Synchronization and Communication in the T3E Multiprocessor

Steven L. Scott

Cray Research, Inc.

sls@cray.com

Abstract

This paper describes the synchronization and communication primitives of the Cray T3E multiprocessor, a shared memory system scalable to 2048 processors. We discuss what we have learned from the T3D project (the predecessor to the T3E) and the rationale behind changes made for the T3E. We include performance measurements for various aspects of communication and synchronization.

The T3E augments the memory interface of the DEC 21164 microprocessor with a large set of explicitly-managed, external registers (E-registers). E-registers are used as the source or target for all remote communication. They provide a highly pipelined interface to global memory that allows dozens of requests per processor to be outstanding. Through E-registers, the T3E provides a rich set of atomic memory operations and a flexible, user-level messaging facility. The T3E also provides a set of virtual hardware barrier/eureka networks that can be arbitrarily embedded into the 3D torus interconnect.

1 Introduction

The goal of kiloprocessor multiprocessing presents a number of challenges. Fundamentally, it requires software capable of *exposing* parallelism in an application, and hardware capable of *exploiting* that parallelism by providing the necessary communication and synchronization support.

Parallelism may be exposed explicitly, using the message passing model (*e.g.*: Parallel Virtual Machine (PVM) [14] or Message Passing Interface (MPI) [31]), or implicitly, using the shared-memory programming model (*e.g.*: High Performance Fortran (HPF) [19] or the Alpha AXP architectural model [10]). The shared-memory model is widely accepted as easier to use, and is better suited for irregular, dynamic parallelism. The message passing model, however, is currently more portable (PVM and MPI run on a wide variety of machines) and makes the detection of parallelism and optimization of data layout significantly easier for the compiler.

For either programming model, however, the best *performance* is likely to be delivered by a tightly-coupled, shared-memory system. The choice of shared memory for the T3D and T3E was not an endorsement of the shared memory programming model over the message passing model, but was made because it minimized synchronization and communication overhead.

As Amdahl's Law illustrates, lower synchronization and communication overhead have the following direct results:

1. a greater number of processors can be used to solve a given problem at a given efficiency, or
2. a finer granularity of work can be performed with a given number of processors.

Depending upon the application, communication bandwidth or communication/synchronization latency may drive the overhead. A highly scalable multiprocessor must address both.

Most multiprocessors are built with commodity microprocessors, which offer rapidly increasing performance and excellent price performance. Microprocessors, however, are generally designed for workstations and modestly parallel servers. A large-scale multiprocessor creates a foreign environment into which they are ill-equipped to fit.

The most striking limitation of most microprocessors is their memory interface. The interfaces are cache line based, making references to single words (corresponding to strided or scatter/gather references in a vector machine) inherently inefficient. More importantly, they typically allow only one or a small number of outstanding references to memory, limiting the ability to pipeline requests in large systems. For example, the DEC 21064 [11] and 21164 [12], on which the Cray T3D and T3E are based, allow a maximum of one and two outstanding cache line fills from memory, respectively.

Microprocessors often lack sufficiently large physical address spaces for use in large-scale machines. The DEC 21064, for example, implements a 33-bit physical address[1], while the maximum physical memory in the T3D, is over 128 GB.

TLB reach is another potential problem. A TLB that is sufficiently large for a powerful workstation may be insufficient for a machine with a thousand processors and a terabyte of physical memory.

Microprocessors are designed to cache data that they reference. While this is usually beneficial, it is sometimes desirable to make non-cached references to memory. When writing to another processor's memory in a message-passing program, for example, it is far better for the data to end up in the recipient processor's memory than in the sending processor's cache!

In general, microprocessors are designed with an emphasis on latency reduction rather than latency toleration. While this is an effective approach for many codes, it is ineffective for scientific codes with poor locality, and it does not support high-bandwidth communication in large-scale multiprocessors.

This paper discusses the Cray T3E multiprocessor, which is based on the DEC Alpha 21164 microprocessor. We describe the "shell" that surrounds the processor to make it fit comfortably into a kiloprocessor machine, and discuss features designed to support highly-parallel, fine-grained programming. The paper focuses on communication and synchronization, giving little consideration to the processor, network, memory system or I/O system.

1. A 34th bit is used to distinguish between memory and I/O space.

The T3E is the second in a line of scalable multiprocessors, following the Cray T3D [8][35], which first shipped in late 1993. Section 2, discusses lessons from the T3D project. Section 3 gives a general overview of the T3E. Sections 4 through 7 discuss global communication, atomic memory operations, message passing support and barrier/eureka synchronization. Section 8 presents performance measurements, Section 9 discusses related work, and Section 10 concludes.

2 Lessons from the T3D

The T3D connects up to 2048 DEC Alpha 21064 microprocessors via a 3D torus network, with two processors per network node. Each processor contains up to 64 MB of local memory, and the memories of all processors are accessible via a shared address space.

To extend the 33-bit physical address space of the 21064, a "DTB Annex"[1] is maintained in the shell outside the processor. The upper bits of a load or store address contain an index into the Annex, and the corresponding Annex entry provides the PE bits for the address. Although remote memory can be cached, the on-chip cache tags will contain an Annex index rather than PE bits, so the line must be flushed if the Annex entry is changed. A more complete description can be found in [24] or [8].

The T3D has several strengths that have been carried forward into the T3E. First among these is the shared address space. Although the shared memory efficiently supports CRAFT [38], Cray's version of parallel Fortran, the most widely used programming models on the T3D have been PVM, and, for performance-critical communication, Shmem [9]. Shmem is a shared-memory-based message passing library that supports direct memory-to-memory transfers without involving the operating system. Researchers at Illinois have also found the shared memory instrumental in achieving good messaging performance [22].

The interconnection network has also proven to be a strength. The 3D torus is wiring-efficient [1] and scales well to large numbers of processors, providing sub-microsecond access latencies and a bisection bandwidth of over 70 GB/s with 1024 processors. The T3D is the *only* machine with a complete set of published NAS Parallel Benchmarks results for greater than 128 processors (results up to 1024 processors have been published) [43].

The T3D barrier network consists of a four-wire-wide, degree-four spanning tree over the entire machine. It provides full machine barrier synchronization in less than 2 μs[2]. While this has proven useful, especially for CRAFT programs in which global synchronization is quite frequent, it appears to be a case of over-engineering. We have yet to encounter an application in which barrier time is a large fraction of total run-time, and the dedicated barrier network is expensive. In addition, we have found the management of the physical barrier resource to be burdensome.

The T3D has several weaknesses, many of which have been reported in [3]. The largest of these is the relatively low single node performance. This is caused by a fixed clock (150 MHz), which has not tracked improvements in the 21064 processor, and by lack of a board-level cache (each processor uses only its 8KB on-chip data cache). This last feature, however, does allow the T3D to provide significantly higher memory bandwidth; the STREAM single processor copy benchmark on the 150 MHz T3D yields over 4 times the bandwidth of the DEC 2100 A500-4/200 using the same processor clocked at 200 MHz [29][28].

The T3D implements three different ways to access remote memory: direct loads and stores, an explicit prefetch queue that allows up to 16 outstanding single-word references, and a block transfer engine (BLT) that provides bulk, asynchronous data transfers between processors' memories. Load/store performance highlights the memory pipelining issue. Since only a single outstanding cache line fill is allowed, sustainable load bandwidth is fairly low (about 30 MB/s in a 256-processor machine). Sustainable store bandwidth is much higher (about 120 MB/s, independent of size), since the stores are acknowledged right away by the processor shell, and an unlimited number may be pipelined in the interconnect.

The prefetch queue is used by both the CRAFT compiler, to fetch remote data in loops, and the Shmem libraries, to increase memory copy bandwidth. Its main limitation is that only a single stream can be prefetched, making it difficult to coordinate its use among multiple parties. Our compiler writers would have liked multiple queues.

