The base message passing model we described uses this technique to protect the translations it uses at both the sending and receiving nodes. The main drawback to invalidation is that it fails to protect physically-addressed message components in transit. So if invalidation were used for memory copy, and a page of the receiver buffer changed mapping, the user message component in flight could reference the *incorrect physical address*. In addition to this problem, the invalidation technique incurs overhead to enqueue and dequeue the translations in use and if translations change during operations, MAGIC must interrupt the processor to request a new translation. In contrast, hold-off can protect memory copy and similar operations by preventing translations from changing during a complex operation. The drawback to hold-off is that it cannot protect arbitrary transactions, but only ones that are guaranteed to complete without the need for interaction from other processes. However, hold-off can be more efficient for operations that are appropriate for it.

A software TLB allows MAGIC to use virtual addresses and avoid many of the problems the other techniques solve. The advantages of a software TLB are: it permits faster initiation since the processor does not have to explicitly provide physical addresses, it provides translations faster than a processor interrupt (on a hit), it doesn't require hold-off, and it can allow efficient use of virtual addresses in some handlers. Some of the drawbacks are: it consumes a portion of MAGIC's memory, it requires several cycles to perform a translation and tag check, and it may need to be filled with translations in advance to achieve good hit rates. The decision whether to implement a software TLB will depend on a detailed evaluation of the costs which is currently underway.

4.3 Protection

There are three kinds of protection that we consider in this section. The first is to prevent random processes from accessing (reading or writing) a given process's memory through message passing primitives. The second is to prevent a process from forging a user message as another process or receiving a user message that is not destined for it. Finally, the third is to limit the set of processes each process can send a user message to.

We achieve memory protection as part of the access rights that are provided through the address translation mechanism supported by the main processor and OS. Therefore, only processes that have been given permission can read or write another process' memory space. Furthermore, MAGIC code is trusted and is guaranteed to maintain this protection.

The second form of protection, preventing user messages from being forged or received by an incorrect process, requires MAGIC to have access to authentic PIDs. As we mentioned before, MAGIC is always notified by the operating system regarding the PID of the currently running process at context switch points. On the sender side, user messages are always tagged with the authentic PID of the sender, which prevents processes from forging user messages. Similarly, on the receiver side, the user message data is placed into the address space of the receiver PID and so memory protection disallows other processes from observing the user message.

The third kind of protection can be supported by restricting user messages to be among the processes in the same application. As mentioned earlier, user messages are sent using virtual PIDs which are limited to the number of processes in the application. MAGIC maintains a table that maps the virtual PID to a processor number and an operating system PID for these valid destinations. Therefore, other processes outside the application simply cannot

be reached. Our current virtual PID translation strategy implicitly assumes that processes do not migrate from one physical node to another during their execution. In the presence of migration, extra functionality is required to allow the translations to change dynamically during the course of the execution. We are currently exploring solutions for efficiently supporting migration, in addition to efficient mechanisms to allow processes in different applications to communicate while maintaining the appropriate level of protection.

4.4 Operating System Support

Some of the mechanisms we use in FLASH require support from the operating system. We first summarize the support we have already mentioned, and then discuss some additional interactions with the OS. As we will see, many of the changes are localized to specific functions within the OS. The first change to the operating system is to notify MAGIC whenever there is a context switch on the main processor. This allows MAGIC to authentically identify a process that attempts to send or receive a user message. Second, MAGIC must be notified on address translation changes. As part of this notification, the OS must wait for MAGIC to respond before it proceeds with the change. This allows MAGIC to implement the hold-off, invalidation, and software TLB techniques. Third, the OS must notify MAGIC when it creates or kills a process. This allows MAGIC to allocate and initialize data structures for subsequent message passing by that process, or to deallocate such structures when they are no longer needed.

