

Performance of the CRAY T3E Multiprocessor

Ed Anderson, Jeff Brooks, Charles Grassl and Steve Scott

Cray Research, Inc.

{eca, jpb, cmg, sls}@cray.com

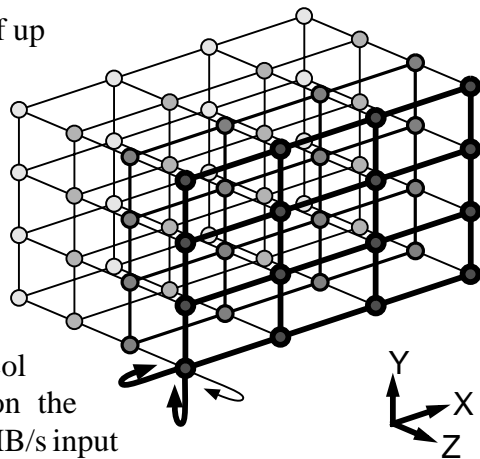
1 Introduction

The CRAY T3E is a scalable shared-memory multiprocessor based on the DEC Alpha 21164 microprocessor. The system includes a number of novel architectural features designed to tolerate latency, enhance scalability, and deliver high performance on scientific and engineering codes. Included among these are **stream buffers**, which detect and prefetch down small-stride reference streams, **E-registers**, which provide latency hiding and non-unit-stride access capabilities, barrier and fetch_and_op synchronization support, and a scalable, high-bandwidth interconnection network.

This paper reports our experiences with the CRAY T3E and presents a variety of performance measurements. Section 2 provides a brief overview of the system architecture. Section 3 describes the latency-hiding features (caches, stream buffers and E-registers) in more detail, assesses their performance impact, and discusses coding techniques for using them. Section 4 presents single-processor performance results. Finally, Section 5 discusses system scalability.

2 CRAY T3E Overview

The CRAY T3E provides a shared physical address space of up to 2048 processors over a 3D torus interconnect (illustrated at right). Each node of the system contains an Alpha 21164 processor, system control chip, local memory and network router (see Figure 1). The system logic runs at 75 MHz, and the processor runs at some multiple of this (300 MHz for the CRAY T3E, 450 MHz for the CRAY T3E-900). Torus links provide a raw bandwidth of 600 MB/s in each direction, with payload bandwidths ranging from 100 to 480 MB/s after protocol overheads, depending upon traffic type. I/O is based on the GigaRing channel [5], with sustainable bandwidths of 267 MB/s input and output for every four processors.



The Alpha 21164 microprocessor is capable of issuing four instructions per clock period, of which one may be a floating-point add and one may be a floating-point multiply, giving it a peak rate of 600 Mflop/s per processor at 300 MHz and 900 Mflop/s per processor at 450 MHz. There are two levels of caching on chip: 8 KB first level instruction and data caches, and a unified, 3-way associative, 96 KB second level cache. Only local memory may be cached. 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 caches when necessary to invalidate lines or retrieve dirty data.

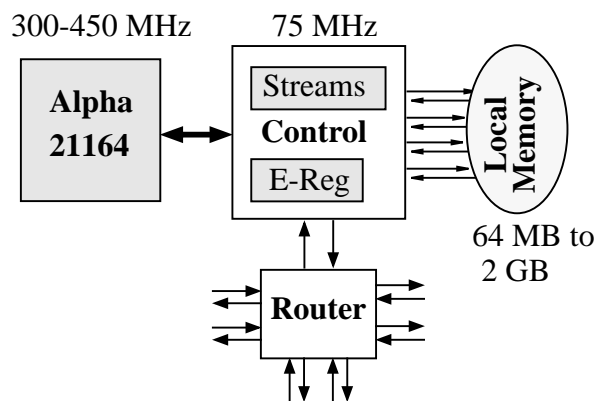


Figure 1: Block diagram of a CRAY T3E node

The local memory consists of a set of four memory controller chips, directly controlling eight physical banks of DRAM. Each 64-byte Scache line is spread across the eight banks, one 64-bit word per bank. The paths between the main control chip and the memory controllers provide a maximum transfer rate of 32 bits per channel per system clock, for a maximum rate of $4 \times 4B \times 75 \text{ MHz} = 1.2 \text{ GB/s}$. The 128-bit processor bus also has a peak rate of 1.2 GB/s, but can sustain a maximum of 80% of peak (960 MB/s).

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. There is no board-level cache on a CRAY T3E node, but the stream buffers can achieve much of the benefit of a large, board-level cache for scientific codes at a small fraction of the cost [4].

The CRAY T3E node augments the memory interface of the Alpha 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. They allow a very large number of outstanding requests for latency hiding, and provide efficient support for non-unit-stride memory accesses by gathering up single words that can then be loaded stride-one across the microprocessor system bus. The reader is referred to [6] and [7] for further details on the CRAY T3E design.

3 Latency Hiding Hardware

Over the past 30 years, the true memory latency of high-end computer systems has remained virtually unchanged, while processor performance has increased by several orders of magnitude. Computer architects have responded to this challenge by devising memory hierarchies of increasing complexity, and by providing features with lower latency for certain memory access patterns. Compiler writers have increased the depth to which code segments are evaluated in order to find more operations for latency hiding. Application developers have turned to new algorithms that better exploit data locality. While many applications have benefited from one or more of these innovations, the challenge of hiding the memory latency and achieving a significant fraction of the peak speed on scientific and engineering codes continues to grow.

