The Design and Performance Evaluation of the DI-multicomputer *

Lynn Choi
Center for Supercomputing R & D
University of Illinois
Urbana, IL 61801-1351
lchoi@csrd.uiuc.edu
(217)333-0969
(217)244-1351 (fax)

Andrew A. Chien
Department of Computer Science
University of Illinois
Urbana, IL 61801
achien@cs.uiuc.edu
(217)333-6844
(217)244-6500 (fax)

Abstract

In this paper, we propose a new multicomputer node architecture, the DI-multicomputer which uses packet routing on a uniform point-to-point interconnect for both local memory access and internode communication. This is achieved by integrating a router onto each processor chip and eliminating the memory bus interface. Since communication resources such as pins and wires are allocated dynamically via packet routing, the DI-multicomputer is able to maximize the available communication resources, providing much higher performance for both intra-node and internode communication. Multi-packet handling mechanisms are used to implement a high performance memory interface based on packet routing. The DI-multicomputer network interface provides efficient communication for both short and long messages, decoupling the processor from the transmission overhead for long messages while achieving a minimum latency for short messages. Trace-driven simulations based on a suite of message passing applications show that the communication mechanisms of the DI-multicomputer can achieve up to four times speedup when compared to existing architectures.

1 Introduction

Parallel computer systems contain processors, memory modules, and interconnection networks which tie them together. While many parallel systems have impressive peak processing rates, they cannot approach maximum performance on application programs unless computation and communication performance are balanced. In many cases, the imbalance between computation and communication performance is not due to poor performance of the core network, but rather a poor coupling of the network and the processing node. In this paper, we address design issues of a multicomputer node architecture, particularly the network interface and its interaction with the local memory hierarchy. The goal of a multicomputer node architecture is to support high performance in both message passing and local computation.

Two major varieties of multicomputer node architecture have emerged (see Figure 1). The first interfaces the network to a local bus, allowing network-memory data transfers and preserving microprocessor interface compatibility. We term this approach the *medium-grained* approach, and it is exemplified by commercial machines such as the Intel Paragon XP/S [25], Thinking Machine CM-5 [34], and Fujitsu AP1000 [22]. Using a stock microprocessor as a building block typically produces poor coupling of processor and network, increasing the software overhead for communication. For example, in the Intel Paragon XP/S, the average hardware network latency is less than one μs, yet the minimum process to

*A preliminary version of some of this work appears in [10].

process communication delay is over 15 μs. Such high communication delay confines the machine to the exploitation of medium-grained parallelism, limiting the application scope and scalability of these machines. In addition, medium-grained machines exhibit other critical problems. First, sharing a bus between local memory access and internode communication limits the available bandwidth for both activities. Second, the medium-grained machines typically transfer incoming messages into the memory and then to the processor, producing long response times for short messages.

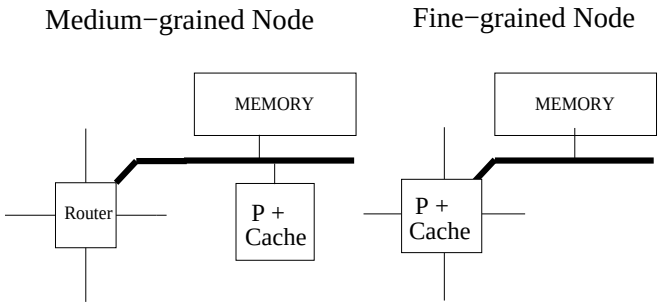


Figure 1: Medium-grained and Fine-grained Node Organizations.

The second approach, termed the *fine-grained* approach, addresses the problems of the medium-grained architectures by providing a more tightly coupled network interface. By integrating the router onto the processor chip and deeply into the processing core, fine-grained architectures can dramatically reduce communication overhead. However a fine-grained approach requires changes to the microprocessor interface, and significant redesign of the processor. The *fine-grained* approach, is exemplified by the MIT J-machine [15, 16], Intel iWARP [29], and Inmos Transputer [35].

Fine-grained architectures also have several critical performance problems. First, by integrating a router on the processor, they statically partition the processor pin bandwidth between local memory access and internode communication. For each of the fine-grained designs mentioned, this reduces local memory performance and therefore limits local computation speed. Second, most fine-grained architectures map messages into the register address space, providing rapid response to short messages. However, this register-based message handling forces the processor to execute instructions to send and receive long messages and copy them through memory hierarchy. This increases memory hierarchy traffic, producing significant runtime overhead for long messages.

We propose a new multicomputer node architecture, the *DI-multicomputer*, which addresses the problems of the existing architectures by integrating the node memory hierarchy and the routing network. The basic idea is to use packet interfaces for both the local memory access and interprocessor communication, joining the node seamlessly with the interprocessor interconnection network. The packets are routed on a dense regular pattern of point-to-point interconnects which is chosen to saturate the wiring media. To make efficient use of pin bandwidth, the memory access by node processor uses mechanisms for multi-packet send and receive. This realizes a local processor memory hierarchy which matches or outperforms existing bus-based interfaces. Figure 2 illustrates a DI-multicomputer node organization on a two dimensional mesh network.

While using packet routing exclusively changes the processor interface dramatically, merging the memory hierarchy with network in multicomputers can produce significant benefits. For example, the combined interface can use all the pins and wires for one communication task, increasing the peak bandwidth available for both local memory access and the routing network. In addition, with the uniform interconnect the packet-based memory interface is now powerful enough to handle interprocessor messages. Local to remote block transfers as well as remote memory access can be done in the same way as local memory operation with no additional cost. With the powerful packet-based interface,

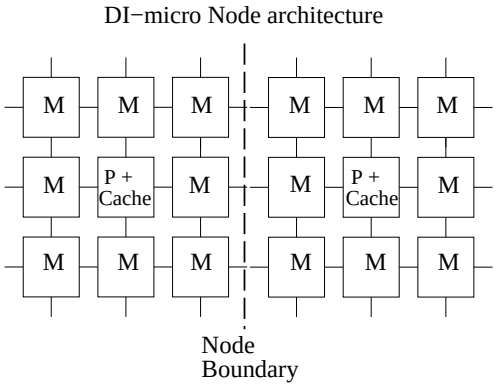


Figure 2: A DI-multicomputer node.

the DI-multicomputer network interface directs different types of messages to different levels of the memory hierarchy under software control, achieving both low-latency response for short messages and high-bandwidth transfers for long messages.

In this paper, we describe the design of the DI-multicomputer, an architecture based on dynamic interconnection and compare its performance to several existing multicomputer architectures. To evaluate the memory hierarchy performance of a node based on DI, we compare the bus interface of an existing microprocessor to the packet-based memory interface of the DI-multicomputer processor. Also, we evaluate the communication performance of the DI-multicomputer by running a trace driven simulation on existing message passing applications. Our results indicate that a node architecture based on dynamic interconnection has significant performance advantages in the areas of processor memory bandwidth and both short and long message passing.

Overview The rest of paper is organized as follows. In Section 2, we discuss some critical performance limitations of existing architectures that motivate the design of a new communication architecture. Section 3 introduces the basic ideas behind the DI-multicomputer and describes the register file architecture of its processor. Section 4 and 5 present the novel mechanisms which allow the DI-multicomputer to support both a high-performance local memory hierarchy and low-latency interprocessor communication. In Section 6, we compare the performance of a DI-multicomputer node to that of traditional bus-based nodes, showing that the DI-multicomputer attains much higher performance in both the memory hierarchy and message-passing operations. Section 7 discusses the key implementation issues for a DI-multicomputer node. Finally, Section 8 and 9 discuss related work and summarize the research results.

2 Motivation for A New Communication Structure

The relentless increase in computing performance of microprocessors continues to raise their input/output requirements. Latency hiding and avoidance techniques such as prefetching and multithreading further increase input/output requirements, making communication resources such as pins and wires a performance critical bottleneck. As a result, the communication structure which interconnects hardware modules has an increasingly significant contribution to overall system performance. In this section, we discuss two critical problems of traditional interconnection structures and present dynamic interconnection as an alternative structure for future multicomputers.

2.1 Static Interconnection

Computer systems today rely primarily on static interconnection – outputs are wired directly to all of the inputs they drive. These static and direct connections form an electrical network that is typically dedicated to a single use such as a column-address strobe, an address line, or even a bus line. However, static interconnection has two clear sources of inefficiency which limit the processor performance in a multicomputer node.

1. Partitioning communication resources between router and memory interface prevents efficient utilization of available pin bandwidth.

2. Irregular or multi-tap interconnects (buses) increase capacitive loading and signal reflections, limiting the maximum switching speed [18, 31, 20] and reducing available pin bandwidth.

Table I demonstrates the problem of static pin allocation in existing microprocessors. First, in off-the-shelf microprocessors such as Intel i860XP and DEC Alpha, only about 40% of signal pins are allocated for data transfer due to address and control signals. This static pin allocation limits the peak memory bandwidth of a machine substantially. Second, the percentage of data pins is further reduced to 10% to 30% in custom multicomputer processors due to signal pin partition between the memory interface and the network interface. An unavoidable consequence is that it impossible to support both high performance network and memory interfaces in multicomputer building blocks such as the iWARP [29] and MDP [16]. They all compromise, yielding mediocre network or memory system performance. The difference signalling rates of buses versus point to point interconnects is well documented and typically a factor of 3 or 4 in clock rate [18, 31].

Table I: Pin allocation and input/output performance. The multicomputer chips (iWarp and MDP) have low memory bandwidth and only moderate communication bandwidth.

	i860XP	DEC Alpha	iWarp	MDP
Package pins	262	431	271	168
Signal Pins	154	291	217	119
Address/Data Pins	29/64	29/128	24/64	11/12
Network Pins	n.a.	n.a.	112	90
% Data Pins (Memory BW)	41.5% (400 MB/s)	43.9% (1.2 GB/s)	29.5% (160 MB/s)	10.1% (24 MB/s)
% Network Pins (Network BW)	n.a.	n.a.	51.6% (320 MB/s)	75.6% (192 MB/s)

2.2 Dynamic Interconnection

We propose a new communication architecture, *dynamic interconnection*, which avoids the problems of the conventional interconnection structures. Dynamic interconnection systems exhibit the following key characteristics at a low-level of hardware interconnection.

1. Use regular point-to-point interconnections to saturate the wiring media.

2. Share communication resources (pins and wires) amongst a variety of communication tasks using low-latency packet routing.

Dynamic interconnection (DI) addresses the key problems with static interconnection systems. Regular point-to-point interconnects maximize signalling speeds by minimizing the capacitive and the inductive loads of each wire. Packet routing allows communication resources to be shared efficiently.

Advances in packet routers allow them to attain channel utilizations in excess of 90% [14] and extremely low latency [33]. These characteristics enable dynamic interconnection systems to achieve comparable communication latencies and much higher bandwidth.

Dynamic interconnection systems have two major advantages. First, pooling communication resources among several tasks eliminates the resource idle time in the static interconnection system. This allows the entire bandwidth to be focused on either the memory or network interface, giving both much higher peak performance. Second, though using regular, point-to-point interconnection requires communications to be routed dynamically, incurring additional delay, dynamic interconnection not only allows the wiring media to be saturated, maximizing the wire bisection, it also allows the wires to be switched at maximum speed, maximizing the bandwidth of each wire. Together, these two features maximize the communication capacity of the system.

3 A Multicomputer Node Architecture based on Dynamic Interconnection

Using dynamic interconnection requires modification to the input/output interface of each multicomputer node. A DI-multicomputer node consists of three elements: a processor, memory units, and routers. However, rather than connecting the elements via a shared node bus, each element is embedded in a low-latency packet routing network, requiring a small router[1] per element. In addition to message passing, all local memory operations such as cache refills are achieved by sending and receiving packets (see Figure 7).

3.1 Design Concepts

The DI-multicomputer processor, called the *DI-microprocessor*, is a RISC processor extended with a network interface, router and a number of memory packet send and receive buffers, replacing the bus-based memory interface. The DI-microprocessor uses a DEC Alpha microprocessor architecture [19] as its base RISC processor. The DI-microprocessor instruction set architecture uses a subset of the DEC Alpha microprocessor instruction set augmented with instruction support for message passing, address translation and synchronization. [2]

The high-level organization of the DI-microprocessor is shown in Figure 3. The specialized memory-packet send and receive buffers allow the memory interface to handle multiple memory packets simultaneously, enabling the full utilization of the processor chip bandwidth. The network interface distinguishes memory packets with communication packets and is also responsible for flow control between router and both memory and network interfaces.

- **Memory interface: Multi-packet Handling** To build a high performance memory hierarchy, it is critical to utilize all the pin bandwidth at the processor chip boundary. The multi-packet handling of the DI-microprocessor's memory interface enables efficient use of the pin bandwidth with a simple hardware.

- **Network Interface: Distinct Mechanisms for Short and Long Messages** The DI-multicomputer's parallel interconnect allows messages to be routed directly to an appropriate level of the memory hierarchy, supporting high performance for both short and long messages. In particular, short messages are routed directly onto the processor chip, allowing them to be handled with low latency. Long messages are routed directly from local memory to remote memory under software control.

[1] As a variety of designs have shown [17], a router need not require a large amount of hardware. For example, the three dimensional router used in the J-machine [15] requires only 29,000 transistors.