Our plan is to hand off exceptional (unexpected or error) cases that can arise during message transfer to the OS for recovery. While the protocol is designed to rapidly recover from more frequent exceptional cases, the OS is better suited for dealing with the less common or more complicated ones. This allows us to off-load complexity from MAGIC to the OS, and also allows recovery to be more uniformly handled through OS mechanisms. An example of a "complex" exception is if a message send fails multiple times. When such a problem occurs, MAGIC logs the error state in memory and interrupts the processor to invoke the OS. We have designed a clean interface between MAGIC and the OS to make this efficient and usable by all MAGIC operations.

Finally, since the OS manages the physical memory of the system, it needs to allocate memory to MAGIC for storing various protocol state for message passing. There are two different options we are considering. One method is for the OS to allocate a large block of memory for MAGIC at boot time (this is the option we have adopted for the coherence protocol state). The alternative is for the OS to allocate individual pages to MAGIC as they are needed. Though the latter choice allows MAGIC's region of memory to dynamically expand, it introduces extra complexity since memory regions may no longer be contiguous. We are currently evaluating these two alternatives.

4.5 PP Handler Restrictions

PP handlers act in many ways as extensions of the OS code onto MAGIC. In fact, the software handlers that are executed on the PP have direct access to many of the key resources in the system. Therefore, incorrect actions by these handlers can easily crash the system, cause deadlock, or compromise protection. To avoid such problems, there are some basic requirements handlers must satisfy. Below, we describe some of the requirements that need to be satisfied by handlers to protect against such problems.

The first requirement is that MAGIC should only execute trusted system code. Arbitrary user code is not allowed for several reasons.[9] First, user code cannot be trusted to guarantee the protection model we desire. One obvious

[9] We are currently considering software mechanisms to provide the required protection for executing a limited class of user handlers on the PP.

example is that MAGIC provides direct physical memory access, so user code could easily circumvent memory protection. Furthermore, MAGIC lacks the ability to preempt a user handler that fails to relinquish the PP on its own. This means the PP would have no mechanism to let other code execute if the user handler didn't voluntarily preempt itself.

Even though we limit ourselves to system handlers, the handlers must still ensure fair access to the PP in order to avoid deadlock. One of the critical requirements is that the handler yields the PP in a *bounded amount of time*. There are several ways a handler could wait for an unbounded duration, all of which are prohibited to avoid deadlock. For example, handlers cannot spin in the PP waiting for a response from another node. Similarly, handlers cannot wait for space on outgoing queues to become available. This is because the handler may indirectly prevent the queues from emptying if it does not yield the PP.[10] In both cases, the handler should suspend itself on the software queue (like the transfer handler) and yield the PP. The condition can be checked again when the handler is next invoked.

5 Preliminary Performance Estimates

This section presents some preliminary performance estimates for message passing on FLASH. We begin by stating some system assumptions. Then we provide estimates for the performance of the message passing implementation discussed in Sections 3 and 4. Finally, we discuss the performance of simple active messages such as a Fetch-and-Op.

5.1 System Assumptions

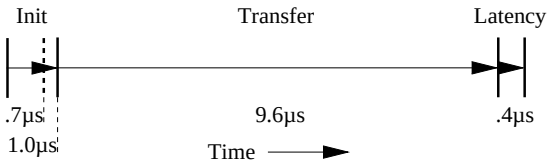
The performance estimates provided in this section are based on the following assumptions and system parameters. The protocol processor is being designed to run at a cycle time of 10 nanoseconds. With its two-way superscalar data path it can thus provide a peak instruction issue rate of 200 MIPS. The memory system is designed to deliver cache line fetches of 128 bytes every 16 cycles (for a peak bandwidth of 800 MB/s), with a latency of 30 cycles. Finally, we assume peak network bandwidth of 400 MB/second over each link and an average latency of approximately 400 nanoseconds across the network. This corresponds to an "average" trip of 8 hops (at 50 ns/hop) for a 256 node machine arranged in a 3D network of dimensions 8x8x4. The above parameters are subject to change as the design of FLASH proceeds.

