Empirical Evaluation of the CRAY-T3D: A Compiler Perspective

Remzi H. Arpaci, David E. Culler, Arvind Krishnamurthy,
Steve G. Steinberg, and Katherine Yelick
Computer Science Division
University of California, Berkeley

Abstract

Most recent MPP systems employ a fast microprocessor surrounded by a shell of communication and synchronization logic. The CRAY-T3D[1] provides an elaborate shell to support global-memory access, prefetch, atomic operations, barriers, and block transfers. We provide a detailed empirical performance characterization of these primitives using micro-benchmarks and evaluate their utility in compiling for a parallel language. We have found that the raw performance of the machine is quite impressive and the most effective forms of communication are prefetch and write. Other shell provisions, such as the bulk transfer engine and the external Annex register set, are cumbersome and of little use. By evaluating the system in the context of a language implementation, we shed light on important trade-offs and pitfalls in the machine architecture.

1 Introduction

In 1991 and 1992 a wave of large-scale parallel machines were announced that followed the "shell" approach [25], including the Thinking Machines CM-5 [15], Intel Paragon [8], Meiko CS-2 [1], and CRAY-T3D [11]. In this approach the core of each node is realized by a state-of-the-art commercial microprocessor and its memory system, surrounded by a shell of additional logic to support global operations, such as communication and synchronization. Based on the announced designs, a simple parallel extension to the C language was designed with the goal of extracting the full performance capability out of this wave of machines[6]. The basic approach was to provide a full C on each node operating out of the local memory, augmented with a rich set of assignment operations on the collective global address space. As the announcements were followed by delivery of the machines, the experiment of implementing the language on the machine and assessing its performance was conducted. For numerous reasons, the T3D provides a very interesting case study: the shell is extremely elaborate, the semantics of the hardware primitives for global operations are at essentially the same level as the language primitives, and many distinct mechanisms exist to perform the same function.

The difficulty this "architecturally interesting" design presents for language implementation is two-fold. First, we need to map language primitives onto the hardware within the compilation framework. Second, to choose the best primitives, we need to establish their performance, which may be a result of complex interactions between the microprocessor and the shell. Thus, our language implementation approach begins by establishing the actual performance of the machine and then tries to minimize the additional costs. To do this, we follow a "gray-box" methodology, where design documents are used to establish the functional characteristics of the hardware and a set of micro-benchmarks are used to characterize its performance empirically. Together these dictate the code-generation strategy.

In this paper we document the results of a "gray-box" language-implementation study on the CRAY-T3D. The language is Split-C, constructed as an extension of gcc, but the study would apply to many languages with similar goals, such as CC++ [4] and HPF [9].

In the remainder of this section we outline the language and the basic machine architecture. In Section 2 we explain our micro-benchmarking methodology and characterize the data access performance of the individual node, including a comparison with a standard workstation using the same microprocessor (DEC Alpha 21064). In Section 3 we explain how the language concept of a global address is mapped to the analogous hardware concept and identify performance concerns that arise. We extend our micro-benchmarking to characterize reads and writes to the global address space in Section 4, and identify semantic problems attributable either to the Alpha or T3D shell. In Section 5 we consider memory operations that overlap communication with computation, and in Section 6 we investigate mechanisms for bulk transfer. We examine a family of synchronization issues in Section 7, and finally come to a close in Section 8, where we illustrate final program performance using a scalable application kernel. In each of the sections, we outline the requirements of the language model as well as the structure, constraints, or performance characteristics of the machine that dictate how the language is implemented.

[0] This work is supported in part by the Advanced Research Projects Agency (N00600-93-C-2481 and DABT63-92-C-0026), by the Department of Energy (DE-FG03-94ER25206), and by the National Science Foundation (CCR-9210260). The authors can be reached at: (remzi,culler,arvindk,sgs,yelick)@CS.Berkeley.EDU

[1] CRAY-T3D is a registered trademark of Cray Research, Inc.

1.1 Language overview

Split-C is a simple parallel extension to C for programming distributed memory machines using a global address space abstraction[6]. It has been implemented on the CM-5, Paragon, SP-1, and a variety of networks of workstations, using Active Messages to implement the global address space[17, 26, 19]. The language has the following salient features:

- A program is comprised of a thread of control on each processor from a single code image.
- Threads interact through reads and writes on shared data, referenced by *global pointers* or *spread arrays*. The type system ensures that the compiler can distinguish local accesses from global accesses, although a global access may be to an address on the local processor. Threads may also synchronize through global barriers.
- To allow the long latency of remote access to be masked, split-phase (or non-blocking) variants of read and write, called *get* and *put*, are provided. For example, given global pointer `P` and local variable `x`, `x := *P` initiates a *get* to the global address `P`, whereas `*P := x` initiates a *put*. The left-hand side is undefined until a *sync* statement is issued; *sync* then waits for completion of all pending *gets* and *puts*.
- Bulk transfer within the global address space can be specified in either blocking or non-blocking forms.
- A form of write, called *store*, is provided to expose the efficiency of one-way communication in those algorithms where the communication pattern is known in advance. Threads can synchronize on the completion of a phase of stores, as in data-parallel programs, or the recipient of stored values may wait for a specified amount of data, as in message-driven programs.

Given the one-to-one nature of threads of control and processors, we often refer to either as the processor without confusion.

1.2 The CRAY-T3D

The CRAY-T3D is a massively parallel processor, consisting of up to 2,048 Alpha nodes with 16 to 64 MB of memory each and a "shell" of support circuitry to provide global memory access and synchronization as part of the interface to the network. Here we discuss the key features of the design; see [11] or [12] for an overview and [5] for a complete functional description.

The DEC Alpha 21064[24] is a 64-bit, dual-issue, second generation RISC processor, clocked at 150 MHz (6.67 ns cycle), with 8 KB instruction and data caches, each with 32-byte lines. The Alpha operates on 64-bit data values, whether integer or floating point, and has only word (32-bit) and long word (64-bit) memory operations. Accesses to smaller data types use powerful byte manipulation instructions. Stores are non-blocking and loads and stores may be reordered, so a memory barrier instruction is required to serialize memory references. The 21064 supports a 43-bit virtual address space, but can only address 4 GB of physical memory.

Cray Research has designed a shell around the microprocessor, and the shell provides several features to support global operation under a primarily shared-memory paradigm; the simplest of these lets a processor access any memory location in the machine through a standard load or store instruction. However, since the physical address space is small, the remote processor number is obtained from one of 32 external registers, called the DTB Annex, indexed by five bits of the physical address. Additional fields in each Annex entry control the mode of remote operation. The Alpha load-locked and store-conditional instructions are used to read and write the Annex registers. The shell also supports an atomic-swap between a shell register and memory and two fetch&increment registers per processor, as well as global-OR and global-AND barriers. The Alpha fetch hint instruction is interpreted by the shell as a binding prefetch into a prefetch FIFO, which is popped by the processor through loads from a memory-mapped address. The Alpha memory barrier instruction ensures that writes and prefetches have been delivered to the shell; an additional status bit indicates whether any remote accesses are outstanding. The shell provides a system-level block transfer engine, which can DMA-transfer large blocks of contiguous or strided data to or from remote memories. Finally, the shell provides a user-level message send FIFO; arrival of a message at the receiver either places the message into a user-level message queue or invokes a specific user thread. Processors are grouped in pairs, share a network interface and block-transfer engine, and all 2-processor nodes are connected via a three-dimensional torus network with 300 MB/s links. The network uses dimension-order routing and incurs a very small latency per hop. These machine features will be discussed in more detail in later sections.

2 Local-Node Performance

In this section we introduce our micro-benchmarking methodology and use it to examine the structure and costs of the local memory system. The results prove critical to understanding the costs and consistency issues of global operations because the memory system is the primary gateway to the shell.

