To appear in the *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, 1996

Coherent Network Interfaces for Fine-Grain Communication

Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood

Computer Sciences Department
University of Wisconsin-Madison
Madison, Wisconsin 53706-1685 USA
{shubu,babak,markhill,david}@cs.wisc.edu

Abstract

Historically, processor accesses to memory-mapped device registers have been marked uncachable to insure their visibility to the device. The ubiquity of snooping cache coherence, however, makes it possible for processors and devices to interact with cachable, coherent memory operations. Using coherence can improve performance by facilitating burst transfers of whole cache blocks and reducing control overheads (e.g., for polling).

This paper begins an exploration of network interfaces (NIs) that use coherence—coherent network interfaces (CNIs)—to improve communication performance. We restrict this study to NI/CNIs that reside on coherent memory or I/O buses, to NI/CNIs that are much simpler than processors, and to the performance of fine-grain messaging from user process to user process.

Our first contribution is to develop and optimize two mechanisms that CNIs use to communicate with processors. A cachable device register—derived from cachable control registers [39,40]—is a coherent, cachable block of memory used to transfer status, control, or data between a device and a processor. Cachable queues generalize cachable device registers from one cachable, coherent memory block to a contiguous region of cachable, coherent blocks managed as a circular queue.

Our second contribution is a taxonomy and comparison of four CNIs with a more conventional NI. Microbenchmark results show that CNIs can improve the round-trip latency and achievable bandwidth of a small 64-byte message by 37% and 125% respectively on the memory bus and 74% and 123% respectively on a coherent I/O bus. Experiments with five macrobenchmarks show that CNIs can improve the performance by 17-53% on the memory bus and 30-88% on the I/O bus.

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, NSF PYI Award CCR-9157366, NSF Grant MIP-9225097, an I.B.M. cooperative fellowship, and donations from A.T.&T. Bell Laboratories, Digital Equipment Corporation, Sun Microsystems, Thinking Machines Corporation, and Xerox Corporation. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the University of Wisconsin Graduate School. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

.

1 Introduction

Most current computer systems do not efficiently support fine-grain communication. Processors receive data from external devices, such as high-speed networks, through DMA and uncachable device registers. A processor becomes aware of an external event (e.g., a message arrival) via interrupts or by polling uncached status registers. Both notification mechanisms are costly: interrupts have high latency and polling wastes processor cycles and other system resources. A processor sends data with an uncachable store, a mechanism that is rarely given first-class support. Both uncachable loads and stores incur high overhead because they carry small amounts of data (e.g., 4-16 bytes), which fails to use the full transfer bandwidth between a processor and a device. Optimizations such as block copy [42] or special store buffers [42, 23] can help improve the performance of uncachable accesses by transferring data in chunks. However, these optimizations are processor-specific, may require new instructions [42, 23], and may be restricted in their use [42].

Snooping cache coherence mechanisms, on the other hand, are supported by almost all current processors and memory buses. These mechanisms allow a processor to quickly and efficiently obtain a cache block's worth of data (e.g., 32-128 bytes) from another processor or memory.

This paper explores leveraging the first-class support given to snooping cache coherence to improve communication between processors and network interfaces (NIs). NIs need attention, because progress in high-bandwidth, low-latency networks is rapidly making NIs a bottleneck. Rather than try to explore the entire NI design space here, we focus our efforts three ways:

- First, we concentrate on NIs that reside on memory or I/O buses. In contrast, other research has examined placing NIs in processor registers [5,15,21], in the level-one cache controller [1], and on the level-two cache bus [10]. Our NIs promise lower cost than the other alternatives, given the economics of current microprocessors and higher integration level we expect in the future. Nevertheless, closer integration is desirable if it can be made economically viable.
- Second, we limit ourselves to relatively simple NIs—similar in complexity to the Thinking Machines CM-5 NI [29] or a DMA engine. In contrast, other research has examined complex, powerful NIs that integrate an integer processor core [28, 38] to offer higher performance at higher cost. While both simple and complex NIs are interesting, we concentrate on simple NIs where coherence has not yet been fully exploited.
- Third, we focus on program-controlled fine-grain communication between peer user processes, as required by demanding parallel computing applications. This includes notifying the receiving process that data is available without requiring an interrupt. In contrast, DMA devices send larger messages to remote memory, and only optionally notify the receiving process with a relatively heavy-weight interrupt.

We explore a class of *coherent network interfaces* (CNIs) that reside on a processor node's memory or coherent I/O bus and participate in the cache coherence protocol. CNIs interact with a

coherent bus like Stanford DASH's RC/PCPU [30], but support messaging rather than distributed shared memory. CNIs communicate with the processor through two mechanisms: *cachable device registers* (CDRs) and *cachable queues* (CQs). A CDR—derived from cachable control registers [39, 40]—is a coherent, cachable block of memory used to transfer status, control, or data between a device and a processor. In the common case of unchanging information, e.g., polling, a CDR removes unnecessary bus traffic because repeated accesses hit in the cache. When changes do occur, CDRs use the underlying coherence protocol to transfer messages a full cache block at a time. Cachable queues (CQs) are a new mechanism that generalize CDRs from one cachable, coherent memory block to a contiguous region of cachable, coherent blocks managed as a circular queue to amortize control overheads. To maximize performance we exploit several critical optimizations: *lazy pointers*, *message valid bits*, and *sense-reverse*. Because CQs look, smell, and act like normal cachable memory, message send and receive overheads are extremely low: a cache miss plus several cache hits. Furthermore, if the system supports prefetching or an update-based coherence protocol, even the cache miss may be eliminated. Because CNIs transfer messages a cache block at a time, the sustainable bandwidth is much greater than conventional program-controlled NIs—such as the CM-5 NI [44]—that rely on slower uncachable loads and stores. For symmetric multiprocessors (SMPs), which are often limited by memory bus bandwidth, the reduced bus occupancy for accessing the network interface translates into better overall system performance.

An important advantage of CNIs is that they allow main memory to be the *home* for CQ entries. The home of a physical address is the I/O device or memory module that services requests to that address (when the address is not cached) and accepts the data on writebacks (e.g., due to cache replacements). Using main memory as a home for CQ entries offers several potential advantages. First, it decouples the logical and physical locations of network interface buffers. Logically, these buffers reside in main memory, a relatively plentiful resource that eases problems of naming, allocation, and deadlock. Physically, they can be located in processor or device caches to allow access at maximum speed. Second, it provides the same interface abstraction for local and remote communication. The sender cannot distinguish if the receiver is local, nor can the receiver tell. Third, it can exploit future processor and system optimizations, such as prefetching, replacement hints, or update protocols, that can further reduce the overheads of accessing NI registers or data buffers.

To expose the CNI design space, we develop a taxonomy reminiscent of Dir_iX [2]. We denote traditional network interface devices as NI_iX and coherent network interface devices as CNI_iX. The subscript *i* specifies the portion of an NI queue visible to the processor. The default unit of *i* is memory/cache blocks, but can also be specified in 4-byte words by adding the suffix 'w'. The placeholder X is either empty, Q, or Q_m. X empty represents the simple case where the NI exposes only part or whole of one message to the processor. As a result there are no explicit head or tail pointers to manage the NI queue. X = Q represents the more complex case where the exposed part of the NI queue is actually managed as a memory-based queue with explicit head and tail pointers. X = Q_m denotes that the home of the explicit memory-based NI queues is main memory.

We then evaluate four CNIs—CNI_4, $CNI_{16}Q$, $CNI_{512}Q$, and $CNI_{16}Q_m$—and compare them with NI_{2w}—an NI that uses uncached accesses to its data buffers and device registers, derived from the Thinking Machines CM-5 NI. We consider placing the NIs on both a coherent memory bus and a slower coherent I/O bus. Microbenchmark results show that compared to NI_{2w}, CNIs can improve the round-trip latency and achievable bandwidth of a small 64-byte message by 37% and 125% respectively on a memory bus and 74% and 123% respectively on a coherent I/O bus. Experiments with five macrobenchmarks show that a CNI can improve the performance by 17-53% on the memory bus and 30-88% on the I/O bus.

We see our paper having two main contributions. First, we develop cachable queues, including using lazy pointers, message valid bits, and sense-reverse. Second, we do the first micro- and macro-benchmarks comparison of alternative CNIs—exposed by our taxonomy—with a conventional NI.

A weakness of this paper, however, is that we do not do an in-depth comparison of our proposals with DMA. The magnitude of this deficiency depends on how important one expects DMA to be compared to fine-grain communication in future systems. Some argue that DMA will become more important as techniques like User-level DMA [3] reduce DMA initiation overheads. Others argue DMA will become less important as processors add block copy instructions [42] (making the breakeven size for DMA larger) and as the marginal cost of adding another processor diminishes [48] (making it less expensive to temporarily waste a processor).

The rest of this paper describes CDRs and CQs in detail (Section 2), presents CNI taxonomy and implementations, (Section 3), describes evaluation methodology (Section 4), analyzes results (Section 5), reviews related work (Section 6), and concludes (Section 7).