5.2 Message Passing

We make several simplifying assumptions for the purpose of estimating performance of the base message passing protocol. First, although we include the overhead of using the invalidation technique described in Section 4, we assume that no translation changes occur during the user message transfer. Second, we assume the send and the receive buffers are each a single 4 KB page, and both are cache line-aligned (thus there is no alignment overhead). Third, we assume local coherence is upheld. Given these assumptions, we hand-coded the important PP handlers in assembly code. This code provides an estimate of the PP occupancy for critical handlers. The results presented below are preliminary; we are currently evaluating the performance of these protocols with detailed simulations.

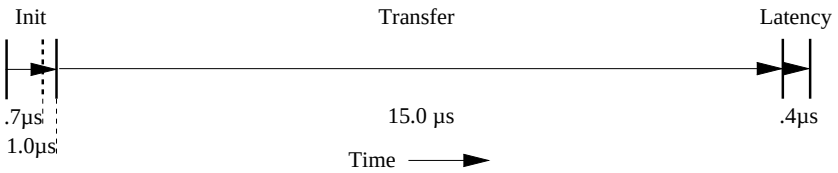
The first measure of performance to consider is the latency to initiate a user message. The time spent by the processor in the `send` library to issue the commands until it receives the reply is approximately 0.7 μs, at which

[10] The dispatcher in MAGIC only executes a handler on the PP when it can guarantee some minimum outgoing queue space. If a handler wants to send more messages than the guaranteed minimum, it must explicitly check for available space in the outgoing queue.



Useful transfer bandwidth:	approximately 372 MB/sec
PP occupancy:	30 cycles on each side per memory line

Figure 13: Performance of a page transfer (page entirely clean in memory).



Useful transfer bandwidth:	approximately 249 MB/sec
PP occupancy:	47 cycles on each side per memory line

Figure 14: Performance of a page transfer (page entirely dirty in sender processor's cache).

time the user message has been described to MAGIC. Before the user message transfer begins, however, the PP must do approximately 0.3 μs of preparation for the transfer (e.g., storing addresses in the invalidation table). Therefore, the user message transfer doesn't actually start until approximately 1 μs after the initiation protocol begins. A 1 μs initiation latency is reasonable, considering it is only 2 – 3 times longer than the local memory latency.

Once the user message description phase completes, MAGIC can begin transferring the data. First we describe the performance for the case when the data buffers are clean in memory at both the sender and receiver. This case provides the highest performance since no coherence actions are necessary. We next consider the other extreme where all the send data is dirty in the sender's cache and the receive buffer is dirty in the receiver's cache. In practice, the performance will lie somewhere between these two extremes depending on how much of the data is actually dirty in caches.

The resulting performance of the user message transfer depends on three different factors. We first need to consider how fast the sending MAGIC chip can process and send individual memory lines. Similarly, we need to consider the speed at which the receiving MAGIC chip can process and store components. Finally, we are also limited by the bandwidth of the network. Our preliminary estimates indicate that the system is fairly balanced. More complicated protocols would likely be limited by PP occupancy, however. Below, we first present the performance of our system as limited by the processing on the MAGIC chips at either end ignoring the network bandwidth limitation. This represents an upper bound on the performance we can achieve using fast networks. We then discuss the actual performance that is possible given the assumed network bandwidth.

For a send buffer that is clean in memory, the sending MAGIC can process one component—128 bytes of data and 16 bytes of header—approximately every 30 cycles. The time to receive the memory line at the other side is in

the same range. This amounts to a data transfer rate of 426 MB/second and a total transfer rate of 480 MB/second (including headers). For a 4 KB user message, this transfer takes approximately 10 μs. Once the transfer has completed from the sender side, the receiver will still receive data for another 400ns (i.e., the latency of the network we assumed). Figure 13 shows that the send takes a total of about 11 μs from start to finish, achieving a net useful bandwidth of approximately 372 MB/sec. Compared to 11 μs to transfer a 4KB page, servicing a remote read miss (a single 128 byte cache line) on FLASH takes approximately 2 μs. This implies that message passing can be expected to efficiently speed up bulk transfer with respect to the performance available through shared memory accesses.

