FUGU: Implementing Translation and Protection in a Multiuser, Multimodel Multiprocessor

Kenneth Mackenzie, John Kubiatowicz,
Anant Agarwal and Frans Kaashoek*
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

October, 1994

Abstract

Multimodel multiprocessors provide both shared memory and message passing primitives to the user for efficient communication. In a multiuser machine, translation permits machine resources to be virtualized and protection permits users to be isolated. The challenge in a multiuser multiprocessor is to provide translation and protection sufficient for general-purpose computing without compromising communication performance, particularly the performance of communication between parallel threads belonging to the same computation. FUGU is a proposed architecture that integrates translation and protection with a set of communication mechanisms originally designed for high performance on a single-user, physically-addressed, large-scale, multimodel multiprocessor.

Communication in FUGU is based on the mechanisms of the Alewife machine [1]. The mechanisms are shared memory with hardware cache coherence, user-level message sends and receives, and a user-controlled DMA facility integrated with messages for bulk transfers. This paper presents a design that integrates translation and protection with these communication mechanisms. Three components of the design are novel. First, we propose maintaining TLB coherence as a side-effect of cache coherence on page table entries. Second, we describe how to permit user-launched and user-handled messages without dedicating physical memory for buffering at the sender or at the receiver. We propose to use a rudimentary, second, system-only network to avoid deadlock of the user-accessible network. Third, we show how to integrate user DMA with virtual memory to permit bulk transfers without prenegotiation and global locking of physical memory.

*MIT LCS TM 503, October 1994. This research has been funded in part by NSF grant # MIP-9012773, in part by DARPA contract # N00014-91-J-1698, and in part by a NSF Presidential Young Investigator Award.

1 Introduction

Recent parallel processors unify support for two models of processing by providing support for shared addressing of data and for message passing. This class of "multimodel" machines is of great interest because it combines the best of two worlds: the efficiency of message passing, for bulk data transfer and for combining data with synchronization, and the flexibility and convenience of shared memory for fine-grained and dynamic computations [13]. Multiprocessors, for example Alewife [1], achieve good performance by providing direct access to the underlying hardware. Shared-memory machines initiate communication with ordinary, user-level load and store instructions. User-level messaging interfaces permit user-level code to launch and to handle messages by directly manipulating the network interface hardware.

A multiuser multiprocessor requires protection checks on accesses to hardware resources, e.g. accesses to memory or to incoming messages, in order to isolate users. Translation on all accesses permits providing the user with the convenient illusion of a virtual machine. Adding translation and protection without compromising performance is a challenge. The goal of FUGU is to include hardware and software mechanisms for translation and protection that support general-purpose, multiuser computing on large-scale, multimodel multiprocessors while providing parallel application performance equivalent to running on dedicated, single-user hardware.

1.1 The Problem

The problem of translation and protection for shared memory accesses can be solved using a relatively straightforward application of standard virtual memory. One new problem is that of keeping the translations coherent in a scalable way. Protection for user-level messaging and translation for messages that access memory is less clear. Current solutions for user-level messages fall into three categories, none of which is entirely satisfactory because of the impact on performance:

1. User-initiated message sends can be permitted to globally named, pre-negotiated areas of physical memory at the receiver, for instance as remote-write operations. Translation and protection are handled in analogy to virtual memory. SHRIMP [5] and bulk transfers in FLASH [15, 10] use remote-write.

2. User-level access to the network hardware can be preserved if the machine is rigidly partitioned and all hardware in the partition, including the network, is context switched. The CM-5 adopts this solution [17].

3. User-initiated transfers between memories can use explicit acknowledgements to manage sender-side buffer space for each message. Software protocols such as IP typically use this approach. FLASH general messages and FUNet [11] provide the acknowledgements in hardware.

1.2 FUGU Overview

FUGU provides hardware support for three communication mechanisms. A coherent shared memory system communicates implicitly via messages synthesized and interpreted by hardware. A user-level messaging facility allows user-code to send messages to and handle messages from the network directly with no buffering. Finally, a direct memory access (DMA) facility is built on top of the basic messaging

Mechanism	Translation	Protection
Shared memory	Memory mapping. TLB coherence via PTE coherence.	Memory mapping, page-level protection.
Short messages	*(compiler-generated only)*	GID-stamped messages. Receive handler timeout. Rudimentary second network.
Messages w/DMA	Uses the processor's TLB. Implicit frame-locking, checked by the page cleaner	*(same as short messages)*

Table 1: Translation and protection for each communication mechanism.

facility to permit efficient bulk transfers. We propose to provide translation and protection for each of these communication mechanisms using the techniques summarized in Table 1.

Shared memory Translation and protection for shared memory is provided by memory mapping via a single translation-lookaside buffer (TLB) per node placed at the processor. Placing the TLBs at the processor requires that the page table entries (PTEs) cached in the TLBs be kept consistent. We propose to implement TLB consistency as a side effect of data cache coherence on the PTEs stored in shared memory. FUGU cache coherence is directory-based and uses invalidation. The only required change to support TLB consistency is to provide an alternate, software invalidation action for memory lines that store PTEs. The alternate action invalidates the PTE cached in the TLB as well as in the main cache.

Short messages Translation and protection for messages is handled as follows. Translation of node names for messages is performed by the compiler or user-level runtime system. That is, if translation is required, the runtime system takes a virtual node number and looks up the physical node number for each message sent. Protection for user messages is enabled by adding an unforgeable process group identifier (GID) to all messages and providing two-way demultiplexing in hardware at the receiver. Messages intended for the currently running process at a receiver are handled at full speed while messages with non-matching GIDs are diverted to the kernel. A timeout on message handlers protects the machine from errant message handling code. A second, system-only, logical network allows kernel software to deal with errant user code that floods the network by providing a channel for communicating scheduling information and, in extreme situations, for paging. Importantly, the second network need only be rudimentary in construction and performance because it is rarely used.

Bulk transfer messages DMA for messages interacts heavily with the virtual memory system. Translation for the DMA facility is provided by the processor's TLB. Blocks of memory in use for DMA are implicitly locked in memory without prenegotiation and without adding overhead to the messaging operations. Notably, our scheme locks physical frames, an operation local to a node, rather than virtual pages visible across the machine. Fast receive operations without prenegotiated receive buffers are made possible by taking advantage of the explicit coherence semantics of DMA writes: the user cannot expect to read good data until the DMA operation reports completion so blocks directed to a missing virtual page can be quickly written to a freshly allocated physical page and then reconciled after the write but before they are made available to the user. Protection for messages with DMA is based on GID checks, the same as for short messages without DMA.

This paper makes three contributions by proposing three techniques for integrating translation and protection with an efficient set of multiprocessor communication mechanisms. First, we propose maintaining TLB coherence as a side-effect of PTE coherence. Second, we describe how to permit user messages without dedicating physical memory for buffering at the sender or at the receiver. Third, we show how to integrate user DMA with virtual memory to permit bulk transfers without prenegotiation and global locking of physical memory.

Section 2 summarizes our assumptions and the programmer's model for the system. Sections 3, 4 and 5 form the core of the paper, describing how to provide translation and protection for shared memory, for short messages and for long messages with DMA, respectively. Shared memory is described first because the other systems interact closely with virtual memory. Section 6 describes the hardware we propose to construct to evaluate FUGU. Section 7 describes related work and Section 8 concludes.

2 Programmer's Model

This paper is chiefly concerned with two low-level abstractions and the mechanisms that connect them. The first abstraction is the user-level instruction set architecture as seen by the compiler back-end and the user-level part of the runtime system. The second abstraction is the raw, kernel-level instruction set architecture of the machine. This section describes the user-level abstraction we want to achieve and the rest of the paper describes the kernel-level hardware and runtime techniques used to implement it. The term *user-level* in this context refers to the protection domain. An application programmer generally will not write programs at this level of abstraction but will instead use higher-level abstractions provided by the compiler and runtime system [12].

The Alewife machine serves as our prototype of a single-user, multimodel multiprocessor [1, 14]. The machine consists of identical processing nodes connected via a message-passing network. FUGU (Figure 1) is arranged similarly, although with a second logical network. The Communications and Memory Management Unit (CMMU) serves to implement shared memory and serves as the network interface for message passing. Communication between the processor and the CMMU is both implicit, via loads and stores that the CMMU intercepts to implement shared memory, and explicit, via memory-mapped registers. The CMMU resides as close as possible to the main processor, in this case on the processor's external cache bus. FUGU also includes hardware facilities to provide translation and protection for user communications. These facilities are referred to collectively as the User Communication Unit (UCU), although they are in fact integrated with either the processor or the CMMU. For instance, the UCU includes the processor's TLB.

2.1 Communication Models

FUGU, similar to Alewife, exports three basic communication mechanisms to user level code: a coherent shared memory, a messaging interface efficient for short, processor-to-processor messages and a DMA facility integrated with the basic message interface for long, memory-to-memory messages. This subsection describes the user-level view of each communication mechanism.