The BLT is shared between the two processors at a node and requires a system call to use. It takes on the order of 1000 6-ns processor clocks to start up a transfer, and as a result has been of little use. Even if the BLT startup was more reasonable, its value would be questionable. We have found that having three ways to access remote memory is more of a liability than a benefit. It means that the compiler, library and/or user must always decide *how* to access memory, an optimization problem for which the necessary information is seldom available.

The DTB Annex has proven useful for library routines, but difficult for the compiler to exploit. Without global information, the Annex entries are generally set up each time they are used. Since the overhead to change an Annex entry is small, a single entry would have likely sufficed.

Several features in the T3D require special management, including the barrier network and the two dedicated fetch_&_inc registers and one dedicated message queue at each processor. Since these are special hardware resources, they must be protected by the operating system. The message queue also requires OS involvement on the receiving side, as user and OS messages share the same queue, significantly increasing message latency. The Illinois messaging implementation [22] did *not* use the dedicated messaging hardware.

The DTB Annex allows a single DTB entry to map a physical page on all processors in a parallel program[3], but every processor must use the *same* mapping. So while DTB coverage is significantly amplified, memory management is inflexible; moving a shared page on one processor requires stopping all processors in the program and moving their pages too. This is similar to the TLB shootdown problem [4], but significantly more expensive.

In summary, the shared memory and fast 3D network have been very useful, and non-cached stores and the prefetch queue have proven to be very effective for pipelined remote memory access. But there are too many ways to access remote memory, remote load bandwidth is poor, and several special-purpose hardware features have proven cumbersome to manage and/or inflexible to use. The design of the T3E was largely guided by these experiences.

1. DTB stands for Data Translation Buffer, DEC's term for a TLB.
2. Most of this time is in the library software; performance is almost independent of machine size.
3. The software must explicitly manage the Annex, however, to access all PEs.

3 T3E overview

The T3E implements a logically shared address space over physically distributed memories (up to 2 GB per processor). Each processing element (PE) contains a DEC Alpha 21164 processor connected to a "shell", consisting of a control chip, a router chip and a local memory (see Figure 1). The system logic runs at 75 MHz, and the processor runs at some multiple of this (initially 300 MHz).

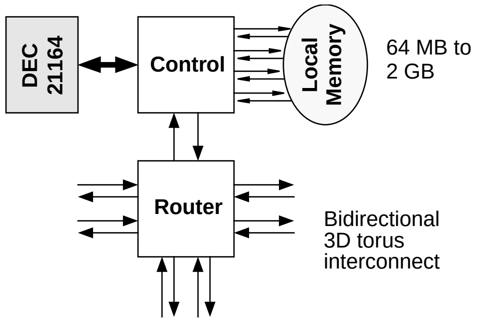

Figure 1. T3E PE block diagram

Like the T3D, up to 2048 processors are connected by a bidirectional 3D torus, but each node of the torus contains only a single processor, and the network implements fully adaptive, minimal-path routing [45]. The network links are time multiplexed at five times the system frequency, and can deliver one 64-bit word of payload each sysclock (13.3 ns).

The T3E is a self-hosted machine[1] running Unicos/mk, a serverized version of Unicos based on the Chorus microkernel [7]. I/O is based on the GigaRing channel [44], with sustainable bandwidths of 267 MB/s input and output for every four processors.

Like the T3D, the T3E contains no board-level cache, but the 21264 processor has two levels of caching on chip: 8KB first level instruction and data caches, and a unified, 3-way associative, 96 KB second level cache. As with the T3D, memory bandwidth is higher than would be possible with a board-level cache. Measured performance on the STREAM copy benchmark of 470 MB/s is over twice that of the DEC 8400 5/300 (186 MB/s), which uses a 21164 processor running at the same frequency [29][28].

The 21164 allows two outstanding 64-byte cache line fills. Local memory bandwidth is enhanced by a set of hardware stream buffers. These buffers automatically detect consecutive references to multiple streams, even if interleaved, and prefetch additional cache lines down each stream. They can achieve much of the benefit of a large, board-level cache for scientific codes at a small fraction of the cost [36].

Only local memory is cached in the T3E. The on-chip caches are kept coherent with local memory through an external backmap, which filters memory references from remote nodes and probes the on-chip cache when necessary to invalidate lines or retrieve dirty data.

The T3E augments the memory interface of the DEC 21164 microprocessor with a large set (512 user plus 128 system) of explicitly-managed, external registers (E-registers). All remote communication and synchronization is done between these registers and memory.

The E-registers take the place of the T3D's DTB Annex, prefetch queue, block transfer engine and remote loads/stores. They also facilitate the removal of dedicated fetch_&_inc registers and message queues. Instead, synchronization variables and message queues are stored in normal user memory, allowing them to be managed via the existing address translation mechanism and substantially increasing their flexibility. In a similar vein, the dedicated barrier/eureka network has been virtualized, easing the task of managing the barrier trees and providing multiple logical barrier networks.

The goals of the T3E design were to integrate and simplify the various features of the shell, make messaging, synchronization and memory management more flexible, and significantly increase the amount of pipelining in the memory system, both for cacheable references to local memory and non-cached references to remote memory.

1. The T3D requires a Cray vector machine as a front end.

4 Global communication

This section explains the use of E-registers for global communication. E-registers provide two primary benefits over a more straight-forward load/store mechanism for accessing global memory: they extend the physical address space of the microprocessor to cover the full physical memory of the machine, and they radically increase the degree of pipelining attainable for global memory requests. They also provide efficient single-word bandwidth, an integrated centrifuge for flexible data distribution, and a convenient mechanism for messaging and atomic memory operations.

The 21164 implements a cacheable memory space and a non-cacheable I/O space, distinguished by bit 39 of the 40-bit physical address. Local memory loads and stores in the T3E use cacheable memory space. Address translation takes place on the processor in the usual fashion, and physical addresses are passed through the shell directly to the memory.

The T3E uses I/O space to access memory-mapped registers, including the E-registers. There are two primary types of operations that can be performed on E-registers:

- Direct loads and stores between E-registers and processor registers.
- Global E-register operations.

Direct E-register loads/stores are used to store operands into E-registers and load results from E-registers. Global E-register operations are used to transfer data to/from global (meaning remote *or* local) memory and perform messaging and atomic operation synchronization.

4.1 Address translation for global references

For global E-register operations, a global virtual address (shown in Figure 2) and virtual PE number are formed *outside* the processor in the shell circuitry. The virtual PE number goes through a translation mechanism at the source processor to identify the physical PE, and the global virtual address is transmitted across the network, where it goes through a virtual-to-physical translation using a *global translation buffer* at the target PE.

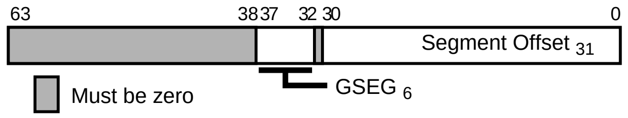

Figure 2. Global virtual address (GVA)

The T3E supports the data distribution features of many implicit programming languages [38][19][6][13] via an integrated hardware *centrifuge*. The virtual address for global references is formed using a mask, index and base. The mask bits indicate which of the bits in the index represent PE bits, and which bits represent address within a PE. For each bit set in the mask, the corresponding bit in the index is extracted. The extracted bits are compacted to form a virtual PE number, and the remaining bits are compacted and added to the base to form a virtual address. Typically a base and mask are set up for each shared distributed array and then the index is varied.