The other extreme in performance occurs when the entire send buffer and receive buffer are dirty in their respective local processor's caches. Since FLASH does not have a snoopy bus (as in DASH, for example), MAGIC needs to check the directory state for each memory line and explicitly request the dirty lines from the processor cache. The extra time to fetch the data from the processor's cache decreases the transfer rate on the sender MAGIC chip to approximately one component per 47 cycles. This corresponds to a data transfer bandwidth of 272 MB/sec (306 MB/sec including headers). As illustrated in Figure 14, the total time to do the transfer (including initiation time) grows to approximately 16.4 μs, leading to a net useful bandwidth of approximately 249 MB/sec. Thus, we lose about a third of the effective bandwidth compared to the case where the entire send data is clean in memory.

Now that we have established an upper bound on the transfer performance due to MAGIC occupancy, we take into account the effect of limited network bandwidth (400 MB/s in our system). Without this limit, the sender MAGIC can sustain data transfer of a clean page at 480 MB/s (including headers). With the network limit in place this transfer is limited to 355 MB/s for data and 45 MB/s for the header information. This stretches the clean transfer region to 11.5 μs (from 9.6 μs) and the net useful bandwidth is reduced to approximately 317 MB/sec (as compared to 372 MB/s without the network limitation). In the case where the user message is entirely dirty, the network bandwidth is not the limitation, and the performance is unchanged when we take it into account.

5.3 Fetch-and-Op

To illustrate the performance of a simple active message, we describe the latency of a Fetch-and-Op performed on a remote memory location. Similar to message passing, the processor uses a special command to specify a Fetch-and-Op operation to its local MAGIC. The PP forwards a Fetch-and-Op message to the home node; this occupies the local PP for approximately 15 cycles. Once the message reaches the home node, it is decoded and the home PP performs the appropriate operation on the designated memory location (assume it is clean in memory). In addition, the home PP returns the result in a reply message to the requester. The occupancy of the home PP to accomplish these tasks is approximately 19 cycles plus 30 cycles to read the memory line. Eventually, the requesting MAGIC chip receives the reply and places the response at a notification address specified through the Fetch-and-Op command. This last step occupies the requesting PP for approximately 12 cycles.

The total occupancy of the local and remote PPs is about 88 cycles if we combine the latencies through all relevant units. After adding in network latency, the total latency (with no contention) of a remote Fetch-and-Op is about 2 μs, which is nearly the same as for a remote read operation.

6 Discussion

This section discusses the design choices in FLASH and compares our design to other related work. We also discuss our ongoing work and some extensions to the ideas presented in this paper.

6.1 Related Work

The J-machine [DCF+89, NWD93] supports fine grain communication and computation, with messages forming activations similar to active messages. Message initiation is through user-level instructions on the sender processor, with the instructions providing the message data from registers. At the receiving node, an interrupt interface invokes a handler to service a message when it arrives. Unfortunately, the explicit processor interaction that is required for sending and receiving every word of the message causes high processor overhead in handling messages. Furthermore, messages longer than 4KB are difficult to support since the J-machine uses the on-chip memory (only 4KB) to queue messages. In practice, since they implement several queues in the memory and want to accommodate several messages in each queue, messages are limited to approximately 128 bytes [Noa93].

The Thinking Machines CM-5 consists of nodes connected by two networks: a fat tree and a broadcast network [Thi92]. Messages are initiated through a user-level memory mapped interface, and are limited to 24 bytes per message. The processor checks a status register to determine if a message was successfully launched before issuing the next one. The receiver is either interrupted or polls for message arrival, and similar to the sender, the receiver processor is involved with handling each packet that arrives. Therefore, the processors at both the sender and the receiver are involved in servicing all data that is transferred by a message. This effectively limits the overlap between computation and communication.

The *T architecture [NPA92] consists of two processors, a data processor (dP) used for computation and a synchronization processor (sP) used to service communication and synchronization. The sP relieves the dP from handing message transfers. Motorola's implementation of *T is based on a variation of the 88110 processor (the 88110MP) and utilizes two such processors per node [Bec92]. The dP at the sending node can initiate messages through commands to its local sP. The receiver can either poll for message arrival, or receive an explicit continuation invocation (similar to the dataflow model). The additional processor at each node allows for communication to be carried out in parallel with computation. However, one disadvantage of the *T design is that the second processor is not tightly coupled to the memory and network as compared to the dedicated controller in FLASH.