2.1 Micro-benchmarking

Benchmarking massively parallel processors usually refers to measuring the execution time of a set of applications (such as ParkBench [20] or Perfect Club [3]) designed to represent a meaningful workload. This measures the performance of the system as a whole, including the processor, the memory system, and the compiler, which is appropriate if one is taking all of these as fixed. However, the results tell us little about the performance of the individual components of the system, which is important if we are developing a new compiler or, in many cases, optimizing applications.

In this paper we take a different path toward performance evaluation inspired by Saavedra's micro-benchmarks [22, 21]. We treat the system as a "gray box" – admitting that we have some *a priori* knowledge of the system, but that it is both incomplete and unverified. Simple probes are used to determine the parameters and characteristics of the machine. We work from the bottom-up, analyzing the simplest primitives first so the results can be used to help understand increasingly more complex mechanisms. Our probes are written in assembly language so that measurements reflect actual hardware costs, not overhead inflicted by the compiler or message passing library.

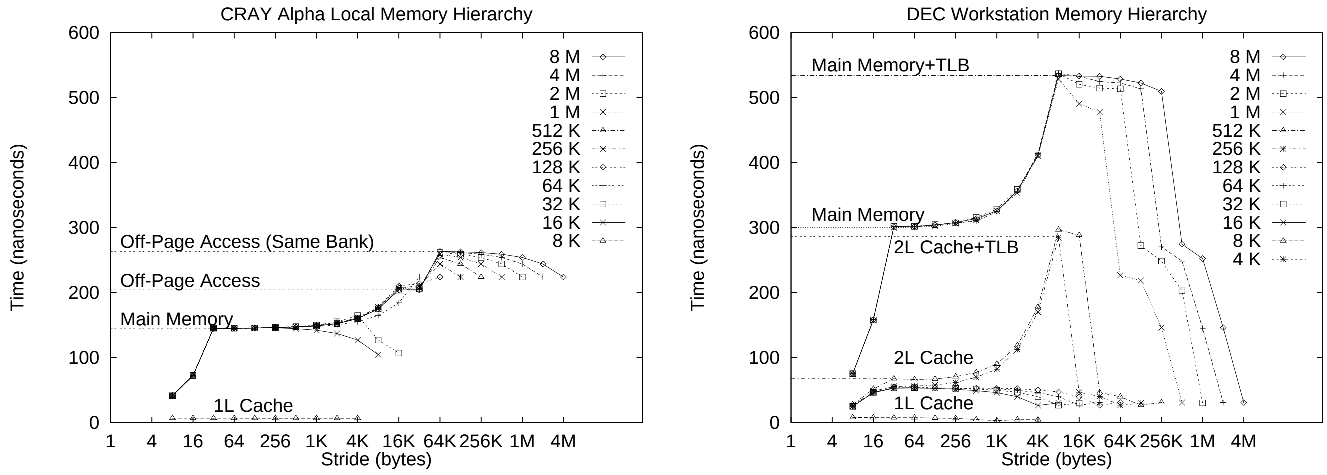

Figure 1: *Local Memory Hierarchy Comparison.* Average read latency for CRAY-T3D and DEC Alpha.

2.2 Local Read Latency

Our first experiment characterizes the latency of a local read. The idea is to generate a controlled stream of addresses to the memory system, *i.e.*, a stimulus, and to observe the response in terms of the average latency per memory access. By varying the parameters that define the address stream, *i.e.*, by varying the stimulus, and observing the variations in the response we can infer specific properties of the memory system. The probe, derived from [22] with slight modifications, simply steps through an array of a given size with a given stride. By increasing the size of the array, we increase the range of addresses within the stream. By increasing the stride, we increase the frequency at which the address varies from low to high. The stimulus is a sawtooth wave, and can be described with the pseudo-code:

```
for (arraySize = 4 KB; arraySize < 8 MB; arraySize *= 2)
  for (stride = 1; stride <= arraySize/2; stride *= 2)
    for (i = 0; i < arraySize; i += stride)
      MEMORY OPERATION ON A[i];
```

We surround the innermost loop with an additional loop that repeats the experiment to mitigate timer granularity and obtain a suitable confidence level. All loop and address calculation overhead is subtracted out so that the reported time reflects only the time of the requisite memory operation. We use two separate probes: reads from memory location A[i], and writes to memory location A[i]. Note that A[i] is an 8-byte word. In each case we plot the average latency curve as a function of stride for a range of array sizes. The results for the read experiment are shown in the left portion of Figure 1.

The graph shows that reads take an average of 6.67 nanoseconds for array sizes up to 8 KB, matching the cycle time (150 MHz) of the microprocessor and the published size of the on-chip first-level data cache, respectively. (Note that while a read issues in one cycle, the result is not accessible for two more cycles.) When the size of the array exceeds the size of the data cache, the reads begin to generate misses. The average access time is now the weighted sum of the hit time and miss time, so an inflection point occurs when every read generates a miss. This point reveals the cache-line size of 32 bytes and the full memory access time of roughly 145 ns (22 cycles). We can tell that the cache is direct mapped because the access time does not drop to the cache-hit time for large strides. If the cache had an associativity of two, for example, there would have been a drop when the stride was half of the array size since the two addresses being accessed would fit in a single set.

As the array size and stride continue to increase we see another rise in latency. This effect is due to the internals of DRAM: strides of 16 KB or greater result in off-page DRAM accesses with each subsequent load. The net result is an additional 60 ns (9 cycles) of latency. At 64 KB strides, the effect is slightly worse, again due to the organization of the memory system. Since there are 4 memory banks, every access with a stride of 64 KB accesses the same bank, thus exposing the full memory-cycle time. This brings the total worst-case memory access to 264 ns (40 cycles).

In most systems, this secondary increase in latency is indicative of the translation look-aside buffer (TLB), which caches a limited number of virtual to physical translations. However, on the T3D the rise occurs at too small an array size (32 KB) for it to be caused by TLB costs, as this would imply a 2-entry TLB with the smallest possible page size of 8 KB. The absence of a rise in latency attributable to the TLB indicates that the T3D designers have chosen to use very large page sizes, possibly reflecting their heritage of not supporting virtual memory. This resolves a potentially difficult code-generation issue regarding global pointers, and will be discussed in Section 3.

The graph also shows that there is no second-level (L2) cache on the T3D; an L2 cache would reveal itself as an intermediary latency between the L1 cache cost and the full memory access time. For comparison, Figure 1 also shows the read latency profile of a DEC Alpha workstation, which contains the same microprocessor (DEC 21064) as the T3D but a different memory system. This graph shows three distinct sets of curves, corresponding to the 8-KB L1 cache, the 512-KB L2 cache, and main memory. The inflection point at a stride of 8 KB is due to TLB misses and indicates the page size used by this workstation.

Notice that a main memory access require 300 ns (45 cycles) on the workstation, but only 145 ns (22 cycles) on the T3D. This supports the vendors' claim that eliminating the L2 cache allows for higher memory bandwidth when streaming through very large data sets, as is typical of vector-style scientific codes[12]. The T3D can deliver roughly 220 MB/s from memory into the processor and the workstation only about half that amount.

2.3 Local Writes

Our second probe, which updates the array in the inner loop, produces the write latency profile shown in Figure 2. The difference between the read and write profiles is dramatic, but not surprising. Much of the difference is due to the write buffer hiding the latency of writes to memory.

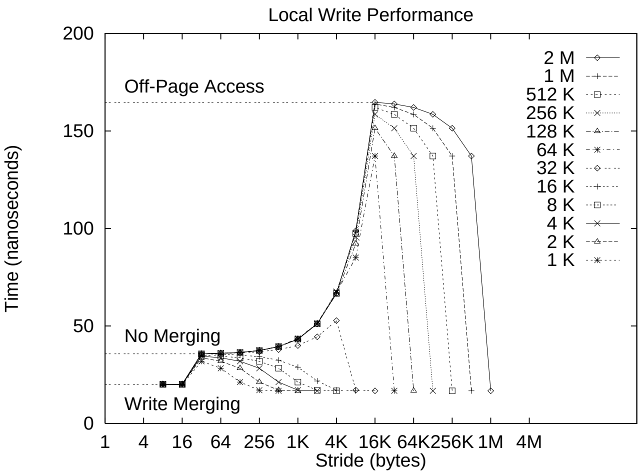

Figure 2: *Local Write Cost.*

The Alpha 21064 has a write-through, read-allocate data cache. We observe the presence of the write buffer by noting that every write to the cache does not incur the full memory latency. Note that the inflection point at a stride of 16 KB occurs because each successive store to the array causes an off-page DRAM access in the memory controller.