Figure 3 illustrates the centrifuge operation for an array distributed over 64 PEs. The bits in the index corresponding to the ones in the mask are pulled out to form the virtual PE number (PE 37 in this case). The remaining bits of the index form an offset which is added to the base. Since the PE field starts at bit 6, each successive cache line (64 bytes) in the shared array maps to a new PE.

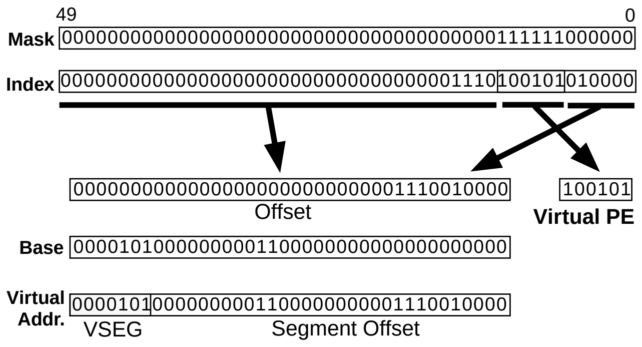

Figure 3. HW centrifuge operation example: Array interleaved by cache line over 64 PEs

A typical "distributed memory" (message passing) program would use a single mask and base for all E-register operations. The base would be set to zero and the mask would have a block of set bits in the upper part of the address. The index would thus contain a complete address; the lower part would simply be a virtual address within a PE, and a field in the upper part would represent a PE number.

The full address translation path for a global E-register reference is shown in Figure 4. The operation is performed by performing a *store* in I/O space. The address of the store encodes a command (*e.g.*: read a word from remote memory into an E-register) and a source or destination E-register. The 64-bit word written onto the data bus includes the *index* for the remote memory location and a *pointer* to an aligned block of four E-registers containing the mask and base for the centrifuge and up to two additional arguments. Before performing the operation, the mask and base must have been stored into the E-registers. This need only be done once for each distributed array (or at least is done outside the inner loop). A single general-purpose mask/base pair may also be set up for all miscellaneous data references, or for all references in a message-passing program as described above.

The index is centrifuged with the mask and base to produce a virtual address and virtual PE number (this is the PE number that an application uses; virtual PE space always goes from 0 to *n*-1 in an *n*-processor job). The virtual address includes a virtual segment number, which indexes into a segment translation table. The segment translation table produces a global segment (GSEG), a base PE (which corresponds to section of the machine in which the application is running), a PE limit and protection information. The virtual PE is added to the base PE to produce a logical PE number, which is presented to a routing lookup table to produce a physical routing tag[1].

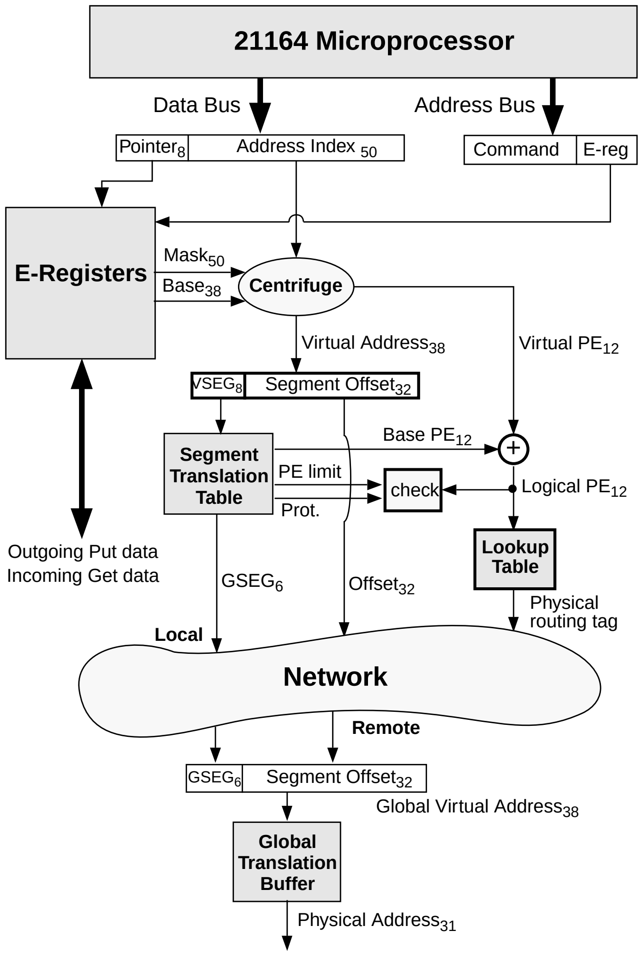

Figure 4. Address translation for global (E-register) references

The segment lookup on the source node guarantees that user-generated addresses only access authorized GSEGs on authorized PEs. Segment range violations are detected at the remote node. The 6-bit GSEG space allows multiprogramming; different jobs (with possibly common VSEGs) sharing memory at a node are assigned different GSEGs.

The GSEG and segment offset form a GVA (refer to Figure 2) that is transmitted across the network with the reference. At the remote node, the GVA goes through a translation to produce the actual physical address at that node. The global translation buffer performs page-based translation with flexible pages sizes (64 KB - 128 MB), and is hardware-loaded from a complete page table in memory (so cannot fault under normal conditions).

The remote translation step allows each node to manage its own physical memory; it can move its part of a shared segment independently from the other processors. This eliminates TLB shootdowns entirely.

To further support this, an integrated hardware engine we call the "magical memory mover" can perform a local memory copy operation in the background and allow memory requests to that page to be serviced *while* the copy is in progress (the reference is serviced

1. The logical PE to physical routing tag translation allows spare PEs to be mapped in to replace broken PEs.

from the old or new location depending upon whether that particular word has been transferred yet). This allows the operating system to move a page of a shared segment without delaying any of the processors in the parallel job accessing the page.

4.2 Get and Put operations

The global operations to read memory into E-registers or write E-registers to memory are called *Gets* and *Puts*, respectively. There are two forms of Gets and Puts: single word and vector. Both can operate on either 32-bit or 64-bit words. Vector Gets and Puts transfer 8 words, with an arbitrary stride. The stride operand is stored in the block of E-registers that contains the mask and base.

Access to E-registers is implicitly synchronized by a set of state flags, one per E-register. A Get operation marks the target E-register(s) empty until the requested data arrives from memory, at which time they are marked full. A load from an E-register will stall if the E-register is empty until the data arrives. A Put from an E-register will also stall if the E-register is empty until the data becomes available. A global memory copy routine might perform Gets from the source area of memory into a block of E-registers and subsequently Put the data from the E-registers to the target memory area. The implicit state flag synchronization protects against the RAW hazard in the E-registers.

Since there are a large number of E-registers, Gets and Puts may be highly pipelined. The bus interface allows up to four properly-aligned Get or Put commands to be issued in a two-cycle bus transaction, allowing 256 bytes worth of Gets or Puts to be issued in 26.7 ns. This issue bandwidth is far greater than the sustainable data transfer bandwidth, so the processor is not a bottleneck. Data in a memory-to-memory transfer using E-registers does not cross the processor bus; it flows from memory into E-registers and out to memory again.

In addition to providing a highly-pipelined memory interface, the E-registers provide special support for single-word load bandwidth. Row accesses in Fortran, for example, can be fetched into contiguous E-registers using strided vector Gets. The resulting blocks of E-registers can then be loaded broadside into the processor in cache-line-sized blocks, making significantly more efficient use of the bus than would be possible with normal cache line fills.

The maximum data transfer rate between two nodes using vector Gets or Puts (as determined by the network) is 480 MB/s, and E-register control logic further limits the bandwidth to something less than this, depending upon the operation. At this rate Little's Law[1] indicates that 128 E-registers provide sufficient pipelining to hide the round-trip packet latencies plus command issue times (on the order of 1-2 µs).