2 Coherent Network Interface Techniques

In this section, we describe two techniques for implementing CNIs: Cachable Device Registers (CDRs) and Cachable Queues (CQs). A CDR is a single coherent cache block used by a processor to communicate information to or from a CNI device. A CQ generalizes this concept into a contiguous region of coherent cache blocks. We describe the major issues in successfully exploiting CDRs and CQs. We describe their operation assuming write-allocate caches kept consistent by a MOESI write-invalidate coherence protocol [43].

2.1 Cachable Device Registers

Cachable Device Registers (CDRs) combine the traditional notion of memory-mapped device registers with the now-ubiquitous bus-based cache-coherence protocols supported by all major microprocessors. Reinhardt, et al., [39, 40] first proposed CDRs to communicate status information from a special-purpose hardware device to a processor. We extend their work to use coherence to efficiently communicate control information and data both to and from a network interface.

A CDR is a coherent, cachable memory block shared between a processor and a coherent network interface (CNI) device. The CNI sends information to the processor—i.e., to initiate a request or update status—by writing to the block. The CNI must first obtain write permission to the block in accordance with the underlying coherence protocol. The processor receives the information by polling the block. Unlike existing polling schemes, the CDR block is cachable, so in the common case of unchanging information, the processor's unsuccessful polls normally hit in the local cache.[1] Bus traffic only occurs when a device updates the information. Figure 1 illustrates this case under a write-invalidation based MOESI coherence protocol, assuming both the CNI and processor caches start with a read-only copy of

1. Cache conflicts can cause replacements, which affect performance but not correctness.

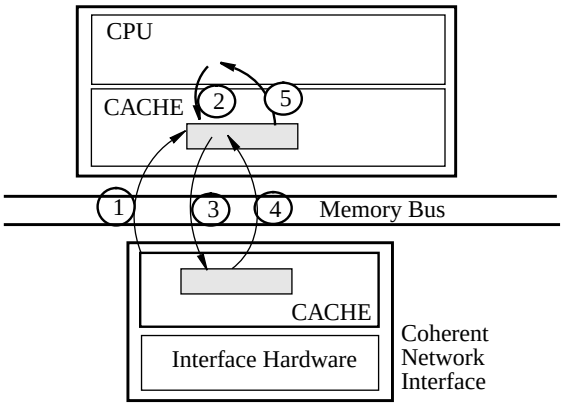

FIGURE 1. Coherent Network Interface with Cachable Device Register

the CDR. The CNI generates an invalidation to obtain write permission (arc 1), and the processor incurs a cache miss to fetch the CDR on its next poll attempt (arcs 2–5). Because a CDR consists of a whole cache block, an entire small message can be communicated between processor and CNI in a single bus transaction, amortizing the fixed overheads across multiple words.

A CDR can also transfer information from the processor to the device, in a logically symmetric way. Processor writes to the CDR are treated just like for a normal coherent cache block, obtained using the standard coherence mechanisms. The CNI device receives the information by reading the block, in a manner equivalent to polling. However, because the device observes the coherence protocol directly, it knows when the processor requests write permission to the block. Thus it need not poll periodically, but can read the block back immediately after the processor requests write permission. The device can provide a system programmable backoff interval to reduce the likelihood of "stealing" the block back before the processor completes its writes to the CDR. This technique, called *virtual polling*, is necessary because few processors can efficiently "push" data out of their caches. For processors (e.g., PowerPC [47]) that do support user-level cache flush instructions, the CDR can be directly flushed out of the cache.

CDRs allow a processor to efficiently transfer a full cache block (e.g., 32–128 bytes) of information to or from a CNI. For smaller amounts of data, e.g., a 4-byte word, CDRs are less efficient. For most processors, fetching a single word from an uncached device register takes roughly the same time as from a CDR; this is because the CNI responds with the requested word first which is then bypassed to the processor. However, the CDR still has higher overhead since it will displace another block from the cache, potentially causing a later miss. CDRs do even less well for small transfers to a device. Because most modern processors have store buffers, a single uncached store is more efficient than transferring that word via a CDR. For most processors and buses the breakeven point typically occurs at two or three double words. Hence, our CNI designs still use uncached stores to transfer single words of control information from the processor to the device.

For messages larger than a cache block, we require some method to reuse the CDR. For example, after the processor has read the first block of a message, it may want to read the second block using the same CDR. Conventional device registers often solve this problem using implicit *clear-on-read* semantics, where the register is cleared after an uncached read. For example, the CM-5 network interface treats the read of the hardware receive queue as an implicit "pop" operation. Clear-on-read works because processors guarantee the atomicity of individual load instructions; that is, the value returned by the device is guaranteed to be written to a register.

Clear-on-read does not work well for CDRs, since most processors do not provide the same atomicity guarantees for cache blocks. The load that causes the cache miss should be atomic (to close the "window of vulnerability" [27]); however, there are no guarantees for the remaining words in the block. Before subsequent loads complete, a cache conflict (e.g., resulting from an interrupt) could replace the block. With clear-on-read semantics, the remainder of the data in the CDR would be lost forever.

Instead, CDRs require an explicit clear operation by the receiver to enable reuse of the block. Under a MOESI protocol even this clear operation requires a slow three-cycle handshake between the processor and CNI to make sure that the processor sees new data when it re-reads the CDR. In the first step of this handshake, the processor issues an explicit clear operation by performing an uncached store to a traditional device register. In the second step, the processor must ensure that the CNI has seen the clear request. Since most modern processors employ store buffers, this step may incur additional stalls while a memory barrier instruction flushes the store out to the bus. When the CNI observes the explicit clear operation, it invalidates the CDR by arbitrating for and acquiring the memory bus. The third step of the handshake is for the processor to ensure that the invalidation has completed. It does this by reading, potentially repeatedly, a traditional uncached device status register.[1] Consequently, while CDRs efficiently transfer a single block of information, they perform much less well for multiple blocks. We address this problem by introducing cachable queues.

2.2 Cachable Queues

Cachable Queues (CQs) generalize the concept of CDRs from one coherent memory block to a contiguous region of coherent memory blocks managed as a queue. CQs are a general mechanism that can be used to communicate messages between two processor caches or a processor cache and a device cache. A key advantage of CQs is that they simplify the reuse handshake and amortize its overhead over the entire queue of blocks. Liu and Culler [31] used cachable queues to communicate small messages and control information between the compute processor and message processor in the Intel Paragon. We show how CQs can be used to communicate directly between a processor and a network interface device. We first describe the basic queue operation, and then introduce three important performance optimizations.

Cachable queues follow the familiar enqueue-dequeue abstraction and employ the usual array implementation, illustrated in Figure 2. The head pointer (`head`) identifies the next item to be dequeued, and the tail pointer (`tail`) identifies the next free entry. The queue is empty if `head` and `tail` are equal, and full if `tail` is one item[2] less than `head` (modulo queue size). If there is a single sender and single receiver for this queue, the case we consider in this paper, then no locking is required since only the sender updates `tail` and only the receiver updates `head`[3].

A processor sends a message by simply enqueuing it in the appropriate out-bound message queue. That is, it first checks for overflow, then writes the message to the next free queue location and increments `tail`, relying on the underlying coherence proto-

1. A somewhat more efficient handshake is possible if the processor provides a user-accessible cache-invalidate operation. Issue clear operation, flush store buffer, and invalidate cache entry.

2. We assume fixed size network messages in this paper, but CQs can be generalized to variable length messages in a straight-forward manner.

3. Memory barrier operations may be necessary to preserve ordering under weaker memory models.

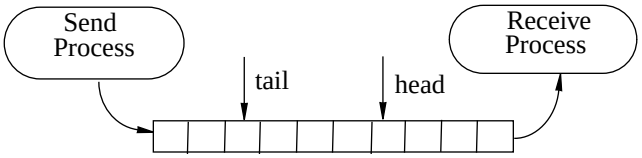

FIGURE 2. Local Cachable Queue

col to bring the block(s) local to the cache. A processor receives a message by first checking for an empty queue, then reading the queue entry pointed to by the `head`. The message remains in the queue until the receiver explicitly increments `head`. The head and tail pointers reside in separate cache blocks.

Because CQs are simply memory, they have the property that the message sender and receiver have the same interface abstraction whether the other end is local or remote. A local CQ, illustrated in Figure 2, is simply a conventional circular queue between two processors. A remote CQ consists of two local CQs, each between a processor and CNI device, as illustrated in Figure 3. The head and tail pointers are also managed as cachable memory. A CNI that uses CQs simply acts like another processor manipulating the queue.

The head and tail pointers of the CQs are a much simpler way to manage reuse than the complex handshake required by CDRs. If there is room in the CQ, then the tail entry can be reused; if the CQ is non-empty, then the head entry is valid. However, even though no locking is required to access the head and tail pointers, a straight-forward implementation induces significant communication between sender and receiver. This occurs because the sender must check (i.e., read) the head pointer, to detect a full queue, and the receiver must check the tail pointer, to detect an empty queue. Because the queue pointers are kept in coherent memory, cache blocks may ping-pong with each check.