[2] The instruction set design and the memory and message packet formats of the DI-microprocessor are described in Appendix A and B.

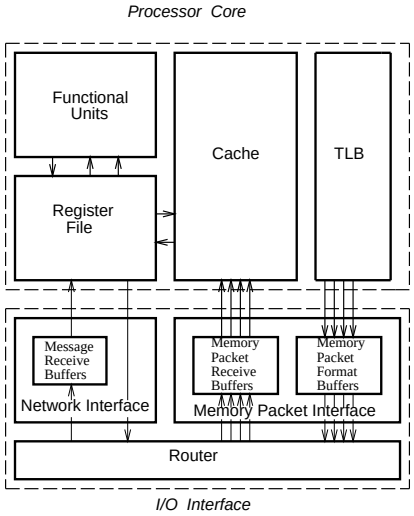


Figure 3: The processor chip of a DI-multicomputer (not to scale).

Details of the memory and network interfaces are described in Sections 4 and 5. Implementation issues for the DI-multicomputer are discussed in Section 7.

3.2 DI-microprocessor Architecture Overview

RISC Architecture The DI-microprocessor is a RISC architecture that is designed with particular emphasis on support for building massively parallel systems.

Support for Message Passing Current message passing machines are optimized either for short messages or for long messages. Register-based message transmission used in custom multicomputer processors such as the MDP of the MIT J-Machine reduces startup cost for short messages but may incur significant processor overhead for long messages in which each word must be handled explicitly. On the other hand, DMA-based message passing used in most commercial multicomputers are usually effective for long messages but may have large startup overheads due to operating system interfaces. This is especially significant for short messages.

The DI-microprocessor solves this problem by having distinct mechanisms both for short and long messages. Its message handling instructions allow high performance communication for both long and short messages without DMA or any system call to message passing library routines. Long messages are transferred from local memory to remote memory by executing a MOVE instruction at the source node, which initiates the transfer. This eliminates the costly transmission and reception overhead in case of a long message. In contrast, short messages are transferred directly from the local processor to the remote processor without interacting with the memory system. Generation, transmission and reception of short messages are based on the register file. This approach not only minimizes the startup cost and reception overhead for short messages but also allows fast message formation using the register file as a scratch-pad message buffer.

Naming and Translation Every module [3] in the network is addressed linearly from 0 to the maximum number of modules allowed. Therefore, memory module addresses are in the same address space as

[3] A module refers a node in the interconnection network.

processor module addresses. Figure 4 shows the address formats for virtual and physical addresses. As usual, virtual address consists of page number and the offset within the page. Unlike conventional machines, the physical page number consists of the memory module address and the page number within the module.[4] As in traditional multicomputers, each processor has a private virtual address space. Local memory is accessed via 64 bit virtual addresses. A translation look aside buffer (TLB) supports virtual to physical address translation for local memory.

Protection and virtual memory support for blocked message transfer (MOVE instruction) are enforced by memory access mechanisms. For example, a page fault for the message transfer is treated as same as the page fault for any memory operation. Based on the address of the local memory and the size of the block, TLB entry should be checked on the memory operation before the message transfer. The TLB miss or a page fault should be handled by the operating system. Therefore, the page fault should not occur during the message transfer.

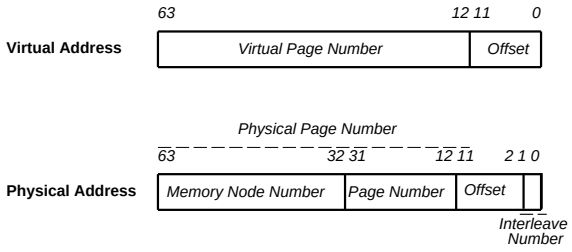


Figure 4: Virtual and physical address format

There is no global virtual address space supported by hardware. In other words, there is no hardware support for global TLB coherence. Software schemes for a global TLB coherence can be used to provide a global virtual address space. On the other hand, unlike conventional machines, the DI-multicomputer provides a global physical address space. Because address translations are not shared, the address translation for the remote node is performed by short message transfers between the local node and the remote node.

Maintaining global cache coherence is possible but an orthogonal issue for the DI-based systems, although each node has locally coherent caches. [5] Either hardware directory-based coherence protocols [2, 26] or software coherence schemes [7, 11] can be used to provide a cache coherent shared address space.

Internode Synchronization A node can communicate with other nodes with message passing. In DI-multicomputer, short and long messages use different mechanisms for synchronization as they are differently handled for message passing. While a short message creates a handler thread at the destination, a long message sent to remote memory needs to be synchronized with the receiving thread at the destination processor. On long message arrival, remote memory nodes generate acknowledgment messages to the destination processor node. And these messages are handled by memory interface at the destination processor as acknowledgment messages for a previously issued store operation. DI-microprocessor provides memory instructions to detect the long message arrival without accessing off-chip memory (see section 5.2).

Block Multithreaded Architecture The DI-microprocessor is a block multithreaded architecture which has multiple hardware contexts to facilitate fast context switching and trap handling. A similar

[4] For interleaved memory organization (see Section 4), the low order bits are used to denote memory interleave module.

[5] To maintain cache coherence locally on DMA access, an I/O processor should send messages to both memory and cache modules.

design we know is the processor architecture of the MIT Alewife [1], which is based on the SPARC architecture. The DI-microprocessor architecture is based on the DEC Alpha processor core. The only modification to the Alpha architecture is its DI-based memory interface and the addition of the multiple hardware contexts. Multiple hardware contexts and the register-file based message handling are chosen to minimize the message handling overhead at the lowest level. It is designed to support message passing operations at the same efficiency as local memory operations.

An incoming message is received into an empty context, creating a new thread at the destination. By receiving messages into hardware contexts, we eliminate both the time to copy messages into memory and the time to load the messages (contexts) from the memory. Note that this is done with no instruction overhead for the DI-multicomputer by copying the messages into hardware contexts directly from network with cycle stealing of register file access ports. This significantly alleviates the overhead of message reception in the conventional message passing machines since the message reception does not involve the context save and restore of the currently running context. Moreover, it allows the computation to overlap with message reception. This message driven reception mechanism is similar to that of J-machine [15]. However, it differs from J-machine in that the reception is based on the register file rather than memory, so it's not necessary to load the messages from memory except in the special cases. A dedicated hardware context moves the incoming messages into memory for message overflow cases. Additional contexts are used to buffer incoming messages or to increase processor utilization. Context switching can occur on a cache miss, incoming message overflow or at a thread completion.

3.2.1 Register File Architecture

The visible state of the processor is an extension to the microprocessor core of the DEC Alpha processor with the following modification to the register file architecture.

Context State The register file is partitioned into multiple contexts. Figure 5 shows the register file organization of DI-microprocessor. Each hardware context has thirty two 64-bit registers that can be used as local scratch-pad registers as well as message buffers. The registers are grouped into sets of 4 contiguous registers (row registers). Also, the on-chip cache is also organized as an array of 4-word rows. *Row mode load-store operations* allow 4 contiguous words of data to be transferred at a time between memory and the register file. These operations allow high bandwidth cache access, which is useful both for fast context save and restore and for fast message buffering for message overflow.

On a short message reception, the message is deposited directly onto an empty context by the network interface hardware. The first word of a message always contains the instruction address of the context that will be created. After copying an entire message, the context becomes active and can be scheduled by the scheduling mechanism. A new thread starts with a jump R0 instruction. Each context has a special register called PSR (Program Status Register).

- **Program Status Register (PSR)** The PSR contains the local state of the corresponding context. In addition to its existing information such as thread state, thread id and pointer to data storage in memory, it also contains other state information specific to the architecture. The *interrupt enable bit, e*, shows the interruptability of the context on a exceptional case such as message reception. Resetting this flag disables all exceptional cases which require the handling of the trap context. Therefore on such exceptional cases, context switching does not occur and the thread continues its execution until a cache miss occurs. And the *schedule bit, s*, shows whether the thread is ready or blocked[6]. This state details the thread state information. If this flag is not set, it can not be dispatched to the processor until the blocking condition is resolved. The register is memory-mapped to system address space and can be accessed by system thread using memory access operations.

[6] on a cache miss or on a long message reception

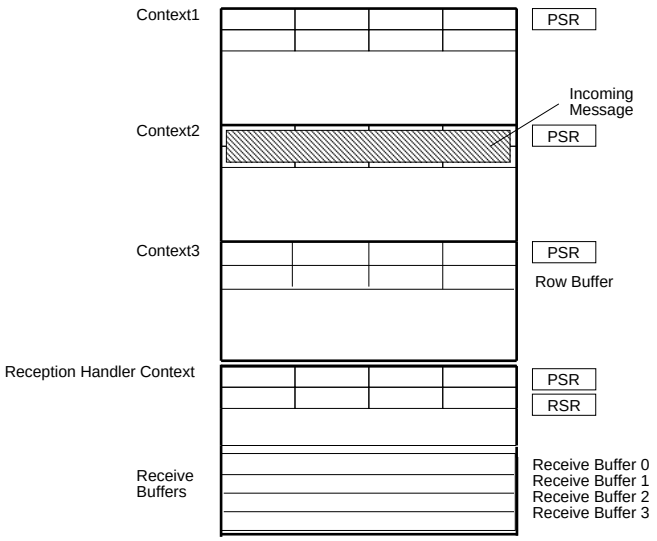


Figure 5: Register file organization

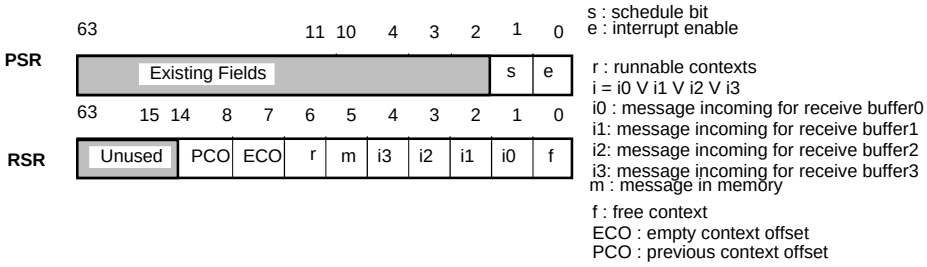


Figure 6: Special registers

Trap Handler Context A hardware context is dedicated to the trap handler thread, allowing fast message reception for special cases. This context includes message receive buffers for incoming messages, enabling the handler to utilize row mode operations to transfer the incoming messages between the cache and the receive buffers. Since the network interface may receive up to 4 messages simultaneously, four receive buffers are statically allocated for message reception. Each receive buffer is a row of 4 contiguous registers.

- **Reception Status Register (RSR)** The RSR shows the message reception status of the processor. When the *runnable context bit, r*, is 0, there is no ready context in the register file. It implies all the hardware contexts are waiting on memory access or waiting for the reception of a long message transfer. The *incoming message bits, i0, i1, i2, i3*, specify whether there are incoming messages in the four corresponding receive buffers. The *message in memory bit, m*, is set when there are unprocessed messages in the message overflow area in memory. The *free context bit, f*, shows whether there is an empty context which can receive a short message. The empty context offset (ECO) field contains the offset of an empty context in the register file. The field is controlled by the scheduling hardware. The previous context offset (PCO) field specifies the offset of the previously running context by which STT_ROW instruction (refer Appendix B) can identify the

source row register. Like ECO field, it is controlled by the scheduling hardware. The RSR can be read as a source integer operand but can not be changed by any instruction. The RSR is used by the trap handler to identify the trap condition as well as to determine the location of an empty context. Like PSR, the RSR is memory-mapped to system address space.

Scheduling A thread can be created by message send or by system. Since the DI-microprocessor has multiple hardware contexts, the context switching does not always imply the context save and restore. A context switching to other thread can occur on a cache miss, message overflow trap, I/O or at a thread completion. However, except the I/O, the context switching will not incur the context save and restore of the previously running context. Once a thread is scheduled in the hardware context, it will hold the context until the I/O occurs.

4 The Memory Interface

Memory hierarchy performance is a critical factor in local processing performance. The DI-microprocessor uses a specialized memory interface to generate and process packets for memory operations, synthesizing a high-performance memory hierarchy from the packet routing network. For the purposes of discussion, we assume a 2-D mesh topology and memory banks interleaved in sets of four. These choices are not essential to the architecture and may differ between implementations.

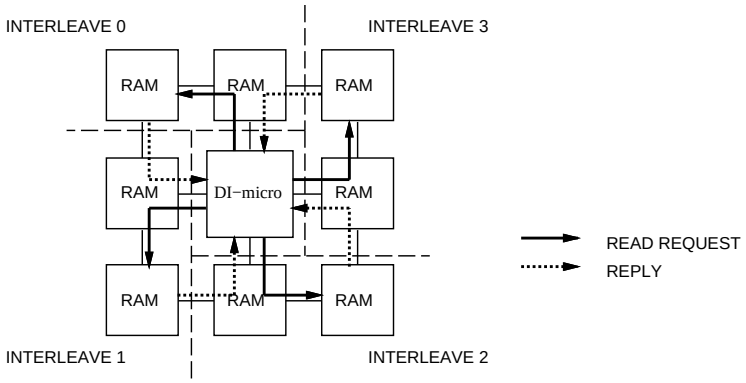


Figure 7: Memory operation example in DI-multicomputer node.