For very small strides, the average access time is about 20 nanoseconds (3 cycles), but at a stride of 32 bytes (the block size of the cache), the average access time goes up to 35 nanoseconds. We can draw two conclusions from these measurements. First, at very small strides, successive writes to the same line are written to the same entry in the write buffer. This is a feature known as *write-merging* [24]. Second, since main memory access time is roughly 145 nanoseconds, dividing this by 35 nanoseconds gives an estimated write buffer size of 4. This is corroborated by the Alpha 21064 Reference Manual[7].

This section has provided a detailed examination of the local memory system. We determined that the cost of an off-chip memory access is 23 cycles, that the large page size essentially eliminates TLB costs, and that the write buffer contains four entries and supports write-merging. Later sections will discuss how these costs and mechanisms affect the language implementation.

3 Global Pointers

In this section we turn to the first of our code-generation challenges: the representation of pointers into the global address space and the operations on these pointers. This issue involves a complex aspect of the T3D shell, which provides a level of physical address translation and interacts with the TLB.

3.1 Language Storage Model Requirements

In Split-C, any processor may access any location in the global address space and each processor owns a specific region of the global space, its local region. The local region contains the stack for automatic variables, static or external variables, and a portion of the heap. *Global pointers* reference the entire address space, while standard pointers reference only the portion local to the accessing processor. The following operations are supported on global pointers.

- *Dereference:* The location referenced by the pointer is read or written.
- *Transfer:* The pointer may be passed as a parameter or stored in an object.
- *Arithmetic:* The reference may be manipulated by performing address arithmetic of two forms. *Local addressing* treats the global address space as segmented among processors and acts on a global pointer as the corresponding addressing operations would act on a standard pointer, *i.e.*, an incremented pointer refers to the next location on the same processor. *Global addressing* treats the global address space as linear with the processor component varying fastest. Addresses wrap around from the last processor to the next offset on the first processor.
- *Extraction and construction:* The global pointer can be taken apart to obtain its processor number and local address components. A global pointer can also be constructed from these components.
- *Null test:* A global pointer can be tested for null just as a standard pointer, *i.e.*, by equality with 0.

3.2 T3D Design Constraints

The 21064 implementation of the Alpha architecture supports 43-bit virtual addresses and 32-bit physical addresses (two additional high bits support memory mapped devices). This has a significant impact on how the machine supports shared memory. Many shared-memory machines map all memory accessible to a process into the virtual address space and a virtual address is translated into a *global* physical address [2, 16, 10, 13, 14]. The memory system extracts the node number and physical location from the global physical address and performs either a local memory access or a message transaction with a remote memory controller. For a fully configured T3D, this would require at least 37 bits of physical address. Since there not that many bits available, the T3D shell performs an additional level of address translation using a set of 32 segment registers known as the DTB annex (Figure 3). Each Annex register specifies a processor number and function code. Annex registers are updated at user level with the revised store-conditional instruction at a measured cost typical of off-chip access, 23 cycles. Annex register 0 always refers to the local processor.

The page tables are constructed to provide shared stack and heap segments containing 32 regions of 128 MB each, one per Annex register. The virtual-to-physical translation performed in the Alpha carries the Annex register index through and translates the remainder of the address to a 27-bit physical address, which must be valid on all processors. Thus, one must view a virtual address as a *temporary* global address; its meaning is dependent on the configuration of the Annex. [2]

[2] An alternative would have been to provide truly global virtual addresses and have the operating system manage the Annex transparently. The page tables would associate addresses for the currently mapped remote processors with the appropriate Annex indexes and a fault would occur on reference to an un-mapped remote processor.

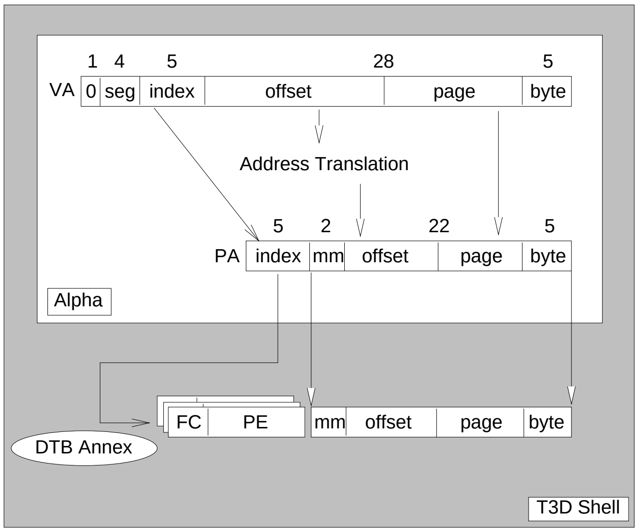

Figure 3: *The DTB Annex.* A 43-bit Virtual Address (VA) is translated to a 34-bit Physical Address (PA), followed by the index into the Annex. Each of 32 Annex entries specifies a function code (FC) and a remote processor (PE). The two mm bits are used to support memory-mapped devices.

A remote access involves storing the destination node number into one of the Annex entries, and then accessing the desired offset in the corresponding segment of the virtual address space. Thus, the Annex presents a new set of registers that the compiler must manage and a short sequence of instructions is required to perform a remote access.

3.3 Global Pointer Representation and Operations

A natural choice for the global pointer representation is a 64-bit entity with the local address stored in the lower 48 bits and the processor number stored in the upper 16 bits. This is the same size as a local address, so transfer is efficient. The Alpha has powerful byte manipulation instructions, so extracting the components of a global pointer and performing arithmetic are also fast. In fact, with the particular virtual memory mapping in the T3D, the 42nd bit of any virtual address is zero. Thus, local address arithmetic is performed on global pointers exactly as it is on local pointers; the result should never overflow into the processor portion of the address. The meaning of a global pointer is independent of the processor that dereferences it, so they can be freely placed in shared data structures.

On dereference, the processor number is extracted and stored in an Annex register, and the annex entry number is inserted in the appropriate bit-field to generate a valid virtual address for use in a load or store instruction. Local arithmetic can be performed on this "internal" global pointer and multiple references can be made without incurring the cost of annex setup.

3.4 Annex Register Management

A key question underlying the use of global pointers is how the compiler manages the Annex registers. The simplest approach is to use only one Annex register and update it on each global access, skipping the Annex update if the compiler can determine that successive accesses are to the same processor. An alternative is to use several Annex registers and keep a runtime table of their entries, with a table lookup added to each global access. This could be combined with compiler analysis to eliminate the table lookup when there is sufficient static information about the pointers.

On the surface, using multiple Annex registers appears to be the better alternative, but it leads to subtle semantic problems and in the end there is no clear performance advantage. The semantic problem occurs if two Annex registers specify the same processor. Because the Annex performs address translation on physical addresses, this allows synonyms: two physical addresses that differ only in their Annex index may map to the same location. Both the cache and write buffer use physical addresses to determine when two accesses are to the same location, so synonyms potentially lead to inconsistent copies. Inconsistencies from caching do not arise on the T3D because the annex entry appears in the high order bits of the address and the cache is direct-mapped, so two synonyms always map onto the same cache line. Unfortunately, the write buffer does admit inconsistencies, since reads can bypass writes. If a write is followed by a read to a synonym, the read may see a stale value from memory while the write is caught in the write buffer. We have produced probes that exhibit this unpleasant phenomenon. Thus, in order to use multiple Annex registers, the compiler must recognize aliases between global pointers or it must generate runtime checks to prevent synonyms.