We can greatly reduce this overhead using three techniques: *lazy pointers*, *message valid bits*, and *sense reverse*. Lazy pointers exploit the observation that the sender need not know exactly how much room is left in the queue, but only whether there is enough room. The sender maintains a (potentially stale) copy of the head pointer, `shadow_head`, which it checks before each send. `Shadow_head` is conservative, so if it indicates there is enough room, then there is. Only when `shadow_head` indicates a full queue does the sender read `head` and update `shadow_head`. If the queue is no more than half full on average, then the sender needs to check `head`—and incur a cache miss—only twice each time around the array.

Lazy pointers work much less well for the tail pointer. The receiver must check `tail` on every poll attempt, to see if the queue is empty. Whenever a message arrives, the receiver's cached copy of `tail` gets invalidated. Thus in the worst case, each message arrival causes a cache miss on `tail`. Instead, we use message valid bits—stored either as a single bit in the message header or in a separate word—to allow the receiver to detect message arrivals without *ever* checking the tail pointer [10, 31]. The valid bits indicate whether a cache block contains a valid message, or not. On a poll attempt, the receiver simply examines the first message in the queue (i.e., the one pointed to by `head`); if it's invalid, the queue is empty. Thus no bus traffic normally occurs in this case. When a valid message is written to the queue, the sender will invalidate the receiver's cached copy, causing a cache miss when the receiver polls again. To complete the handshake, the receiver must clear the message valid bit when it advances `head`.

Clearing the message valid bit requires the receiver to write the queue entry; thus under a MOESI protocol, the receiver becomes owner of the queue entry's cache block, rather than simply having a shared copy. This normally requires an additional bus transaction. We can avoid this transaction (and avoid clearing the valid bit) using a technique known as *sense reverse*. The key idea is to alternate the encoding of the valid bit on each pass through the queue. Valid is encoded as 1 on odd passes, and encoded as 0 on even passes. The sender and receiver both have an additional state bit, stored in the same cache blocks as their respective pointers, indicating the sense of their current pass. Figures 4 and 5 present pseudo-code for the simple case where the valid bit is stored in a separate word in the header. The sender first checks if the CQ has space and then writes the message followed by its current sense as the message valid bit. The receiver compares its current sense to the valid bit in the message, with a match indicating a valid message. Sense reverse has been previously used for barriers [34] and asynchronous logic, but to our knowledge has never been used for messaging.

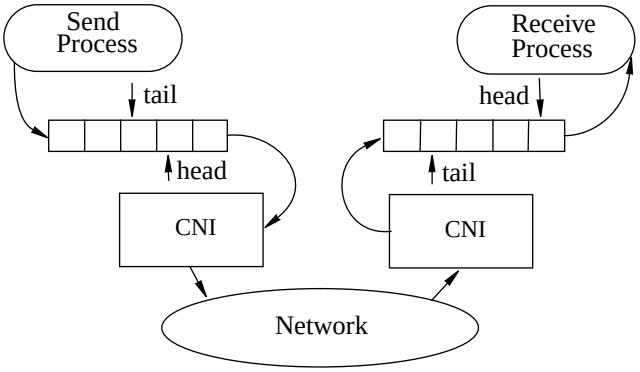

FIGURE 3. Remote Cachable Queue

Combining all three optimizations minimizes the bus traffic required by CQs. Under a write-invalidation based MOESI protocol, each block of a message requires one invalidation, to obtain write permission for the sender, and one read miss, to fetch the block for the receiver. The head pointer requires only two invalidation-miss pairs for each pass around the circular queue, assuming the queues are no more than half full on average. The tail pointer is private to the sender and generates no coherence actions.

2.3 Home for CQ entries

In most computer systems, all legal physical addresses map to a *home* device or memory module. If a block is cachable, for example, then the home is where data are written on cache replacement. Should the home for CDRs or CQ entries be at the CNI, as with a regular device register, or in main memory?

Since CDRs are each a single block and most devices will employ only a few, the logical choice is to provide the home within the device itself. This can also simplify the implementation for some memory buses, because the device may not have to implement all cases in the coherence protocol [36].

CQs, on the other hand, will benefit from being large. For example, Brewer, et al., have demonstrated that remote queues can significantly improve performance by preventing contention on the network fabric [6]. If the CQ's home is main memory—a less precious resource than hardware FIFOs—then its capacity is essentially infinite. Large queues help simplify protocol deadlock avoidance, at least for moderate-scale parallel machines. Having the CQ home in memory also helps tolerate unreliable network fabrics, since messages need not be removed from the send queue until delivery is confirmed.

To place the CQ home in main memory, we must address three operating system issues. First, a CNI needs a translation scheme to translate the CQ virtual addresses to physical addresses in main memory. In this paper, we assume that the operating system allo-

```
if (tail == head &&
      sender's sense != receiver's sense)
{
   Queue is full, stall or return error
}
else
{
   enqueue message at tail + 1;
   *tail = sender's sense;
   advance tail modulo CQ size;
   if (tail == 0)
   {
      /* reverse the sense */
      sender's sense = sender's sense xor 1;
   }
}
```

FIGURE 4. Pseudo code for enqueue with sense reverse

cates CQ pages contiguously, allowing CNIs to use a simple base-and-bounds virtual-to-physical address translation. If the operating system cannot guarantee this, then a more complicated translation mechanism may be necessary [19]. Second, a CNI must ensure that CQ pages always reside in main memory, or be prepared to fetch them from the swap device. For our implementations we assume that CQ pages are "pinned," so that the operating system does not attempt to page them out. Alternatively, we could adopt a more flexible scheme [3, 19] at the expense of a more general address translation mechanism (e.g., a TLB). Finally, there must be some mechanism for the rare case in which even the large amount of memory allocated for a CQ fills up. The simplest option is to block the sender; however, this may lead to deadlock. Alternatively, as proposed for MIT Fugu [32], the CNI device can interrupt the processor, causing it to allocate free virtual memory frames and drain the CQ.

Making main memory the home is generally infeasible with a coherent I/O bus. Coherent I/O buses [18] allow memory residing on the I/O bus to be cached by the processor. However, they do not allow an I/O device to coherently cache data from the regular processor memory space. It is difficult to change this in the near future because the speed mismatch makes it hard for the I/O devices to respond to memory bus snoop requests in a timely fashion.

2.4 Multiprogramming

The demands of multiprogramming require additional support from a network interface. In traditional networking, an NI is a single shared resource virtualized by the operating system. For example, in TCP/IP the operating system multiplexes the hardware device to send and receive network messages to and from many processes. Unfortunately, the operating system's overheads severely limit performance, especially for small messages.

Many multicomputers reduce or eliminate this overhead by mapping the NI directly into the user's address space [1,15,29]. Thus the operating system normally need not get involved when messages are sent and received. However, user-mapped NIs significantly complicate support for multiprogramming. Possible solutions range from disallowing multiprogramming [15], to taking special actions at context switch time (to context switch the NI and network state) [44], to optimistically assuming a message is destined for the current process (reverting to operating system buffering if it is not) [32]. Furthermore, these solutions do not easily generalize to symmetric multiprocessing (SMP) nodes, where multiple processes may concurrently access the NI.

CNIs vastly simplify the multiprogramming problem by using memory-based queues as the interface abstraction, rather than memory-mapped device registers. Each process communicates

```
if (*head != receiver's sense)
{
    Queue is empty, stall or return null
}
else
{
   dequeue message at head + 1;
   advance head modulo CQ size;
   if (head == 0)
   {
      /* reverse the sense */
      receiver's sense = receiver's sense xor 1
   }
}
```

FIGURE 5. Pseudo code for dequeue with sense reverse

with a CNI using its *own* data queues and treats them like regular cachable memory. The CNI device need only maintain a small amount of state for each queue, e.g., base and bound translations, head and tail pointers, and a process identifier. The process identifier is used to ensure that messages sent from one node are placed on the appropriate queue on the receiving node. Since this state is typically much smaller than the data queues, a CNI can support more active processes than a memory-mapped NI with comparable hardware. If there are more active queues than the CNI state can support, two options have potential. First, the operating system can unmap additional queues and accept faults when they are accessed. Second, the operating system can allocate a memory-based data structure that CNI hardware can use to find the state for all active queues (like page tables for a TLB fill).

3 CNI Taxonomy and Implementations

This section proposes a taxonomy of network interfaces (NIs) and describes the implementation of five NIs that we evaluated in this paper. We use the NI queue structure as the main component to enumerate a taxonomy of network interfaces. NI queues are the primary carriers of messages between a processor and its NI. A processor sends messages to the NI through the *send queue* and receives messages from the NI through the *receive queue*. For our taxonomy of CNIs we assume that both the NI queues have the same structure.

Our taxonomy is modelled after Agarwal et al.'s classification of directory protocols [2]. We use the notation NI_iX for traditional NIs and CNI_iX for coherent network interfaces that cache the NI queues. The subscript *i* denotes the size of the NI queue exposed to the processor. The default unit of *i* is memory/cache blocks, but can also be specified in 4-byte words by adding the suffix 'w'. The placeholder X could either be empty, Q, or Q_m. X empty represents the simple case where a network interface exposes only part or whole of one network message. For CNIs a network message is exposed via CDRs. CDR reuse is managed by the explicit handshake described in Section 2.1. X = Q represents the more complex case where the exposed portion of the NI queue is managed as a memory queue with explicit head and tail pointers. X = Q_m denotes that the home of the explicit memory-based NI queues are in main memory. The absence of an '*m*' implies that the device serves as the home for the NI queues.