While previous fine-grained multicomputer chips could use packet routing to extend their memory hierarchies, the DI-microprocessor is unique in that it can send and receive multiple memory packets simultaneously. This hardware support is critical in synthesizing a memory hierarchy which makes efficient use of pin bandwidth. Specifically, the DI-microprocessor can 1) compose and send multiple packets and 2) receive and destructure multiple packets. Hardware support for the memory interface includes multi-packet formatting, multi-packet reception, and an extension to the TLB to identify the multiple memory node numbers.[7]

A sample memory transaction is shown in Figure 7. When a cache miss occurs, the processor loads a new line by sending request packets to a set of memory modules and storing the replies in the cache. To minimize reload time, four read request packets are sent simultaneously, saturating its four network channels. When the memory modules receive a `read request`, they respond with `read reply` packets which are destructured at the processor's packet interface and their data written into the cache. Write

[7] In addition, the router must have multiple source ports to accept and route multiple packets from/to the processor.

operations are performed in a simlar fashion, with all `write requests` being acknowledged by a `write reply` so the processor can detect write completion.

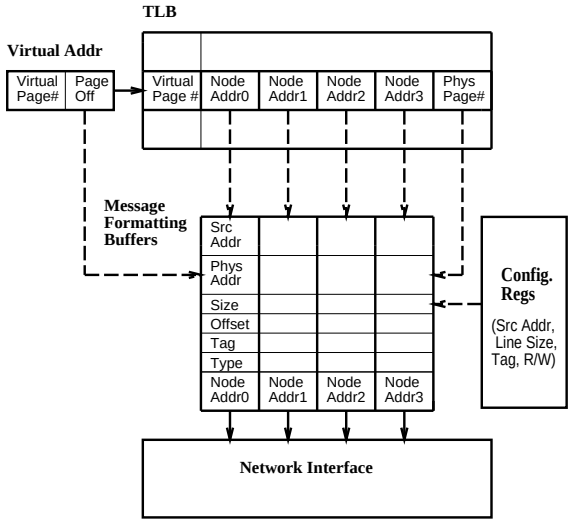


Figure 8: Multi-packet generation on a cache miss

Multi-Packet Send Hardware (Cache Refill Request) Figure 8 shows the hardware required for memory packet generation on a read miss. The node address, type, tag, offset, and size fields determine memory node, packet format, memory interleave number, memory request number[8], and line size respectively. The physical address field determines the memory location at the destination memory node. And the source address is used as a return address for the reply packets.

While forming and sending four arbitrary messages simultaneously would be difficult, a cache reload only requires the generation of four packets with basically the same content. The fields of the `read request` messages are derived from the instruction which caused the cache fault, configuration registers[9], and the translation look aside buffer (TLB) entry corresponding to the faulting address. Memory-packet generation requires a set of memory packet send buffers, and added fields in each TLB entry to address the appropriate memory nodes as shown in Figure 8. Our hardware design can format and launch the request messages within one CPU cycle following a cache miss. This high speed is possible based on the following optimizations: 1) TLB access is overlapped with the cache access and 2) tags, offsets, cache line size can all be written in advance. If the evicted cache line is dirty, a write-back operation will be initiated immediately following transmission of the read request packets, overlapping the writeback and read operations.

Multi-Packet Reception Hardware (Cache Refill Response) Each memory operation produces four reply messages which are identified, destructured, and sorted for presentation to the cache. Since there can be several outstanding memory requests; the reception hardware strips off the header and sorts packets by their tag and offset. Together, these two fields specify a unique location in the memory packet buffers as shown in Figure 9. All of the responses for a particular memory operation are mapped into a single row. Full rows can be written to the cache. The reception hardware uses a complete 4 by 16 interconnection to process any four memory packets simultaneously.

[8] The memory interface supports multiple outstanding memory requests. And this number identifies which outstanding memory request it is in order to match replies to requests.

[9] These define source node address, cache line size, etc.

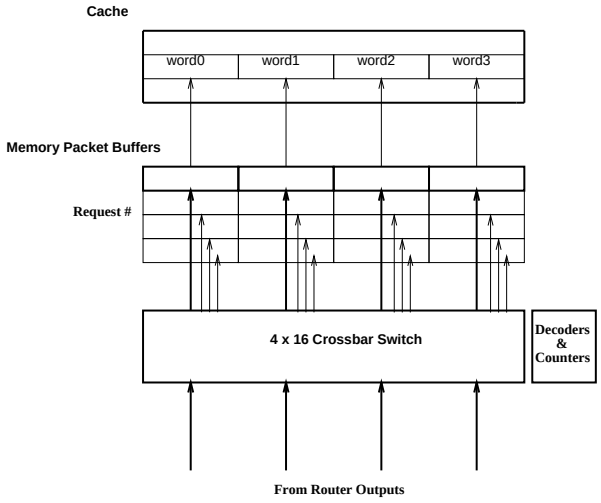


Figure 9: Multi-packet reception hardware

Though several fine-grained machines [16, 29] incorporate on-chip network interfaces, they typically lack high performance memory interfaces. Without multi-packet mechanisms the communication bandwidth of a single router channel is far less than a typical bus-based memory interface, providing insufficient bandwidth to compete with a bus-based memory hierarchy. For example, a router channel in the iWarp is 40 MB/s, far below the 400 MB/s memory bus bandwidth of its contemporary microprocessor (the i860XP). In such systems, packet routing provides insufficient bandwidth to supplant a bus-based memory hierarchy.

5 Message-passing Interface

Messages in the DI-multicomputer system can be classified into three different types: processor-to-memory messages, memory-to-memory messages and processor-to-processor messages, based on the source and destination of the messages. The processor-to-memory messages are transferred between a processor node and a memory node. These messages are called *memory messages* and handled by the memory interface of a processor or by the packet interface of a memory node. The creation, transmission and reception of the messages are entirely handled by hardware and hidden from a user. The other two types are used for interprocessor communication and synchronization. These are called *interprocessor messages*. The second type of messages are used to transfer a long message from local memory to remote memory. These are initiated by the user but handled by the packet interface of source and destination memory nodes. They are called *long messages*. The third type of messages are used to transfer a short message from source processor to destination processor. These messages are handled by the network interface and they are the only messages accessible at user level. Those are called *short messages*.

This section details how the creation, transmission and reception of interprocessor messages are handled by the message operations specified in Appendix B.

Effect of Computation Grain Size on Communication Communication among processors in multicomputers can occur at different levels of memory hierarchy, the choice of which affects the latency of communication and impact on local computation. In fact, the level of memory hierarchy at which the communication occurs determines the granularity of concurrency which can be exploited.

Fine-grained concurrency requires both low overhead and rapid response to short messages. To achieve such efficient interaction, short messages should be injected at a high level of the memory hierarchy such as on-chip registers or caches. Otherwise, the memory hierarchy traffic will increase the startup cost as well as the response time for the messages.

Medium and coarse-grained concurrency requires high bandwidth for long messages and overlapped computation and communication. To achieve a good performance, long message handling should be decoupled from the processor, transferring messages directly into memory. Sending and receiving long messages at the register level is inappropriate as it incurs transmission overhead and produces memory hierarchy traffic to move data up and down the memory hierarchy, increasing processing overhead. Moreover, long messages received directly by processor causes cache pollution, slowing the ongoing computation.

Existing machines do not address this *memory hierarchy traffic overhead* problem as they each attach the network to only one level of the memory hierarchy. The DI-multicomputer addresses this problem by allowing interprocessor messages to be directed to different levels of the memory hierarchy under software control. Thus, the DI-multicomputer can achieve both low-latency response for short messages and high-bandwidth transfers for long messages without cache pollution.

5.1 Short Messages

Message Transmission The DI-microprocessor uses its general purpose registers to form, send and receive messages. This approach allows low overhead message passing by eliminating memory operations moving with the memory hierarchy. To send a message, the processor issues a SEND instruction which specifies the register containing the first word and size of the message. The message should be contiguous in the local context as shown in Fig 10. For example, SEND R4, #5 transmits a five word message in registers R4, R5, R6, R7 and R8. The message size of short messages is limited by the size of the local context.

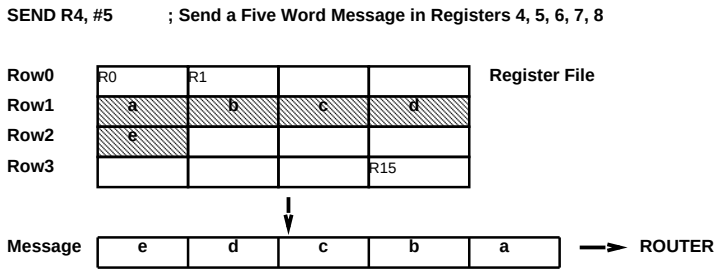


Figure 10: Multi-send vs. send operation

Message Reception There are two types of messages coming into a processor node in the DI-multicomputer system: memory messages and short messages. On a message reception, the destination node address is stripped off by the router, and the network interface decodes the packet type of the incoming message, distinguishing two different message types.[10] If the incoming message is identified as a short message type, the message is deposited directly into an empty context (register set) by the network interface. After the entire message is copied, the context can be scheduled by the hardware. Since the first word of the short message contains the instruction address of a context, when the context is scheduled, the message will be executed by jumping to that address. Memory messages are directed to memory receive buffers and handled by memory interface instead.

[10] The detail message packet types and formats used in the DI-multicomputer systems are described in Appendix A.

Trap Handling There are two exceptional cases which require special treatment. The first case occurs when there are incoming messages but no empty context in the register file (i0, i1, i2, or i3 = 1 and f = 0). It requires the trap handler to copy the messages from the receive buffers to the message overflow area in memory. This is called a *reception trap*. The second case occurs when all the active threads are blocked and there is an available context which can hold an unprocessed message in memory (r = 0 and m = 1 and f = 1). Instead of idling the processor, the trap handler is invoked to load an unprocessed message from memory to an empty context in the register file. This trap is called a *reload trap*. The loaded message can be scheduled by the scheduling hardware as the ordinary messages arrived from the network. These trap handling makes the message buffering for message overflow cases transparent to the scheduling mechanism.

Table II: Exception conditions and the corresponding state bits. The character X denotes the don't care condition. i denotes i0 or i1 or i2 or i3

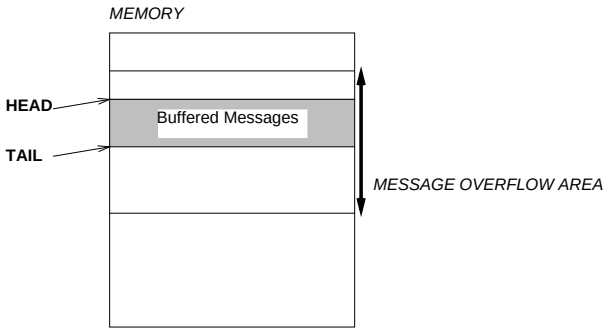
state bits				exception case description
f	i	m	r	
0	1	X	X	no empty context during message reception
1	X	1	0	no runnable contexts and unprocessed message in memory

The network interface handles these exceptional cases through a fast software trap mechanism. Since this trap handler is resident at one of the hardware contexts, we can achieve fast trap handling in the case of an exception, and the overhead of this software trap is just flushing the pipeline, which requires only a couple of CPU cycles.

Reception Trap If there is no available context in the register file, the incoming short message is copied into the corresponding receive buffer row by row by the network interface. After the first four words of the message is copied into the receive buffer, the message incoming bit associated with the receive buffer is set by the interface. At the same time, the message reception trap occurs and context switches to the trap handler context. These messages are called *overflow messages* and copied into message overflow area in memory by the trap handler as shown in Figure 11. The trap handler checks the status of message reception and stores the message into memory row by row. The trap handler includes the four receive buffers as part of its local context (row 4, 5, 6, 7) so that it can use ST_ROW instruction to buffer the incoming messages in memory. The message incoming bit remains set if there is a remaining part of the message or more incoming messages. The trap handling continues until there is no more incoming message, which is indicated by the message incoming bits. Each time more than one messages can be received by the trap handler. The message incoming bits are cleared by the network interface when there are no more incoming messages. At this time, the trap handling stops and context switches to the previously running context. The reception trap handler only needs to check the message incoming bits to stop the trap handling. In addition to the message incoming bit, each receive buffer has an additional full-empty bit to provide flow control for reception. When the network interface copies a row of message from router into a receive buffer, it sets the full-empty bit of the corresponding receive buffer. And ST_ROW operation from the receive buffer clears this bit. In other words, reads or writes to these receive buffers has additional semantics for flow control. Read from empty receive buffers will block until the buffer is full, writes to full buffers will block until a read occurs.

The Figure 11 illustrates the reception handling for the reception trap. In the example code shown in Figure 11, the handler manages the message overflow area as a circular buffer[11]. The handler checks the message incoming bit for each receive buffer and copies the message if one is present.