If a runtime table of Annex entries is kept, the Annex register will need to be selected by hashing the processor portion of the global pointer. Even a simple table lookup requires a memory read and a branch, so the savings relative to a 23-cycle Annex update are small.

A final consideration in using multiple Annex registers is whether this interacts poorly with other aspects of the memory system. Since the Annex register number appears in the high order bits of the virtual address, remote accesses will consume TLB entries. Our concerns on this point where allayed when we determined that the processor was configured to use huge pages and therefore few TLB entries.

3.5 Summary

Global pointers are easily represented on the T3D in a form almost identical to that of local pointers and address arithmetic on global pointers is extremely fast. This capability is due to the 64-bit architecture of the Alpha as well as its powerful byte manipulation instructions. However, the small physical address space of the 21064 presents a serious problem for large shared memory multiprocessors, even when the compiler explicitly manages the global address space. The use of external segment registers to extend the effective physical addressing interacts poorly with the memory system and complicates code generation. The provision of many external segment registers appears to be of small value.

4 Remote Reads and Writes

The shell of the T3D provides a large set of mechanisms for reading and writing to the global address space, each with its own semantics and costs. The compiler writer must determine which of these mechanisms can be used to implement a language primitive and then choose the fastest. This section describes the implementation of the simplest Split-C remote access primitives, *read* and *write*. We define the semantics required by Split-C, characterize the performance of similar T3D mechanisms, and then describe our resulting

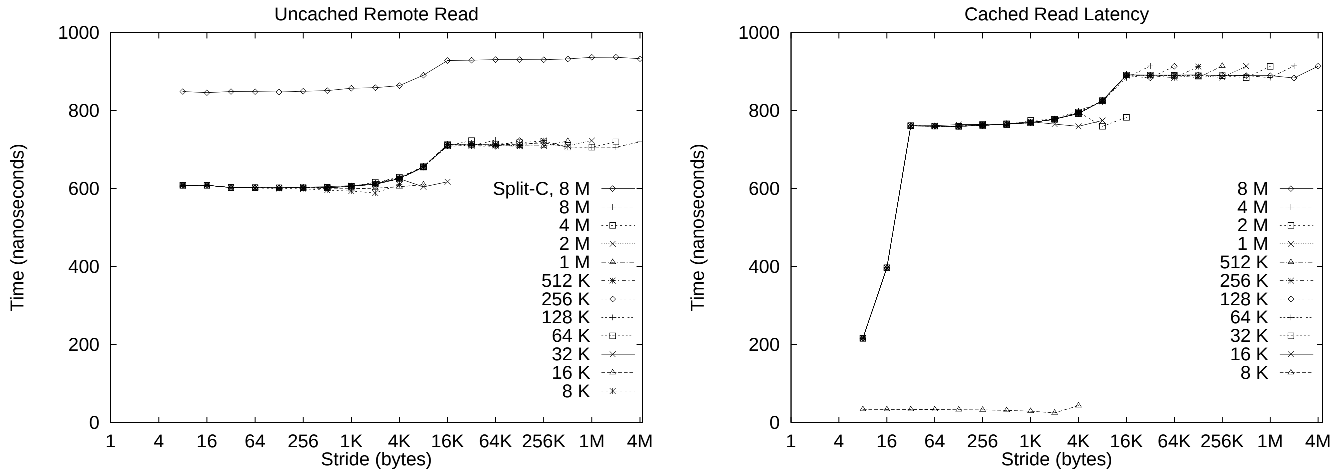

Figure 4: *Remote Read Latency.* Uncached and Cached Read Latency.

implementation. We end with a discussion of two semantic mismatches – discrepancies between the behavior defined by the language and that implemented by the machine – that required awkward workarounds.

4.1 Split-C Global Memory Access

An assignment statement with a global pointer on the right-hand side causes a remote *read*, and one with a global pointer on the left-hand side causes a remote *write*. Reads and writes are blocking operations; a read waits for the requested value to arrive from the remote processor, and a write waits for an acknowledgement that the write is complete. Therefore, these operations appear sequentially consistent to the programmer.

4.2 T3D Remote Reads

The T3D provides two types of remote reads: cached and uncached. Both use the standard Alpha load instruction on a global address, but a function code in the DTB Annex entry specifies the type of read. A cached read updates the requested line (32 bytes) in the the cache and places the requested word into the specified register. An uncached read fetches only the requested word and does not modify the cache.

We determine the latency of remote reads by modifying the local micro-benchmark so that it strides through remote memory. The results are shown in Figure 4. An uncached read costs roughly 610 ns (91 cycles) while a cached read costs 765 ns (114 cycles). As we saw with local-memory operations, strides of 16 KB or greater cause an off-page DRAM access (this time in the *remote* node's memory controller) which increases the average access time by about 100 ns (15 cycles). The cost of cached reads depends on the contents of the local cache. In our micro-benchmark, array sizes that fit in the cache result in local-cache memory-access times since the first time through the array brings the data local for all successive iterations. Cached reads also perform slightly differently for strides of 8 and 16 bytes. By bringing over entire cache lines, cached reads essentially prefetch the next 1 or 3 accesses. Note that all of our measurements are to an adjacent node, with only one processor active; measuring the additional latency through the network reveals roughly a 13 to 20 ns (2-3 cycle) cost per hop.

To put these costs in perspective, remember that a remote uncached read is only three to four times slower than a read from local memory. In fact, the latency to remote memory on the T3D is only 80 ns higher than a main memory access (including TLB miss) on the DEC workstation. This also is significantly faster than similar MPPs: the cost of a read to a remote node on the DASH multiprocessor is roughly 3 μs, and is about 7.5 μs on the KSR [23].

4.3 T3D Remote Writes

The next primitive we examine is the remote write. Just as the remote read is an extension of the load instruction, remote writes use the Alpha store instruction on a global address.

Because stores are handled by the write-buffer, and are therefore non-blocking, we must explicitly poll for the remote acknowledgement in order to match the semantics of the language. The acknowledgement is automatically sent by the T3D hardware and causes a bit in a local shell status register to be cleared. Figure 5 shows the write-latency profile, with blocking remote writes completing in roughly 880 nanoseconds (130 cycles).

One subtlety arose while implementing the blocking write that illustrates the unwanted interactions that can occur between the custom shell and the commodity core. The remote-write status bit is set when there are writes that have left the processor and not returned, but it is clear if there are pending remote writes in the write buffer. Therefore, to poll on this bit, one must first guarantee that the corresponding write(s) have left the buffer. This can be done by issuing a memory barrier instruction or a sufficient number of writes, which effectively increases the cost of the operation.

4.4 Compiler Implications

It was not immediately obvious which mechanism should be used to implement the Split-C *read* primitive. Cached reads offer a higher bandwidth, but are difficult to use because the machine does not guarantee coherence of cached values. If the processor that owns the cache line updates it, the change is not reflected on remote processors that are caching the same line. Reads to the cache line would then obtain stale data. Therefore, if cached reads were used for reading remote values, the cache line would have to be

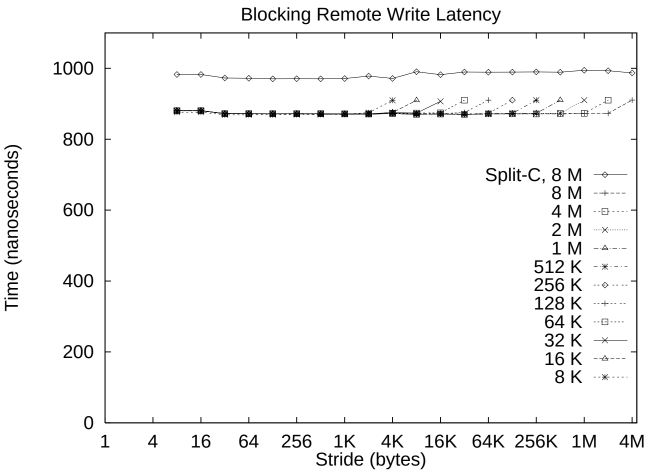

Figure 5: *Remote Write Latency.* The time to write a word into the network and get an acknowledgement.

flushed after the read. This restriction makes cached reads less advantageous since a cache-line flush costs an extra 23 cycles (equivalent to accessing main memory). Languages such as HPF might be able to perform some type of global analysis to determine when data-sharing occurs (and thus when to flush), but this is not possible in a C-like language. Therefore, the Split-C implementation uses uncached reads. Figure 4 shows the total latency as seen by the programmer, which is about 850 ns (128 cycles). The majority of this is attributed to the remote read and Annex set-up time.