5 Atomic memory operations

The T3E expands upon the atomic SWAP feature of the T3D to provide a rich set of atomic operations. While SWAP operations in the T3D can only be performed on dedicated SWAP registers, atomic operations in the T3E can be performed on arbitrary memory locations, allowing an unlimited number of synchronization variables, easing the job of the compiler, and removing the involvement of the operating system.

Table 1 lists the atomic memory operations (AMOs) provided by the T3E. Fetch_&_inc, fetch_&_add, and compare_&_swap are well known synchronization primitives. Masked_swap provides test_and_set and clear operations on individual bits, by swapping in ones or zeros in specified locations. It also provides a mechanism to perform atomic byte (or other size) stores.

Atomic Operation (operands)	**Description**
Fetch_&_Inc (none)	Add one to memory location and return original memory contents.
Fetch_&_Add (addend)	Add integer addend to memory location and return original memory contents.
Compare_&_Swap (comperand, swaperand)	If comperand equals contents of memory, then store swaperand into memory. Return original contents of memory.
Masked_Swap (mask, swaperand)	For each bit set in mask, store corresponding bit of swaperand into memory. Return original contents of memory.

Table 1. Atomic Memory Operations

Herlihy has shown [17] that compare_&_swap is a universal primitive, meaning that it can be used to construct a wait-free implementation[2] of any sequential object (*e.g.*: shared work queues). It is also necessary or beneficial for a variety of scalable synchronization algorithms [30][32]. Load-linked/store-conditional, implemented in several architectures [10][41][27], is also a universal primitive, and in fact can allow more straight-forward implementations of some concurrent objects [18]. However, most load-linked/store-conditional implementations place restrictions on the types of operations that can be performed in the critical section (e.g.: no memory operations), and the primitive does not scale well to large numbers of processors under variable contention.

To perform an AMO in the T3E, any necessary operands are first written to E-registers. The operation is then triggered via a store to I/O space, as described in Section 4.1. The AMO command is specified on the address bus. The necessary operands are read from the aligned block of E-registers that is used for the mask and base. An atomic memory operation packet is then sent to the specified global memory location, where the operation is performed. The result is returned to the E-register specified on the address bus of the AMO command.

Most AMOs in the T3E require a read-modify-write of DRAM, resulting in a minimum repeat time of 11 sysclocks (147 ns) for a given synchronization variable (8M AMOs per second). High bandwidth fetch_&_inc operations are supported via a buffer at the memory controller of each node. Successive fetch_&_incs to the same word are satisfied out of the buffer, allowing a repeat time as low as 13.3 ns, or 75 M fetch_&_incs per second.[3]

6 Messaging

Message queues in the T3D and T3E are intended to support distributed memory applications and inter-process communication within the operating system.

The T3D provides a single message queue at each processor that is shared by both user and system messages. The queue is of fixed size (256 KB) and is located at a fixed location in memory. Mes-

1. N = X•R, where X ≡ throughput, R ≡ response time, and N ≡ number outstanding.

2. One in which no blocked (*e.g.* swapped out) process can impede the progress of any other process.

3. The buffer was originally intended to support all atomic operations, but due to implementation constraints, only fetch_&_incs were supported.

sages are 32 bytes plus header information. While the hardware transmission latency involves only a single network traversal, all incoming messages generate interrupts and are examined by system software. This adds a significant latency penalty to message receipt, calling into question the efficacy of using the special mechanism rather than constructing message queues in normal shared memory.

The T3E allows an arbitrary number of message queues to be created by either user or system code. Queues are mapped into normal memory space and can be of any size up to 128 MB. Messages are 64 bytes (no header is stored). The queues can be set to interrupt on arrival, never interrupt (in which case messages are detected via polling), or interrupt only when some threshold number of messages have arrived. T3E message queues integrate the desirable features of message passing (one-way network traversal latency, no distributed buffer management) with the flexibility of a shared-memory implementation.

6.1 Message Queue Control Word

A message queue is created by simply constructing and storing a Message Queue Control Word (MQCW) at the address of the desired location of the queue. The 64-bit MQCW has four fields, as shown in Figure 5.

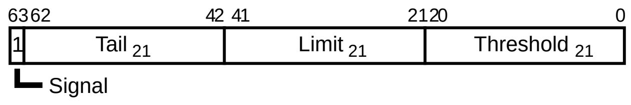

Figure 5. Message Queue Control Word

Tail is a relative offset that is added to the address of the MQCW to identify the next available slot in the queue. Tail must be initialized to a value greater than zero to avoid having the first arriving message write over the MQCW. The Tail pointer has a granularity of 64 bytes. It is incremented by one each time a message is stored into the queue.

Limit is a 21-bit value that indicates the size of the message queue. Sizes up to (2^{21} - 2) 64-byte messages are supported. When a message arrives, Limit is compared to Tail. If Tail $\geq$ Limit, the message is rejected and the Tail pointer is not incremented. If Tail < Limit, the Tail value is added to the global virtual address (GVA) of the MQCW to generate a new GVA for the message. This GVA is then translated by the GTB and checked for range errors prior to storing the message. If the message is accepted, an ack is returned, the message is stored and Tail is incremented. If the address is illegal, a nack is returned.

Threshold is a soft limit which is generally set to a value $\leq$ Limit. When a message is accepted, Tail is incremented and compared to Threshold. If Tail = Threshold, then an interrupt is delivered to the local processor, and the Signal bit is set to facilitate identification of the interrupting message queue. Messages are not rejected when Tail $\geq$ Threshold.

6.2 Sending a message

Messages are transmitted by first assembling them in an aligned block of 8 E-registers and then issuing a SEND command. A SEND is similar to a Put command, except the memory address of the SEND must be a valid MQCW in memory.

The block of 8 E-registers is delivered to the specified address, where it is stored into the message queue as described in Section 6.1. The read-modify-write of the MQCW and storage of the message are performed atomically, so no arbitration is required when multiple processors are transmitting to the same queue.

When the SEND is issued, the state flags associated with the 8 E-registers are set to empty. When the response is received, if the message was accepted, the flags are set to full, else the flags are set to “full-send-rejected”. This can be detected by the sending processor and the message can be retransmitted.

6.3 Message queue management

The head of a message queue (next unread message) is maintained by software. As the local processor consumes messages, it increments its head pointer until equal to the tail pointer. The processor is responsible for re-allocating message queue space when Tail approaches (or reaches) Limit.

An atomic memory operation performed on a MQCW exactly affects the flow of messages to the corresponding message queue. If a swap is performed to redirect messages to a different portion of the queue, for example, the returned MQCW will represent the last message stored to the queue. No messages will be lost.

A typical algorithm for managing a message queue is as follows. The processor first sets Tail and Limit to point to the first half of the queue. As Tail approaches Limit, the processor performs a SWAP to set Tail and Limit to point to the second half of the queue. It then consumes the residual messages from the first half of the queue. When Tail again approaches Limit, the processor performs a SWAP to switch to the first half of the queue, and so on. Other algorithms are of course possible.

The MQCW/SEND mechanism allows users to set up multiple message queues of arbitrary size. Since message queues are held in normal memory space, no special access protection need be provided. By polling, users programs can use messaging with no operating system intervention, significantly reducing overhead. Measurements on the T3E have demonstrated one-way message latencies (half of a round-trip message exchange) to PEs three network hops away of 2.7 µs, including software overhead.

7 Barrier/eureka synchronization

Barriers allow a set of participating processors to determine when all processors have signalled some event (typically reached a certain point in their execution of a program). Eurekas allow a set of processors to determine when any one of the processors has signalled some event. Users might use eurekas to signal the completion of a parallel search. The operating system might use eurekas to interrupt some or all remote processors.