[11] The codes for boundary checking is not included in the example




```
/* Reception Trap Code */

RECEIVE0:  BLBS      R15, CREATE          ; if  f = 1, then handle 'reload trap'
           SRA       R15, 1, R14
LOOP0:     BLBC      R14, RECEIVE1        ; check receive buffer0 (if i0 = 1)
           ST_ROW    ROW4, R3             ; MEM[TAIL++] <-- receive buffer0
           ADD       R3, 4, R3            ; R3 is a TAIL
           BR        LOOP0                ; continue reception from receive buffer0

RECEIVE1:  SRA       R15, 1, R14
LOOP1:     BLBC      R14, RECEIVE2        ; check receive buffer1(if i1 = 1)
           ST_ROW    ROW5, R3             ; MEM[TAIL++] <-- receive buffer1
           ADD       R3, 4, R3
           BR        LOOP1

RECEIVE2:
           ....

RECEIVE3:  SRA       R15, 1, R14
LOOP3:     BLBC      R14, RECEIVE         ; check receive buffer3(if i3 = 0)
           ST_ROW    ROW5, R3
           ADD       R3, 4, R3
           BR        LOOP3

/* Reload Trap Code */

CREATE:    BLBC      R15, DONE            ; if f = 0, then DONE
           LDQ       R5, 1(R2)            ; load the message size, R2 is a HEAD
           LDT_ROW   ROW0, R2             ; load the message into an empty context
           ADD       R5, 4, R5            ; R5 = message size
           BLT       R5, DONE             ; if message size <= 4, then DONE
           ADD       R2, 4, R2
           LDT_ROW   ROW1, R2
           ADD       R5, 4, R5
           BLT       R5, DONE             ; if message size <= 8, then DONE
           ....
DONE:
```

Figure 11: Handler codes for both reception trap and reload trap

Since an incoming message can create a new thread even if there are unprocessed messages in memory, messages are not executed in the order in which they arrive. However, this reduces the frequency of the message buffering, resulting in faster message reception.

Reload Trap On the second exceptional case, the trap handler loads a message from memory into the register, executing them just as if they had come directly from the network. The trap handler should be able to specify the registers of other contexts to copy a message into an empty context directly. A special row mode operation called LDT_ROW is used for this purpose, i.e. to load a row of words in memory into an empty context. Instead of using the local context offset, the instruction uses the ECO (empty context offset) in RSR for row operand. The trap handler routine shown in Figure 11 loads a message from memory into an empty context and creates a new thread.

5.2 Long Messages

The packet-based memory interface of the DI-multicomputer supports the PUT/GET primitives of shared-memory libraries [12] directly in hardware. This allows a high-bandwidth and low overhead

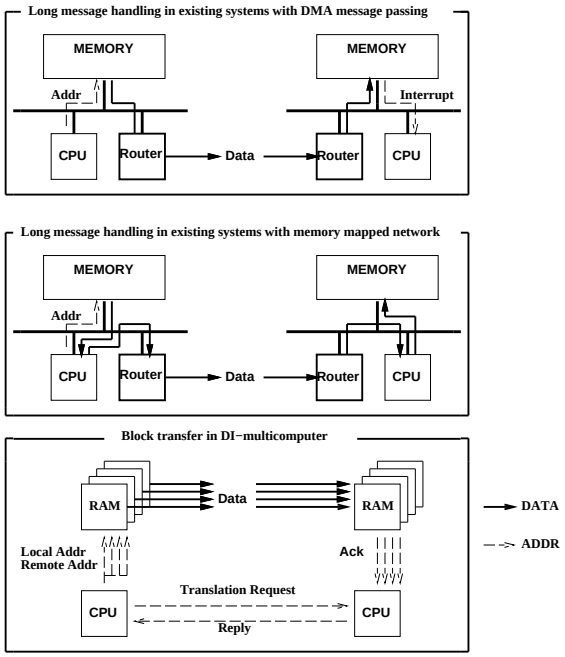


Figure 12: Long message transfer in the DI-multicomputer versus traditional multicomputer architectures. For clarity, the buffer request messages in the traditional architectures are not shown.

communication for long messages for both internode (PUT/GET) and intra-node (local memory-to-memory copy) communication.

Figure 12 contrasts the data movement (PUT) for a long message transfer in the DI-multicomputer with those of other multicomputer node architectures. The primary drawbacks of both bus-based architectures are unnecessary data movement in the memory hierarchy, the use of only one network channel, and the data transfer across the bus once or several times at both the source and destination nodes. These characteristics reduce message passing and local computation performance. In contrast, the DI-multicomputer transfers messages from local to remote memory in parallel with computation. The packet-based memory interfaces allow direct inter-memory transfer and the parallel interconnect provides parallel communication paths, enabling higher bandwidth transmission.

Transmission of a long message is achieved in two steps: 1) allocation of a buffer at the remote node and 2) a parallel block transfer into that buffer. Buffer allocation is achieved using short messages, and block transfers are achieved using a MOVE instruction.

Address Translation and Buffer Allocation Since address translations are not shared and the source node has no information about the availability and the location of memory space at the destination node, before sending a long message, the source processor needs to request the buffer allocation as well as the physical address for the buffer. It is accomplished by sending a short message called translation request message to the destination processor.

The code sequences for long message transmission is illustrated with message contents in Figure 13. To initiate a long block transfer, first, the sending thread initiates a translation request message including the virtual address for the remote node and the size of the block transferred. To send a long message, a source node first sends a short message, requesting a buffer at the destination node, which returns the physical buffer address (four addresses, one for each interleave) as a reply. The buffer allocation is used for flow control for large messages, and it also allows the sender to achieve direct long message transfer

by writing data into the receiver node's memory.

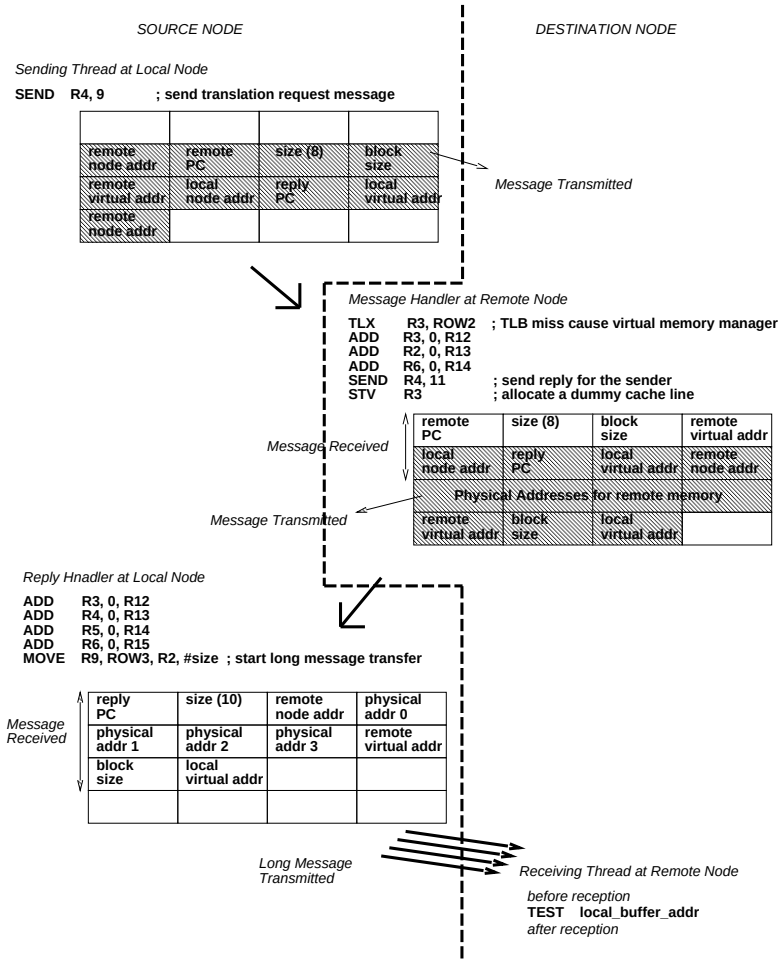


Figure 13: Handler codes for long message transmission

At the destination, it creates a handler thread, which allocates the buffer space for the message if it has not been allocated and replies with the physical address of the buffer at the destination. The address translation at the remote node is done by using TLX instruction (see Appendix B). Since the memory is 4-way interleaved, TLB returns multiple physical addresses for the 4 interleaved memory nodes. The virtual memory manager needs to guarantee that the the remote memory buffer will not span two sets of memory nodes so that a single memory to memory transfer will suffice.

Synchronization of Long Message Reception When the remote node handles the translation, it also allocates a dummy cache line using the virtual store operation, STV, to receive the acknowledgments from memory for later long message reception. The receiving thread at the destination can check the status of the long message reception by polling the status of the cache line using a TEST operation for the remote buffer address. If the cache line contains all the acknowledgments from its local memory nodes, the message reception is done and the receiving thread can access the block. Otherwise, the instruction will return a failure signal (cache miss) until all the acknowledgments arrive.

Block Transfer Request Packet Transmission On reception of this translation reply message, the reply handler performs a MOVE instruction, which transmits four MOVE requests to the four local memory nodes. The TLB provides the local memory addresses and the row operand in the MOVE instruction provides the remote memory addresses for the MOVE request packets. The instruction initiates the TLB translation for the local address and generates the physical memory addresses which contain the four memory node numbers. Similar to the memory request message generation on a cache miss, the memory interface generates four memory MOVE request packets using the memory packet generation buffers. Instead of using the source node address as a return address, the MOVE message includes the destination processor number and remote memory address as the return address.

Intermemory Block Transfer On receiving the MOVE request, each memory node generates a *write* request packet with appropriate data and sends it to the appropriate remote memory node directly. This is like a parallel DMA transfer. Because of the parallel memory banks and interconnect, a four-fold speedup is possible. On remote memory nodes, the incoming data are stored and write acknowledgment messages are sent to the destination remote processor.

The synchronization of acknowledgment packets is achieved with the dummy cache line that is allocated at the time of buffer allocation. On the reception of four acknowledgments from the memory nodes, a destination node thread blocked on a TEST operation can resume execution.

The DI-multicomputer's long message transmission mechanism has several advantages. First, one MOVE instruction causes four memory banks to transmit at the full network rate, exploiting the parallel interconnect with minimal instruction execution overhead. Second, the long message transmission mechanism does not complicate the memory nodes as the MOVE request packets are handled identically to local memory access packets[12]. Finally, the synchronization of long message arrival occurs in the receiving processor's on-chip cache not in the off-chip memory, minimizing the synchronization latency.

6 Performance Evaluation

In this section, we study the performance of the DI-multicomputer's memory hierarchy and communication subsystem, comparing them to existing message passing machines. The goal is to explore if the DI-multicomputer's novel memory communication primitives translate into better application performance. By comparing cache reload times, we show that the superior bandwidth of dynamic interconnection can produce a higher performance memory hierarchy than a bus-based approach. By using trace-driven simulation, we compare execution time and message handling overhead on the DI-multicomputer to several existing multicomputer architectures. These studies show that the DI-multicomputer's mechanisms produce significant performance benefits, even on modest size problems and machines.

6.1 Memory Interface Performance

A memory hierarchy based on dynamic interconnection can match and in some cases outperform bus-based approaches. As a case study, we compare the bus-based memory interface of the Intel i860XP [24] to our DI-microprocessor memory interface. To make the comparison fair, we assume the processors have the same internals, with single level on-chip cache and approximately the same number of I/O pins (see Figure 14). The i860XP's memory interface uses 139 pins: 64 data lines, 29 address lines, and 46 lines for parity and bus control. The DI-microprocessor uses a 152 pins: four 38 line channels with 32-bit bidirectional data links, and six control lines for parity and control each.

To evaluate memory interface performance, we compare cache refill times over a range of line sizes. The performance numbers assumed for the calculation are shown in Table III and they are derived from our hardware design studies [4] including SPICE simulations of multi-tap bus lines. We further assume

[12] The only difference lies in that the destination addresses for reply packets become remote memory nodes instead of the local processor.

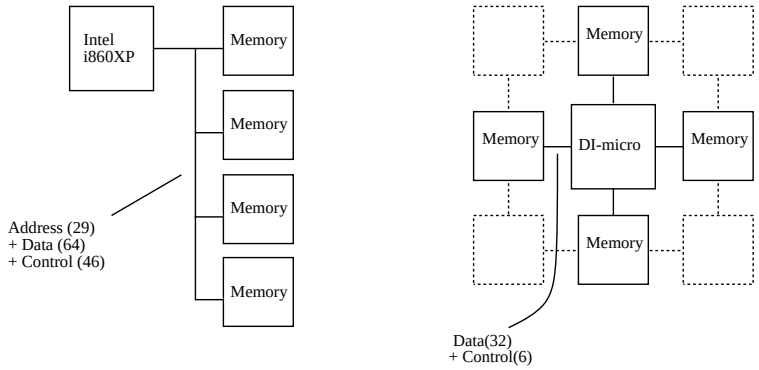


Figure 14: Bus-based (i860XP) and DI-based systems.

that the DI-microprocessor reloads its cache lines from its four nearest neighbors, minimizing the routing delay. The higher network clock rate for the DI-microprocessor memory interface is due to the electrical advantages of point-to-point interconnects over multi-tap bus lines [31, 18]. Router delay is based on a number of published implementation studies [17, 32] and our own designs [4, 8]. The actions required to complete a cache line reload in each system are illustrated in Figure 15.

Table III: Memory and interconnect performance numbers assumed for the evaluation.