The Split-C *write* primitive uses the write mechanism followed by polling and requires a total latency of 981 ns (147 cycles), as seen in Figure 5. However, the operation has a subtle associated cost. When a processor initiates a remote write, the object being written to might be cached by the remote processor that owns the object. In order to maintain coherence, the T3D allows the caches to be operated in a *cache-invalidate* mode. When a remote write request is received, the corresponding cache line is flushed even if that particular cache line isn't currently cached. In other words, we are forced to operate in this mode in the absence of higher-level information, and doing so may generate spurious cache flushes in order to preserve coherence.

4.5 Semantic Mismatches

There are two idiosyncrasies of reads and writes on the T3D, caused by tension between the machine's shell and core, that led to serious semantic mismatches between the machine and the language:

Byte Writes: The Alpha does not support byte store operations. The designers chose to simplify the memory interface by not supporting byte read/write operations, but instead provided a family of byte manipulation instructions that operate on register values [24]. A byte store operation could therefore be implemented as a read-modify-write sequence. However, on a multiprocessor like the T3D, we cannot guarantee correct execution of a byte-store operation when multiple processors are updating the same word. If two processors attempt to update a byte at the same time, one update will clobber the other. Note that a solution using the load-locked and store-conditional instructions is no longer possible, since these instructions were consumed by annex manipulation. Section 7 looks at an alternative way to support byte updates.

Global-Local Consistency Issues: Writes through global pointers wait for the operation to complete irrespective of whether the location being accessed is local or remote. However, writes through normal local pointers, which appear as standard Alpha store operations, are buffered in a write buffer. This could result in access order violations since the global address space is also accessible through standard local pointers. A *read* through a local pointer could overtake a *write* operation issued earlier also through a local pointer but to a different location. In such a situation, another processor could observe this reordering of accesses, resulting in a violation of sequential consistency semantics. To avoid consistency violations, we could either ensure that writes through local pointers are not caught in the write buffer by using the memory barrier instruction or by explicitly privatizing the global address space. We currently choose the latter, and require the programmer to insert explicit calls around any region where shared global data may be accessed through local pointers.

5 Split-Phase Accesses

5.1 Split-C Get and Put Operations

The Split-C *get* and *put* primitives are split-phase, non-blocking operations that can be used to overlap communication and computation. A *get* operation initiates a prefetch from a remote address to a local address, while a *put* operation initiates a non-blocking write to a remote location. The *sync* operation waits for all outstanding split-phase accesses to complete. The *get* and *put* operations lead to weak consistency semantics because the accesses initiated between two *syncs* can no longer be ordered.

5.2 T3D Binding Prefetch

The T3D's prefetch mechanism uses the Alpha fetch instruction to provide non-blocking reads of remote addresses. The fetch instruction informs the support circuitry to fetch a word from a remote node while the processor continues computation. The fetch request is placed in the write buffer and eventually sent to the specified remote processor. When the processor actually needs the requested data it must pop the word(s) from the 16-entry memory-mapped prefetch queue by issuing a load instruction.

To examine the performance of the prefetch mechanism we used a simple benchmark that measures the average latency of prefetching one or more words. The benchmark groups n remote read requests into a series of prefetch instructions and then pops the results from the prefetch queue and stores them to local memory. The results indicate the extent to which the prefetch mechanism can overlap remote access costs (Figure 6).

One (prefetch, pop from queue, store) sequence is slower than a blocking read by about 15 cycles, but issuing four prefetches, popping the queue four times, and storing all the results to local memory is significantly faster than four blocking reads. The prefetch essentially allows the network to be pipelined, thereby masking most of its latency. An important conclusion we can draw from Figure 6 is that the remote latency is almost entirely hidden as the size of the group approaches 16. Therefore, the choice of 16 for the size of the prefetch queue seems to be a reasonable one. Note that when less than 4 prefetches are issued, a memory barrier must be inserted before the pop from queue. This guarantees that the prefetch has left the processor.

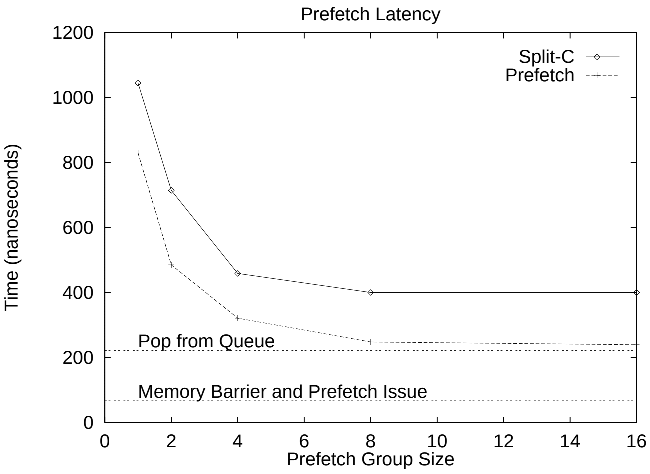

Figure 6: *Prefetch Latency.* Average latency for prefetches based on number issued.

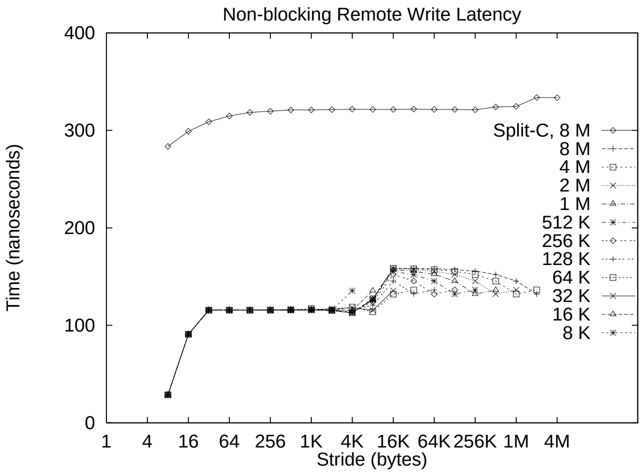

Figure 7: *Non-blocking Write Latency.* The average time to write to a remote processor (no acknowledgement).

Further instrumentation of the prefetch mechanism reveals the following cost breakdown:

```
Prefetch Issue: 4 cycles
Memory barrier: 4 cycles
Round Trip:     80 cycles
Prefetch pop:   23 cycles
```

This breakdown shows that the prefetch mechanism allows about 75% of the cost of a remote fetch operation to be overlapped with other useful work. When prefetches are issued in groups of 16, we spend 31 cycles per prefetch/pop operation. Subtracting the 23 cycles required for the pop and the 4 cycles required for the prefetch issue leaves only 4 cycles of network latency and address manipulation overhead.

5.3 T3D Non-blocking Writes

The Alpha store instruction is a non-blocking write operation; the processor simply issues a store to the write-buffer and continues computation. Figure 7 shows the results of the familiar micro-benchmark modified to write to a remote processor. Writes with a stride of less than 32 bytes reveal the write-merging behavior of the Alpha (similar to Figure 2). Larger strides show an average cost of 115 nanoseconds (17 cycles) per write. At 16K, we once again see the sensitivity to remote-memory DRAM page misses.