Barriers are heavily used in many parallel applications, and their performance can affect the ability to the scale the application. As an example, we have worked with proprietary meteorological codes that perform on the order of one barrier every 200 µs in a 128-processor system. At this rate, and additional 15 µs to perform a software barrier (see Section 8) would add over 7% to the application runtime.

7.1 Barrier/Eureka Synchronization Units

The T3E provides a set of 32 barrier/eureka synchronization units (BSUs) at each processor. The BSUs are accessible as memory-mapped registers and are allocated and protected via the address translation mechanism. A set of processors can be given access to a particular BSU through which they can perform barrier and/or eureka synchronization. Multiple disjoint sets of processors may reuse the same logical BSU.

A BSU at a processor can be in one of several states. Processors can read this state and perform operations on the BSU via load and store operations. Tables 2 and 3 show a subset of the local states and operations.

State	Description
S_EUR	A eureka event came
S_EUR_I	A eureka came, interrupt signalled
S_ARM	Barrier is armed
S_ARM_I	Barrier is armed, an interrupt will occur on completion
S_BAR	Barrier just completed
S_BAR_I	Barrier just completed, interrupt signalled

Table 2. Barrier/Eureka Synchronization Unit States

Operation	Description
OP_EUR	Send eureka
OP_INT	Set to interrupt when a eureka event occurs
OP_BAR	Arm Barrier
OP_BAR_I	Arm Barrier, interrupt on completion
OP_EUR_B	Send eureka and arm barrier

Table 3. Barrier/Eureka Synchronization Unit Operations

Figure 6 (a) shows the state transitions for a simple barrier. An OP_BAR takes a given BSU from the S_BAR state to the S_ARM state. When all participating processors have armed their barriers, the network delivers completion notifications that takes the BSUs to the S_BAR state, at which point the BSUs are ready for the next barrier synchronization. The barrier can be made to interrupt upon completion by joining it with the OP_BAR_I operation.

A simple, re-usable eureka event, shown in Figure 6 (b), is a three-state transition that includes a barrier to establish that all processors have seen the eureka before performing another eureka. Starting in the S_BAR state, a single processor performs an OP_EUR. This takes its BSU to the S_EUR state, and causes the network to deliver eureka events to all other participating BSUs, taking them to the S_EUR state as well. As processors observe the eureka, they indicate this by performing an OP_BAR. The triggering processor can perform a combined OP_EUR and OP_BAR using an OP_EUR_B.

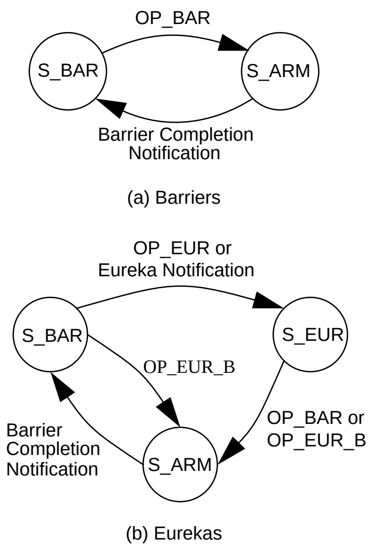
OP_BAR
S_BAR
S_ARM
Barrier Completion Notification
(a) Barriers
OP_EUR or Eureka Notification
S_BAR
S_EUR
OP_EUR_B
OP_BAR or OP_EUR_B
Barrier Completion Notification
S_ARM
(b) Eurekas

Figure 6. Simple barrier and eureka local transitions

Once all processors have joined the barrier, the network delivers barrier notifications that place all BSUs in the S_BAR state, ready for the next eureka event. As with barriers, eurekas can optionally be set to interrupt upon notification.

Note that the BSU interface allows “fuzzy” barriers, in which a processor can perform unrelated work between joining a barrier and checking for completion.

7.2 Embedded barrier/eureka trees

Rather than dedicate physical wires for barrier/eureka synchronization, the T3E embeds logical barrier/eureka networks into the regular 3D torus interconnect. Small barrier/eureka packets are passed over the network to signal events. Barrier/eureka packets use their own virtual channel and are transmitted with highest priority. This scheme keeps global barrier/eureka latency to less than that of a single remote memory reference, while making more efficient use of limited network wires.

To embed the barrier/eureka trees in the network, each network router maintains a register for each of the 32 BSUs. This register allows the node to be configured as an internal node in the BSU’s logical tree. The register indicates which of the six network directions plus the local processor are children in the tree, and which direction (if any) is the parent. It also keeps track of the set of children that have signalled a barrier.

When all children have signalled a barrier, or when any child signals a eureka, a corresponding signal is sent up to the parent (by sending the parent a barrier/eureka packet), or, if the node is the root of the tree, completion signals are sent to all of the children (also via barrier/eureka packets). Completion signals are broadcast hierarchically to all children in a barrier/eureka tree, and result in appropriate changes to the child BSUs, optionally interrupting the leaf processors.

8 Performance

This section presents performance measurements taken on early hardware. These measurements do not represent a complete performance profile of the T3E, but rather are intended to illustrate the efficacy of the primitives discussed in this paper. The absolute values of many of these measurements are likely to evolve, and, in particular, the DRAM timing parameters of the measured systems were set to less aggressive values than production systems will use.

We used a series of small, micro-benchmarks to measure various communication and synchronization latencies and throughputs. Code was written in C and compiled using Cray’s standard T3E compiler. Standard Shmem library routines were used for barriers, atomic memory operations and memory-to-memory copies. Message-passing was performed by manipulating the MQCW and shared memory directly from the C code. The memory pipelining and strided reference benchmarks used assembly-language kernels in order to efficiently schedule Gets and loads.

With the exception of the barrier benchmark, all measurements were taken on a 20-processor machine, with 300 MHz processors. The barrier benchmark was run on a 64-processor machine built with prototype parts running at 200 MHz (50 MHz sysclock); results were scaled to reflect times on a full-speed system.

Figure 7 shows the effect of pipelining on global memory bandwidth. The benchmark loads an array of 16K entries (128 KB) from a node three network hops away using vector Gets and E-register loads. The number of E-registers used to hide the latency is varied from 1 to 256. For 32 or more E-registers, a loop preamble first issues Gets to all the E-registers. The main loop then repeatedly loads a block of 32 E-registers, issues Gets into the vacated E-registers and increments the block pointers. A loop postamble loads the remaining E-registers values.

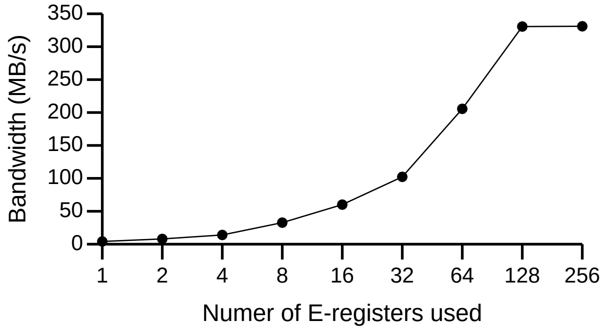

Figure 7. Effect of pipelining in the memory interface

Realized Get bandwidth increases with the number of E-registers used. Using 8 E-registers, the realized bandwidth is 32.8 MB/s. The round trip latency (time to store out the vector Get command, perform the Get operation from remote memory, load the 64 byte result into processor registers, plus loop overhead) is thus 64B/(32.8 MB/s) = 1.86 µs. At this latency, 128 E-registers (1 kilobyte) provide sufficient buffering to sustain the maximum transfer rate, which appears to be limited by a bottleneck in the E-register control logic.

The relatively high latency for remote references[1] will limit bandwidth for smaller transfers. Figure 8 shows the effect of startup latency on realized bandwidth for memory-to-memory copies using the shmem_get() and shmem_put() library calls. Source and target nodes are three network hops away (average distance in a 64 processor machine).