Several existing NIs can be classified with this taxonomy. The Thinking Machines' CM-5 [44] NI is NI_{2w} since it exposes two words of a message to the receiver. Similarly, the Alewife [1] NI is NI_{16w} [26]. The network interface in *T-NG [10], which devotes 8 KB for an NI queue and consists of 64-byte cache blocks, is $NI_{128}Q$.

We examine five different network interface devices in this paper, summarized in Table 1, to send 256-byte network messages. The first is a conventional network interface modelled after the

NI/CNI	Exposed Queue Size	Queue Pointers	Home
NI_{2w}	2 words		
CNI_4	4 cache blocks		device
$CNI_{16}Q$	16 cache blocks	explicit	device
$CNI_{512}Q$	512 cache blocks	explicit	device
$CNI_{16}Q_m$	16 cache blocks	explicit	main memory

TABLE 1. Summary of Network Interface Devices

Thinking Machines CM-5 NI. Messages are sent by first checking an uncachable status register, to ensure there is room to inject the message, then the message is written to an uncachable device register backed by a hardware queue. Similarly, receives check an uncached status register to see if a message is available, then read the message from an uncachable device register. Because all accesses to the NI queues are non-cachable, and two four-byte words of the message are exposed, we classify this device as NI_{2w}.

The second NI extends this baseline device by using four CDRs to expose a 256-byte network message. This device, denoted by CNI_4, exploits the memory bus's block transfer capability to move a message between the processor and the device. However, the status and control registers are uncached. After receiving a message, the processor issues an uncached store to explicitly pop the message off the queue. By always checking the uncached status register—which does not indicate message ready again until the cached copy has been invalidated—the processor and CNI_4 device perform a three-cycle handshake that prevents false hits.

The three-cycle handshake limits the bandwidth achievable by CNI_4. The third and fourth NIs solve this problem by employing CQs for message data and regular memory for control and status information (head and tail pointers). $CNI_{16}Q$ and $CNI_{512}Q$ cache up to 16 and 512 blocks, respectively. The memory that backs up the caches resides on the devices themselves. The larger capacity of $CNI_{512}Q$ reduces the number of flow control stalls, increasing performance for applications with many messages in flight.

Sending messages to a CNI_iQ device involves three steps: checking for space in the CQ, writing the message, and incrementing the tail pointer. The send is further optimized by sending a message ready signal to the CNI device through an uncached store. As discussed in Section 2.1, uncached stores are more efficient than cache block operations for small control operations. Hence for the send queue, the CNI device does not use virtual polling. Instead, the CNI_iQ uses the message ready signal to keep a count of pending messages. This count is incremented on each message ready signal and decremented when the device injects a message into the network. As long as this counter is greater than zero, the CNI_iQ device pulls messages out of the processor cache (unless the blocks have already been flushed to their home in the device) and increments the head pointer. On the receive side, the processor polls the head of the queue, reads the message when valid, then increments the head pointer. Both sender and receiver toggle their sense bits when they wrap-around the end of the CQ.

The last device, $CNI_{16}Q_m$, caches up to 16 cache blocks on the network interface device, and overflows to main memory as necessary. The total size of the memory-based queue is 512 cache/memory blocks. Having main memory as home for the CQ simplifies software flow control. Specifically, for the other NIs, whenever the sender cannot inject a message it must explicitly extract any incoming messages and buffer them in memory [6]. Conversely, the $CNI_{16}Q_m$ does this buffering automatically when the CNI cache cannot contain all the messages. The $CNI_{16}Q_m$ taxonomy allows for memory overflow to occur on both the sending and receiving CNIs. However, for simplicity, this paper only examines memory buffering at the receiver.

Operation	Cache Bus	Memory Bus	I/O Bus
Uncached 8-byte load from NI	4	28	48
Uncached 8-byte store to NI	4	12	32
Cache-to-cache transfer from CNI to processor (64 bytes)		42	76
Cache-to-cache-transfer from processor to CNI (64 bytes)		42	62
Memory-to-cache transfer (64 bytes)		42	

TABLE 2. Bus Occupancy for Network Interface and Memory Access in Processor Cycles

The CNI devices implement a variant of virtual polling to minimize the number of bus transactions on the critical path. Specifically, under the bus's write-invalidation based MOESI protocol, the processor must generate an invalidation signal to acquire ownership of a cache block before it can write to it. Since our CQs are filled in FIFO order, an invalidation signal for all blocks other than the first block of a multi-block message implies that the processor is done writing the previous cache block. When the CNI device detects an invalidation signal it issues a coherent read on the previous cache block of the same message. Thus part of the message is transferred to the CNI cache before the processor has completed writing all the cache blocks of the message.

4 Methodology

This section describes the system assumptions and benchmarks used to evaluate the five network interface designs. Section 5 presents results from the evaluation.

4.1 System Assumptions

Our simulations model a parallel machine with 16 nodes, each with a 200 MHz dual-issue SPARC processor modelled after the ROSS HyperSPARC, 100 MHz multiplexed, coherent memory bus, 50 MHz multiplexed, coherent I/O bus, and a network interface (NI_{2w} or one of the four CNI_iXs). Both buses support only one outstanding transaction. The memory bus's coherence protocol is modelled after the MBus Level-2 coherence protocol [24]. Coherence on the I/O bus resembles that of the coherent extension to PCI [18]. An I/O bridge connects the memory and I/O buses. The bridge buffers writes and coherent invalidations, but blocks on reads. When transactions are simultaneously initiated on the two buses, the I/O bridge NACKs the I/O bus transaction to prevent deadlock. Fairness is preserved by ensuring that the next I/O bus transaction succeeds.

The single-level processor cache is 256 KB direct-mapped, with duplicated tags to facilitate snooping and 64-byte address and transfer blocks. The CNI caches are also direct-mapped with 64-byte address and transfer blocks. The CNI cache sizes vary according to the subscript *i* in the CNI_iX nomenclature. Table 2 shows the bus occupancy for our network interface and memory accesses in processor cycles. Since the I/O bus is connected to the processor via the memory bus, the bus occupancy numbers for the I/O bus includes the corresponding memory bus occupancy cycles.

Network topology is ignored and network message size is fixed at 256 bytes. All messages take 100 processor cycles to traverse the network from injection of the last byte at the source to arrival of the first at the destination. We model hardware flow control at the end points using a hardware sliding window protocol. A processor can send up to four network messages per destination before it blocks waiting for acknowledgments. To avoid dead-

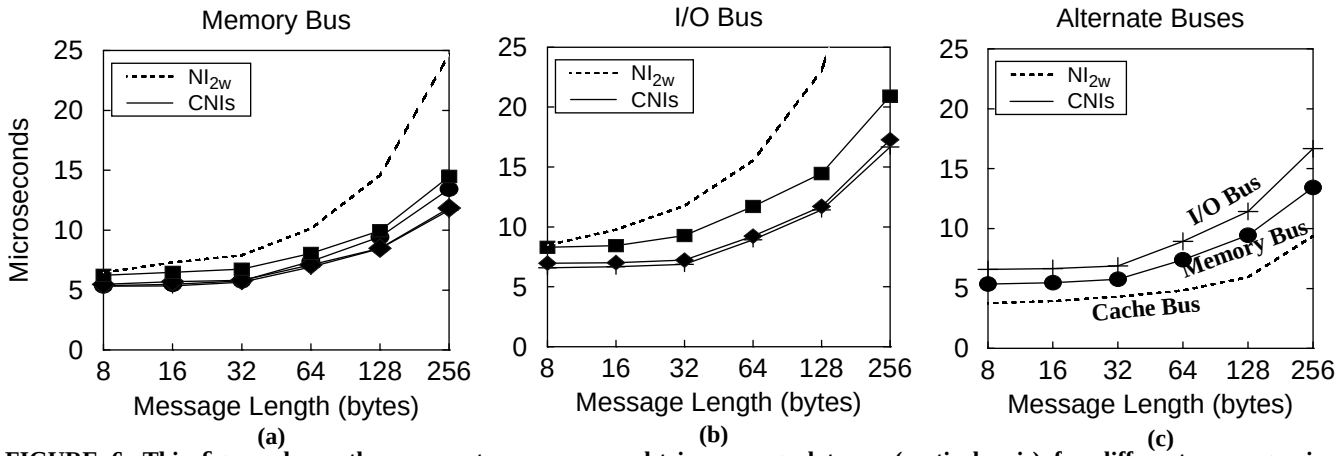

FIGURE 6. This figure shows the *process-to-process* round-trip message latency (vertical axis) for different message sizes (horizontal axis). (a) shows the round-trip message latency for NI_{2w}, CNI_4, $CNI_{16}Q$, $CNI_{512}Q$, and $CNI_{16}Q_m$ on the memory bus. (b) shows the same (except $CNI_{16}Q_m$) on the I/O bus. (c) compares $CNI_{512}Q$, $CNI_{16}Q_m$, and NI_{2w} on the I/O, memory, and cache buses respectively. [■ = CNI_4, ◆ = $CNI_{16}Q$, + = $CNI_{512}Q$, ● = $CNI_{16}Q_m$]

locks, if a processor blocks on a message send, it reads messages from the NI and buffers them in user space (except for $CNI_{16}Q_m$ in which messages automatically overflow to main memory from the CNI cache).