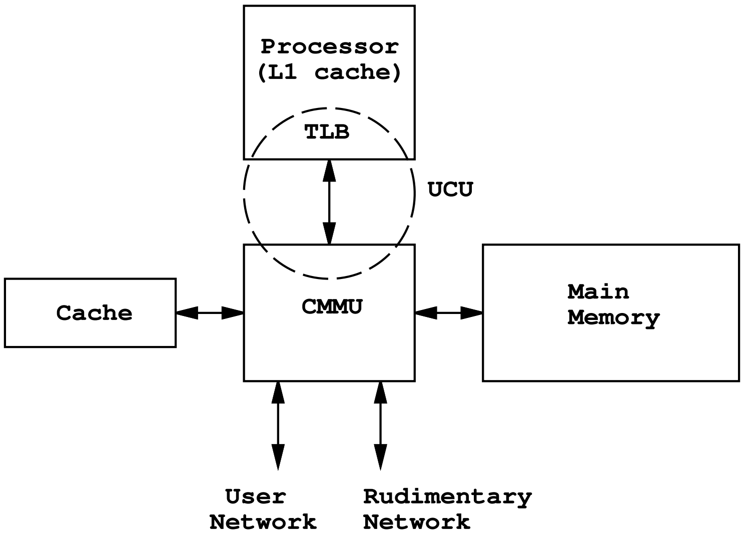

Figure 1: A FUGU node. The Communications and Memory Management Unit (CMMU) resides on the processor's external cache bus. The Translation Lookaside Buffer (TLB) is integrated with the processor. The User Communication Unit (UCU) refers to the set of hardware features that provide translation and protection for the hardware communication facilities.

Shared memory Shared memory communication operations are initiated implicitly in response to ordinary load/store operations. Each application has its own, private address space. Addressing is processor-independent across the processors executing an application. This addressing model is the usual shared-memory model. With virtual memory, processor-independent addressing implies that the logical translations for shared pages must be the same on all nodes.

Although the shared memory is accessed uniformly across all processors, it is non-uniform in implementation. Consequently, programmers have two underlying sharing mechanisms to be aware of: First, fine-grain (cache block size) renaming of locations for locality is provided by the caches. Second, coarse-grain (page size) renaming is provided by the virtual memory system. There is thus an opportunity for the programmer (i.e., the compiler and user runtime system) to optimize for locality by advising the kernel of a preferred static placement of pages and by choosing the partitioning of data within these pages. Alternatively, locality management can be left entirely to the caching hardware and the page migration software.

Short messages Messages are initiated and handled by explicitly communicating with the CMMU. The CMMU component appears as a set of memory mapped registers on the processor's external cache bus. CMMU registers are read and written with colored load and store instructions[1], `ldio` and `stio`, which are presumed to bypass internal processor caches and proceed at the speed of the external cache. Details of the interface to the FUGU CMMU are collected in Appendix A.

The message send interface consists of 16 CMMU registers that serve as a window into the network output queue and a `launch` operation. An outgoing message is composed in the window and then `launch` atomically commits the message to the network. The first word of the message is distinguished

[1] Such as provided by the SPARCLE processor [2] and the general SPARC architecture.

as the header and is interpreted by the hardware for routing. A minimal message can be launched from processor registers with three instructions:

```
stio   rheader, cmmu-output-registers[0]
stio   rvalue,  cmmu-output-registers[1]
launch
```

At the receive side, the processor is notified of the arrival of a packet either via polling or via interrupts, at the option of the user code. An incoming message appears in a complementary set of 16 CMMU registers forming a window into the network input queue. The processor uses the window to examine the incoming packet. A `storeback` operation disposes of packets after they have been read. Note that the receive interface is particularly raw: no buffering is provided beyond the CMMU's internal queues. Incoming messages block the network input port and potentially the network itself until they are examined and disposed of.

Bulk transfer messages Long messages with DMA are initiated using the same network interface as used for short messages. The difference is that descriptors for regions of memory (begin/end address pairs) are written into the outgoing window instead of values. Descriptors are distinguished from values by using another colored store instruction, `stdesc`. Single blocks are restricted to one page in size for reasons explained in Section 5. A block of memory could be included in a message as follows:

```
stio   rheader, cmmu-output-registers[0]
stio   rvalue,  cmmu-output-registers[1]
stdesc rbegin,  cmmu-output-registers[2]
stdesc rend,    cmmu-output-registers[3]
launch
```

At the receive side, a handler can invoke DMA to dispose of a message by writing descriptors for the destination blocks of memory into a third set of CMMU registers before invoking the `storeback` operation. The DMA interface is examined in detail in Section 5.

2.2 Scheduling Assumptions

We will call the basic unit of computation a *thread*. Each thread resides in a protected address space and has its own stack and program counter. A *process* is a collection of threads executing in the same address space on a single processor. Finally, a *group* is a collection of processes coherently sharing the same address space across several processors. Threads are scheduled by a user-level *thread scheduler* and processes and groups by a kernel-level *process scheduler*.

We expect multiprocessors to be used to run multiple groups simultaneously, as in Figure 2. Although communication may occur between groups, as in a client-server relationship, we assume that most communication occurs between threads of the same group. This assumption has two consequences. First, although we provide a means for general, inter-group messaging, we will optimize for efficient message transmission between threads in the same group (and protection domain). Second, we will assume that the process scheduler loosely gang-schedules processes within a group. Note that this latter stipulation is merely for performance reasons; we do not advocate enforced space sharing or hard-partitioning of

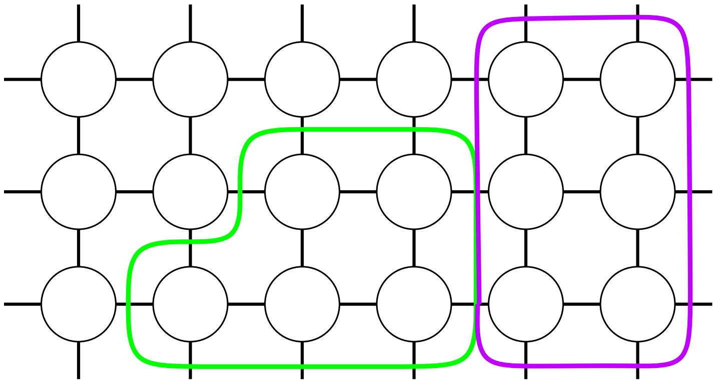

Figure 2: Two process groups scheduled together on separate nodes of the same multiprocessor.

the machine and gang-scheduling of partitions. Scheduling processes together appears to be desirable in general for multiprocessing [3, 26, 7].

2.3 Protection Model

User process groups should act as if they are running on a private, virtual multiprocessor. Multiple groups can be multiplexed on the hardware without interference, except in terms of performance. The protection model as viewed by the user can be summed up as follows:

1. Groups are isolated from each other in separate address spaces.
2. Groups appear to have a private network for intra-group messages.
3. Inter-group messages can be reliably identified by the receiver.

In addition, there is an implicit assumption about protecting the system and other users from errant user code:

- User bugs may affect performance but do not compromise the integrity of the machine.

In particular, we do not attempt to provide fair access to the network in hardware. We guarantee forward progress to the process scheduler which in turn is expected to guarantee forward progress to user processes. The assumption here is that in a general-purpose machine it is sufficient for a severely misbehaving application to be detected and controlled by administrative means.

2.4 Fault Model

We currently assume that all components are reliable. The network delivers all packets correctly. The nodes all share the same environment and administrative control, i.e., nodes are not shut down or altered independently and network packets cannot be generated or read by an adversary. This model is adequate for a large-scale multiprocessor as the environment it is in is generally less hostile than the environment for a network of workstations.

3 Translation and Protection for Shared Memory

This section describes our implementation of translation and protection for shared memory, in other words virtual memory. First, we show that the general translation scheme is justified. Next, the hardware and software structures for storing translations are explained. The page tables are arranged so that purely local operations can be separated from operations which might require communication with other nodes. Third, we show how to piggyback translation coherence on basic cache coherence provided there exist hooks from the coherence hardware to software.

3.1 Translation at the Processor

In FUGU, translation conceptually takes place at the processor as it emits an address. Translation at the processor means two things. First, translation at the processor is a memory model issue. Translating addresses immediately permits complete flexibility to map pages to arbitrary locations in the machine. Memory mapping provides a form of physical transparency by isolating the user software from the size and layout of physical memory in the machine. In addition, memory mapping enables the operating system to provide coarse-grain shared memory via static page distribution and dynamic page migration. We expect the operating system to make use of these coarse-grain sharing techniques to complement the fine-grain sharing provided by the native coherent caches. Various groups have experimented with tradeoffs between multiple shared memory mechanisms and found page-grain sharing beneficial on shared-memory machines without caching [22, 16, 7].

Second, translation at the processor is an implementation issue that provides a simplification and a complication. The simplification is that performing all translation at the processor permits the rest of the memory system, including the hardware shared memory system, to operate in terms of physical addresses. No translation information needs to be recognized or managed by the hardware in the memory system beyond the TLB. Virtual pages that are shared are mapped to physical frames of shared memory managed by the hardware. The cache coherence directories are associated with these physical frames so the frames must be localized ("cleaned") before a page can be swapped out.[2] The complication is that translation at the processor accelerated by a TLB introduces a coherence problem since translations cached in the TLBs must be kept consistent.

3.2 Translation Structures

Translation on memory references is performed using paged memory mapping, i.e., by referencing a page table by virtual page number to find the physical page number for each memory access. The lookup is accelerated by a TLB at the processor, as usual, so that in the common case no actual memory reference is required to perform the translation. In addition, in a multiprocessor, it is occasionally important to resolve references to local pages that miss in the TLB without incurring any potentially expensive references to global memory. For instance, in FUGU, the DMA mechanism, described in Section 5, takes different actions depending on whether a page is local or remote and needs to make the determination as quickly as possible.

[2] Note that translation at the processor does not necessarily restrict the system to using physically-indexed caches. For instance, virtually-indexed caches with physical tags may be supported by recording the active cache page number in the coherence directory associated with each memory line and permitting only one cache page per memory line to be active at a time.