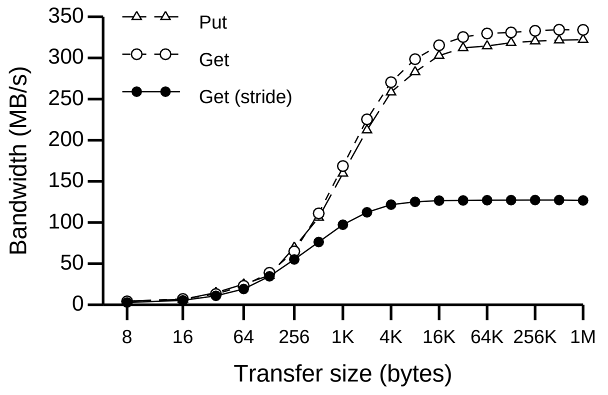

Figure 8. Effect of startup latency on realized bandwidth

The Get transfer performs stride-1 Gets from remote memory and Puts to local memory. The Put transfer performs stride-1 Gets from local memory and Puts to remote memory. The strided Get transfer performs stride-10 Gets from remote memory and stride-1 Puts to local memory; each vector Get command is broken up into 8 single-word Get packets that traverse the network separately.

The stride-1 transfers achieve near asymptotic bandwidth for lengths of about 16 KB and beyond. The $N_{1/2}$[2] is approximately 1 KB. Due to the lower asymptotic rates, the strided Get transfer achieves near peak bandwidth at about 4 KB, and has an $N_{1/2}$ of approximately 256 bytes.

1. This is one metric by which the T3E is worse than the ECL-based T3D.
2. Length for which 1/2 asymptotic bandwidth is achieved.

Figure 9 illustrates the ability to load strided (or gathered) data into the microprocessor by first fetching it into aligned blocks of E-registers using strided vector Gets. The graph shows the asymptotic bandwidth of local loads through E-registers vs. the stride of the reference stream. With cacheable references, of course, realized bandwidth falls off as stride increases because an increasing fraction of the cache line is ignored, resulting in a large-stride BW of only 1/8th (for an 8-word cache line) of the stride-1 bandwidth. Using Gets, however, stride-independent bandwidth is possible.

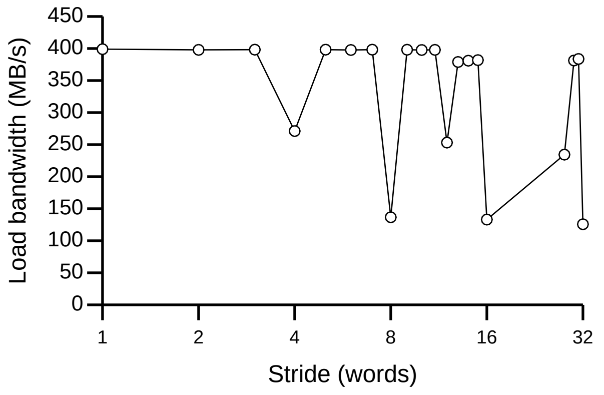

Figure 9. Performance of strided memory references using E-registers

The reduced bandwidth for certain strides in Figure 9 is an artifact of the local memory system, which contains 8 independent, single-word banks. All strides up to 16 are shown, as well as 28, 31 and 32. Strides that are a multiple of 8 load all data from a single bank, and strides that are a multiple of 4 load all data from a pair of banks. Other strides use four or eight banks and achieve full bandwidth.

As discussed in Section 4, E-register data is loaded with I/O space loads, which are less efficient than cacheable loads in the 21164 processor. Cacheable memory loads can be performed at roughly twice the bandwidth of E-register loads in the T3E, so are preferable for stride-1 or stride-2 reference streams.

Figure 10 shows the performance of atomic memory operations. All 16 processors in this benchmark perform AMOs to the same synchronization variable located at processor 0. The graph shows average latency for an AMO vs. rate of operations. The maximum sustained rate of fetch_&_add operations is approximately 4.5 Mops/s (222 ns per AMO). Fetch_&_inc has a lower latency, due

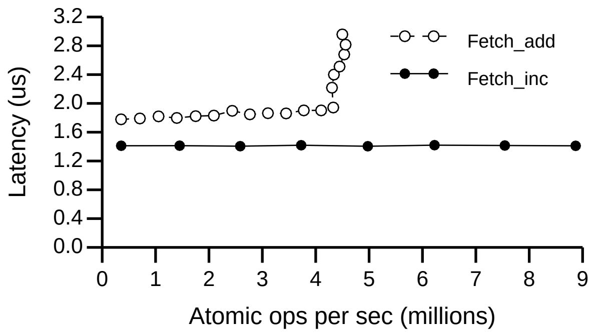

Figure 10. Atomic memory operation performance

to its smaller network packet size and the fetch_&_inc buffer at the memories, and has a substantially higher sustainable bandwidth. Sixteen processors making one reference at a time were unable to saturate the memory system. By making pipelined requests, 16 processors were able to saturate the system at approximately 26 M fetch_&_incs/s (39 ns per fetch_&_inc).

Figure 11 shows the performance of the SEND/MQCW messaging mechanism. In this benchmark, processors 1 through 15 exchange pairs of messages with processor 0. The graph shows average round trip latency (processor x sends to processor 0, processor 0 receives message and sends a response message back to processor x, processor x reads the response message) vs. the rate of message exchanges. Round trip latency is approximately 5.5 µs and a maximum exchange rate of 932M/s was achieved. This corresponds to an occupancy of 1.07 µs at processor 0 to receive a message and send a reply.

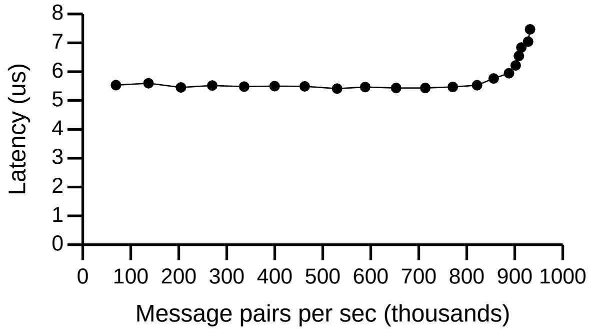

Figure 11. Messaging performance

Figure 12 compares the performance of a software barrier with the T3E hardware barrier as the number of participating processors is varied. The software barrier is an efficient, log2(n) stage barrier based on Puts to shared memory. The latency shown is the average time to perform a global barrier over 50 consecutive barriers. At 56 processors, the hardware barrier has approximately 1/7th the latency of the software barrier. Extrapolation of the curves indicates that this factor will be about 15 for a 1024-processor system.

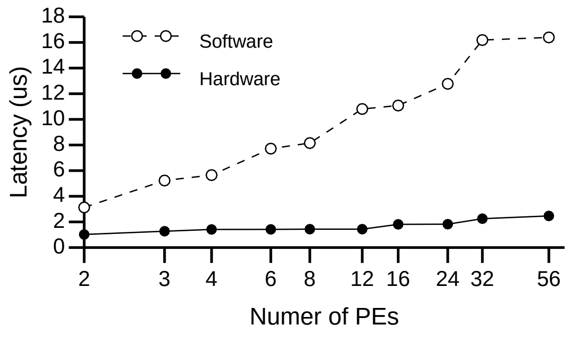

Figure 12. Barrier performance

9 Related Work and Discussion

We discussed the direct predecessor to the T3E in Section 2. This section discusses some other related work.

Early message-passing machines, such as the NCUBE [37] or iPSC/2 [5], required operating system calls to perform any communication between processors. More recent systems have provided user-level messaging facilities. The Connection Machine CM-5, for example, provides a user-level network interface via memory-mapping [46]. Message receipt, however, requires cooperation of the remote processor, interfering with its work and necessitating tight coupling of the sending and receiving processors [22].