All benchmarks are run on the Tempest parallel programming interface [22]. Message-passing benchmarks use only Tempest's active messages [37]. Shared-memory codes on Tempest also use active messages, but assume hardware support for fine-grain access control [39]. Codes with custom protocols use a combination of the two.

4.2 Macrobenchmarks

Table 3 depicts five macrobenchmarks used in this study. Spsolve [12] is a very fine-grained iterative sparse-matrix solver in which active messages propagate down the edges of a directed acyclic graph (DAG). All computation happen at nodes of the DAG within active message handlers. The messaging overhead is critical because each active message carries only a 12 byte payload and the total computation per message is only one double-word addition. Several active messages can be in flight, which can create bursty traffic patterns.

Gauss is a message-passing benchmark that solves a linear system of equations using Gaussian elimination [9]. The key communication pattern is a one-to-all broadcast of a pivot row (two kilobytes for our matrix size).

Em3d models three-dimensional electromagnetic wave propagation [13]. It iterates over a bipartite graph consisting of directed edges between nodes. Each node sends two integers to its neighboring nodes through a custom update protocol [16]. Several update messages (with 12 byte payload) can be in flight, which like spsolve, can create bursty traffic patterns.

Benchmark	Key Communication	Input Data Set
spsolve	Fine-Grain Messages	3720 elements
gauss	One-To-All Broadcast	512x512 matrix
em3d	Fine-Grain Messages	1K nodes, degree 5, 10% remote, span 6, 10 iter
moldyn	Bulk Reduction	2048 particles, 30 iter
appbt	Near neighbor	24x24x24 cubes, 4 iter

TABLE 3. Summary of macrobenchmarks

Moldyn is a molecular dynamics application, whose computational structure resembles the non-bonded force calculation in CHARMM [7]. The main communication occurs in a custom bulk reduction protocol [35], which constitutes roughly 40% of the application's total time with NI_{2w} as the network interface. One execution of the reduction protocol iterates as many times as there are processors. In each of these iterations, a processor sends 1.5 kilobytes of data to the same neighboring processor through Tempest's virtual channels.

Appbt is a parallel three-dimensional computational fluid dynamics application [8] from the NAS benchmark suite. It consists of a cube divided into subcubes among processors. Communication occurs between neighboring processors along the boundaries of the subcubes through Tempest's default invalidation-based shared memory protocol [38].

5 Results

This section examines the network interfaces' performance with two microbenchmarks and five macrobenchmarks. On the memory bus we simulated all four CNIs plus NI_{2w}. On the I/O bus we simulated all but $CNI_{16}Q_m$, since $CNI_{16}Q_m$ cannot be implemented with current coherent I/O buses (Section 2.3). Since coherence is usually not an option on cache buses, we did not simulate CNIs there. For each microbenchmark and macrobenchmark we compare the performance of NI_{2w} on the cache bus with the best of the CNI alternatives—$CNI_{16}Q_m$ on the memory bus and $CNI_{512}Q$ on the I/O bus. Since NI_{2w} on the cache bus is closest to the processor, it provides a rough upper bound to the performance achievable with different coherent network interfaces.[1]

5.1 Microbenchmarks

This section examines *process-to-process* round-trip message latency (Figure 6) and bandwidth (Figure 7) for our five network interface implementations.[2] These numbers include the messaging layer overhead for copying a message from the network interface

1. Raw data at URL `http://www.cs.wisc.edu/~wwt/cni96`.

2. Each message is broken into one or more 256-byte network message(s). Each network message has a 12-byte overhead for header information. The message sizes in both Figure 6 and Figure 7 do not include this header.

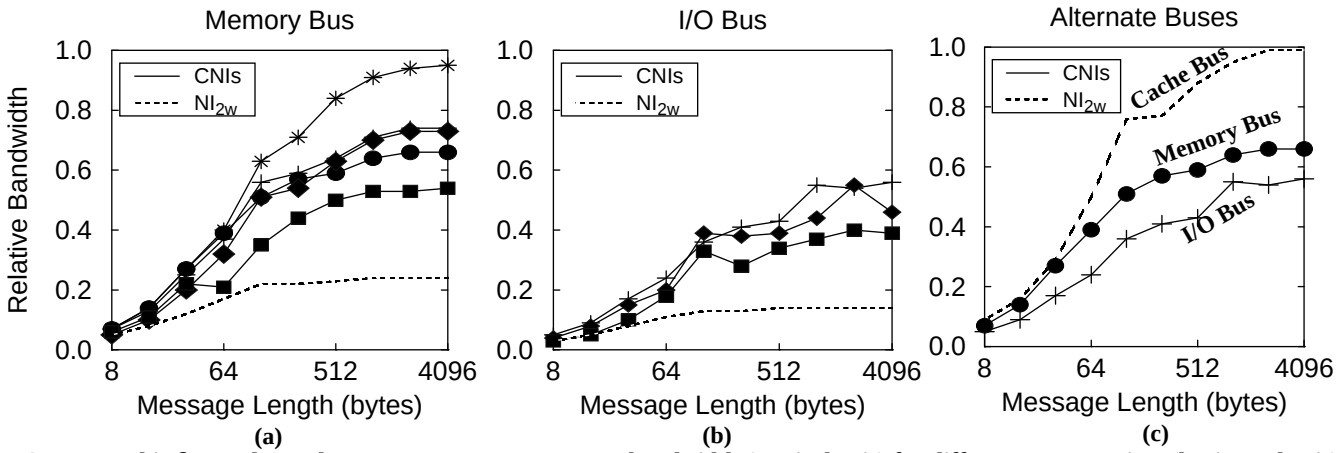

FIGURE 7. This figure shows the *process-to-process* message bandwidth (vertical axis) for different message sizes (horizontal axis). The message bandwidth is expressed as a fraction of the maximum bandwidth two processors on the same coherent memory bus can sustain using a local memory queue (Figure 2). (a) shows the message bandwidth for NI_{2w}, CNI_4, $CNI_{16}Q$, $CNI_{512}Q$, and $CNI_{16}Q_m$ (with and without snarfing) on the memory bus. (b) shows the same (except $CNI_{16}Q_m$) on the I/O bus. (c) compares $CNI_{512}Q$, $CNI_{16}Q_m$, and NI_{2w} on the I/O, memory, and cache buses respectively. [■ = CNI_4, ◆ = $CNI_{16}Q$, + = $CNI_{512}Q$, ● = $CNI_{16}Q_m$, ✻ = $CNI_{16}Q_m$ with snarfing]

to a user-level buffer, and vice versa. Thus data begins in the sending processor's cache and ends in the receiving processor's cache, rather than simply moving from memory to memory.

5.1.1 Round-Trip Latency

Figure 6 shows the round-trip latency of a message for each of NI_{2w} and the four CNI_iXs. It shows two important results. First, CNIs reduce messaging overheads significantly. For small messages, between 8 and 256 bytes, $CNI_{16}Q_m$ is 20-84% better than NI_{2w} on the memory bus (Figure 6a) and $CNI_{512}Q$ is 29-141% better than NI_{2w} on the I/O bus (Figure 6b). Second, $CNI_{16}Q_m$ on the memory bus increases the latency over NI_{2w} on the cache bus by only 43% (Figure 6c). This is significant, because the CNIs do not require modifications to the processor or processor board.

The four CNIs have similar latencies with minor variations among them. CNI_4 performs worst because it polls an uncached status register and must use the expensive three-cycle handshake to invalidate the previous message from the processor cache. $CNI_{16}Q$ and $CNI_{512}Q$ consistently have the lowest latency due to efficiently polling the cached message valid bit and by using explicit head and tail pointers to amortize the reuse overhead across the entire queue of messages. $CNI_{16}Q_m$'s latency is slightly worse (on the memory bus) because when its cache overflows, it must flush (i.e., writeback) messages to main memory. A better replacement policy and/or a writeback buffer can help take these flushes off the critical path. However, as we will see later with the macrobenchmarks, $CNI_{16}Q_m$ consistently outperforms the other three CNIs on the memory bus due to its ability to overflow messages to main memory instead of backing up the network.

5.1.2 Bandwidth

Figure 7 graphs the bandwidth provided by the five network interfaces. The vertical axes are normalized to the maximum bandwidth two processors on the same coherent memory bus can sustain while transfering data from one cache to the other. For our simulation parameters (Table 2), this bandwidth is 144 MB/s. This is the maximum bandwidth the four CNIs can hope to achieve with our simulation parameters.