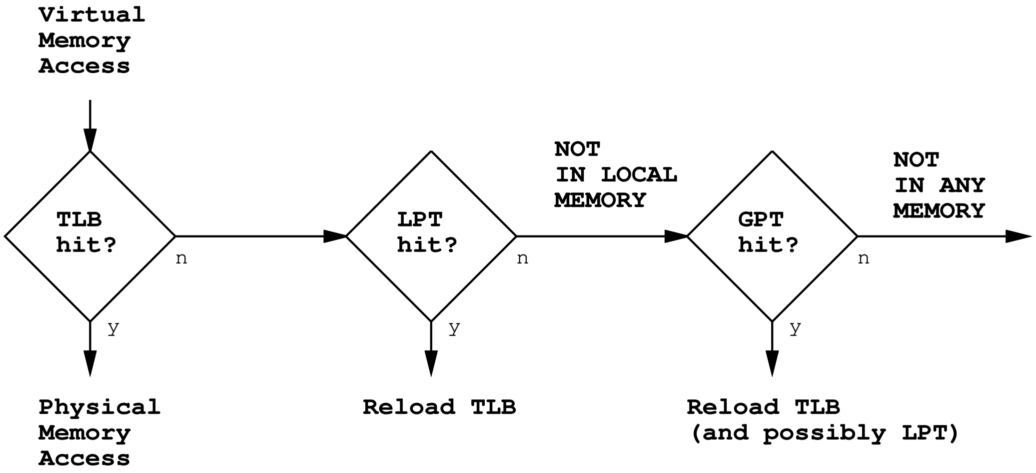

Figure 3: Address translation procedure.

FUGU uses a variation of the page table layout used by Cox and Fowler for PLATINUM [8]. The page tables are split into local page tables (LPTs), stored inverted in the private, physical memory of each processor, and a global page table (GPT), stored in distributed, shared memory. All pages that are physically resident on the node (whether private or shared) are listed in the LPT. The page translation procedure proceeds as illustrated in Figure 3. Translations that miss in the TLB are looked up in the LPT at the cost of a small number of references to local memory. Translations not found in the LPT are looked up in the GPT and potentially require accesses to global memory.[3] In essence, there are two levels of page faults reported: *not-in-local-memory* and *not-in-any-memory.*

3.3 Translation Coherence via PTE Coherence

The problem of maintaining a coherent view of a global virtual address space across a multiprocessor is generally called the TLB consistency problem [4, 23, 24]. Translations for global pages are read from entries in the global page table (GPTEs). In FUGU, GPTEs may be stored in several places aside from the processors' TLBs, including the main caches and the LPTs. Given that GPTEs are cached in multiple places, we want to unify the consistency mechanism for these entries for simplicity and in order to minimize the number of messages required to maintain coherence. If the GPT is stored in coherent global memory, then the machine already provides a coherence mechanism for GPTEs stored in the main caches. We find it possible to selectively extend the cache coherence mechanism to maintain all the translation information derived from the GPTEs and thus provide TLB coherence as a side effect of GPTE coherence.

Fugu's cache coherence mechanism, like Alewife, uses a directory-based invalidation protocol. A directory is associated with each memory line listing all nodes with possible cached copies of the memory line. When an operation that requires invalidation, such as a write, occurs, the memory side launches invalidation messages to each listed node. Each cache that receives an invalidation removes the line, if present from the cache and replies with an acknowledgement. In the common case, these messages are generated and interpreted in hardware, as in Figure 4.

[3]Both LPT and GPT accesses are accelerated by the main cache. In addition, GPT accesses can be accelerated by caching GPT entries in the LPT and explicitly managing the LPT as a cache.

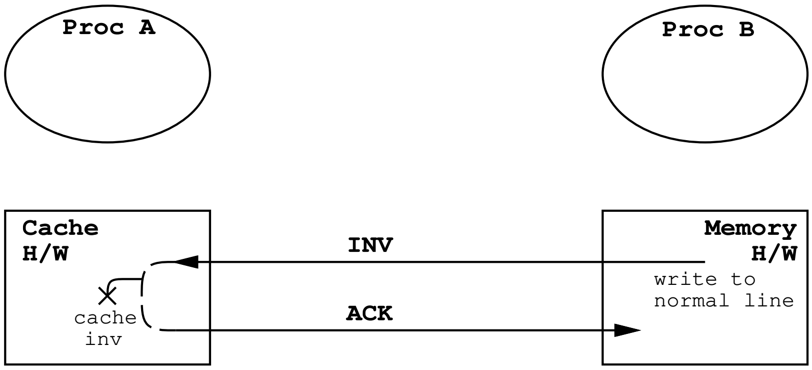

Figure 4: Normal invalidation is performed by hardware. Node *A* has cached a read copy of a memory line from Node *B*. When a write occurs, The hardware at Node *B* invalidates the read copy using messages generated and interpreted by hardware.

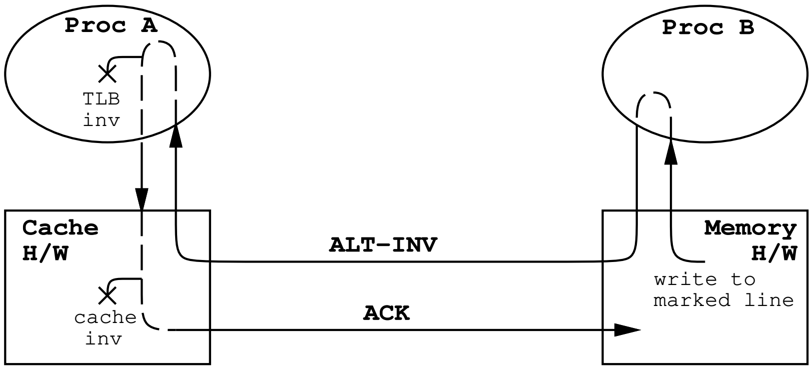

Figure 5: Alternate invalidation is performed partly by software. Node *A* has cached a read copy of a PTE from Node *B*. On a write to the PTE, Processor *B* is interrupted, allowing software to generate an alternate invalidation message that Processor *A* interprets as an invalidation to both the TLB and the cache.

Coherence for global page tables can be piggybacked on the cache coherence of the memory line on which the GPTE resides by causing an alternate invalidation message and action to be used for that line. The alternate invalidation message must cause any GPTE found in the TLB as well as the cache to be invalidated. A simple hardware mechanism that permits alternate invalidations to be added to the cache coherence protocol is a mode bit associated with each memory line that causes a trap to software on the processor associated with the memory whenever a write is made to that memory line. Figure 5 illustrates the use of this mechanism when the mode bit is set for all GPTEs. The memory-side trap handler then interprets the directory and sends alternate invalidation messages to the listed nodes. The message handler at the cache side flushes the entry from the TLB and then proceeds with the cache flush, responding to the memory side with an ordinary acknowledgement message.

Trapping to software at the memory side has been proposed for hybrid hardware/software implementations of cache coherence and for implementing special case coherence protocols [6]. Translation coherence is a case of an alternate cache coherence protocol: the trap is used to send alternate invalidation messages so that side effects may be added at the cache side. Note that hardware support for collecting and handling read requests, collecting acknowledgement messages, etc., will all be reused to implement the cache coherence.[4]

Directories for translation coherence have generally been kept explicitly in software. PLATINUM maintains a directory for each GPTE. The standard "TLB shootdown" algorithm maintains one directory for an entire page table [4, 23]. Maintaining directories at a grain finer than a page table is important to scalability because it generally reduces the number of nodes that must receive invalidation messages for any given invalidation. FUGU maintains a directory for several GPTEs together on a cache line. Lumping GPTEs together introduces false sharing over a scheme that maintains GPTE directories individually, but adds hardware support to accelerate directory maintenance. The actual performance of FUGU's translation coherence remains to be investigated.

3.4 Memory Protection

Protection on shared memory communication is provided by the usual memory mapping techniques. Process groups are placed in different address spaces and individual pages may be marked inaccessible or read-only. In terms of the taxonomy of protection mechanisms for communication discussed in the introduction, protection by memory mapping amounts to permitting user-initiated communication between pre-negotiated regions of physical memory.

4 Translation and Protection for Short Messages

Providing direct access to network interface hardware minimizes the costs of communication but increases the effort required to multiplex users on the hardware. In addition, the exposure of the system to problems caused by user bugs is greatly increased by permitting user access to hardware and by relying on the user to handle hardware events. The protected user-level network interface is the core of the FUGU design.

Translation for short messages is minimal and is described briefly at the beginning of the section. The remainder of the section describes protection for messages. Protected access to a user-level network

[4] An alternative mechanism using more hardware could flag memory lines containing GPTEs to automatically send alternate invalidation messages in hardware.

interface is a specific instance of a general problem of protected user-level access to any I/O device. We describe the mechanisms for exporting interrupts and polling to the user and then the mechanisms for providing protected access to the network interface. Finally we discuss how system software uses the hardware mechanisms to implement the protection model.

4.1 Translation of Node Names

Short messages are sent and received without references to memory, so translation on memory addresses is irrelevant. However, full physical transparency for user code requires that node names be virtualized as well as memory locations. In other words, some translation mechanism is required for the node destination addresses written at the head of the packet. In FUGU, we currently leave this translation to the compiler or runtime system. As a consequence, there is minimal control over where a user process directs its messages. We apply protection only at the receiver to detect code that launches a message to an inappropriate destination, as described at the end of this section.