Architecture component	Characteristics	Performance number assumed
Memory Module	Access Time	20 ns
Network	Switching speed	200 MHz (5 ns cycle time)
	Routing Delay	20 ns (4 cycles per hop)
Bus	Switching Speed	50 MHz (20 ns cycle time)

The memory node processes a request packet in six network cycles: two to strip the header and extract the the physical address, four to access the memory and form a reply. [13] Slower memories affect only the memory access time. For example, with 60 ns memory access time the reload times for both the DI-microprocessor and the i860XP would increase by 40 ns.

The DI-based memory interface has a larger cache refill time for a small cache line (32 bytes) due to

[13] Off-chip secondary caches can be implemented by SRAM memory modules in organization similar to Figure 14.

Table IV: Cache reload times for the bus-based and DI-based systems.

Line size	DI-micro(32bit links)		i860XP(64bit bus)		i860XP(128bit bus)		i860XP(256bit bus)	
	Cycles	Time(ns)	Cycles	Time(ns)	Cycles	Time(ns)	Cycles	Time(ns)
32 bytes	29	145	6	120	4	80	3	60
64 bytes	31	155	10	200	6	120	4	80
128 bytes	35	175	18	360	10	200	6	120
256 bytes	43	215	34	680	18	360	10	200

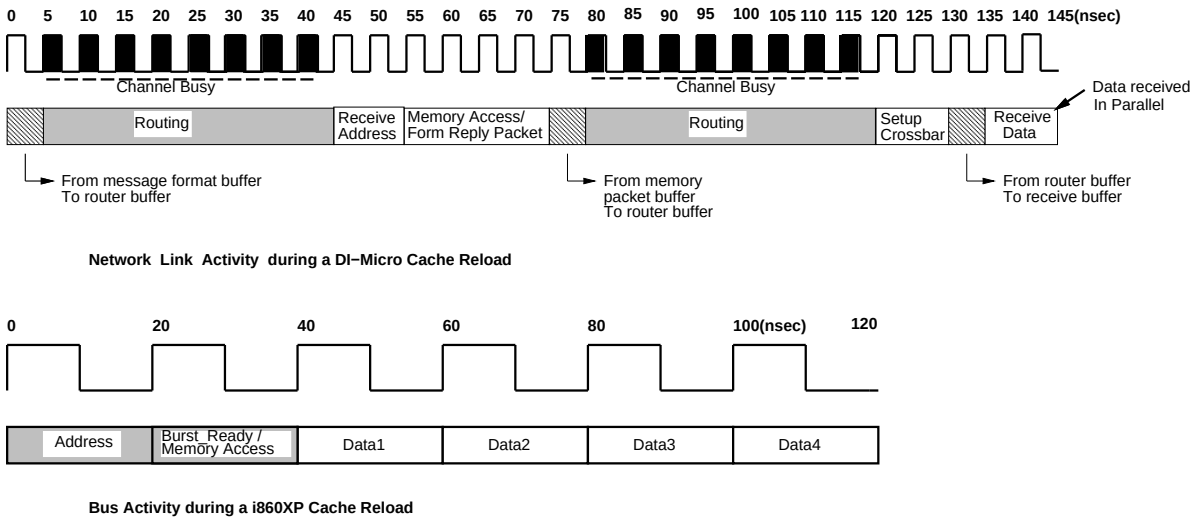


Figure 15: Cache line reload for DI-based and bus-based systems.

its increased routing delay. But for large cache lines, the higher data bandwidth of the DI-based memory interface masks the latency incurred by packet routing, and outperforms the i860XP bus (see Table IV). Doubling the data bus width of the i860XP would make the i860XP competitive for larger line sizes, but doing so requires dramatically larger numbers of pins, often expensive resources.

Table V: Bandwidth of two memory interfaces. The signal pins for i860XP include only memory interface pins.

	DI-microprocessor (32-bit channels)	i860XP (64-bit bus)
# of Signal pins (data)	152 (128)	139 (64)
Clock rate	200 Mhz	50 Mhz
Signal pin bandwidth	3.8 GB/s	0.87 GB/s
Data/Signal ratio	84.2 %	46 %
Peak memory bandwidth		
32 byte lines	0.91 GB/s	0.32 GB/s
256 byte lines	2.44 GB/s	0.39 GB/s

The major performance benefit of a DI-based interface is the increased bandwidth derived from higher signalling speeds and flexible pin allocation. In Table V, we compare the available pin bandwidth and utilization for memory transfers. The DI-microprocessor's data rates are two to six times those of the bus-based system. The peak memory transfer rates in Table V show the highest data rates achievable with overlapped transfers of the specified line size (addressing and header overhead deducted). An excess of memory bandwidth allows memory hierarchies to be managed aggressively: making the advanced memory system techniques such as data prefetching and multithreading more attractive.

6.2 Parallel Computer Performance

In this section, we compare the message passing performance of the DI-multicomputer to the existing fine-grained and coarse-grained multicomputer architectures. Trace-driven simulation using iPSC/2 traces [23] are used to address the following questions: (1) How much do the DI-multicomputer communication primitives speed up applications?, (2) How much performance improvement do the distinct mechanisms for short and long messages give?, and (3) How do machine size (number of processors) and application granularity affect these tradeoffs? Our results show that by eliminating most of message handling overhead, the DI-multicomputer increases overall application performance by 7% to 485%. Distinct message passing mechanisms for short and long messages reduce memory hierarchy traffic, improving application performance by 5% to 77% compared to register based message handling. These results argue for network interfaces that can direct messages either to the processor or to the memory as appropriate. Such distinct mechanisms allow a multicomputer node to give good performance for both coarse and fine-grained applications. The DI-multicomputer's mechanisms also give more robust performance for a variety of system size and data sets.

Experimentation Methodology The trace driven simulation is chosen as an experimentation method since it allows the simulation under real work load. Based on the machine dependent parameters such as processor speed, network implementation and message passing cost, etc., the simulator can simulate different distributed memory machines and provides an experimentation framework that allows the comparative evaluation of the DI-multicomputer to existing distributed memory machines.

The performance impacts of the DI-multicomputer mechanisms depend on several factors such as target machine architectures, communication characteristics of the applications, and the grain sizes of the data sets used. To validate this experimentation, traces are collected both for different machine sizes and for different data sets. Also, two existing target coarse-grained multicomputer architectures, Intel iPSC/2 and Delta, as well as a fine-grained multicomputer architecture similar to the J-Machine are compared to the DI-multicomputer.

The performance evaluation focuses on the performance comparison of communication architectures of different multicomputer organizations. Neither the implementation technology nor relative computation speeds are considered for the comparison. To achieve this, we performed the simulation as follows. First, the message passing traces consist of message send/receive events and computation times in-between, all of which are directly measured by the application runs from the iPSC/2. By taking the computation time directly from the traces, all multicomputer architectures assume the same computation performance. Therefore, the local memory system performance is also not considered for the performance comparison. Second, to factor out the impact of different implementation technology used in different multicomputer architectures, the simulations of DI-multicomputer architectures assume the same network implementation (the same network bandwidth and the same routing latency per hop) as the target machine architectures such as the iPSC/2 or Touchstone Delta.

Simulation Model The execution time of parallel programs can be divided into computation time, communication time and idle time due to the synchronization. Even though the machine can support ideal communication performance (zero overhead and zero latency communication), there is an idle time due to the synchronization among the multiple processes running on the different nodes in the machine.

In the simulation, the computation time is derived from the traces while the software and hardware latencies of the communication and the idle time due to the synchronization are modeled by the simulator. In the current simulation, network load is assumed to be zero, i.e. no network traffic is modeled.

Applications The communication traces are collected from the following seven parallel applications (2 VLSI CAD applications, 4 numerical applications and 1 event-driven simulator). The applications are described in Table VI. The traces are derived from [23].

Table VI: The description of parallel applications that are used for simulation.

Application	Description
Fast Fourier Transform (FFT)	Parallel implementation of discrete fourier transform
Standard Cell Placement (PLACE)	Cell placement program based on simulated annealing
QR Factorization (QR)	Parallel version of QR factorization algorithm
Gaussian Elimination (GAUSS)	Parallel gaussian elimination based on tree broadcasting
Hypercube Router (ROUTER)	Parallel event-driven simulation of hypercube
VLSI Circuit Extraction (EXTRACT)	Circuit extraction based on two phase data distribution
Eigenvalue Computation (TRED)	Generated by semi-automatic parallelizing technique, ignoring communication cost

Table VII: Grain size and message passing overhead for different applications. The data is collected from a trace-driven simulation of a 16-node iPSC/2. The programs FFT and QR are simulated with two data sets: 2^{12} (l) and 2^8 (s) for FFT, and 128 x 128 (l) and 64 x 64 (s) for QR.

Applications	# of Comm.	Avg. Msg. Size	Total Msg. Vol.	Instructions Per Message	Comm. Traffic (Byte/1K Instr.)	Processor Utilization
FFT (l)	800	2048	1638.4KB	70950	28.87	88.4%
FFT (s)	800	128	102.4KB	4088	31.31	58.0%
PLACE	15889	87	1382.3KB	2533	34.35	7.5%
EXTRACT	1626	5656	9196.7KB	154982	36.49	70.4%
TRED	11165	120	1339.8KB	469	255.86	9.5%
ROUTER	10537	10	105.4KB	623	16.05	26.4%
GAUSS	1869	210	392.5KB	3060	68.63	44.7%
QR (l)	2032	1026	2084.8KB	7067	145.18	53.5%
QR (s)	504	514	259.1KB	3730	137.8	40.9%

Table VII summarizes the communication characteristics of the above applications. *Number of communications* shows the total number of messages sent and received, counting each send or receive as a separate communication event. *Instructions per message* presents the number of user-level instructions per message, showing the grain size of the application traces. *Communication traffic* shows the message volume transmitted or received per a thousand user-level instructions, showing the communication to computation ratio. And the last column, *processor utilization*, shows the ratio of computation time to total execution time in the iPSC/2.

Speedup over iPSC/2 and Touchstone Delta We present simulation results comparing the DI-multicomputer to the iPSC/2 and the Touchstone Delta, showing the speedup due to the DI-multicomputer communication mechanisms. Figures 16 and 17 compare the simulation results for a 16-node DI-multicomputer to the iPSC/2 and the Touchstone Delta. [14] The *Ideal* architecture is a point of reference which denotes the minimum communication time, assuming zero overhead and zero latency communication. It corresponds to the maximum performance improvement possible by improving communication performance.

[14] Our DI-multicomputer uses a mesh-based packet-routing network, but the iPSC/2 uses a circuit-switched hypercube network. To minimize the impact of network topology, a mesh-based packet routing network is used for simulation of both systems. But, we use the same network bandwidth and the same routing latency per hop as the original iPSC/2 network for the mesh network. For a fair comparison, we also assume a conservative communication distance (twice the communication distance of the original trace) for the mesh network.

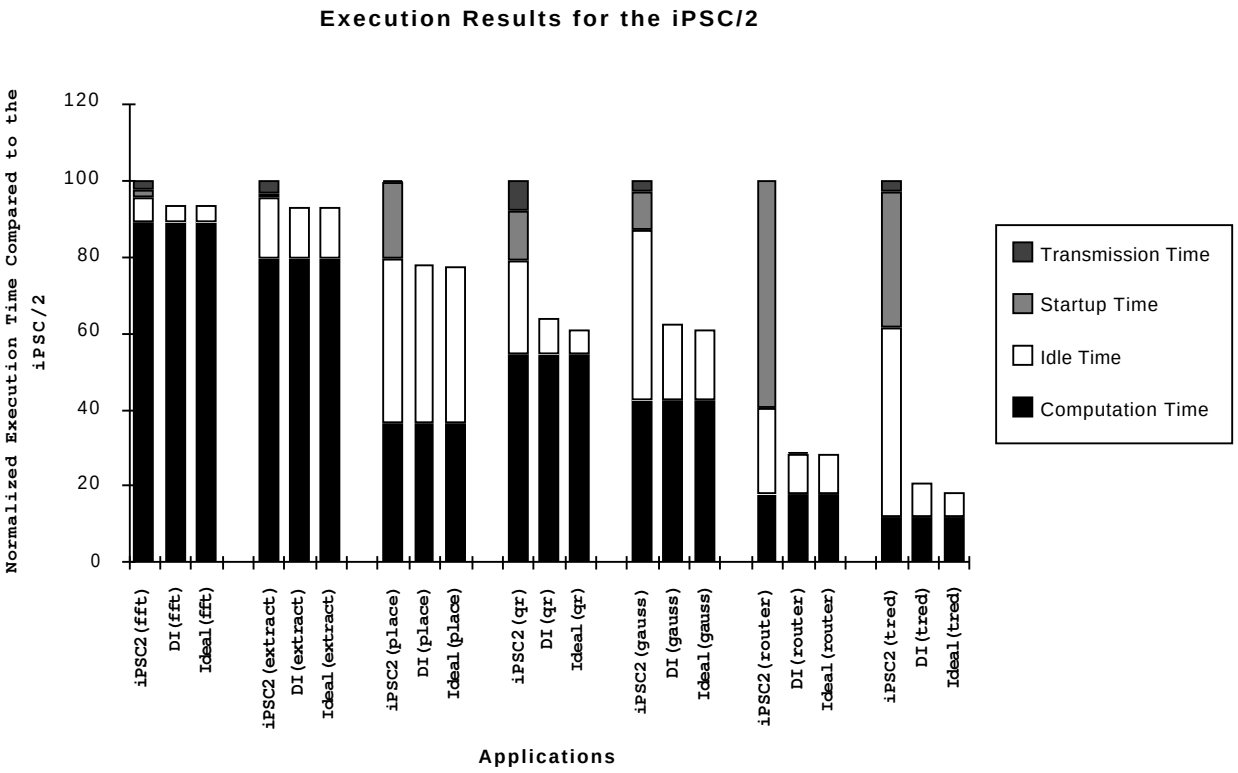


Figure 16: Execution results for 16 nodes iPSC/2

In the figures, application execution time is divided into computation and communication time. The communication time consists of both processing overhead for message passing (fixed startup overhead plus transmission time) and idle time due to synchronization and load imbalance. In the simulation, the computation times are derived from the traces while the message passing overhead and idle time are measured by simulation.