The Alewife project at MIT addressed that issue in its messaging support by providing a fast interrupt mechanism in the Sparcle processor [2]. Similarly, the J machine [34] and the proposed interface of Henry and Joerg [16], integrate message handling mechanisms directly into the processors. These designs, of course, prohibit the use of commodity processors. Other system designs, such as the Intel Paragon [21], Meiko CS-2 [20] and *T [33] delegate message processing to a dedicated coprocessor.

All of the above designs require that messages be processed in order as they arrive. The T3E, on the other hand, deposits incoming messages directly into their specified queues in user memory. This decouples message receipt from the compute processor, allowing it to process messages when it is ready, and allows efficient receipt of messages not belonging to the current process. By controlling the MQCW interrupt thresholds or polling frequency, software can also listen to different message queues with varying attentiveness. This mechanism combines the flexibility of shared memory with the one-way latency of message passing. The price for this flexibility can be increased latency over a more tightly integrated design; all messages go through the local memory before being consumed by the processor.

The mechanism in the T3E for *sending* messages is quite similar to those in Alewife, the NIC (network interface chip) [16] and others; the message is first assembled by writing it into a set of external registers and then atomically launched into the network. This mechanism is as fast as can be expected without custom modification of the processor.

A number of other recent, large-scale machines have provided direct, hardware support for shared memory. These include the Stanford DASH [26], Kendall Square KSR-1 [23], MIT Alewife and HP/Convex Exemplar. One noticeable difference between these systems and the T3E is that they cache remote data, while the T3E does *not*. These machines rely on locality of reference, and provide very little pipelining in the global memory subsystem. In addition, cache coherence interactions can degrade performance of explicitly parallel codes with software-managed communication.

The T3E instead emphasizes sustainable memory and communication bandwidth. E-registers support pipelined global memory requests and the stream buffers and lack of a board-level cache increase local memory bandwidth. Performance on the T3E is likely to suffer by comparison on dynamic, irregular codes, and shine on memory-intensive codes and/or codes with large amounts of inherent communication.

We are not aware of other machines that include centrifuge support for controlling data distribution. The RP3 had a simpler mechanism that allowed pages to be either allocated at one node or interleaved across some number of nodes in their circular, dance-hall interconnect [39]. The mechanism for performing E-register commands in the T3E is similar to that in the NIC [16]. Both of these designs use the *address* of a load or store to specify a command and an external register number. In the NIC, the data path is used to move message data between the processor registers and network interface messaging registers (similar to E-register loads and stores in the T3E). The T3E also uses the data path for storing global memory addresses for E-register operations. This method of extending the physical address space of the microprocessor by storing virtual addresses and using remote hardware translation is unique as far as we know.

The Stanford FLASH [25] and Wisconsin Typhoon [42] designs incorporate fully functional, programmable protocol processors at each node. This provides the flexibility to implement any number of protocols and communication/synchronization mechanisms. While this is a particularly valuable feature for a research vehicle, and offers the promise of protocols tailored for specific applications, the performance of a protocol processor in unlikely to match that of a hardwired implementation. The estimates in the FLASH paper [25], for example, indicate that protocol processor occupancy in most cases is just hidden by the transfer time for the 128-byte cache lines. A smaller transfer unit or a faster memory pipe would expose the processing latency and create a bottleneck. The T3E provides support for single-word (8 byte) memory transfers, necessitating the hardwired approach. The FLASH and Typhoon projects also differ from the T3E in their focus on global cache coherence and lack of emphasis on pipelining in the memory system.

Previous machines have provided mechanisms for global processor synchronization. The CM5 has a separate control network that provides barriers and other synchronization [46]. Our opinion (as evidenced by the change from the T3D) is that the performance of a separate dedicated network does not justify the cost. [1] The CM-5 control network is also single user, requiring that it be drained and saved on a context switch. T3E barrier state does not require saving on a context switch, and all synchronization variables are held in normal user memory. The RP3 [39] and NYU Ultracomputer [15] both included multi-stage combining networks in their designs. While general combining networks are quite powerful, they can be expensive and/or complex to implement (the IBM group estimated that it would increase their switch cost by 6-42 times in the technology of the day [40][2] and later dropped the combining network). The (inexpensive) T3E barrier network is of course a special case of general combining networks. Coupled with a rich set of atomic memory operations and scalable, software synchronization algorithms [30][32] where necessary, this appears to be a good solution.

10 Summary

This paper described the communication and synchronization features of the Cray T3E, a distributed shared memory multiprocessor scalable up to 2048 processors using a high-bandwidth, 3D torus interconnect. The T3E uses a commodity microprocessor surrounded by a custom shell – based on a large set of external registers (E-registers) – that allows the processor to fit more naturally into a large-scale system.

E-registers provide two main features: they extend the address space of the microprocessor to support very large machines, and they dramatically increase the available pipelining in the memory system. They also serve as the interface for message sending and atomic memory operation synchronization.

The E-registers support up to several kilobytes of outstanding global memory references and can efficiently support single word accesses (strides and gathers). The integrated centrifuge and remote address translation allow the entire physical memory of the machine to be accessed without a TLB fault. The centrifuge supports simple addressing for message-passing codes (PE field in the upper bits of the address) and more complicated distributions for languages such as HPF.

Messaging in the T3E is tightly integrated with the shared memory and is performed at user level. Message queues reside in normal shared memory, and can be of arbitrary size and number. Message transmission, however, involves only a single, one-way traversal of the interconnect; arbitration for queue space, storage of the message, and notification of the remote processor (by interrupt or by polled status) are performed atomically by the hardware. Of course, using the shared memory and synchronization primitives, other implementations of messaging such as "receiver-pull" are possible.

The T3E provides a rich set of atomic memory operations, including the universal primitive compare_&_swap. These operations can be performed on any memory location and have an observed sustainable rate of approximately one per 200 ns (one per 40ns for fetch_&_inc).

Perhaps the most important synchronization primitive in the T3E is the hardware barrier. The hardware barrier outperforms a hierarchical software barrier by a factor of 7 with 56 participating processors, and extrapolations indicate that the factor will grow to approximately 15 with 1024 participating processors. Moreover, because the barrier trees are virtual, using packets over the existing data network, the hardware barrier is almost free. The only cost is a set of registers and a small amount of logic on each of the router and control chips.

Finally, a number of features have been designed to ease the task of the operating system. Remote memory translation and the magical memory mover allow each node to independently manage its own physical memory. This eliminates the TLB shootdown problem and significantly simplifies shared memory allocation. Virtualizing the barrier networks relieves the burden of managing access to the physical barrier network; there are multiple virtual networks and their embedding is completely flexible. Lastly, several features of the T3D have been moved from the system to the user domain: bulk data transfer is now performed using E-registers, and message queues and synchronization variables are held in normal user memory.

Acknowledgements

Design of the T3E was a large undertaking involving lots of talented people. Credit for much of the work presented in this paper belongs to the other architects of the T3E, Steve Oberlin and Rick Kessler. Thanks also to Mark Birrittella, Eric Fromm, Dan Hendrickson, Randy Passint and Greg Thorson for their design work and architectural contributions. Thanks to Bill Dally for his consultation on this project. Karl Feind and Al Rivers were particularly helpful in obtaining performance measurements for this paper. The work presented in this paper was supported in part by the Advanced Research Projects Agency under Agreement No. MDA972-95-3-0032 dated 12 January, 1995.

References

[1] Agarwal, A., "Limits on Network Performance," *IEEE Trans. on Parallel and Distributed Systems*, pp. 398-412, October, 1991.