4.2 Protection for User-Level Messages

User control over the network is a specific instance of user control over an I/O device. I/O devices are ordinarily protected by reserving access to the operating system kernel and having the kernel check protections on every request. Kernel-mediated access to the device can be tolerated only if the device is relatively slow. User-level messages over fast VLSI interconnection networks must avoid the kernel.[5]

The interface to an I/O device (an interconnection network, in our case) requires four features to make the device user-level:

- *Notification:* I/O device operation is asynchronous and requires a mechanism for notification of events. For networks, both interrupts and polling are desirable mechanisms, as described below. The user must have a means to *disable* notification in order to provide atomicity with respect to interrupts and to select polling operation.
- *Isolation:* device events and incoming data associated with the I/O device must be known to belong to the user.
- *Permitting sharing:* there must be a means to context-switch the I/O device to multiplex between users.
- *Enforcing sharing:* there must be a means to enforce access to shared resources. The I/O device and associated resources may be shared with other users and with the kernel.

An existing, simple example of user control over an I/O device is a bit-mapped display. It makes sense to permit direct user access to this device because the bandwidth requirements are very high. There are no notification issues because it is an output-only device. Sharing is provided by memory-mapping the memory associated with the display. The device is isolated by mapping it to one process at a time.

[5]Note that for our purposes, a device is at *user-level* if code running at user-level can affect hardware resources without any checks made by the kernel. With this definition, for instance, the feature that makes an interrupt user-level is not whether the actual code paths go through the kernel or not. Avoiding the kernel is an optimization. The key concepts are that the interrupt code is user-provided and that user has control over whether or not the interrupt is signaled.

Requirement	Mechanism
Notification	Kernel-delivered user interrupts. User-controlled polling.
Isolation	GID stamped on messages at sender. Demultiplex on GID at receiver.
Permitting sharing	Atomic `launch` operation. Buffer and relaunch.
Enforcing sharing	Receive handler timeout. Buffer and scheduling using the second network.

Table 2: Mechanisms to permit and protect user access to the network interface.

Sharing is permitted, because the device can be mapped to any process. Sharing is enforced because the kernel controls the maps.

FUGU provides means for users to control the network interface directly yet share it using the mechanisms listed in Table 2. We address the issues of *notification*, *isolation*, *permitting sharing*, and *enforcing sharing* for FUGU's network interface separately below.

Notification FUGU supports both polling and interrupts on network events at user level. User traps are delivered through the kernel, but may be disabled by the user. Two CMMU registers support selective polling on network events and atomicity with respect to network events. The `user-trap-pending` and `user-trap-mask` registers both contain a bit for each type of event. The bits in `user-trap-mask` indicate which events should cause traps and the bits in `user-trap-pending` indicate which events have occurred. If the trap for an event is disabled, the user can poll on `user-trap-pending`.

The trap enable for message input has additional functions. When message reception is disabled, instructions that might block for network resources, e.g., shared memory loads and stores, are made illegal and cause a kernel trap immediately. Additionally, a timeout counter counts whenever message reception is disabled in order to enforce sharing. We describe interrupts and polling in general in this section and come back to the extra features of the message input enable under *enforcing sharing*, below.

FUGU provides both polling and interrupts because each provides the best performance in its operating domain. Interrupts provide the lowest overhead if events are infrequent because polling checks are not required. On the other hand, polling provides the lowest overhead in the case where an event is expected. Consider the several alternate notification sequences in Figure 6. These sequences illustrate three possible message receptions: using interrupts only, using interrupts with the GID check wired into the hardware, and using polling. Polling requires that the GID check be wired into hardware. Interrupts which pass through the kernel incur a number of extra cycles of overhead even for the most minimal code. Interrupts delivered directly to user level are potentially much faster but are not available on current processors. Potentially even more expensive for interrupts is the disruption of processor state.

We provide polling to take advantage of its minimal latency in cases where the data are expected. Such cases arise (1) in code with rigid static scheduling, (2) in code with communications phases that exchange many messages in bursts and (3) on server nodes that handle many short messages.

An example of a server process is a lock-server: a process on a node that maintains locks accessed by messages. A simple experiment on a simulated Alewife machine changed the lock-handler in a

Kernel interrupt	Kernel interrupt + H/W GID check	User polling + H/W GID check
(6) Interrupt	(6) Interrupt	
(3) Check GID		
(8) Jump to user	(8) Jump to user	

Figure 6: Message delivery latency for various hardware options. Numbers are approximate cycle counts on a SPARC processor.

synthetic application from purely interrupt-driven operation to polling operation. This was done by having the interrupt handler optimistically check for a message before returning from the handler. Using interrupts alone, the throughput of the lock-server averaged one lock every 100 cycles. Switching to polling increased the throughput to one lock serviced every 48 cycles on average. When polling, the server avoids unnecessary copying of state information and can keep common variables in registers across invocations of the handler.

Isolation Isolation is achieved by tagging every message with the sender's group identifier (GID) and disabling network access at the receiver until the tag has been verified. A `GID` field is defined for all user messages and an `input-enable` control bit is associated with the user network input interface. When `input-enable` is clear, user accesses to the network input interface are illegal and cause a trap.

Because the user has direct hardware access at the sending side, we stamp the sender's GID in the `GID` field of outgoing messages by hardware to prevent forgery. This operation requires a hardware copy of the GID to the message at the time of the `launch` operation. At the receive side, the GID must be compared to the `GID` register before delivery to the user to provide isolation. We perform this comparison in hardware to permit user polling without the overhead of a kernel trap. If the GIDs match (i.e., the message is an intra-group message), the network receive interface is enabled (`input-enable` is set), delivering the message directly to the processor. The `input-enable` is automatically reset when the message is disposed. If the GIDs do not match, a trap is caused to deliver the message to the kernel to be buffered as described in Section 4.3. Also, a subset of user-generatable message types representing remote supervisor requests are always delivered to the kernel.

The hardware stamping and checking of GIDs on intra-group messages provides the illusion of a private network. Inter-group messages are achieved by having the kernel provide protection. An inter-group message uses a message type that causes it to be delivered to the kernel at the receiving node. The kernel software then applies arbitrary protection checks and, if the message passes the checks, enables the interface by setting a `input-enable` control bit. The receiving user can reliably identify the sender via the GID in the message.

Permitting Sharing The FUGU network interface permits sharing by providing mechanisms to transparently unload and reload state from the network on context switches. Context switches are invoked by the kernel-level process scheduler and may be invoked by a user-level thread scheduler. Partially composed outgoing messages and any incomplete incoming message are the most significant pieces of state.

At the sending side, the outgoing message window can be saved and restored at any time because the `launch` operation atomically commits the outgoing message to the network. Up until the time of the launch, the CMMU registers used to compose messages can be unloaded and reloaded as necessary.

At the receiving side, the input interface can be drained by using DMA to store the incoming message to buffer memory and then the message can be handled at a later time, as described in section 4.3 below. Our experience with Alewife suggests that the network input port is not likely to be context-switched by a user-level thread scheduler because the user message handlers will ordinarily pull the message from the network with user interrupts disabled. Extraordinary circumstances such as page faults in the user handler code or preemptive switches of the process by the kernel could leave the input interface filled.

Enforcing Sharing The user network is a resource common to multiple users and to the operating system. User code cannot be trusted to share these resources, so mechanisms for forcing sharing are required. We enforce sharing by providing a timeout whenever the user disables interrupts and by providing a rudimentary second network to allow the operating system to recover when users fill the primary network. These mechanisms solve two problems described below.

The first problem is to assure that user handlers on the receive side actually empty the network and that the kernel can recover control from a user that fails to empty the network. We provide a timeout counter that counts user-mode instructions whenever network input interrupts are disabled. The counter is zeroed either automatically when the user disposes of a message or by user request when no input message is waiting to be handled. A kernel trap is invoked if the timer reaches a maximum count. The timeout period is well defined to user code and timeout traps indicate errant user code. For messages handled by interrupt, the timeout forces the message handler to dispose of the message within a timeout period. For polling, the timeout forces the sum of the polling period and the handling time to be less than the timeout period. It is possible to use a less strict timeout, such as the the timeslice interrupt, to recover control from a user handler, but our specialized timer provides the important advantage of a tight, carefully checked bound on the timeout period. As a practical point, closely timing every invocation of the message handler (as opposed to sampling with a less strict timeout) finds user bugs more rapidly and reliably.

The second problem is to control access to the network. Access control is difficult in general without imposing flow control on user messages in hardware. There are two components to this problem. One is that, with only user-enforced flow control, the user can flood the network with messages faster than receivers can handle them, stealing network resources from other users and potentially requiring unbounded amounts of buffer space at the receiver. The other is that there is no mechanism at the sending node to prevent a user process from sending arbitrary messages to any destination node.

We address access control indirectly by using the rudimentary system network to allow the operating system to sink messages indefinitely and to communicate with the sending node's process scheduler, as described in section 4.3 below. The existence of the rudimentary system network enables FUGU to control user access to the network by checking messages only at the receiving side and to guarantee buffering at the receiving side without requiring dedicated physical memory. Since the second network is required only infrequently, it requires less performance and thus less hardware than the main, user network.

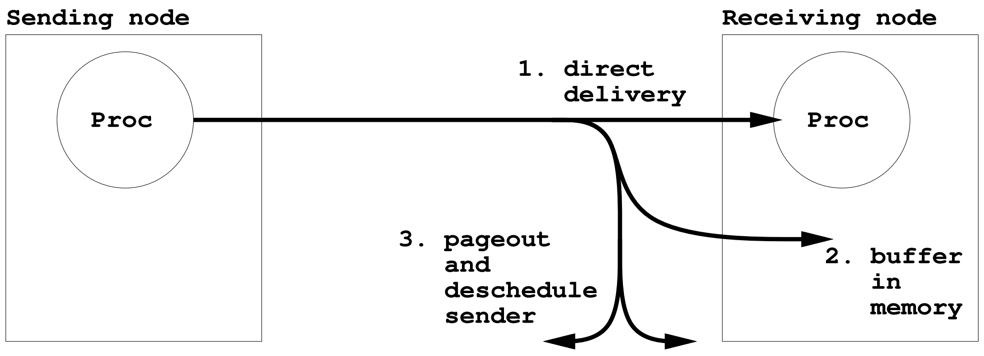

Figure 7: Message reception cases.