For each application, the DI-multicomputer eliminates almost all processing overhead for communication. This is because the DI-multicomputer's communication mechanisms eliminate or decouple message handling overhead from the processor. Short messages are handled on-chip with small processing overhead. Long messages incur little processing overhead as they are transferred memory to memory directly. In FFT, EXTRACT, QR, GAUSS and TRED which deal with only long messages, the DI-multicomputer completely eliminates the messaging overhead. However, because its communication latency is still non-zero, the DI-multicomputer still spends more time idling in QR, GAUSS and TRED than the Ideal architecture. For FFT, PLACE and EXTRACT, idle time is approximately constant across the different architectures. This idle time is dominated by load imbalance, not communication latency.

Overall, the DI-multicomputer achieves a 7% to 79% reduction in total execution time, a 1.1 to 4.9 times speedup due to its improved message-passing performance. In most of the applications, the DI-multicomputer nearly matches the performance of the Ideal architecture. In general, the speedup over the target architecture is more pronounced when the application is communication-intensive (ROUTER and TRED; see Table VII) or when the application is tightly synchronized (QR and GAUSS).

To further evaluate the DI-multicomputer's communication mechanisms, we performed a trace-driven simulation of a second target machine, the Touchstone Delta. Although this machine has a much faster

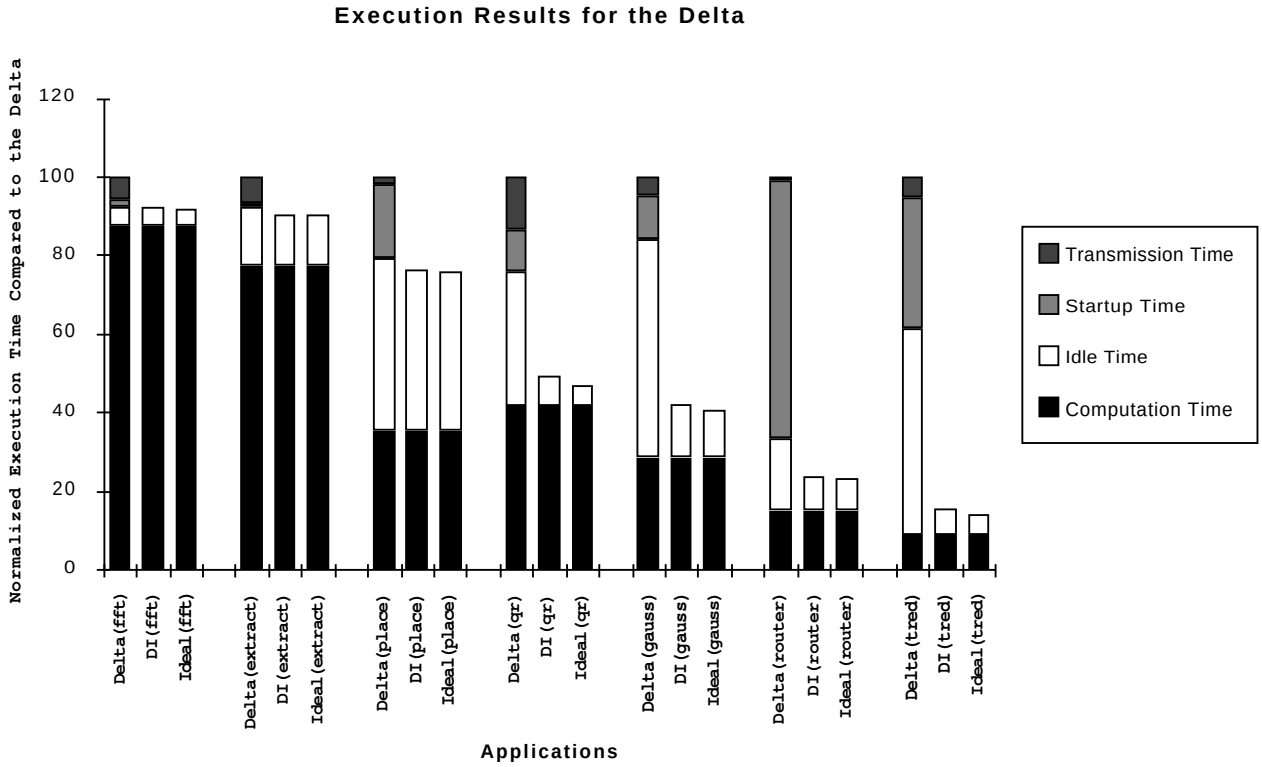


Figure 17: Execution results for 16 nodes Touchstone Delta

CPU than the iPSC/2, its ratio of computation to communication performance is approximately the same. Figure 17 shows the execution results for the Delta and the DI-multicomputer assuming the same computation speed and the network implementation. The execution time distribution as well as the relative speedup of the DI-multicomputer over the target machine gives almost identical as in the iPSC/2 simulation. Overall, applications become slightly more communication intensive, reflecting the fact that the architecture's ratio of computation to communication performance has increased slightly.

Speedup versus Register Based Message Handling To study the performance impact of the memory hierarchy traffic overhead of the register based message handling, we simulate an another architecture, called Short architecture, which is similar to several fine-grained architectures [15, 29] and has user level message handling based on the register file. And we compare its result to that of the DI-multicomputer. In this experiment, since message passing is performed at user level in both architectures, we eliminate the effect of the software startup cost and are able to factor out the memory hierarchy traffic overhead of the register based message handling. Figure 18 shows the simulation result of the Short architecture compared to that of the DI-multicomputer. In the Figure 18, we count instruction execution overhead for message load, message store and message transmission as transmission overhead. Therefore, the differences in transmission time between both architectures accounts for added memory hierarchy overhead of the Short architecture. Even though register-file based message passing can reduce the startup overhead of message passing, it increases the instruction execution overhead due to transmission and memory hierarchy traffic. As a result, especially for the applications with large amounts of communication traffic (QR, TRED and ROUTER, see Figure VII), processing overhead for the memory hierarchy traffic becomes more significant in the Short architecture. On the other hand, the

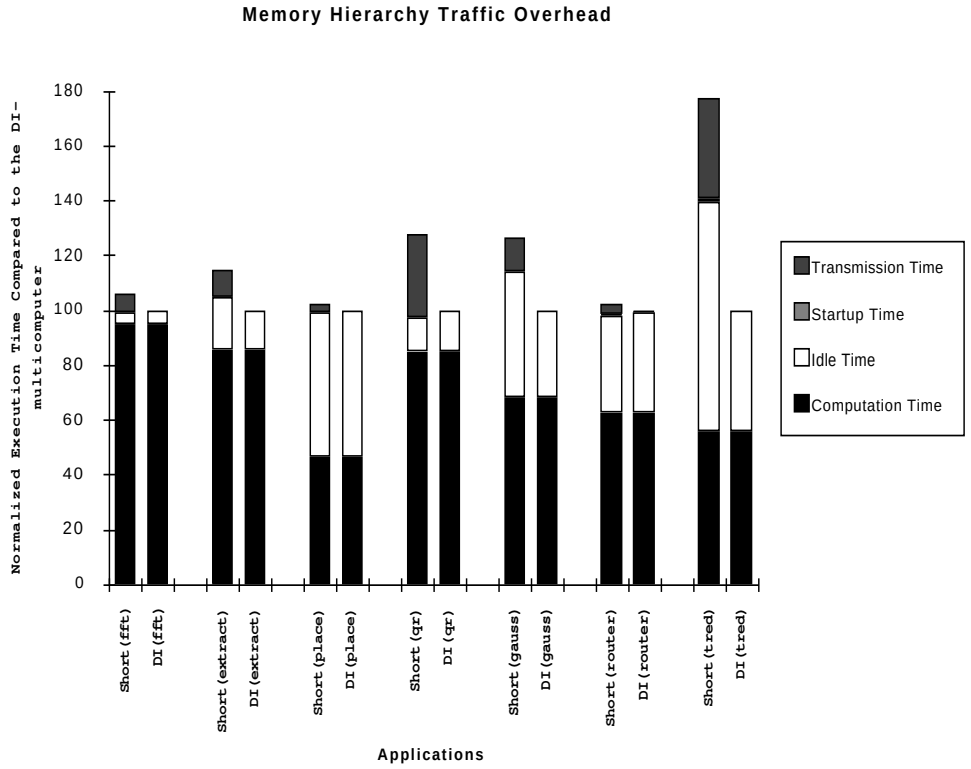


Figure 18: Effect of memory hierarchy traffic overhead. Comparison to a fine-grained Architecture

DI-multicomputer can eliminate this overhead completely for long messages since they are transferred directly from memory to memory. Note that the applications PLACE and TRED mostly deal with short messages. Since both architectures handle short messages based on the register file, the performance difference is minimal. For those two applications, the DI-multicomputer still performs better due to its higher bandwidth communication. For applications GAUSS and TRED, note that the memory hierarchy traffic also increases the idle time, further increasing the communication overhead. However, by mapping messages to an appropriate level of memory hierarchy, the DI-multicomputer can reduce the total execution time of the applications by 5% to 77% relative to the Short architecture which uses register-file based message handling exclusively. Considering the poor memory bandwidth of existing fine-grained architectures due to the pin limitation problem (see Section 2), the performance difference between the two architectures is even more pronounced.

Effect of Machine Size and Application Granularity Scalable performance is a major goal of multicomputer design. In this section, we examine how communication performance affect application and machine scalability. Figure 19 shows the distribution of total execution time for three different applications for smaller data sets. The smaller data set makes the applications more communication intensive, increasing the importance of efficient communication mechanisms. This is reflected in the increasing speedup of the DI-multicomputer over the iPSC/2.

Another way to examine the same issue is to fix the application data set size and scale the number of processors. Figure 20 shows the relative speedup achieved by the DI-multicomputer and iPSC/2 (normalized to 4-node iPSC/2 performance) for three applications as machine size is increased. Increasing machine size usually increases communication overhead, expanding the performance gap between the DI-

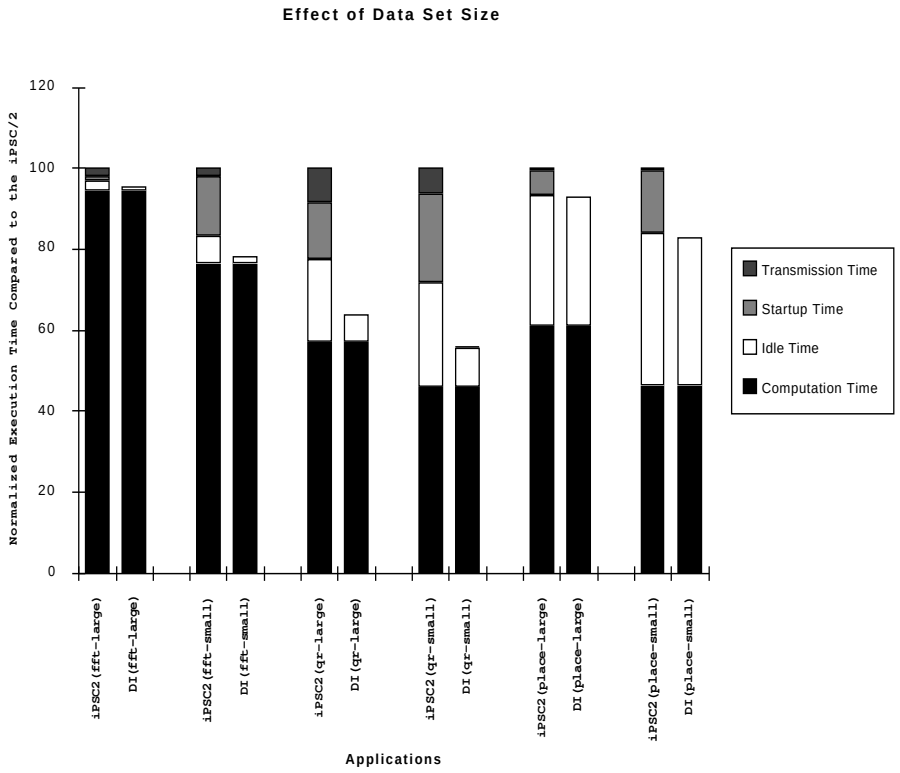


Figure 19: Execution time versus data sizes for applications FFT, QR and PLACE. Two data sets are shown for each application, showing the smaller data set to the right.

multicomputer and the iPSC/2. With low overhead communication mechanisms, the DI-multicomputer improves the scalability of the applications and approaches the performance of the Ideal architectures. In particular, when running GAUSS, the 16-node iPSC/2 shows poor performance as communication overhead starts to dominate the advantages gained by parallelism. Unfortunately, because the traces are only for 16 processors, we currently cannot extend our studies beyond this point.