Figure 7 shows two interesting results. First, CNIs improve the bandwidth over NI_{2w} significantly, *even for very small messages*. On the memory bus, $CNI_{16}Q_m$ is 59-169% better than NI_{2w} for 8-4096 byte messages (Figure 7a). For the same message sizes, $CNI_{512}Q$ is 51-287% better than NI_{2w} on the I/O bus (Figure 7b). Second, NI_{2w}'s bandwidth on the cache bus is only 50% more than $CNI_{16}Q_m$'s on the memory bus (Figure 7c).

As in the round-trip microbenchmark, all four CNIs have similar bandwidth with minor variations among them. CNI_4 performs worst of the four CNIs because of its high overhead for polling uncached registers and the three-cycle handshake in the critical path of message reception. CNI_4 shows two different knees on the memory and I/O buses respectively. The knee on the memory bus appears when a message crosses the first cache block boundary and writes to the second cache block. The second cache block is partially empty resulting in wasted work by CNI_4, which must still read the entire block. Since CNI_4 reuses CDRs, the processor must wait for CNI_4 to complete the entire read (and the three-cycle handshake) before it can write another message. The CQ-based CNIs do not have this problem because instead of blocking, a processor simply writes to the next queue location. This same knee does not show up on the I/O bus because the higher I/O bus access latencies dominate over the pipelined transfer time for the cache block. On the I/O bus a different knee appears when CNI_4 saturates the I/O bus.

$CNI_{16}Q$ and $CNI_{512}Q$ perform the best due to their low poll overhead and ability to cache multiple messages (a network message fits in four cache blocks). However, when the message size reaches two kilobytes, $CNI_{16}Q$'s performance on the I/O bus dips slightly. This is because the small queue size forces frequent updates to the shadow head pointer on the receive queue, which in turn creates contention at the I/O bridge. $CNI_{512}Q$ does not exhibit this problem because its larger queue requires less frequent updates to the shadow head.

$CNI_{16}Q_m$ achieves slightly lower bandwidth than $CNI_{512}Q$. This is because the message send rate is significantly higher than the message reception rate, causing the receiving $CNI_{16}Q_m$'s cache to overflow. The resulting writebacks to main memory induce moderate bus contention which decreases the maximum communication bandwidth. Unfortunately, because the problem is bandwidth not latency, a writeback buffer will not help with this microbenchmark as it would for the round-trip microbenchmark.

However, an alternative technique, called *data snarfing* [17, 14], can potentially improve both latency and bandwidth. In data snarfing, a cache controller reads data in from the bus whenever it has a tag match (i.e., space allocated) for a block in the invalid state. Thus in our microbenchmark, the processor cache on the

receive side simply snarfs in the cache blocks that $CNI_{16}Q_m$ writes back to memory. This eliminates many of the invalidation misses on the receive cachable queue and improves the bandwidth by as much as 45% (Figure 7a). We also expect that an update-based coherence protocol would have similar behavior. However, while data snarfing significantly improves microbenchmark performance, we found it had little effect on macrobenchmark performance and do not examine it further.

The absolute bandwidth offered by CNIs can improve significantly with a more aggressive system. With our simulation parameters—200 MHz processor, 100 MHz memory bus, 64 byte cache blocks, and 230 ns cache-to-cache transfer—the maximum bandwidth achieved by $CNI_{512}Q$ on the memory bus is 107 MB/s. This represents over 73% of the bandwidth achievable between two processors on the same coherent memory bus. More aggressive system assumptions, such as non-blocking caches, bigger cache blocks, prefetch instructions, support for update protocols, and a pipelined or packet-switched bus, would significantly improve this absolute performance.

5.2 Macrobenchmarks

Figure 8 shows the performance gains from CNIs for the five macrobenchmarks described in Section 4.2.

CNI_4, $CNI_{16}Q$, $CNI_{512}Q$, and $CNI_{16}Q_m$ offer a progression of incremental benefits over NI_{2w}. Unlike NI_{2w}, which can only be accessed through uncached loads and stores, CNI_4 effectively utilizes the memory bus's high-bandwidth block transfer mechanism by transferring messages in full cache block units. $CNI_{16}Q$ and $CNI_{512}Q$ further reduce overhead by amortizing the three-cycle handshake over an entire queue of messages. The larger capacity of the CQs also helps prevent bursty traffic from backing up into the network. $CNI_{16}Q_m$ further simplifies software flow control in the messaging layer by allowing messages to smoothly overflow to main memory when the device cache fills. This avoids processor intervention for message buffering, which, otherwise, could significantly degrade performance [25].

Block Transfer. The increase in bandwidth obtained by transferring messages in whole cache block units has a major impact on performance. *Gauss* and *moldyn* do bulk transfers and *appbt* communicates with moderately large (128-byte) shared-memory blocks. *Gauss* performs a one-to-all broadcast of a 2KB row, while *moldyn's* reduction protocol transfers 1.5KB of data between neighboring processors. CNI_4 improves *gauss*'s performance by 39% and 46%, *moldyn*'s performance by 42% and 20%, and *appbt*'s performance by 10% and 11% on the memory and I/O buses respectively. Even for *spsolve* and *em3d* that send small messages (12-byte payload), CNI_4's performance improvement over NI_{2w} is significant (between 13-21%).

CNI_4's performance improvement for *moldyn* on the I/O bus is not as high as on the memory bus because of contention at the I/O bridge. The NI_{2w} device never tries to acquire the memory or I/O bus because it is always a bus slave. However, the CNI_4 cache competes with the processor cache to acquire the memory and I/O buses. Simultaneous bus acquisition requests at the I/O bridge from the processor cache and CNI_4 cache creates contention. This effect is severe in *moldyn* because message sends, message receives, and polls on uncached device registers are partially overlapped in *moldyn*'s bulk reduction phase. Thus, the memory bus occupancy for a system with CNI_4 on the I/O bus compared to a system with NI_{2w} on the I/O bus decreases by 41% in *gauss*, but by only 15% in *moldyn*.

Overall, for the five macrobenchmarks, CNI_4 improves the performance over NI_{2w} between 10-42% on the memory bus and 11-46% on the I/O bus. This amounts to 28-92% of the total gain

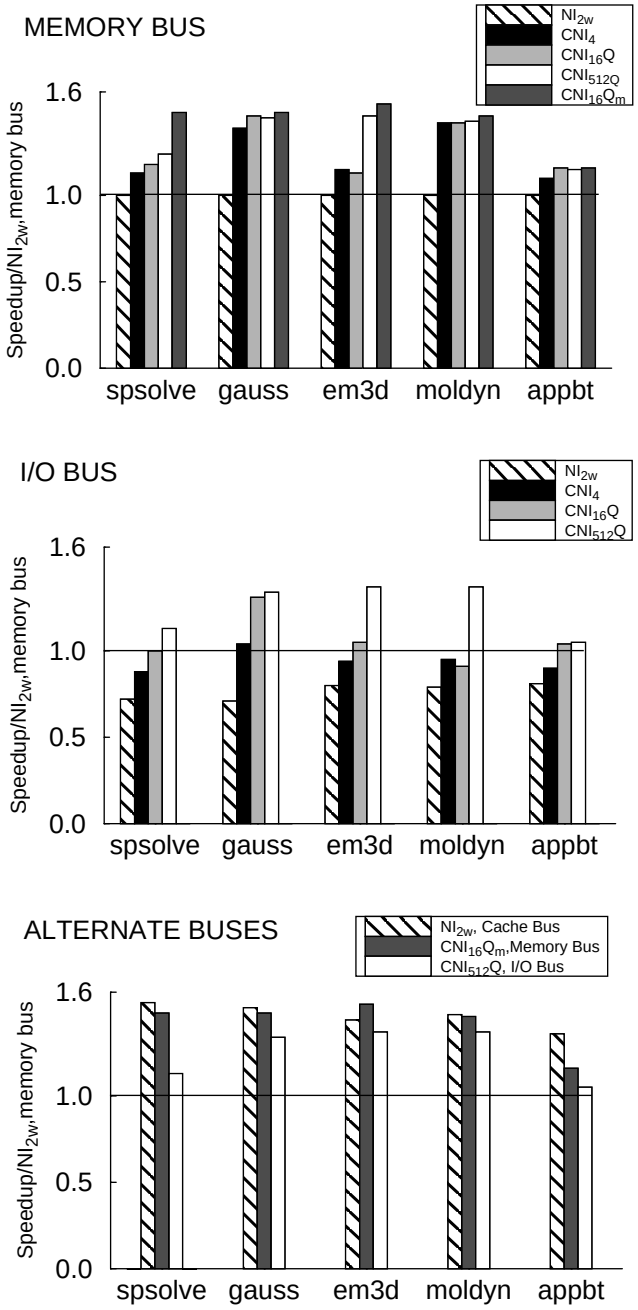

FIGURE 8. This figure compares our five network interface implementations on the memory, I/O, and cache buses for five benchmarks. The vertical axes of the graphs show the speedup over NI_{2w} on the memory bus. (a) compares NI_{2w}, CNI_4, $CNI_{16}Q$, $CNI_{512}Q$, and $CNI_{16}Q_m$ on the memory bus. (b) compares NI_{2w}, CNI_4, $CNI_{16}Q$, and $CNI_{512}Q$ on the I/O bus. (c) compares NI_{2w} on the cache bus, $CNI_{16}Q_m$ on the memory bus, and $CNI_{512}Q$ on the I/O bus.

achieved by our CNIs on the memory bus and 25-52% of that on the I/O bus.