4.3 Message Buffering and Scheduling in Software

Figure 7 illustrates the possible actions at the a receiving node when a message arrives. In the best case, (1), the message GID matches the running process and the message is handled directly by the processor. In the second case, (2), if the GID is mismatched, the message is buffered by the kernel in private memory. In the worst case, (3), when excessive buffering is required, the rudimentary network provides a channel to send a message to the sender's process scheduler and, if required, allows the kernel to swap virtual pages to provide more space for buffering.

The process scheduler is responsible for assuring that the best case is also the most common case. User code is responsible for handling messages in a timely fashion and for providing flow control for its own messages. The other rare cases exist because neither the scheduler nor the user can be considered completely reliable.

Buffering A receiving node must deal with messages that user code cannot handle immediately. In FUGU, the kernel always buffers such messages at the receiver so that the sender can consider a message delivered when the `launch` operation commits the message to the network. Ideally, messages are buffered in local physical memory at the receiver. Unfortunately, it is difficult to bound the amount of buffer space required and we don't want to devote buffering resources to what we expect to be an infrequent case.

The solution is to buffer incoming messages in the sending group's virtual memory. Virtual memory provides an effectively unbounded buffer for incoming messages. Ordinarily, buffered messages will be placed in a local page already allocated or local pages freshly allocated from the virtual memory manager's free list. The virtual memory manager attempts to keep a supply of free frames available for its own use. By using virtual memory, FUGU provides buffering in local memory when required without locking down or pre-negotiating physical memory on the receive side.

In the extraordinary event that free local page frames are exhausted and the user network is blocked, the rudimentary system network provides a channel for paging. Applications that perform such a poor job of flow control that they overflow the network onto backing storage are suffering from performance bugs. The use of virtual memory allows a graceful degradation in performance that permits debugging for correctness separately from debugging for performance.

Buffered messages must be eventually delivered to and handled by the user. An appropriate software architecture provides an efficient solution. If user-level handlers are written with an abstraction that hides the hardware message storage mechanism, a compiler can generate two sets of binary handler objects from a single source code. One binary object processes messages directly from the network interface and the other to reads messages from the buffer in memory. Any poll for incoming messages must check both the network interface and the buffer. Messages diverted asynchronously while a user handler is actually running are tolerated by having the protection trap that occurs on accesses to a disabled receive interface emulate accesses to the interface.

Scheduling FUGU makes it possible to rely on the kernel scheduler and the rudimentary network to provide a gross form of flow control for errant user processes. We identify events that indicate that problems exist and we provide a channel (the system network) for propagating this information back to the sending node's kernel. Direct control over the sending process is left to the scheduler on the sending node.

First, the process scheduler may be used to throttle processes that flood the network with messages. Suppose a process is performing a poor job of flow control and is requiring messages to be buffered at the receiver. The virtual memory system can advise the system scheduler of the excessive traffic. Suppose an incoming message has to be buffered and requires a new free page. Further suppose the allocation of a new free page causes the number of free pages on the node to sink below a low-water threshold. This event is detected by the kernel and causes the kernel to advise the sending node, via the system-only network, to deschedule the sending process and thus throttle the excess traffic.

Second, note that there is no mechanism at the sending side to prevent a user process from launching a message to an arbitrary destination. Detection of unexpected messages is provided by kernel code at the receiving side. Suppose a runaway user process sends an inter-group message to a non-existent process. The message will be delivered to the kernel at the receiving node. The kernel at the receiver can inform the kernel at the sending side which then takes appropriate action, such as terminating the sending user process identified by the GID in the unexpected message.

5 Translation and Protection for Messages with DMA

Support for bulk transfer message is provided in FUGU by DMA facilities built on top of the basic messaging interface. FUGU incorporates two identical DMA engines, one for transmit and one for receive on the user network. As described briefly in Section 2, the messaging model exports these DMA facilities directly to the user. Such direct access is uncommon in most hardware platforms, where access to DMA mechanisms is restricted to I/O device drivers. Not surprisingly, the use of DMA in FUGU raises several issues which must be considered for correct operation: First, the user's notion of memory is confined to virtual addresses. As a result, virtual addresses from the user must be translated to physical addresses for the DMA hardware. Once this translation has occurred, the resulting physical addresses must remain valid throughout the duration of the DMA operation.

Second, the use of DMA in conjunction with cache-coherent shared memory raises questions about the level of memory coherence which can be expected from DMA-transferred data. This is the DMA-coherence problem introduced in [14].

Third, the use of DMA implies some form of asynchronous notification mechanism to indicate to the user that previously requested DMA operations have completed. Ideally, all three of the above issues

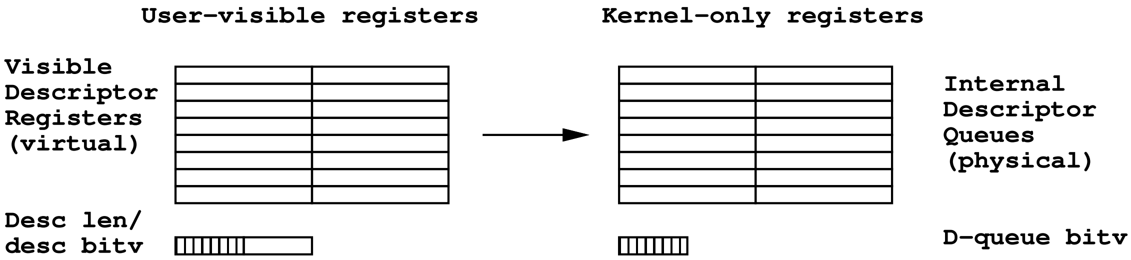

Figure 8: Registers associated with a network output port. Identical interfaces are used for DMA at the input.

should be addressed in a way which does not compromise the efficiency of direct, user-level access to the hardware. These issues are the topic of this section.

5.1 Translation for DMA

This section proposes a mechanism for combining fast DMA (such as in the FUGU network interface) with virtual memory. We handle the common case in hardware and trap to software otherwise. In particular, we consider the common case to be:

- Source regions are mapped to pages that are local to the processor on which output DMA is initiated.
- Destination regions are mapped to pages that are local to the processor on which input DMA is initiated.

If both of these criteria are satisfied, then translation and data transfer can occur directly in hardware. In the common case, pages are implicitly locked during the DMA transfer and unlocked after the DMA completes, with *no intervention by the operating system*. Should DMA be attempted to or from non-local pages, the we employ software trap handlers to present to the user the illusion that DMA has occurred in a single request.

Hardware support for DMA translation First, we would like to examine the method by which the user describes and initiates DMA requests. Section 2 discussed the user's view of a DMA mechanism. This consisted of an array of descriptor registers which were written with starting and ending addresses for DMA operations. Figure 8 shows a more extended view of the registers which participate in DMA requests. Each user-visible descriptor register is shadowed by a register which is visible only to the kernel. In fact, these shadow registers are part of a queue of descriptors which contain all previously queued but pending DMA operations.

To describe a DMA operation, the user stores virtual addresses into the user-visible descriptor registers using the `stdesc` instruction. The `stdesc` instruction translates a virtual address using the processor's TLB or page-fault handlers, then attempts to simultaneously write the virtual address into the requested descriptor register and the physical address into the corresponding shadow register.[6] Dur-

[6]These two pieces of information are sent on the data bus and address bus respectively. The register is selected by a different set of bits, such as the SPARC ASI bits

ing the process of the write, the physical address is examined to see if it represents a local physical address. If so, the write simply finishes. If not, then the write is synchronously faulted; these "nonlocal" faults pass control to special trap handlers that handle the uncommon DMA cases.

Virtual addresses in the user-visible descriptor registers are not used by the DMA hardware and are discarded as soon as a DMA request is committed by `launch` or `storeback`. The virtual addresses are present so that user-level thread schedulers and interrupt handlers can gain access to the DMA descriptor queue by saving and restoring the state of "in-progress" DMA descriptions. In contrast, the queue of physical addresses (which includes those page frames which are part of on-going DMA descriptions) *are* used by the DMA mechanism. Furthermore, this entire queue is visible to the operating system. A bit mask is available to indicate which of the queue entries contain "live" physical addresses, namely those which are part of pending DMA operations. Addresses which are visible in this way are considered implicitly "locked", as are the physical page frames they reference. Although we have not yet described what it means for a frame to be locked, notice that the set of locked frames is updated dynamically as DMA operations are queued and subsequently finished.

Output DMA operations Give the above hardware, we can export a uniform output DMA interface to the user. We assume that the user performs higher-level synchronization around sections of code which are sending data through DMA. In particular, data at the source of a DMA operation is *assumed to be constant throughout the DMA description and queuing process.* Changes to source data which occur during this process are not guaranteed to be reflected in the results of the DMA.

With these semantics, we can guarantee that an output DMA operation is unaffected by two things:

- Attempts to send data which resides in pages on remote nodes.
- Page migrations and other changes in virtual mappings which occur during the course of the DMA description process and during the period in which DMA is queued.