5.4 Compiler Implications

We can implement the Split-C *get* operation with the prefetch mechanism. However, some additional work is required since the *get* operation also specifies a local address that serves as the target for the fetched value. The target address needs to be stashed away when the fetch is issued. This can be accomplished if the compiler has sufficient information to match the prefetches and the corresponding *sync*s, which might not be possible in the presence of unstructured control constructs. In the general case, we maintain a table for storing these local addresses. The target address is stored in this table at prefetch initiation time. When a *sync* operation is encountered, or when the number of outstanding fetch operations reaches 16, a memory barrier is issued, and the values in the prefetch queue are dequeued and stored in the appropriate target addresses obtained from the table.

The Split-C implementation introduces two additional pieces of overhead over the raw hardware prefetch mechanism: the table update and the store into the local address when the prefetch is complete. The cost of the table update and lookup operation is 10 cycles, and the cost of the local store operation is 3 cycles. Figure 6 reveals the final Split-C *get* cost, which includes annex set-up time, table management, and all other overheads.

The implementation of *put* is straight-forward from the non-blocking write, and requires the annex setup as well as a few additional checks. The performance of the Split-C primitive is shown in Figure 7. From this we see that *put* average latency is about 300 ns (45 cycles).

6 Bulk Operations

6.1 Language Provisions

Bulk transfers occur in Split-C in two different flavors: implicitly when an entire remote structure is accessed by a read, write, get, or put in a single assignment and explicitly through calls to *bulk_read*, *bulk_write*, *bulk_get*, and *bulk_put* for copying contiguous regions of global memory. The compiler transforms structure assignment operations into the explicit bulk operations.

6.2 T3D: Block Transfer Engine

The block-transfer engine (BLT) is essentially a DMA device: given a remote processor number, a pointer to a buffer, and a length argument, the engine initiates a transfer between the two processors' memories. The operation specified can be either a read or a write. The BLT is also capable of performing strided-array accesses. One would think that this would be the natural match for the Split-C bulk routines. However, unlike the hardware primitives we have tested so far, this mechanism for remote memory access is only available through an operating system invocation with high software overheads. As language implementors, we need to compare the BLT with the other remote access mechanisms that the T3D shell provides and answer the obvious question of which mechanism to use in implementing the bulk transfer routines.

To compare the different mechanisms for implementing bulk transfers, the appropriate metric is the bandwidth attained by the mechanisms for large reads. Using each of mechanisms described before (uncached reads, cached reads, the prefetch queue, and the bulk-transfer engine), we implemented and micro-benchmarked four bulk read operations. The left side of Figure 8 compares the resultant bandwidths of each implementation.

As the figure indicates, for very large reads (i.e. greater than 16 KB in size), the bulk-transfer engine achieves the highest transfer rate, peaking at roughly 140 MB/s. For reads between 128 bytes and 16 KB, the prefetch queue attains the maximum bandwidth. At this point, the overhead incurred by the BLT overcomes its superior maximum transfer rate. Though cached reads bring over four words at a time (whereas prefetches can only bring over one), the latency-hiding ability of the prefetch mechanism makes it strictly better than the cached reads, except in the case of 32 and 64 bytes[3]. At these points, the cached read can bring over either one whole cache line or two, and thereby achieve the best performance. Lastly, for very small transfers (8 bytes), the uncached read is optimal.

For bulk writes, we have two mechanisms available: the non-blocking store instruction and the BLT. Figure 8 shows the net difference between the two. Whether the local data to be written is in cache or not, the performance of non-blocking writes is superior to initiating the bulk-transfer engine. The bandwidth for writes from local memory to remote memory (i.e. not in the local cache) peaks at 90 MB/s, and is apparently bus limited.

6.3 Compiler Implications

The implementation of the Split-C *bulk_read* and *bulk_write* routines is an obvious progression from the micro-benchmarks. The key cross-over point occurs at a transfer size of about 16 KB. At this size, the bulk transfer engine should be used instead of the prefetch queue. For simplicity, the prefetch queue is used even for 32 and 64 byte transfers; switching this to cached reads would increase performance. Figure 8 shows the Split-C performance attained.

To implement the bulk get and put operations, we need to compare the initiation time of the various mechanisms. If we use the BLT for implementing a bulk get, we need to wait only for the duration of initiating a BLT operation, and we could overlap the actual transfer with local computation. However, if we use the prefetch mechanism for implementing a bulk get, we find that removing the restriction of completion is not especially beneficial. This follows from the fact that the prefetch queue can only have a maximum of 16 outstanding requests. Unfortunately, since BLT invocation overhead is egregiously high, we must use the prefetch mechanism for implementing bulk gets of certain transfer sizes. The cost of initiating a bulk transfer using the BLT is 180μsec. The prefetch mechanism can read about 7,900 bytes during that time interval. Therefore, the Split-C bulk get uses the prefetch mechanism for all transfer sizes less than 7,900 bytes and the BLT for larger transfers. By similar reasoning, the implementation of bulk put uses the Alpha non-blocking stores for all transfer sizes less than 16,900 bytes and the BLT for larger transfers.

[3] The performance of bulk transfers using cached reads has an inflection point at 8K. This occurs because the cache line flushes (required to maintain coherence) can now be batched into an entire cache flush operation, which is less expensive.

6.4 Summary

We have now seen most of the T3D's shell: cached and uncached reads, writes, prefetching, and bulk transfer. Of these, the prefetch and write are most useful. Prefetch would be the ideal read mechanism if a larger pay-load could be brought over. Since prefetch is so efficient, both cached and uncached reads are of little use. Cached reads are especially difficult to utilize due to the lack of hardware coherence. The block transfer engine is cumbersome, and would be greatly improved if access were from user level. Lastly, writes are the simplest and most efficient data movement mechanism, achieving both low-latency and high-bandwidth data transfer.

7 Bulk-synchronous and Message-driven Computation

In this section we investigate optimizations that are available for applications with structured communication and synchronization patterns. Typically, this structure arises where there is a well-understood global view of the computation allowing information to be *pushed* to where it is needed next. For example, in stencil calculations where the boundary regions must be exchanged between steps, the communication patterns are predefined. In a "bulk synchronous" style, the program proceeds as a phase of purely local computation followed by a phase in which all processors store data into the boundary regions of their logical neighbors. Global barriers between phases enforce all necessary dependences without requiring fine-grain completion of individual stores. In a "message driven" style, a node can proceed with the next phase of computation as soon as it has received its new boundary data. The language provides a means of expressing the lenient synchronization requirements of structured applications, potentially enabling the use of various optimizations at the machine level.

7.1 Language Support: Signaling Stores

Split-C allows the programmer to reason at the global level by specifying clearly how the global address space is partitioned over the processors, as discussed in Section 3. What is remote to one processor is local to another. The `:-` assignment operator, called *store*, stores a value into a global location, but the issuing processor is not necessarily informed of its completion. The extremely weak completion semantics of store mean that only one-way communication is required (*i.e.*, no acknowledgements) and stores can be heavily pipelined. There are two ways of detecting completion of stores. A form of global barrier, *all_store_sync*, returns when all stores issued before the barrier have completed. This is sufficient to support bulk-synchronous execution. Alternatively, completion can be detected locally using *store_sync*, which returns when a specified amount of data has been stored into the region of the address space owned by the local processor. This supports message-driven execution. In many Split-C implementations, signaling stores are the most efficient form of communication when applicable.