Extra Buffering. The CQ-based CNIs provide extra buffering that helps smooth out bursts in message traffic. However, $CNI_{16}Q$ and $CNI_{512}Q$ cannot always take advantage of this feature. The problem is that once a sender blocks, the flow control software aggressively buffers received messages in memory. This results in messages being pulled out of the CNI's cache, even when there was still room for additional messages. Further, because of its

small queue size, $CNI_{16}Q$ frequently updates its shadow head by reading the processor's head pointer, which creates bus contention. Because of these effects, $CNI_{16}Q$ and CNI_4 achieve roughly the same performance on the memory bus. $CNI_{512}Q$'s larger queue reduces the frequency of shadow head updates. For *em3d* this improves $CNI_{512}Q$'s performance over $CNI_{16}Q$ by 29% on the memory bus.

On the I/O bus, the higher latencies mitigate the effects of over aggressive buffering, by slowing down the rate at which messages are extracted and buffered. This allows CQ-based CNIs to exploit their buffering and smooth out the bursty traffic of all five macrobenchmarks. In *spsolve* and *em3d*, several small active messages (with 12-byte payload) can be in flight simultaneously causing bursts in the message arrival. In *gauss* and *moldyn*, periodic bulk transfers cause the bursts. Request-response protocols normally do not cause bursts. However, *appbt* exhibits a hot spot in which one processor receives twice as many messages as other processors. Thus, $CNI_{16}Q$ improves the performance of *spsolve*, *gauss*, *em3d*, and *appbt* over CNI_4 on the I/O bus by 15%, 26%, 11%, and 16% respectively. For *moldyn*, frequent updates of the shadow head causes contention at the I/O bridge and actually slightly reduces $CNI_{16}Q$'s performance. But the extra buffering and infrequent updates of the shadow head result in $CNI_{512}Q$ improving performance by 13%, 31%, and 51%, respectively, over $CNI_{16}Q$ for *spsolve*, *em3d*, and *moldyn*.

Overflow to Memory. $CNI_{16}Q_m$ allows messages to smoothly overflow to memory when the device cache fills up. This eliminates the over aggressive message buffering that was a problem for $CNI_{16}Q$ and $CNI_{512}Q$. This automatic buffering improves *spsolve*'s performance over $CNI_{512}Q$ by 20%. For the other four macrobenchmarks, $CNI_{16}Q_m$ is slightly better than $CNI_{512}Q$. Thus, $CNI_{16}Q_m$ consistently outperforms $CNI_{512}Q$ on the memory bus *even with significantly less memory (i.e., cache) on the device*.

On the memory bus, $CNI_{16}Q_m$ shows the best overall performance improvement (between 17-53%), while $CNI_{512}Q$ shows the best improvement (between 30-88%) on the I/O bus. Also $CNI_{16}Q_m$ on the memory bus comes within 4% of NI_{2w}'s performance on the cache bus for *spsolve*, *gauss*, and *moldyn*, and within 17% for *appbt* (Figure 8). For *em3d*, $CNI_{16}Q_m$ on the memory bus slightly outperforms the cache bus NI_{2w} because NI_{2w} has limited buffering in the device and the processor must explicitly buffer messages in memory. These indicate that $CNI_{16}Q_m$ is an attractive alternative because it is feasible with most commodity processors and requires no change to the processor core or board.

Finally, CNIs significantly reduce the memory bus occupancy. By polling on cached registers and transferring messages in full cache block units, CQ-based CNIs on the memory bus reduce the memory bus occupancy by as much as 66% (averaged over five macrobenchmarks) compared to NI_{2w}. In comparison, CNI_4 reduces the memory bus occupancy by only 23% because it still requires the processor to poll across the memory bus.

6 Related Work

Coherent Network Interfaces differ from most previous work on program-controlled network I/O in three important respects. First, unlike other NIs, CNIs interact with processor caches and main memory primarily through the node's coherence protocol. Second, CNIs separate the logical and physical locations of NI registers and queues allowing processors to cache them like memory. Third, CNIs provide a uniform memory-based interface for both local and remote communication. Table 4 compares network interfaces of different machines with respect to these three issues. The Thinking Machines' CM-5 [44], the Wisconsin Typhoon [38], the Stanford FLASH [28], and the Meiko CS2 [33] multiprocessors provide high latency uncached access to their NIs on the memory bus. Since both Typhoon and FLASH have a coherent cache in their network interfaces, they could both support CQs. The Meiko CS2 network interface supports the memory bus's coherence protocol, but does not contain a cache. The MIT Alewife [1] and FUGU [32] machines provide uncached access to their NIs under control of a custom CMMU unit. The StarT-NG NI [10] is not coherent because it is a slave device on the non-coherent L2 coprocessor interface. StarT-NG NI queues can be cached in the L1 cache, but the processor must explicitly self-invalidate or flush stale copies of the NI queues. Wisconsin T-Zero [39] caches device registers, but not queues, and only uses them to send information from the device to the processor. AP1000 [41] directly DMA's messages from the processor's cache to the NI, but does not receive messages directly into the cache. Princeton SHRIMP's memory bus NI [4] allows coherent caching on the processor, but requires processors to use the higher traffic write-through mode. The DI multicomputer's on-chip NI [11] neither supports coherence nor allows its registers or queues to be cached. The processor chip interfaces with the rest of the system through the NI. Unlike other machines, the DI-multicomputer supports a uniform message-based interface for both memory and the network, whereas CNI uses the same *memory* interface for both memory and network.

Network Interface	Coherence	Caching	Uniform Interface
CNI	Yes	Yes	Memory Interface
TMC CM-5 [44]	No	No	No
Typhoon [38]	Possible	Possible	Possible
FLASH [28]	Possible	Possible	Possible
Meiko CS2 [33]	Possible	No	Possible
Alewife [1]	No	No	No
FUGU [32]	No	No	No
StarT-NG [10]	No	Maybe	No
AP1000 [41]	No	Sender	No
T-Zero [39]	Partial	Partial	No
SHRIMP [4]	Yes	Write Through	No
DI Multicomputer[11]	No	No	Network Interface

TABLE 4. Comparison of CNI with other network interfaces

Unlike many other NIs, our *implementation* of CNIs does not require changes to an SMP board or other standard components. Yet they enable processors and network interfaces to communicate through the cachable memory accesses, for which most processors and buses are optimized. Henry and Joerg [21] and Dally, et al. [15] advocate changes to a processor's registers. MIT Alewife [1] and Fugu [32] rely on a custom cache controller. MIT StarT-NG [10] requires a co-processor interface at the same level as the L2 cache. AP1000 [41] requires integrated cache and DMA controllers. Stanford FLASH [28, 20] uses a custom memory controller with an embedded processor. Other efforts, such as the TMC CM-5 or SHRIMP, use standards components, but settle for lower performance by using loads and stores to either uncachable or write-through memory, instead of using the full functionality of write-back caches.

Three efforts that appear very similar to our work are FLASH messaging [19], UDMA/SHRIMP-II [3], and Remote Queues [6]. We differ from FLASH, because we do not have a processor core in the network interface, we allow commands to use cachable loads and stores, and we can notify the receiving process without an interrupt. We differ from the UDMA/SHRIMP-II, because we use the same mechanisms when the destination is local and remote (whereas SHRIMP-II's UDMA does not handle local memory to

local memory copies), we use only virtual addresses (where SHRIMP-II requires that the sender knows the receiver's physical addresses), we allow device registers to use writeback caching, and we focus on fine-grain user-to-user communication in which the receiving process may be notified without an interrupt. We differ from Remote Queues by being at a lower-level of abstraction. Remote Queues provide a communication model similar to Active Messages [45], except extracting a message from the network and invoking the receive handler can be decoupled. Implementing Remote Queues with CNIs is straightforward and offers advantages over CM-5, Intel Paragon, MIT Alewife, and Cray T3D network interfaces. CNIs support cachable device registers for low-overhead polling (unlike the others), allow network buffers to gracefully overflow to memory (unlike the CM-5), and do not require a second processor (Paragon), custom cache controller (Alewife), or hardware support for globally shared memory (T3D).

Finally, our results conservatively estimate the rate at which processors can move data, given trends toward block move instructions, prefetch support, and non-blocking caches. UltraSPARC-I [42], for example, has instructions that copy a cache block (64 bytes) to or from floating-point registers, SPARC V9 [46] has four prefetch instructions that indicate expected locality (e.g., that a block will be written once and not accessed again), and numerous processors do not stall on the first cache miss. These optimizations can further increase the relative benefits of using CNIs.

7 Conclusions

This paper explored using snooping cache coherence to improve communication performance between processors and network interfaces (NIs). We call NIs that use coherence *coherent network interfaces* (CNIs). We restricted our study to NI/CNIs that reside on memory or I/O buses, to NI/CNIs that are much simpler than processors, and to the performance of fine-grain messaging from user process to user process.