Both of these issues are handled in a straightforward manner.

First, we assume that the version of the `stdesc` instruction which is used to describe output DMA requires only *read* permission to addresses that it translates. Then, any attempt by the user to describe DMA from a remote page is interrupted by a "nonlocal" fault, as described above. The trap handler can deal with a nonlocal DMA request in several ways. We assume that user programs use messages because they know something about where data are located and thus that nonlocal faults are unusual events. As a result, we use the simplest means to recover from a nonlocal fault: the trap handler sends a message to the node with the data requesting that a read copy of the page be migrated to the local node. The DMA proceeds after the page returns and TLB entries are updated to reflect its new location. Note that because message launches are atomic, system software is free to temporarily unload and then later restore the partially constructed message output queue in order to schedule another process or thread while waiting for the page fault.

Second, for correct DMA behavior in the face of changes in virtual address mappings, we simply guarantee that page frames which are actively queued by the DMA interface (i.e. "locked") are not reused until *after* the DMA interface has finished with them. This guarantee is sufficient, since we have asserted that output data must not change during the description and queuing process. To accomplish this feat, we maintain two free frame lists: one which is used for freshly freed frames, and one which is used during page allocations. When necessary, frames are moved from the first free list to the second by verifying that they are not locked in any output DMA queue.

Input DMA operations Similar to the output side of DMA, we wish to guarantee that an input DMA operation is unaffected by three things:

- Attempts to receive data into buffers which are marked read-only because they are read-shared by multiple nodes.
- Attempts to receive data into buffers which reside on remote nodes or backing store.
- Page migrations and other changes in virtual mappings.

Assume that the version of the `stdesc` instruction which is used to describe input DMA requires *write* permission to the addresses that it translates. This check for write permission permits the first situation to be flagged by the page-fault handler. Then, the first and second situations may be handled in similar ways: a fresh frame is allocated off the free list. The `stdesc` instruction is emulated by writing the new physical address directly into the shadow registers. Later, when the DMA operations have completed, we request write copies of the real pages; when they arrive, we merge the new data with the old, thereby logically completing the input DMA operation.

Finally, to prevent page migrations from affecting input DMA operations, we check each migration request to see if it is requesting a page which is mapped to a frame that is locked in some input DMA queue. If so, we permit the migration, but schedule the (newly unmapped) local page frame for future merging with data.

5.2 DMA and Cache Coherence

The use of DMA in a cache-coherent shared memory multiprocessor raises the specter of DMA coherence [14]. The most general possible DMA coherence model is that of *global coherence*, in which data in both the source and destination blocks of memory are fully coherent with respect to all processors. A somewhat restricted DMA coherence model is that of *local coherence*, which guarantees that data at the source and destination are fully coherent with respect to local processors; this means that, if necessary, data will be retrieved from the cache at the source, and invalidated from the cache at the destination.

Local coherence is easier and more desirable to support in hardware for a number of reasons [14]. Furthermore, many uses of messages involve separate message buffers which are reserved exclusively for message traffic [13]. Thus, we propose to support locally-coherent DMA in hardware and to synthesize global coherence with software only when required, just as in Alewife.

Block cleaning However, we also wish to support general paging, including page migration. Further, we wish to support explicit rearrangement of shared data by the user. Both of these applications require some form of global DMA coherence. Consequently, to assist in the synthesis of global coherence, we postulate the existence of a "block cleaning mechanism" which could:

- Perform a *collect* operation before initiating output DMA, by scanning through directories in the source block of memory and invalidating those with outstanding dirty cached copies.
- Perform a *block invalidate* operation before initiating an input DMA, by invalidating *all* outstanding read and write copies.

In both cases, we want some form of notification when the cleaning operations have completed.

Hardware support for block cleaning It is an open question as to whether hardware support is needed for block cleaning, or whether software access to hardware coherence directories is sufficient. This is a question that we plan to explore.

In brief, however, we would like to note that both of these operations, collect and block invalidate, could be accelerated in hardware through the user network output interface. Such *cleaning* requests would be constructed in the output descriptor queue and queued like normal messages, with the exception of a special "cleaning" header. The blocks of memory to be cleaned would specified by the DMA portions of the cleaning request. When processing a cleaning request, the output DMA mechanism would simply scan through each of the specified cache-coherence directories, searching for directories which are not sufficiently "clean" and sending invalidations accordingly. Synchronization for the cleaning operation could then be provided by setting a special *cleaning synchronize* bit in each directory that has been invalidated, retaining a count of the number of directories which were cleaned, and decrementing this count as acknowledgments return.

5.3 DMA and Notification

The last of the three issues that we identified above is that of notifying the user that DMA operations have completed. We make use of hardware-generated transaction identifiers for this purpose. Conceptually, we have two counters, called `trans-head` and `trans-tail` associated with each DMA interface. The `trans-head` register contains the transaction identifier for the next request to be queued; this counter is incremented with each new DMA request. The `trans-tail` register holds the transaction identifier for the DMA operation which is currently in progress; this counter is incremented whenever a DMA request completes.

To handle the fact that input DMA operations are not complete from the user's standpoint until after outstanding merge operations have finished, we export a virtual tail pointer to the user, called the `user-trans-tail`. Under normal circumstances, this register simply tracks the `trans-tail` register. On rare circumstances in which data from an input DMA must be merged with remote data, the page-fault handler can freeze the `user-trans-tail` at a maximum value by writing to the `frozen-tail` register.

With the above mechanism, the user can poll for the completion of a particular DMA operation by saving the transaction identifier at the time that the request is queued and watching the `user-trans-tail`. To support DMA completion interrupts, we provide an `int-trans-id` register, which is set with the current transaction identifier whenever a DMA operation is queued with a request for interrupt. An interrupt is posted when the `user-trans-tail` exceeds this value.

5.4 Protection for DMA

Long messages make use the same delivery mechanisms and thus the same protection mechanisms as short messages to provide isolation between process groups. Memory accesses by DMA are protected by memory mapping in the same manner as ordinary memory accesses. The frame locking technique preserves the protection provided by memory mapping. Specifically, if a process initiates a read from (or write to) a page with DMA and then unmaps it, no other process can map and then erroneously or maliciously write over (or read from) the page because the frame is locked until the DMA operation completes.

6 The FUGU Experimental Platform

FUGU is an experimental architecture and provides a framework for exploring issues in multiuser multiprocessing. FUGU heavily leverages software and hardware from the Alewife machine. Most of the functionality of FUGU described in this paper can be achieved by augmenting Alewife nodes with a single-chip user-level communication unit, or UCU. Work on a detailed simulator based on the Alewife binary-compatible simulator is in progress.

We intend to leverage Alewife hardware to build a prototype that we will use as a near-real-time emulator of a FUGU system. Experiments performed on the prototype will be used to calibrate and verify the results obtained from the FUGU simulator.

Most of the functionality of FUGU can be implemented by adding a TLB, a rudimentary network, and protection checking hardware to an Alewife processor node. Because we are building on top of the existing Alewife system rather than building from scratch, some protection features that would be straightforward to implement from scratch are difficult to provide in this prototype. Rather than add complication to this implementation to make the protection complete, we will rely, for demonstration purposes, on a trusted compiler and runtime system for some protection checks.

This section describes the hardware mechanisms we will actually implement and how they differ from the architecture described in the first part of the paper.

6.1 The User-Level Communications Unit

The hardware mechanisms will be implemented in a single-chip UCU. The chip will be implemented initially in an FPGA (Xilinx 4025). Figures 9 and 10 show a basic Alewife node and a FUGU node which is an Alewife node augmented with the UCU and the rudimentary network. The UCU implements:

1. A coherent TLB.
2. Protection checks on incoming user-level messages.
3. The rudimentary second network.
4. Translation for bulk transfer messages.

The UCU and kernel, in conjunction with compiler and runtime software, support most of the mechanisms for translation and protection described in Sections 3, 4 and 5. The UCU features to support translation and protection for shared memory, short messages and bulk transfer messages are described below.

1. Translation and protection for shared memory are provided by a virtual memory system built around the TLB implemented in the UCU. The TLB coherence mechanism proposed for FUGU can be completely implemented with this TLB.
2. Translation for short messages is via compiler, as proposed earlier. Protection for short messages relies on a trusted compiler for two phases of message transfer. First, the compiler is required to stamp the GID on messages as they are sent. Second, at the receive side, the compiler will provide message handling code that either does not time out or that sets up a timeout counter.

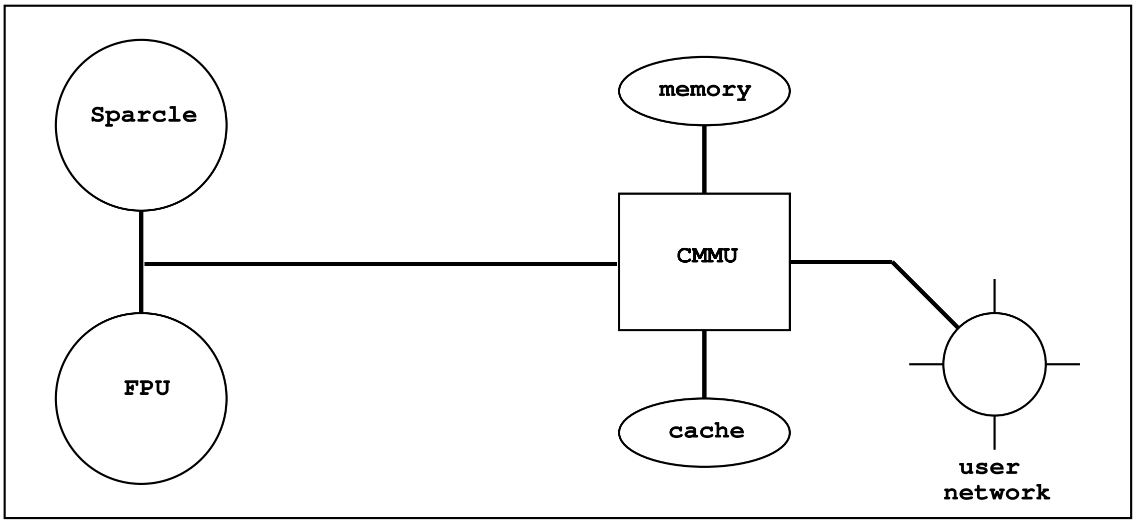

Figure 9: Alewife node architecture.