7 Implementation Issues

All of the processor extensions discussed in this paper have been designed and simulated at the gate or transistor level. Integration of the processor core with a network interface and router has been explored in the iWARP [29] and MDP [16]. We are currently exploring a variety of implementation approaches (gate-array, standard cell, and full custom VLSI). In this section, we briefly discuss the additional hardware requirements and likely speed of these implementations.

Memory Packet Interface The send hardware requires eight 64-bit registers, less than 100 simple gates for control, and an extension of the TLB. The multi-packet receive hardware includes a 4 by 16 crossbar which sorts the replies, sixteen 64 bit registers, and some control logic. Because the control information arrives one network cycle before the data, the crossbar can be set up in time to allow the data word to flow through to the appropriate register in a single cycle. Based on our design studies, the additional hardware required for both the multi-packet send and reception hardware is approximately

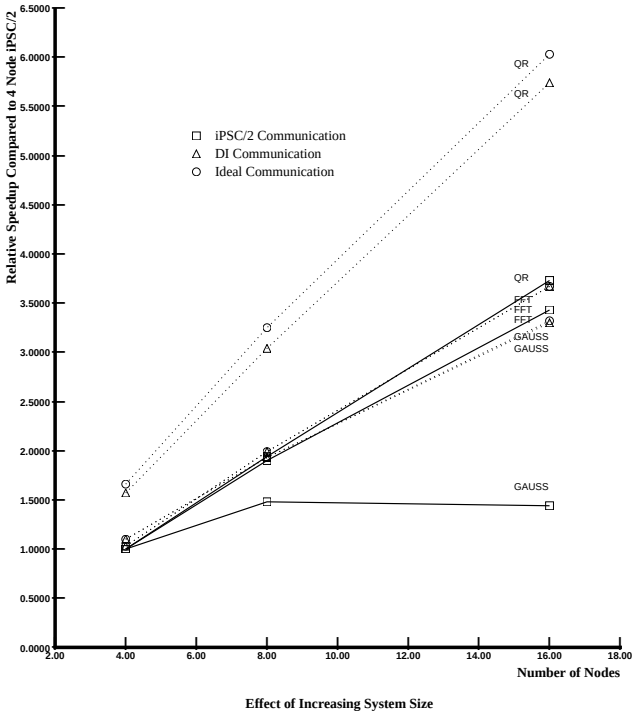


Figure 20: Effect of increasing system size for application FFT, QR and GAUSS

26,000 transistors, representing $\approx 0.6\%$ of the transistor count in a modern microprocessor.

Message Passing interface The short message handling mechanism requires hardware support to copy short messages to and from the register file. Since much of the packet reception hardware such as the crossbar switch and control for path setup is shared with the memory packet reception units, the hardware required for short message handling is only message buffers plus control logic in the network interface. And we expect the implementation of the message passing interface to be simpler than that used in conventional bus-based systems since the uniform interconnect of dynamic interconnection obviates the manipulation of complex bus protocols. We expect the hardware overhead of the network interface to be less than that for the memory packet receive hardware, representing less than 0.5% of the processor chip die area. Long message passing does not require any significant additional hardware since the mechanism is embedded in the memory subsystem.

Packet Router Dynamic interconnection demands routers with three characteristics: First, they must exhibit extremely low latency. Our implementation studies and others [4, 8, 17, 33] show that router latencies in the 5-10 nanosecond range are feasible if chip crossing costs are reduced with advanced packaging. Second, it must have multiple input and output ports, supplying or absorbing several packets at a time. Network implementation studies have shown that wormhole routers can be augmented with additional input and output ports with modest increases in complexity. Third, it must be possible to inject messages into several dimensions. For example, if the network used dimension-order routing, all packets would have to route in X first, making it impossible to inject in the Y dimension. Recent advances in the design of simple adaptive routers indicates that such networks overhead likely to be modest [4, 8]. A comparable router design we completed required only 18 $M\lambda^2$, $\approx 1\%$ of a modern microprocessor's die area. Also, our design study of three adaptive routing schemes [5, 9, 28] shows that

they all require less than 10,000 gates for 2-dimensional network [8]. Finally, they must use bidirectional channels. This allows all of the processor pins to be utilized for dynamic interconnections. A number of routers which use bidirectional signalling have already been constructed [5, 17].

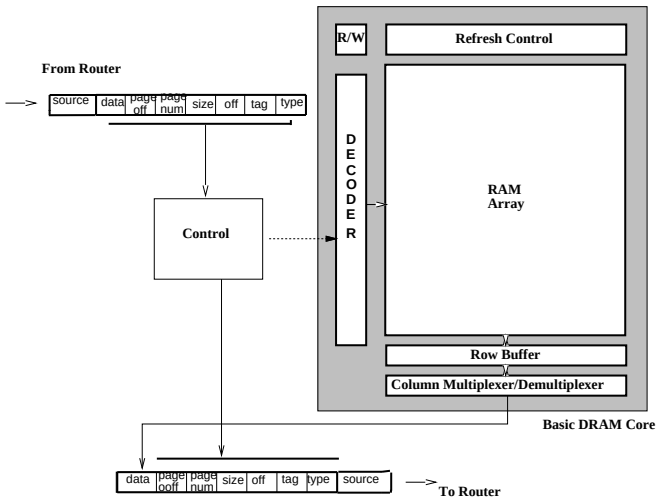


Figure 21: Memory node packet interface in dynamic interconnection

Memory Nodes Dynamic interconnection memory nodes require a router and packet processing hardware. This hardware can be added to individual dynamic RAM chips, or the cost can be amortized over a number of chips. A block diagram of a memory node design is shown in Figure 21. Self-refresh capability is built in. Other designs [30, 20] have shown that the latency due to packet processing is small compared to memory access latency.

8 Discussion and Related Work

Dynamic allocation of communication resources is not a new idea. Several systems use dynamic allocation of communication resources when their efficient utilization is at a premium [6, 27]. Applying network-based dynamic interconnection to low-level interconnection has been discussed in the fine-grain multicomputer community for some time. Seitz was the first to describe the dynamic interconnection idea in print and observe some of its advantages such as the higher signaling speed and the capability of parallel transactions of point-to-point direct networks [33]. However, he introduces the possibility of DI-based systems without discussing memory hierarchy and network interface issues.

The Tera architecture [3] also implements its memory subsystem with interleaved memory units interconnected by a packet-switched interconnection network. However, the Tera architecture is quite different in that there is neither a cache nor local memory; all accesses are mapped to a global shared memory and no message passing is supported. In addition, rather than building a high-performance memory subsystem, they try to hide the long memory latency caused by packet routing with cycle-by-cycle context switching.

A number of new processor memory interfaces have been proposed which address memory bandwidth limitations. The Rambus uses a 9-bit data channel which runs at 250 Mhz and achieves rates of 500MByte/s [30]. Dynamic interconnection is distinguished from Rambus primarily by the use of a general point-to-point interconnect and packet routing and the support for multiple masters and sharing

of memory modules. The Scalable Coherent Interface (IEEE P1596) also uses point-to-point links to achieve high speed signaling [20], but the ring topology used in SCI increases memory access latency.

The J-machine [16] addresses message handling overhead with hardware support, by putting the network interface on chip and providing hardware to queue incoming messages. Processor-network interface studies for the *T project [21] have addressed the issue of how to couple processors with the network. However, like existing fine-grained architectures, their register-file based message handling suffers from the memory hierarchy traffic overhead. The Fujitsu AP1000 [22] includes hardware support for two different message classes. However, its line send mechanism for short messages is based on a hardware managed queue in memory, not an on-chip FIFO as in the DI-multicomputer. This means that incoming short messages still need to be fetched from memory, increasing the latency.

The issue of implementing global cache coherence is an orthogonal issue to the DI-based systems. Either hardware directory-based coherence protocols [2, 26] or software-based approaches [7, 11] can be implemented on top of the DI-based systems to support the cache coherence.

9 Summary

The DI-multicomputer uses dynamic interconnection and novel message handling mechanisms to increase in memory bandwidth and reduce message passing overhead. Dynamic interconnection increases chip input/output bandwidth significantly by sharing pins and using point-to-point interconnects to achieve faster signalling. Our performance comparisons show that DI systems are competitive with conventional memory hierarchies at small cache line sizes (32 bytes) and superior for larger line sizes.

The DI-multicomputer provides distinct mechanisms for short and long messages, achieving a significant reduction in communication overhead. For short messages, register based short message handling eliminates startup costs. For long messages, the parallel memory-to-memory block transfer produces high bandwidth, low latency communication. Using two distinct message passing mechanisms allows messages to be handled at an appropriate level of the memory hierarchy, reducing cache perturbation and avoiding unnecessary data transfers.

Our simulation results show that the novel message passing mechanisms of the DI-multicomputer can achieve up to 4 times speedup. Also, by sending messages to appropriate level of memory hierarchy, the DI-multicomputer can eliminate most of the memory hierarchy traffic overhead, which is inevitable in register based message handling. The two level message passing mechanisms can give up to a 77 % increase in performance compared to register based message passing. The robust communication performance of the DI-multicomputer on enables it to excel existing multicomputer architectures on both fine-grained and coarse-grained applications. Our studies confirm that the DI-multicomputer achieves more robust scalable performance for larger machine sizes than the target architectures.

We have only begun to explore the possibilities of dynamic interconnection-based systems. There are a number of obvious optimizations: express cubes [13] can reduce the routing penalty in the dynamic interconnection and the network/memory hierarchy interference, and extremely large messages can be broken into several MOVE instructions to exploit greater communication parallelism, etc. The global physical address space can support a variety of additional functionality. For example, memory can be pooled amongst processors, providing some advantages of shared memory machines. Alternatively, the free association between memory and processors can be used to build a variable grain-size machine. One major issue that remains is the evaluation of the contention memory and communication traffic and its impact on the performance. We are currently performing a detail simulation of the DI-multicomputer memory hierarchy based on application memory traces to address this question.

Acknowledgements The research described in this paper was supported in part by NSF grants CCR-9209336 and MIP-92-23732, ONR grants N00014-92-J-1961 and N00014-93-1-1086 and NASA grant NAG 1-613. Additional support has been provided by the National Science Foundation under Grant No. MIP 89-20891 and MIP 93-07910 and by a generous special-purpose grant from the AT&T Foundation. We

would like to thank the reviewers for their valuable comments and suggestions. We also wish to thank Michael Peercy at IBM Almaden and Jim Hsu at HP Laboratory for their guidance on the trace-driven simulation, and Professor Prith Banerjee at Coordinated Science Laboratory for providing us the traces. Our warmest thanks to Professor Pen-Chung Yew at University of Minnesota for his encouragement and supports.

References

[1] Agarwal, A., Lim, B.-H., Kranz, D., and Kubiatowicz, J. April: a processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 104–114, May 1990.

[2] Agarwal, A., and et al. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*, 1991.

[3] Alverson, G., Alverson, R., Callahan, D., Koblenz, B., Porterfield, A., and Smith, B. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *Proceedings of the 6th ACM Interational Conference on Supercomputing*, 1992.

[4] Aoyama, K. The cost of adaptivity and virtual lanes in a wormhole router. In *Journal of VLSI Design*, 1994.

[5] Berman, P., Gravano, L., Pifarre, G., and Sanz, J. Adaptive deadlock and livelock free routing with all minimal paths in torus networks. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, 1992.

[6] Cerf, V., and Kahn, R. A protocol for packet network interconnection. *IEEE Transactions on Communications*, 1974.

[7] Cheong, H. Life Span Strategy - A Compiler-Based Approach to Cache Coherence. In *Proceedings of the International Conference on Supercomputing*, 1992.

[8] Chien, A. A. A cost and speed model for k-ary n-cube wormhole routers. multiprocessors. In *Proceedings of the Hot Interconnects*, 1993.

[9] Chien, A. A., and Kim, J. H. Planar-adaptive routing: Low-cost adaptive networks for multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, pp. 268–77, May 1992.

[10] Choi, L., and Chien, A. Integrating networks and memory hierarchies in a multicomputer node architecture. In *Proceedings of the International Parallel Processing Symposium*, April 1994.

[11] Choi, L., and Yew, P. C. A Compiler-Directed Cache Coherence Scheme with Improved Intertask Locality. In *Proceedings of the ACM/IEEE Supercomputing*, pp. 773–782, November 1994.

[12] Cray Research Inc. SHMEM User's Guide. 1994.

[13] Dally, W. J. Express cubes: Improving the performance of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 40(9):1016–1023, 1991.

[14] Dally, W. J. Virtual channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, 1992.

[15] Dally, W. J., Chien, A., Fiske, S., Horwat, W., Keen, J., Larivee, M., Lethin, R., Nuth, P., Wills, S., Carrick, P., and Fyler, G. The J-Machine: A fine-grain concurrent computer. In *Information Processing 89, Proceedings of the IFIP Congress*, pp. 1147–1153, August 1989.

[16] Dally, W. J., Fiske, J. A. S., Keen, J. S., Lethin, R. A., Noakes, M. D., Nuth, P. R., Davison, R. E., and Fyler, G A. The message-driven processor. *IEEE Micro*, April 1992.