7.2 T3D Design Constraints

The T3D offers no store mechanism that avoids acknowledgement upon completion. In this view, the completion

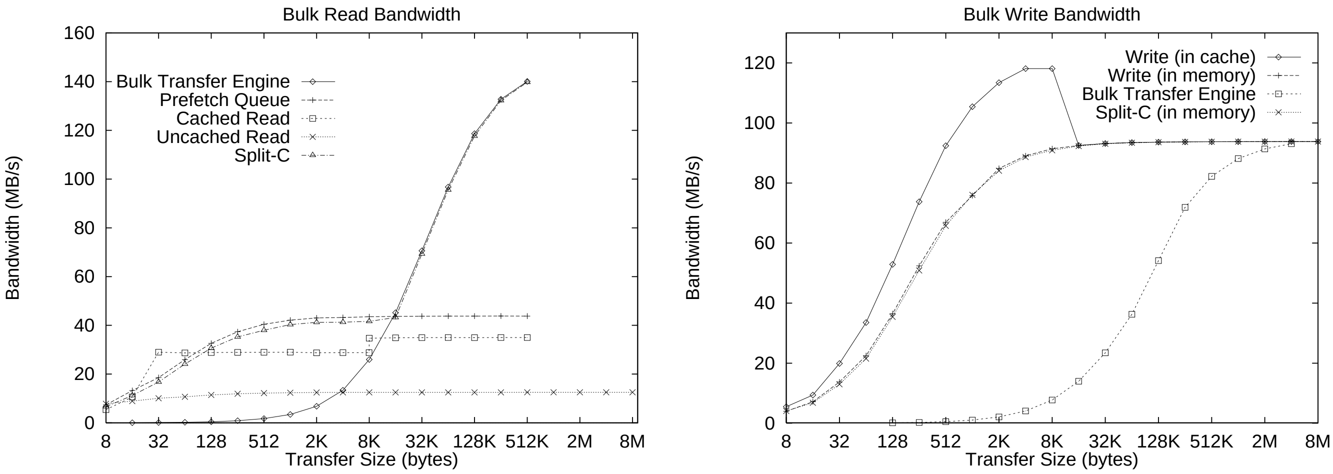

Figure 8: *Bulk Read and Write Comparison.* On the left, all the different remote access mechanisms are used to implement a bulk read. On the right, bulk writes are implemented using the BLT and the store instruction.

semantics provided by the machine are stronger than those specified by the language. There still is a potential performance gain on signaling stores (now essentially puts), since waiting for completion can be deferred and many stores can potentially be issued before waiting.

The key problem with remote reads and writes performed directly in hardware is that the recipient of the data is not informed that the data is in fact present, as required for message-driven programs. This is in contrast to many other Split-C implementations where the global memory operations are constructed upon Active Messages and a store handler can, for example, increment a counter. On the T3D, we must construct essentially the equivalent of Active Messages to provide *notification* to the recipient, apart from the actual data transfer. This brings three other significant components of the T3D shell into the picture: explicit messages, fetch&increment registers, and fuzzy barriers.

7.3 User-level Message Queue

The T3D provides direct access to the network via a message queue. The mechanism is straight-forward: a four word message is composed and a PAL call is issued to atomically put the message into a cache-line sized transfer to the specified destination. PAL code is provided by the Alpha as an architected interface to microcoded functions. We measure the time to inject a message into the network as 813 ns (122 cycles), comparable to the time to perform a remote read.

Unfortunately, the send cost is the fast part in the message transfer. Upon message reception, the processor is interrupted, the message is placed in a user-level queue, and one of the following two actions take place: control is either returned to the original thread or transferred to a specific message handler. The measured cost of the interrupt is 25 microseconds (3700 cycles), and the switch to a message handler adds another 33 microseconds (5000 cycles) on top of this.

7.4 Fetch&Increment Registers

Given the high cost of message receipt due to operating system intervention, we are led to consider ways of constructing message transfers out of the fast shared memory primitives. One approach is to have each node provide a dedicated input queue for every other node. In this case send is again simple (check availability in remote queue, store data, store flag), but determining that a message has arrived requires searching all the queues. What really is required is an N-to-1 queue to bring all the incoming messages or notifications together (which is, of course, what the physical network does). The T3D provides a set of fetch&increment registers to allow such multi-access data structures to be constructed efficiently. The fetch&increment operation is essentially the cost of a remote read, *i.e.*, about 1 microsecond. To deposit a message of four data words and one control word into a remote queue (essentially equivalent to a CMAM Active Message call[26]), takes 2.9 μs, whereas dispatching on the receiving end and accessing the message takes 1.5 μs. As this provides the full power of poll-based Active Messages, it provides a basis for supporting the message-driven *store_sync* as well as the ability to execute a function atomically on a remote processor. We can also provide a correct implementation of *byte_write* using this mechanism, as discussed in Section 4.

7.5 Fuzzy Barriers

Bulk-synchronous execution can be supported, as well, but issuing the *store* is slower than a *put*. The global barrier waits for outstanding stores to complete, performs the start-barrier instruction, and then polls the message queue until all other processors have reached the start-barrier before completing the barrier. This "fuzzy" barrier gives us the ability to place code between the start-barrier, which notifies other processors that the synchronization point has been reached, and the end-barrier, which resets the global-OR bit so the barrier can be used again, and allows the fast hardware barrier to be used while supporting remote memory access *and* user-level message passing. On many other platforms, the Split-C implementation was unable to use the fast native barriers because they did not compose well with other operations.

8 A Case Study: EM3D

In this section we bring together the various components of the language implementation on the T3D in a performance

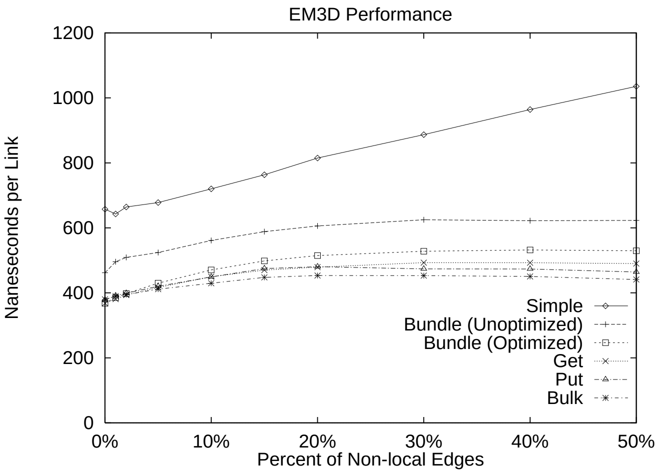

Figure 9: *EM3D Performance.* Performance obtained on several versions of EM3D using a synthetic kernel graph with 16,000 nodes of degree 20 on 32 T3D processors.

study of a scalable Split-C application, EM3D, that models the propagation of electro-magnetic waves through objects in three dimensions [18]. A preprocessing step casts this problem into a simple computation on an irregular bipartite graph containing nodes representing electric and magnetic field values. The computation consists of a series of "leapfrog" integration steps: on alternate half time steps, changes in the electric field are calculated as a linear function of the neighboring magnetic field values and *vice versa.* Specifically, the value of each E node is updated by a weighted sum of neighboring H nodes, and then H nodes are similarly updated using the E nodes. Thus, the dependencies between E and H nodes form a bipartite graph. In the parallel version of EM3D this graph is represented directly using global pointers, and is spread across all of the processors.

We have developed six versions of the application with varying degrees of optimizations using the capabilities offered in Split-C. We consider a collection of synthetic graphs with 500 vertices on each processor with each vertex having a degree of 20, the same inputs seen in [6]. The communication requirements of the problem are scaled by adjusting the fraction of edges in the graph that cross processors in a synthetic graph. The useful performance metric when scaling both problem and machine size is the average time per edge, as shown in Figure 9 for the T3D. This corresponds to reading the value of a neighboring E or H node and a floating-point multiply-add. In the figure, the horizontal axis is the percentage of remote edges.