The Alewife architecture [ACD+91, KA93, KJA+93] 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. However, interrupting the processor may take time away from computation that is executing on the processor. To simplify the design, some functionality is not supported in Alewife. For example, 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 Alewife addresses the issue of coherence of message data, only local coherence is supported in hardware. Global coherence can be achieved only through processor interrupts which can lead to diminished performance. In summary, the Alewife approach depends on fast main processor interrupts to achieve its flexibility and does not provide full support for protection, coherence of message data, or virtual memory.

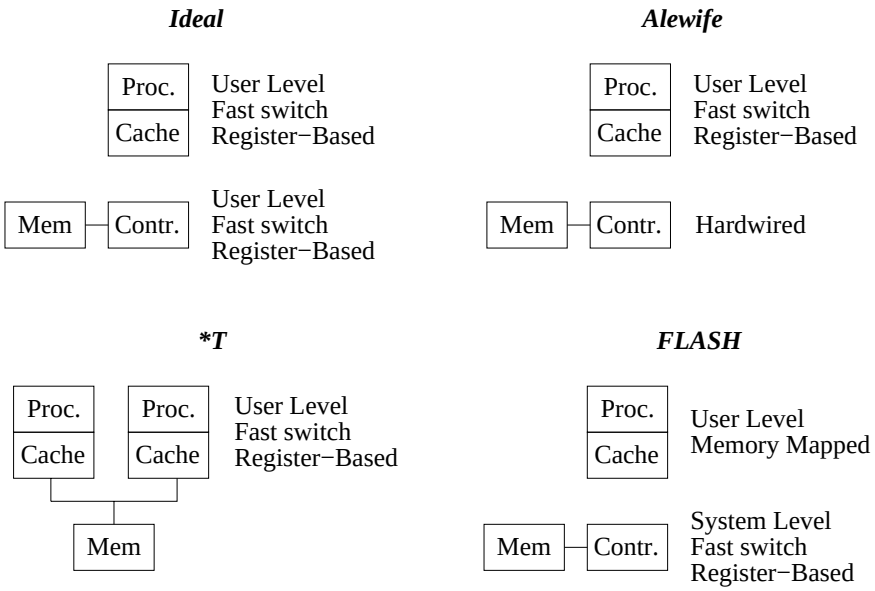


Figure 15: Various options in the design space.

6.2 Design Space

This section provides some insight for the design decisions made in FLASH. Figure 15 illustrates a number of possible system configurations that one may consider. The ideal configuration consists of a main processor and a programmable controller, with both providing user-level programmability, fast context switch support, and efficient register-based communication interfaces (see Henry and Joerg [HJ92] for a detailed taxonomy of communication interface options). The figure summarizes how the Alewife, *T, and FLASH designs differ from this ideal configuration.

Our goal in the FLASH system has been to achieve high performance while using existing commercial microprocessors. Since such processors do not currently provide fast context switch capability and an efficient register-based communication interface, we are limited to using a main processor with a standard memory-based communication interface and with no support for fast context switches. While this is a compromise, we believe we can achieve the majority of the gains attainable by the ideal configuration because the controller in FLASH can be used to support much of the functionality that would otherwise require the main processor.

The limitation of the programmable controller in FLASH compared to the ideal configuration is that FLASH does not allow user-level code to be executed on MAGIC. Even though supporting user-level code is interesting, it would require several additional hardware mechanisms in MAGIC for protecting the system and other users from incorrect or malicious code. By allowing only system-level code, we are able to significantly simplify the design of the controller. Furthermore, common operations can be provided as system code. For operations that are less common, MAGIC can interrupt the main processor to service the operation. However, since these operations are less common, the extra overheads in interrupting the processor are less significant. We are also considering software techniques for supporting user-level code to be executed on the PP while providing appropriate levels of protection.