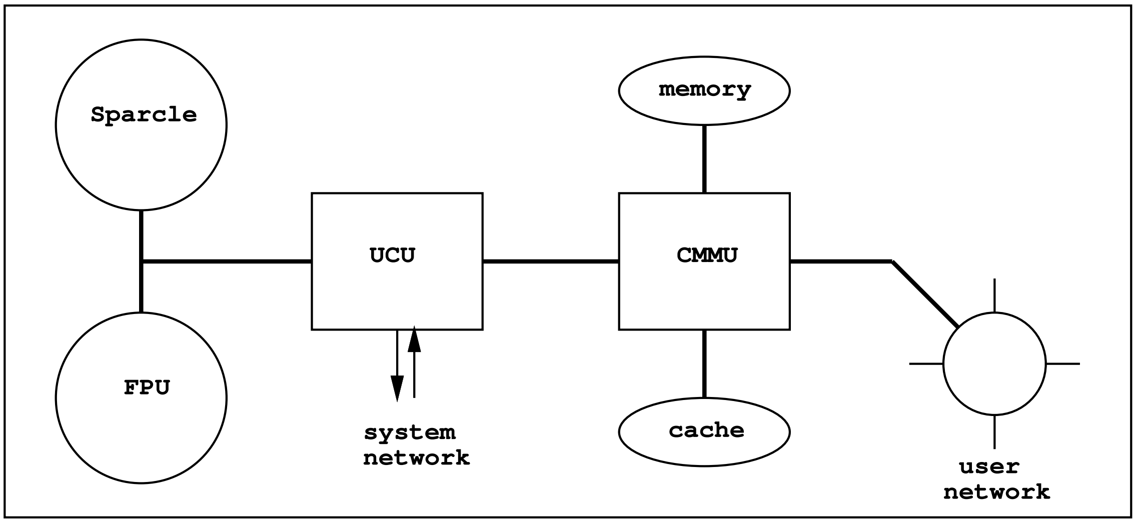

Figure 10: FUGU node architecture.

The GID on a messages is tested in hardware by the UCU at the receiver. We will also experiment with GID testing by low-level kernel code. The UCU protection hardware can be eliminated if the performance is acceptable. The FPGA implementation of the UCU will allow us to change such hardware details.

The second, rudimentary network is used chiefly to throttle incoming messages. The kernel achieves this throttling by sending a control message to the kernel running on the sending node. In extreme cases, the kernel can use the rudimentary network to page out buffered messages. The UCU will include the hardware to implement the second network, as shown in Figure 10.

3. Translation for bulk transfers is provided by a combination of a trusted compiler and the UCU. For space reasons, the number of blocks that can be launched will be limited to two, each of which can not cross a page boundary. User code will probe the UCU for address translations and then copy the resulting physical addresses to the CMMU's DMA queue. The UCU will store the virtual page numbers used in case the DMA queue needs to be transparently unloaded and reloaded. We will also investigate providing the reloadable queue using only software.

   Finally, since the DMA physical address descriptors are not all available, the page frame recycler will have to operate by unloading any message currently under construction and waiting for any pending DMA operations.

6.2 Differences from the FUGU Architecture

The FUGU prototype hardware described above will allow most of the functionality described earlier, but requires that some protection checks be provided by software and requires extra overhead for some operations. The protection checks would be built into hardware in straightforward ways in a production system. We believe that the various extra overheads will not qualitatively impact our results. The differences between the hardware prototype and the FUGU architecture are summarized below.

- In the experimental version of FUGU, protection on messages includes a GID stamp added by a trusted compiler to all outgoing messages at a cost of potentially one extra cycle per message send.
- The stamped GID must be tested each time a message is received. When the GID check is done by the UCU, no extra cycles are incurred on message receives. If done in software, however, short message receives would cost an extra test and branch (at least 2 extra cycles).
- Message reception via polling either must use code guaranteed by the compiler not to time out or must explicitly maintain a timeout counter.
- We enforce the restriction that DMA launches and storebacks cannot cross page boundaries, because implementing a hardware mechanism that provides a notification when this happens requires modifying the CMMU chip.
- The page frame recycler must wait for pending DMA operations instead of just checking descriptors in the CMMU.
- User-notification for DMA doesn't work exactly as described in Section 5. Our implementation will interrupt the processor even if the running process has a different GID, and the UCU or low-level kernel code will perform the protection check.

- Bulk message sends will cost several cycles more per descriptor in our implementation that in a completely optimized version, because translation is performed by software probes of the TLB.

7 Related Work

The FUGU effort attempts to demonstrate a scalable multiprocessor which supports multimodel and multiuser operation. It focuses on hardware and software mechanisms for protection and translation in scalable multiprocessors that provide user-level messaging and coherent shared memory. The FUGU effort relates to other research projects as follows.

Alewife [1] is a single-user, multimodel multiprocessor. It has user-level message handling integrated with coherent shared memory. It supports both short messages and bulk transfer (DMA) messages, with a unified packet interface. Bulk transfer messages require the receiving processor to determine the destination of the data. FUGU extends Alewife features for multiuser operation and uses an exokernel operating system. In FUGU, bulk transfers require translation at the receiving processor, but do not pre-negotiate for pages.

FLASH [15, 10] is a multimodel machine with a microcoded, kernel-level coprocessor for message handling including shared-memory protocol messages. Bulk transfers in FLASH are in the form of remote-writes which avoid using the receiving processor, but require pre-negotiating the sending addresses in shared memory. For instance, there is no notion of bulk transfer between purely private memories. They use implicit locking on DMA operations similar to FUGU, but load the message send operation and TLB coherence algorithm with the overhead of marking and checking for pages in use in the MAGIC chip and introduce deadlock by locking pages instead of frames. FLASH uses two networks but for a different reason than FUGU. FLASH uses two networks for request and reply packets to avoid deadlock. Packets can be sorted into request and reply networks because the message protocols are controlled by the kernel-level coprocessor. FUGU mostly uses one network with mixed kernel and user traffic, but relies on a cost-effective, rudimentary network to avoid deadlock.

Typhoon [21] offers user-level message handling and user-level cache coherence using a second processor dedicated to the network interface. The second processor lengthens the message latency in the best-case situation, although it provides concurrency in most other situations. Typhoon provides network protection by context-switching the network in the manner of the CM-5 [17]. Context-switching the network fails to support a client-server model of interprocess communication.

The *T [18] processor uses a memory coprocessor model as well. *T does not include DMA facilities or coherent caches and thus does not address the interaction of these features with messaging. A recent *T paper [20] has independently proposed protection mechanisms similar to FUGU's for short messages and makes the same assumptions about separating performance from correctness in scheduling. Our focus is on developing a minimal set of mechanisms that are sufficient for user-level message handling and protection, while *T proposes a richer set of features. For instance, *T demultiplexes messages into several receive queues, and the receive queues are implemented as a part of the processor register set.

SHRIMP [5] and Hamlyn [27] propose remote writes for protected user communication between pre-negotiated and pre-locked pages.

Recent work on parallel processing on networks of workstations seeks to identify mechanisms to accelerate communication while retaining protection. Work by Thekkath and others at the University of Washington [25] advocates remote memory access to pre-negotiated buffers as a mechanism to accelerate communication while retaining protection. Similarly, Hybrid Deposit [19] proposes hardware to interpret messages as operations on pre-negotiated buffer areas. FUGU's approach is to add protection while maintaining existing, well-defined user-level communication mechanisms and efficient, distributed shared memory.

The J-machine multicomputer [9] provides two levels of network priorities, user-level access to the network hardware and the ability to relaunch incoming messages from memory transparently. The J-machine is a single-user machine with no support for shared memory or DMA on messages.

The CM-5 multicomputer provides multiuser multiprocessing by rigidly partitioning its network and context switching entire partitions [17]. It does not support shared memory.

8 Conclusion

Multiprocessors which integrate hardware support for message passing and shared memory are of great interest because they combine the best of two worlds: the efficiency of message passing, when communications patterns can be statically determined, and the flexibility and convenience of shared memory for fine-grained and dynamic computations. Providing translation and protection mechanisms required to make such multiprocessors suitable for general-purpose use without compromising performance is a challenge.

This paper has proposed a multiprocessor architecture which integrates translation and protection with cache-coherent shared memory, user-level message passing and user-level DMA on messages. We have described techniques that retain direct, user-level access to the underlying communications network and hardware DMA mechanisms without compromising the integrity of the machine as a whole. Architectures such as FUGU bring us one step closer to the vision of scalable multiprocessors as general-purpose, multiuser machines.

References

[1] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. An extended version of this paper appears as MIT/LCS Memo TM-454, 1991.

[2] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An Evolutionary Processor Design for Multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[3] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 95–109, 1991.