[17] Dally, W. J., and Song, P. Design of a self-timed vlsi multicomputer communication controller. In *Proceedings of the International Conference on Computer Design*, pp. 230–4. IEEE Computer Society, 1987.

[18] Davidson, E. E. Electrical design of a high speed computer packaging system. *IEEE Transactions on Components, Hybrids and Manufacturing Technology*, CHMT-6(3):272–282, 1983.

[19] Digital Equipment Corporation. *Alpha Architecture Handbook*, 1992.

[20] Gustavson, D. B. The scalable coherent interface and related standards projects. *IEEE Micro*, 12(1), Feb. 1992.

[21] Henry, D. S., and Joerg, C. F. A tightly-coupled processor-network interface. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages an Operating Systems*, pp. 111–122, 1992.

[22] Horie, T., et al. AP1000 architecture and performance of LU decomposition. In *International Conference on Parallel Processing*, 1991.

[23] Hsu, J. M. Performance measurement and hardware support for message passing distributed memory multiprocessors. Technical Report UILC-ENG-91-2209, University of Illinois, Center for Reliable and High-Performance Computing, 1991.

[24] Intel Corporation. *i860 XP Microprocessor Data Book*, 1991.

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

[26] D., Laudon, J., Gharachorloo, K., Gupta, A., and Hennessy, J. The Directory-Based Cache Coherence Protocol for the DASH Computer. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 148–159, May 1990.

[27] Metcalfe, R., and Boggs, D. Ethernet: Distributed packet-switching for local computer networks. *Communications of the Association for Computing Machinery*, 19(7):395–404, 1976.

[28] Ni, L., and Glass, C. The turn model for adaptive routing. In *Proceedings of the International Symposium on Computer Architecture*, 1992.

[29] Peterson, C., Sutton, J., and Wiley, P. iWarp: a 100-MOPS LIW microprocessor for multicomputers. *IEEE Micro*, June 1991.

[30] Rambus Corporation. Rambus architectural overview. Product Literature, 1992.

[31] Matsui, N., Satoh, H., and Okada, K. Analysis of high-speed bus lines in printed circuit boards. In *IEEE/CHMT Japan IEMT Symposium*, pp. 156–167. IEEE, 1989.

[32] Seitz, C., and Su, W. A family of routing and communication chips based on the Mosaic. In *Proceedings of the University of Washington Symposium on Integrated Systems*, 1993.

[33] Seitz, C. L. Let's route packets instead of wires. In W. J. Dally, editor, *Proceedings of the 6th MIT Conference on Advanced Research in VLSI*, pp. 133–37. MIT Press, 1990.

[34] Thinking Machines Corporation, Cambridge, Massachusetts. *CM-5 Technical Summary*, October 1991.

[35] Whitby-Strevens, C. The transputer. In *Proceedings of 12th International Symposium on Computer Architecture*, 1985.

A Packet Formats

B Instruction Set Architecture

The DI-microprocessor has a basic RISC instruction set augmented with special communication, synchronization and translation instructions. All registers are addressed relative to a context offset (CO) in the process status register. In this section, only the new instructions added to DEC Alpha instruction set will be described.

All the new instructions are described using a simple format which can be implemented by any instruction format of a modern RISC microprocessor. The operand types of the format are shown in Table VIII. The detail field specification of the operand types is assumed to be specified according to the DEC Alpha instruction format shown in Figure 24.

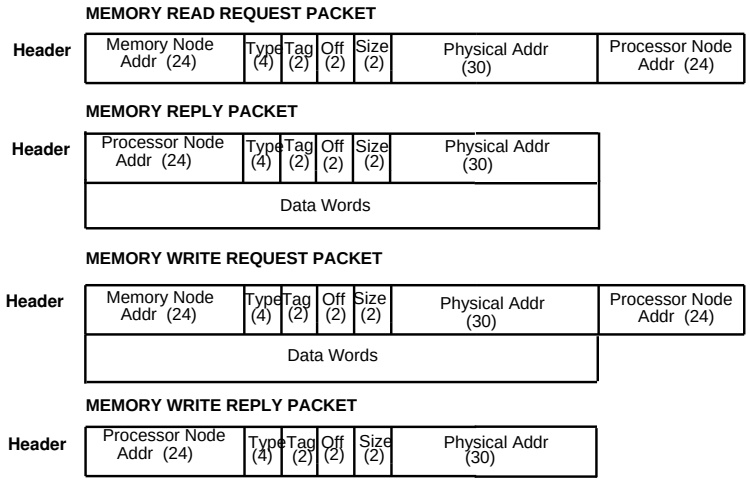


Figure 22: Memory packet formats in dynamic Interconnection

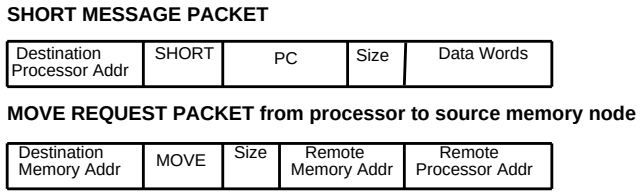


Figure 23: Packet formats used in message handling

B.1 Extended Memory Operations

Row mode memory operations are added to the usual load store operations of the DEC Alpha. Table IX shows the row mode memory operations which move data between a row register and memory. All the row mode memory operations (LD_ROW, ST_ROW, LDT_ROW) use the *Memory Instruction Format* of DEC Alpha. The virtual address is computed by adding register Rb to the sign-extended 16 bit displacement.

LD_ROW: Load a Memory Block into a Row Register

- **Format** *LD_ROW ROWd, Src* (Memory Instruction Format)

Table VIII: Operand Field Description

Mnemonic	Description
Rs, Rd	Register Operands
ROWs, ROWd	Row Operands
Src, Dst	Memory Operands
#Size, #Dest	Immediate Operands

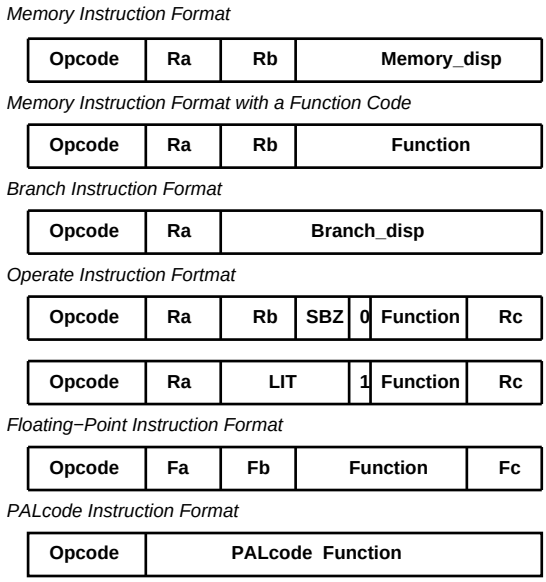


Figure 24: DEC Alpha Instruction Formats

- **Operation** ROWd ← MEM_BLOCK[Src]
- **Description** The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The source memory block is fetched from memory and written to row register ROWd.

ST_ROW: Store a Row Register into Memory

- **Format** *ST_ROW ROWs, Dst* (Memory Instruction Format)
- **Operation** MEM_BLOCK[Src] ← ROWs
- **Description** The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The ROWs operand is written to memory at this address.

LDT_ROW: Load a Memory Block into a Row Register in Empty Context

- **Format** *LDT_ROW ROWd, Src* (Memory Instruction Format)
- **Operation** (ROWd in empty context) ← MEM_BLOCK[Src]
- **Description** The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The source memory block is fetched from memory and written to row register ROWd. Instead of local register sets, the target row register is from an empty context which is specified by the empty context offset in RSR. This is a privileged operation. The trap context only can perform the operation.

STT_ROW: Store a Row Register in Previous Context into Memory

- **Format** *STT_ROW ROWs, Dst* (Memory Instruction Format)

- **Operation** MEM_BLOCK[Dst] ← (ROWs in previous context)
- **Description** The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The ROWs operand is written to memory at the specified address. Instead of local register sets, the source row register is from the previously running context which is specified by the previous context offset in RSR. This is also a privileged operation. The trap context only can perform the operation.

There are two additional memory operations used for long message transfer. One (STV) is used for reception of a long message and the other (TEST) is used for synchronization. Executing an STV instruction allocates a dummy cache line which will hold the acknowledgments from the memory. There is no source operand specified. Therefore, the specified target memory word will remain unchanged by executing this virtual store instruction. The TEST instruction checks whether the location contains all the acknowledgments from the memory nodes. If not, the thread executing the instruction will block until the acknowledgments arrive. This instruction allows synchronization between the receiving thread and message reception at destination node.

STV: Store a Register Data into Memory Virtually

- **Format** *STV Dst* (Memory Instruction Format)
- **Operation** allocate CACHE_BLOCK[Dst]
- **Description** The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. A cache block is allocated for the target address but the instruction will not affect the corresponding memory block. The source register operand field is not used.

TEST: Test memory location

- **Format** *TEST Src* (Memory Instruction Format)
- **Operation** if CACHE_BLOCK[Src] = 'ACK' then {test succeeds and returns} else block
- **Description** The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The register operand field is not used.

B.2 Message Operations

Table IX also shows the message send instructions in DI-microprocessor. SEND is used for short message transfers while MOVE is used for long block transfers. The source operand Rs in the SEND instruction specifies the first register of the transmitted messages in the register file, and the size field specifies the length of the message. All the message handling instructions use the *Operate Instruction Format* of DEC Alpha. All the immediate fields use the literal field of the instruction format. The source address should be local and the destination memory address should be remote in the MOVE instruction. Since the MOVE instruction needs more than 2 source register operands, it may need another cycle to decode and fetch all the register operands specified depending on the implementation.

SEND: Send a short message

- **Format** *SEND Rs, #Size* (Operate Instruction Format)
- **Operation** Router ← Message in

$$R_s, R_{s+1}, ..R_{s+size-1}$$

- **Description** The message should be contiguous in the specified registers. The message is copied into router port by the network interface.

Table IX: Extended Instruction Set for DI-microprocessor

	Mnemonic	Operation
LD_ROW	ROWd, Src	Load a line of words in memory into a row register
ST_ROW	ROWs, Dst	Store a row register into memory
LDT_ROW	ROWd, Src	Load a line of words in memory into empty context (Supervisor mode load for reception context)
STT_ROW	ROWs, Dst	Store a row register in the previous context into memory
STV	Rs, Dst	Virtual store operation for the reception and synchronization of long messages
TEST	Src	Test long message arrival
SEND	Rs, #Size, Rd	Send a short message from the specified registers
MOVE	Rs1, ROWs, #Size, Rs2	Initiate a long block transfer from local memory (Rs1) to remote memory (Rs2) with destination processor in (Rs3)
TLX	Rs, ROWd	virtual (Rs) to physical (ROWd) address translation using TLB

MOVE: Move a long message from local memory to remote memory

- **Format** *MOVE Rs1, ROWs, #Size, Rs2* (Operate Instruction Format)
- **Operation** MEM_BLOCK[ROWs] at remote processor Rs2 ← MEM_BLOCK[Rs1] where BLOCK_SIZE = #Size
- **Description** The virtual address for local memory is specified by the register field Ra. And the Function field is used to specify the row register ROWs which contains the physical addresses for remote memory. The Size field uses the literal field of the format. And the remote processor is specified by the register field Rc in the operate instruction format. The memory interface generates the corresponding move requests to memory.

B.3 TLB Manipulation Operations: TLX

To handle address translation request by software, DI-microprocessor provides a TLB manipulation operation as shown in Table IX. It uses the *Operate Instruction Format* of DEC Alpha. The instruction accesses the TLB and gives the physical memory address (four memory node addresses) corresponding to the virtual address specified.

TLX: Translate a virtual address into a physical address

- **Format** *TLX Rs, ROWd* (Operate Instruction Format)
- **Operation** ROWd ← TLB[Rs]
- **Description** The virtual address is specified by the register field Ra and destination row register is specified by the register field Rc. TLB entry corresponding to the address is fetched from TLB and written into the row register specified. Since there are four physical addresses corresponding to the virtual address, each physical address is written into a register in the specified row register.

Biography

LYNN CHOI is a Ph.D candidate in the Department of Computer Science and a research assistant in the Center for Supercomputing Research and Development of University of Illinois at Urbana-Champaign. He received his B.S. and M.S. degree both in computer engineering in 1986 and and in 1988 respectively from Seoul National University, Seoul Korea. After the study, he worked at Korea Telecom Research Center as a technical staff until 1990. His current research interests include architecture and compilation issues in high performance computer systems, specifically cache and memory system design in large-scale parallel systems. He is a student member of both IEEE and ACM.

ANDREW A. CHIEN received his B.S. degree in Electrical Engineering (1984) and his M.S. (1987) and Ph.D. (1990) degrees in Computer Science all from the Massachusetts Institute of Technology. He is currently an Associate Professor in the Department of Computer Science at the University of Illinois where he leads the Concurrent Systems Architecture Group. His research interests involve the interaction of architecture, system software, compilers, and programming languages in high-performance parallel systems. Andrew Chien has authored numerous research papers in those areas and recently published *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs* with MIT Press. Andrew Chien is a member of both IEEE and ACM.