In the simplest version (Simple) the value associated with an edge is simply read, *i.e.*, a blocking memory read is issued to fetch the possibly remote value. The same value is fetched multiple times if it is required more than once during a single time-step. The other versions rectify this problem by introducing local "ghost nodes" that serve as temporary cache-sites for non-local values. The computation is divided into two phases: the first phase fetches the remote values into the local ghost nodes, and the second phase performs the weighted sum computation based on the ghost node values. With this change, Bundle benefits from reuse of cached values and from better code generation since the communication and compute phases are separated. Next, we optimize the compute phase of the program by loop unrolling and software pipelining. In version Get, we pipeline the remote reads that fill the ghost node values using Split-C *gets.* In Put, we move the responsibility of filling a ghost node from the local processor that maintains the ghost node to the remote processor that maintains the actual value, using *put* operations to update them. The last optimization gathers all the values that need to be sent from one processor to another processor into a single buffer and uses a *bulk_put* transfer to fill the ghost node values.

By introducing ghost nodes and optimizing the local computation, we reduce the cost of processing an edge to 0.37μsec when all the edges are local. This corresponds to a floating point performance of 5.5 MFlops per processor since there are two floating point operations involved in processing an edge. The other versions do not affect this local node performance; they primarily focus on decreasing the communication costs. As expected, we decrease the communication costs by pipelining the reads in the Get version. The Put version performs better than the Get version because *puts* have less overhead than *gets.* Finally, the Bulk version has the best performance since it avoids repeated Annex set-up operations.

9 Analysis and Conclusions

The goal of this study was to derive an efficient implementation of a parallel language, Split-C, on a novel large-scale multiprocessor, the CRAY-T3D, using a systematic micro-benchmarking methodology. This approach is warranted because the T3D provides an elaborate shell around a sophisticated microprocessor, the DEC Alpha, to support global operations which are critical to a parallel language. The performance of the many shell operations was not previously documented and the interactions between them were potentially non-trivial.

In particular, the shell provides loads and stores of remote addresses, but requires manipulation of external segment registers to expand the physical address into one containing a full processor number. The external registers are managed by the compiler, and the usage strategy potentially interacts with the use of the TLB. The Alpha takes an extreme position on the weak ordering of memory operations, so writes are only known to have been committed after an explicit memory barrier instruction and reads can bypass independent writes. The shell provides additional operations to detect completion of remote operations. Various forms of cache-ability are supported, but caching of remote memory is not coherent. The Alpha provides prefetch "hints," which the T3D shell interprets as a binding load into an off-chip prefetch queue. The shell provides a powerful block transfer engine, which can move data to or from remote memory, but requires operating system intervention on start-up. It also supports explicit messaging through a user-level output FIFO, although the receive requires operating system intervention to handle the interrupt. The shell also provides an extremely fast global "fuzzy" barrier and special registers to support fetch&increment operations on remote nodes.

The large menu of primitives meant that there were potentially many different ways to realize the same language primitive on the machine. Careful assessment of the performance trade-offs was required to select between them. In making this selection, we have provided a thorough empirical characterization of the performance of the many primitives supported in the machine. The raw performance of the remote memory operations is very impressive and significantly faster in absolute terms than any previous large scale design. Remote access is performed in less than 1 μs, only three to four times the latency of a local cache miss, depending on whether the local DRAM access is off-page or not. However, the fast clock-rate of the Alpha means that this is still roughly 100 cycles. The prefetch support is very effective and capable of reducing the effective remote latency to 250 ns, where the remaining time is filled with useful work or used to issue additional prefetches.

In many cases we found that either the performance characteristics or subtle interactions between the language and hardware eliminated many of the potential options. Further, architectural oversights in both the shell and the microprocessor were limiting factors. For example, the potential hazards arising from physical synonyms in the write buffer prevent the compiler from using many external segment registers, except in special situations. The potential degradation of TLB efficiency leads to the same conclusion. Updating the configuration of the external registers is fast enough (roughly 23 cycles), that explicit checks can outweigh the cost of simply reloading it. In light of this, a single Annex entry for remote access could have sufficed. The lack of partial word stores make the use of shared data structure with data types that are less than a word, *e.g.*, character arrays, very cumbersome, even when there is no actual interaction on the individual elements. The prefetch mechanism is so fast that there is little performance advantage to providing the load, except that tracking the implicit hardware management of the prefetch queue can be costly. The start-up cost of the block-transfer engine is large, so prefetches and non-blocking writes are the best way to perform bulk transfer, except for very large transfers, in excess of 16 KB. Finally, the cost of message receipt is large, so that it is generally better to construct a remote message queue using the shared memory primitives and the fast synchronization support.

10 Acknowledgements

We would like to thank Mark Potts, Bill Carlson, and David Knaak, for all of their assistance and expertise. We also thank both Andrea Dusseau and David Bader for providing us with valuable comments.

References

[1] E. Barton, J. Cownie, and M. McLaren. Message passing on the Meiko CS-2. *Parallel Computing*, 20(4):497–507, April 1994.

[2] BBN Advanced Computers Inc. *TC2000 Technical Product Summary*, 1989.

[3] M. Berry, P. Chen, P. Koss, and David J. Kuck. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. Technical Report 827, November 1988.

[4] K.M. Chandy and C. Kesselman. Compositional C++: Compositional Parallel Programming. In *5th International Workshop on Languages and Compilers for Parallel Computing*, pages 124–44, New Haven, CT, August 1992.

[5] Cray Research Incorporated. *CRAY Hardware Reference Manual*, 1993.

[6] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel Programming in Split-C. In *Supercomputing '93*, pages 262–273, Portland, Oregon, November 1993.

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

[8] W. Groscup. The Intel Paragon XP/S supercomputer. In *Proceedings of the Fifth ECMWF Workshop on the Use of Parallel Processors in Meteorology.*, pages 262–273, Nov 1992.

[9] High Performance Fortran Forum. High Performance Fortran language specification version 1.0. Draft, January 1993.

[10] Kendall Square Research. *KSR1 Technical Summary*, 1992.

[11] R.E. Kessler and J.L. Schwarzmeier. Cray T3D: a New Dimension for Cray Research. In *Digest of Papers. COMPCON Spring '93*, pages 176–82, San Francisco, CA, February 1993.

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

[13] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *7th ACM International Conference on Supercomputing*, July 1993.

[14] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford Flash Multiprocessor. In *21st International Symposium on Computer Architecture*, pages 302–13, April 1994.

[15] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S. Yang, and R. Zak. The Network Architecture of the CM-5. In *Symposium on Parallel and Distributed Algorithms '92*, pages 272–285, June 1992.

[16] D. Lenoski, J. Laundon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. The Directory Based Cache Coherance Protocol for the DASH Multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148–159, 1990.

[17] Steve Luna. Implementing an Efficient Global Memory Portability Layer on Distributed Memory Multiprocessors. Master's thesis, University of California, Berkeley, May 1994.

[18] Niel K. Madsen. Divergence Preserving Discrete Surface Integral Methods for Maxwell's Curl Equations Using Non-Orthogonal Unstructured Grids. Technical Report 92.04, RIACS, February 1992.

[19] Richard Martin. HPAM: An Active Message Layer for a Network of HP Workstations. In *Hot Interconnects II*, August 1994.

[20] The Parkbench Committee. *PARKBENCH Committee Report-1*, 1993. Technical Report CS-93-213, Computer Science Department, University of Tennesse, Knoxville.

[21] R. H. Saavedra, R. S. Gaines, and M. J. Carlton. Micro Benchmark Analysis of the KSR1. In *Supercomputing '93*, November 1993.

[22] R. H. Saavedra-Barrera. *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking.* PhD thesis, U.C. Berkeley, Computer Science Division, February 1992.

[23] J. P. Singh, T. Joe, J. L. Hennessy, and A. Gupta. An Empirical Comparison of the Kendall Square Research KSR-1 and Stanford DASH Multiprocessors. In *Supercomputing '93*, pages 214–225, Portland, Oregon, November 1993.

[24] Richard L. Sites. *Alpha Architecture Reference Manual.* Digital Equipment Corporation, 1992.

[25] Jim Smith. Using Standard Microprocessors in MPPs. Presentation, ISCA, 1992.

[26] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, pages 256–66, 1992.