[2] Agarwal, A., R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie and D. Yeung, "The MIT Alewife Machine: Architecture and Performance," *Proc. 22nd International Symposium on Computer Architecture*, pp 2-13.

[3] Arpaci, R. H., D. E. Culler, A. Krishnamurthy, S. G. Steinberg and K. Yelick, "Empirical Evaluation of the CRAY-T3D: A Compiler Perspective.", *Proc. 22nd International Symposium on Computer Architecture*, pp 320-331, June 1995.

[4] Black, D. L., R. F. Rashid, D. B. Golub, C. R. Hill and R. V. Baron, "Translation Lookaside Buffer Consistency: A Software Approach," *Proc. ASPLOS-III*, pp. 113-122, April 1989.

1. Another way to view this, is that for the same cost, overall performance would be better served by dedicating *all* the wires to the primary communications network.

2. Surely this would be less expensive today with the use of high-density CMOS ASICs.

[5] Bradley, D. K., “First and Second Generation Hypercube Performance,” Technical Report UIUCDCS-R-88-1455, University of Illinois at Urbana-Champaign, September 1988.

[6] Chapman, B., P. Mehrotra and H. Zima, “Vienna Fortran - A Fortran Language Extension for Distributed Memory Multiprocessors,” ICASE, NASA Langley Research Center, 1991.

[7] Chorus Systems, *CHORUS Kernel v3 r4.2 Specification and Interface*, CS/TR-91-69.1, 1993.

[8] Cray Research, Inc., *CRAY T3D System Architecture Overview*, 1993.

[9] Cray Research, Inc., *Application Programmer’s Library Reference Manual*, SR-2165, 1994.

[10] Digital Equipment Corporation, *Alpha AXP Architecture Handbook*, 1994.

[11] Digital Equipment Corporation, *DECchip 21064-AA Microprocessor Hardware Reference Manual*, 1992.

[12] Digital Equipment Corporation, *Alpha 21164 Microprocessor Hardware Reference Manual*, 1995.

[13] Fox, G., S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng and M.-Y. Wu, *Fortran D Language Specification*, Rice University, 1991.

[14] Geist, A., A. Beguelin, J. Dongarra, R. Manchek, W. Jiang and V. Sunderam, *PVM: A Users’ Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994.

[15] Gottlieb, A., R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, “The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer,” *IEEE Trans. on Computers*, C-32, pp. 175-189, February 1983.

[16] Henry, D. S. and C. F. Joerg, “A Tightly-Coupled Processor-Network Interface,” *Proc. Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 111-122, October 1992.

[17] Herlihy, M., “Wait-Free Synchronization.” *ACM Trans. on Programming Languages and Systems*, pp. 124-149, January 1991.

[18] Herlihy, M., “A Methodology for Implementing Highly Concurrent Data Objects.” *ACM Trans. on Programming Languages and Systems*, pp. 745-770, November 1993.

[19] High Performance Fortran Forum, *High Performance Fortran Language Specification Version 1.1*, Rice University, November 10, 1994.

[20] Homewood, M. and M. McLaren, “Meiko CS-2 interconnect Elan-Elite design,” *Proc. Hot Interconnects*, August 1993.

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

[22] Karamcheti, V. and A. A. Chien, “A Comparison of Architectural Support for Messaging in the TMC CM-5 and the Cray T3D”, *Proc. 22nd International Symposium on Computer Architecture*, pp 298-307, 1995.

[23] Kendall Square Research, *KSR-1 Technical Summary*, 1992.

[24] Koeninger, R. K., M. Furtney and M. Walker, “A Shared-Memory MPP from Cray Research,” *Digital Technical Journal*, 6(2):8-21, 1994.

[25] Kuskin, J., D. Ofelt, M. Heinrich, J. Heinlein, R. SImoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum and J. Hennessy, “The Stanford FLASH Multiprocessor,”, *Proc. 21st International Symposium on Computer Architecture*, pp 302-313, April 1994.

[26] Lenoski, D., J. Laudon, K. Gharachorloo, A. Gupta and J. Hennessy, “The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor,” *Proc. 17th International Symposium on Computer Architecture*, pp 148-159, 1990.

[27] May, C., E. Silha, R. Simpson, H. Warren, editors, *The PowerPC Architecture: A specification for a new family of RISC processors*, Morgan Kaufmann Publishers, Inc., San Francisco, 1994.

[28] McCalpin, J. D., “Memory Bandwidth and Machine Balance in Current High Performance Computers,” *IEEE Technical Committee on Computer Architecture News*, December, 1995.

[29] McCalpin, J. D., “STREAM ‘standard’ results,” `http://perelandra.cms.udel.edu/hpc/stream/`, July, 1996.

[30] Mellor-Crummey, J. M. and M. L. Scott, “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors,” *ACM Trans. on Computer Systems*, 9(1), pp. 21-65, February 1991.

[31] Message Passing Interface Forum, “MPI: A Message-Passing Interface Standard,” *International Journal of Supercomputing Applications and High Performance Computing*, 3/4, 1994.

[32] Michael, M. M. and M. L. Scott, “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms,” *Proc. 15th ACM Symposium on Principles of Distributed Computing*, May 1996.

[33] Nikhil, R. S., G. M. Papadopoulos and Arvind, “*T: A Multithreaded Massively Parallel Architecture,” *Proc. 19th International Symposium on Computer Architecture*, pp 156-167, May 1992.

[34] Noakes, M. D., D. A. Wallach and W. J. Dally, “The J-Machine Multicomputer: An Architectural Evaluation”, P*roc. 20th International Symposium on Computer Architecture*, pp 224-235, May 1993.

[35] Numrich, R. W., P. L. Springer and J. C. Peterson, “Measurement of Communication Rates on the Cray T3D Interprocessor Network,” *High Performance Computing and Networking, International Conference and Exhibition*, Munich, Germany, April 18-20, 1994, Proc. Volume 2: Networking and Tools, W. Gentzsch and U. Harms, eds. Springer-Verlag, 1994, pp. 150-157.

[36] Palacharla, S. and R. E. Kessler, “Evaluating Stream Buffers as a Secondary Cache Replacement,” *Proc. 21st International Symposium on Computer Architecture*, pp 24-33, April 1994.

[37] Palmer, J. F., “The NCUBE family of high-performance parallel computer systems,” *Proc. Third Conference on Hypercube Concurrent Computers and Applications*, pp. 847-851, January 1988.

[38] Pase, D., T. MacDonald and A. Meltzer, “The CRAFT Fortran Programming Model,” *Scientific Programming*, Vol. 3, pp. 227-253, 1994.

[39] Pfister, G. F. W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfielder, K. P. McAuliffe, E. S. Melton, V. A. Norton and J. Weiss, “The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture,” *Proc. International Conference on Parallel Processing*, pp. 764-771, Aug. 1985.

[40] Pfister, G. and V. A. Norton, “Hot Spot Contention and Combining in Multistage Interconnection Networks,”, *IEEE Trans. on Computers*, C-34, pp.943-948, October 1985.

[41] Price, C., *MIPS IV Instruction Set*, MIPS Technologies, Inc., Rev. 3.2, September, 1995.

[42] Reinhardt, S. K., J. R. Larus and D. Wood, “Tempest and Typhoon: User Level Shared Memory,” *Proc. 21st International Symposium on Computer Architecture*, pp 325-336, April 1994.

[43] Saini, S. and D. H. Bailey, “NAS Parallel Benchmarks Results 3-95”, Report NAS-95-011, April 1995.

[44] Scott, S., “The GigaRing Channel,” *IEEE Micro*, pp 27-34, February 1996.

[45] Scott, S. and G. Thorson, “The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus,” *HOT Interconnects IV*, Stanford University, August 1996.

[46] Thinking Machines Corporation, *CM5 Technical Summary*, November 1992.