We developed two mechanisms that CNIs use to communicate with processors. A *cachable device register* allows information to be exchanged in whole cache blocks and permits efficient polling where cache misses (and bus transfers) occur only when status changes. *Cachable queues* reduce re-use overhead by using array of cachable, coherent blocks managed as a circular queue and (optionally) optimized with lazy pointers, message valid bits, and sense-reverse.

We then compared four alternative CNIs—CNI_4, $CNI_{16}Q$, $CNI_{512}Q$, and $CNI_{16}Q_m$—with a CM-5-like NI. Microbenchmark results showed that CNIs significantly improved the round-trip latency and bandwidth of small and moderately large messages. For small message sizes, between 8 and 256 bytes, CNIs improved the round-trip latency by 20-84% compared to NI_{2w} on a coherent memory bus and 29-141% on a coherent I/O bus. For moderately large messages, between 8 and 4096 bytes, CNIs improved bandwidth by 59-169% over NI_{2w} on a coherent memory bus and 51-287% on a coherent I/O bus. Macrobenchmark results showed that $CNI_{16}Q_m$ performed the best on the coherent memory bus and $CNI_{512}Q$ on the coherent I/O bus. $CNI_{16}Q_m$ was 17-53% better than NI_{2w} on the memory bus, while $CNI_{512}Q$ was better than NI_{2w} by 30-88% on the I/O bus. Also, $CNI_{16}Q_m$ on the memory bus came within 17% of NI_{2w}'s performance on the cache bus. This indicates that $CNI_{16}Q_m$ is an attractive alternative because it is feasible with most current commodity microprocessors and requires no change to the processor core or board.

Our experiments use assumptions that are reasonable for commodity parts in the present and near future. In the medium term, our quantitative results will likely be obviated by better memory interconnects that pipeline requests, allow out-of-order responses, or even abandon physical buses. Nevertheless, we expect our qualitative results in favor of CNIs to continue to hold as CNIs continue to exercise memory interconnects with the operations the interconnects are optimized for, namely, coherent block transfers. In the longer term, caches, memory bus, NIs, and memory may move onto processor chips (or, in another view, everything moves onto memory chips). To manage complexity, however, these super chips may resemble boards of old systems with die area devoted to a custom mix of relatively-standard, optimized components (e.g., processors and DRAM) interconnected through well-defined interfaces. While integrating an NI into a processor is possible, CNIs will be interesting as a less expensive (in terms of design and verification costs) way to deliver competitive performance.

Acknowledgements

This work developed in the supportive environment provided by members of the Wisconsin Wind Tunnel Project (`http://www.cs.wisc.edu/~wwt`). Steve Reinhardt and Rob Pfile were instrumental in ideas leading to CDRs. Fred Chong and Shamik Sharma provided us with spsolve. Erik Hagersten, Steve Reinhardt, Jon Wade, Bob Zak, and anonymous referees provided excellent feedback on various drafts of this manuscript. Special thanks to Steve Reinhardt for his generous help with debugging the simulator.

References

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[2] Anant Agarwal, Richard Simoni, Mark Horowitz, and John Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, 1988.

[3] Matthias A. Blumrich, Cesary Dubnicki, Edward W. Felten, and Kai Li. Protected User-level DMA for the SHRIMP Network Interface. In *Second IEEE Symposium on High-Performance Computer Architecture*, February 1996.

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

[5] Shekhar Borkar, Robert Cohn, Geroge Cox, Thomas Gross, H. T. Kung, Monica Lam, Margie Levine, Brian Moore, Wire Moore, Craig Peterson, Jim Susman, Jim Sutton, John Urbanski, and Jon Webb. Supporting Systolic and Memory Communication in iWarp. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 70–81, 1990.

[6] Eric A. Brewer, Frederic T. Chong, Lok T. Liu, Shamik D. Sharma, and John Kubiatowicz. Remote Queues: Exposing Message Queues or Optimization and Atomicity. In *Proceedings of the Sixth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 42–53, 1995.

[7] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D . J. States, S. Swamintathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization, and dynamics calculation. *Journal of Computational Chemistry*, 4(187), 1983.

[8] Doug Burger and Sanjay Mehta. Parallelizing Appbt for a Shared-Memory Multiprocessor. Technical Report 1286, Computer Sciences Department, University of Wisconsin–Madison, September 1995.

[9] Satish Chandra, James R. Larus, and Anne Rogers. Where is Time Spent in Message-Passing and Shared-Memory Programs? In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 61–75, October 1994.

[10] Derek Chiou, Boon S. Ang, Arvind, Michael J. Becherle, Andy Boughton, Robert Greiner, James E. Hicks, and James C. Hoe. StartT-ng: Delivering Seamless Parallel Computing. In *Proceedings of EURO-PAR '95*, Stockholm, Sweden, 1995.

[11] Lynn Choi and Andrew A. Chien. Integrating Networks and Memory Hierarchies in a Multicomputer Node Architecture. In *Proceedings of the Eighth International Parallel Processing Symposium*, 1994.

[12] Fred Chong, Shamik Sharma, Eric Brewer, and Joel Saltz. Multiprocessor Runtime Support for Irregular DAGs. In R. Kalia and P. Vashishta, editors, *Toward Teraflop Computing and New Grand Challenge Applications*. Nova Science Pulishers, Inc., 1995.

[13] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[14] Fredrik Dahlgren. Boosting the Performance of Hybrid Snooping Cache Protocols. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 60–69, 1995.

[15] William J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Michael Larivee, Rich Nuth, Scott Wills, Paul Carrick, and Greg Flyer. The J-Machine: A Fine-Grain Concurrent Computer. In G. X. Ritter, editor, *Proc. Information Processing 89*. Elsevier North-Holland, Inc., 1989.

[16] Babak Falsafi, Alvin Lebeck, Steven Reinhardt, Ioannis Schoinas, Mark D. Hill, James Larus, Anne Rogers, and David Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of Supercomputing '94*, pages 380–389, November 1994.

[17] James R. Goodman and Philip J. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 422–431, 1988.

[18] PCI Special Interest Group. *PCI Local Bus Specification, Revision 2.1*, 1995.

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

[20] Mark Heinrich, Jeffrey Kuskin, David Ofelt, John Heinlein, Joel Baxter, Jaswinder Pal Singh, Richard Simoni, Kourosh Gharachorloo, David Nakahira, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 274–285, 1994.

[21] 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 (ASPLOS V)*, pages 111–122, October 1992.

[22] Mark D. Hill, James R. Larus, and David A. Wood. Tempest: A Substrate for Portable Parallel Programs. In *COMPCON '95*, pages 327–332, San Francisco, California, March 1995. IEEE Computer Society.

[23] MIPS Technologies Inc. *MIPS R10000 Microprocessor User's Manual*, 1995.

[24] Sun Microsystems Inc. *SPARC MBus Interface Specification*, April 1991.

[25] Vijay Karamcheti and Andrew A. Chien. A Comparison of Architectural Support for Messaging in the TMC CM-5 and the Cray T3D. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 298–307, 1995.

[26] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, 1993.

[27] John Kubiatowicz, David Chaiken, and Anant Agarwal. Closing the Window of Vulnerability in Multiphase Memory Transactions. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 274–284, 1992.

[28] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[29] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fifth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, July 1992.

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

[31] Lok Tin Liu and David E. Culler. Evaluation of the Intel Paragon on Active Message Communication. In *Proceedings of Intel Supercomputer Users Group Conference*, June 1995.

[32] Kenneth Mackenzie, John Kubiatowicz, Anant Agarwal, and Frans Kaashoek. Fugu: Implementing Translation and Protection in a Multiuser, Multimodel Multiprocessor. Technical Memo MIT/LCS/TM-503, MIT Laboratory for Computer Science, October 1994.

[33] Meiko World Inc. Computing Surface 2: Overview Documentation Set, 1993.

[34] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[35] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 68–79, July 1995.

[36] Robert W. Pfile. Typhoon-Zero Implementation: The Vortex Module. Technical report, Computer Sciences Department, University of Wisconsin–Madison, 1995.

[37] Steven K. Reinhardt. Tempest Interface Specification (Revision 1.2.1). Technical Report 1267, Computer Sciences Department, University of Wisconsin–Madison, February 1995.

[38] Steven 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*, pages 325–337, April 1994.

[39] Steven K. Reinhardt, Robert W. Pfile, and David Wood. Typhoon-0: Hardware Support for Distributed Shared on a Network of Workstations Memory. In *Workshop on Scalable Shared-Memory Multiprocessors*, 1995.

[40] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996.

[41] Toshiyuki Shimizu, Takeshi Horie, and Hiroaki Ishihata. Low-Latency Message Communication Support for AP1000. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 288–297, 1992.

[42] SPARC Technology Business. *UltraSPARC-I User's Manual, Revision 1.0*, September 1995.

[43] Paul Sweazey and Alan Jay Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 414–423, 1986.

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

[45] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrating Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[46] David L. Weaver and Tom Germond, editors. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.

[47] Shlomo Weiss and James E. Smith. *Power and PowerPC*. Morgan Kaufmann Publishers, Inc., 1994.

[48] David A. Wood and Mark D. Hill. Cost-Effective Parallel Computing. *IEEE Computer*, 28(2):69–72, February 1995.