[4] David L. Black, Richard F. Rashid, David B. Golub, Charles R. Hill, and Robert V. Baron. Translation Lookaside Buffer Consistency: A Software Approach. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems.*, pages 113–122. ACM, 1989.

[5] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings 21st Annual International Symposium on Computer Architecture (ISCA'94)*, pages 142–153, April 1994.

[6] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234. ACM, April 1991.

[7] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and Page Migration for Multiprocessor Compute Servers. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 12–24. ACM, October 1994.

[8] A. Cox and R. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 32–44, December 1989. Also as a Univ. Rochester TR-263, May 1989.

[9] William J. Dally et al. The J-Machine: A Fine-Grain Concurrent Computer. In *Proceedings of the IFIP (International Federation for Information Processing), 11th World Congress*, pages 1147–1153, New York, 1989. Elsevier Science Publishing.

[10] John Heinlein, Kourosh Gharachorloo, Scott Dresser, and Anoop Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 38–50. ACM, October 1994.

[11] James C. Hoe. Network Interface for Message-Passing Parallel Computation on a Workstation Cluster. In *Proceedings of Hot Interconnects II*, August 1994.

[12] Wilson C. Hsieh, Kirk L. Johnson, M. Frans Kaashoek, Deborah A. Wallach, and William E. Weihl. Efficient Implementation of High-Level Languages on User-Level Communication Architectures. MIT/LCS TR-616, MIT, Cambridge, MA, May 1994.

[13] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory; Early Experience. In *Principles and Practice of Parallel Programming (PPoPP) 1993*, pages 54–63, San Diego, CA, May 1993. ACM. Also as MIT/LCS TM-478, January 1993.

[14] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the International Supercomputing Conference (ISC) 1993*, Tokyo, Japan, July 1993. IEEE. Also as MIT/LCS TM-498, December 1992.

[15] 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 Annual International Symposium on Computer Architecture (ISCA) 1994*, Chicago, IL, April 1994. IEEE.

[16] T. J. LeBlanc, B. D. Marsh, and M. L. Scott. Memory Management for Large-Scale NUMA Multiprocessors. TR 311, Rochester, Rochester, NY, March 1989.

[17] Charles E. Leiserson, Aahil S. Abuhamdeh, and David C. Douglas et al. The Network Architecture of the Connection Machine CM-5. In *The Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, 1992.

[18] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 156–167. ACM, 1992.

[19] Randy Osborne. A Hybrid Deposit Model for Low Overhead Communication in High Speed LANs. Technical Report 94-02v3, MERL, 201 Broadway, Cambridge, MA 02139, June 1994.

[20] Gregory M. Papadopoulos, G. Andy Boughton, Robert Greiner, and Michael J. Beckerle. *T: Integrated Building Blocks for Parallel Computing. In *Supercomputing '93*, pages 624–635. IEEE, November 1993.

[21] Steve K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA) 1994*, pages 325–336, Chicago, IL, April 1994. IEEE.

[22] Richard P. LaRowe, Jr. and Carla Schlatter Ellis. Page Placement Policies for NUMA Multiprocessors. *Journal of Parallel and Distributed Computing*, 11:112–119, 1991.

[23] Bryan S. Rosenburg. Low-Synchronization Translation Lookaside Buffer Consistency in Large-Scale Shared-Memory Multiprocessors. *ACM Operating Systems Review*, 23(5):137–146, December 1989.

[24] Patricia Jane Teller. Translation-lookaside buffer consistency in highly-parallel shard-memory multiprocessors. RC 16858, IBM, IBM Thomas J. Watson Research Center, Distribution Services F-11 Stormytown, PO Bos 218, Yorktown Heights, NY 10598, May 1991. RC 16858 (no. 74685) 5/14/91.

[25] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Efficient Support for Multicomputing on ATM Networks. UW-CSE 93-04-03, University of Washington, Seattle, WA, April 1993.

[26] Andrew Tucker. Efficient Scheduling on Multiprogrammed Shared-Memory Multiprocessors. CSL-TR 94-601, Stanford, January 1994.

[27] John Wilkes. Hamlyn — An Interface for Sender-Based Communications. Department Technical Report HPL-OSR-92-13, HP Labs OS Research, November 1992.

Instruction		Description
`ldio`	`CMMU-Rs, Rd`	Load from CMMU register.
`stio`	`Rs, CMMU-Rd`	Store to CMMU register, including output queue.
`stdesc`	`Ra, Rl, CMMU-Dd`	Store to CMMU output queue.
`launch(i)`	`Rd`	Launch packet; this ldio returns the *head* pointer for the DMA channel.
`storeback(i)`	`Skip, Rd`	Discard/storeback input packet; ldio returns *head* pointer.
`uei`		Enable user traps.
`udi`		Disable user traps.

Table 3: Network instructions and operations

Trap	Event that it signals
user-network-input	Incoming message on the user network input port with a matching GID.
user-space-available	More than `space-request` words available at the user network output port.
user-DMA-done	`user-trans-tail` = `int-trans-id` for the user DMA channels. One each for input and output.
user-software-trap	Trap intended to signal input data available in a memory buffer.
mismatch-input	Incoming user message with a mismatched GID or system header on the user input port.
network-timeout	User network input queue stalled more than `timeout-count` cycles.
DMA-done	`trans-tail` = `int-trans-id` for each DMA channel. One per DMA channel for a total of four.
TLB-refill-fault	TLB miss.
nonlocal-fault	`stdesc` instruction referencing remote memory.
protection-fault	Reference to a protected page.

Table 4: Events associated the communication mechanisms in FUGU. The first four types correspond to the user traps.

Appendix A

The basic architecture for FUGU was shown in Figure 1 and places the network interface outside the main processor on the second-level cache bus. The CMMU is accessed using alternate-space loads and stores that bypass the on-chip cache. Operations that communicate with the CMMU are listed in Table 3 and CMMU registers are depicted in Figure 11. Events reported by the CMMU to the processor are tabulated in Table 4.

Several features not described in the body of the text are included here for completeness. The `space-available` register contains the count of doublewords available for writing in a network output queue. A *space-available* event is constructed from this count by adding a `space-request` count and signaling an event when there are more than `space-request` doublewords available. The *user-software-trap* event is intended to signal that network messages have been diverted to a memory buffer. An efficient message send/receive loop can be constructed by polling *space-request, user-network-input* and *user-software-trap* simultaneously.

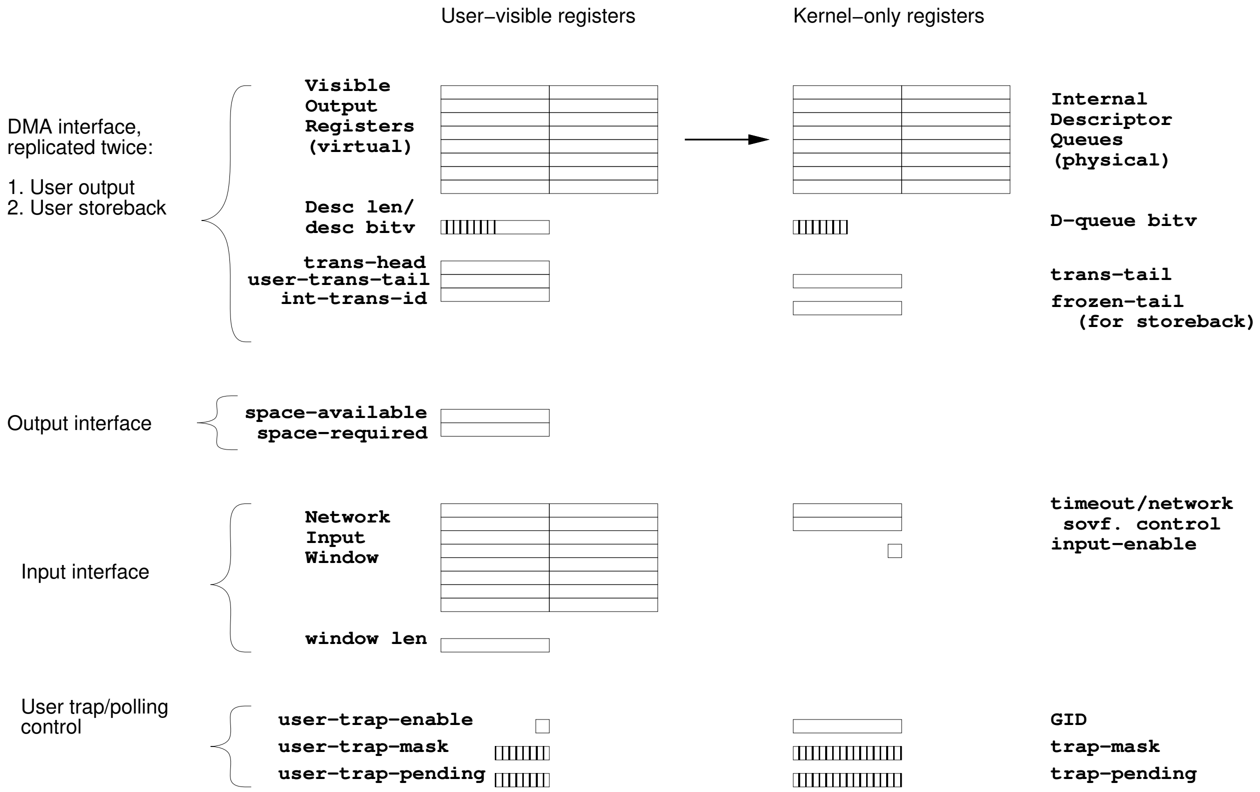

Figure 11: CMMU registers.