6.3 Extensions and Ongoing Work

This section discusses our continuing efforts on implementing and evaluating message passing primitives on FLASH. First, we have developed a detailed functional simulator of the architecture. We plan to implement and evaluate various message passing protocols using this simulation environment. We are also planning to develop applications that exploit the hybrid capability of FLASH by using both the shared memory and message passing functionality and to evaluate the gain from using such a hybrid model.

Besides the features we have already described, there are a few extensions that would help provide a more seamless merge of message passing and shared memory. The data transfer protocols discussed so far assume the home for the send and receive buffers are local to the sender and receiver, respectively. One possible extension is to allow a processor to send and receive data whose home is remote. We could implement this by allowing a node to initiate a user message transfer on another node. When a node's MAGIC chip detects that part (or all) of the send data has its home in a remote memory, it could send a message to the home node requesting that the remote MAGIC perform the transfer on the original node's behalf. Upon completion of the remote transfer, the remote MAGIC chip could send an acknowledgement to the sender, indicating that its portion of the transfer is finished. A similar situation may occur if the receive buffer's home node is not the receiver. The receiver could reply to the sender to redirect the user message data to the node that is the buffer's home.

Finally, we are exploring communication primitives beyond the basic message passing semantics we described. We are following the development of the MPI standard message passing interface, and will consider it as a possible model to support in the future [Mes93]. We are also considering extensions to the message transfer primitives to support multicast and gather/scatter (especially those with a simple stride). We are also looking into efficient implementation of synchronization operations such as tree-based barriers using MAGIC. We plan to also consider reduction (or scan) primitives as well as more complex operations such as fast matrix transposition.

7 Summary

Providing efficient communication support is essential for achieving high performance in multiprocessor systems. Cache coherent shared memory systems provide efficient fine-grain communication through memory reads and writes. On the other hand, message passing systems provide efficient support for bulk transfer of data. In FLASH, we attempt to integrate these two complementary communication paradigms in a single architecture. We achieve this goal by using a programmable controller that provides flexible functionality for supporting either communication model. Compared to other attempts at integrating cache coherent shared memory and message passing, our approach is unique because it provides full support for virtual memory, protection, and the interaction of message data with cache coherence. Our preliminary studies indicate that we can utilize close to the maximum available network bandwidth for data transfers. We achieve this performance by using the same base hardware mechanisms used by our cache coherence protocol.

8 Acknowledgements

Many people have helped us in the various stages of this research. This work would not be possible without the effort that went into the FLASH architecture by the entire Stanford FLASH team, the authors of [KOH+94]. In particular, we thank Joel Baxter, John Chapin, Scott Dresser, Mark Heinrich, John Hennessy, Mark Horowitz,

Jeffrey Kuskin, Mendel Rosenblum, and Steven Woo for insightful comments on the drafts and discussions about many of the ideas. We owe special thanks to Jaswinder Pal Singh, whose detailed comments in the later stages of this report were invaluable, and to David Ofelt for working with us in the early stages of this research. John Heinlein is supported by an Air Force Laboratory Graduate Fellowship (AFOSR). Kourosh Gharachorloo is supported by Digital Equipment Corporation's Western Research Laboratory. Anoop Gupta is supported by DARPA Contract N00039-91-C-0138 and by an NSF Presidential Young Investigator Award.

References

[ACD+91] 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.

[Bec92] 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.

[DCF+89] William J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Michael Larivee, Rich Lethin, Peter Nuth, and Scott Wills. The J-Machine: A fine-grain concurrent computer. In *IFIP Congress*, August 1989.

[HJ92] 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.

[KA93] 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.

[KJA+93] 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.

[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+92] Dan Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, 1992.

[Mes93] Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report No. CS-93-214, University of Tennessee, November 1993.

[Noa93] Michael Noakes, June 1993. Personal Communication.

[NPA92] 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.

[NWD93] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-Machine multicomputer: An architectural evaluation. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 224–235, May 1993.

[Pie88] 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. ACM, 1988.

[Thi92] Thinking Machines Corporation. *Programming the NI*, March 1992.

[vECGS92] 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.