A sampling of past and present high-performance computing systems tells the story:

Table 1: Historical memory latencies

System	Memory latency [ns]	Clock speed [ns]	Ratio	FP ops per clock period	FP ops to cover memory latency
CDC 7600	275	27.5	10	1	10
CRAY 1	150	12.5	12	2	24
CRAY X-MP	120	8.5	14	2	28
SGI Power Challenge	~760	13.3	57	4	228
CRAY T3E 900	~280	2.2	126	2	252

Each of these systems introduced new techniques to effectively reduce or hide memory latency. The CDC 6600 and 7600 decoupled loads and stores from a set of pipelined functional units. The CRAY 1 employed vector registers and a pipelined memory system. The CRAY X-MP added gather/scatter hardware to the basic CRAY-1 design to handle irregular memory access patterns. The SGI Power Challenge employed a large (4 MB) streaming board cache which could effectively pipeline memory operations at high bandwidth.

The CRAY T3E has a smaller cache but adds stream buffers for local data that are accessed with a small or unit stride and E-registers for data accesses that should bypass the cache. In this section, we will demonstrate the performance to be gained and show how to use each of these latency-hiding features.

3.1 Data Caches

The data caches are the first layer of latency hiding in a CRAY T3E node. Data elements that are already in the caches can be supplied to the microprocessor at the peak rate of two loads per clock period, or can be updated at the rate of one store per clock period. Load and store requests that miss in the caches need not stall the processor, and operations on data in the cache, including loads and stores, may continue while cache misses are outstanding. The usual compiler and programming techniques for improving the temporal locality of memory references work well, and the 3-way associativity of the secondary cache helps to mitigate pathological examples of cache-thrashing that could otherwise arise from the direct-mapped instruction and primary data caches. Special issues for the CRAY T3E programmer include the relatively small size of the caches and the random replacement policy of the secondary cache, which may reduce its effective size in some cases.

3.1.1 Description

Each CRAY T3E processor contains an 8 KB direct-mapped primary data cache (Dcache), an 8 KB instruction cache, and a 96 KB 3-way associative secondary cache (Scache) which is used for both data and instructions. The Scache has a random replacement policy and is write-allocate and write-back, meaning that a cacheable store request to an address that is not in the cache causes that address to be loaded into the Scache, then modified and tagged as dirty for write-back later. Write-back of dirty cache lines occurs only when the line is removed from the Scache, either by the Scache controller to make room for a new cache line, or by the back-map to maintain coherence of the caches with the local memory and/or E-registers.

3.1.2 Capability

A load request that hits in the Dcache is returned in two clock periods (CP). If it misses in the Dcache, it enters the Miss Address File (MAF) where it may merge with other load requests to the same 32-byte block before being sent on to the Scache. The latency from Scache is 8-10 CP. A load request that misses in the Scache sends a request to local memory for a 64-byte Scache line. Up to two Scache misses may be outstanding at one time.

The peak rates for Dcache and Scache access are shown in Table 2. These rates correspond to the maximum instruction issue rate of two loads per CP or one store per CP. The peak rate is attainable for floating-point loads, but is less for integer loads due to resource limits in the microprocessor.

Table 2: Peak data transfer rates on the CRAY T3E-900

Type of access	Latency [CP]	Bandwidth [MB/sec]
Dcache load	2	7200
Scache load	8-10	7200
Dcache or Scache Store		3600

3.1.3 Programming implications

The following example is a finite difference kernel which has a small enough working set to benefit from optimization for the cache:

```
REAL*8 AA(513,513), DD(513,513)
REAL*8 X (513,513), Y (513,513)
REAL*8 RX(513,513), RY(513,513)
DO J = 2,N-1
  DO I = 2,N-1
    XX = X(I+1,J)-X(I-1,J)
    YX = Y(I+1,J)-Y(I-1,J)
    XY = X(I,J+1)-X(I,J-1)
    YY = Y(I,J+1)-Y(I,J-1)
    A = 0.25 * (XY*XY+YY*YY)
    B = 0.25* (XX*XX+YX*YX)
    C = 0.125 * (XX*XY+YX*YY)
    AA(I,J) = -B
    DD(I,J) = B+B*A*REL
    PXX = X(I+1,J)-2.*X(I,J)+X(I-1,J)
    QXX = Y(I+1,J)-2.*Y(I,J)+Y(I-1,J)
    PYY = X(I,J+1)-2.*X(I,J)+X(I,J-1)
    QYY = Y(I,J+1)-2.*Y(I,J)+Y(I,J-1)
    PXY = X(I+1,J+1)-X(I+1,J-1)-X(I-1,J+1)+X(I-1,J-1)
    QXY = Y(I+1,J+1)-Y(I+1,J-1)-Y(I-1,J+1)+Y(I-1,J-1)
    RX(I,J) = A*PXX+B*PYY-C*PXY
    RY(I,J) = A*QXX+B*QYY-C*QXY
  END DO
END DO
```

The inner loop of this kernel has 47 floating point operations, 18 array reads and 4 array writes. The reads are of two 9-point stencils centered at $X(I,J)$ and $Y(I,J)$, and the writes consist of unit-stride stores to the independent arrays AA, DD, RX and RY. The two 9-point stencil array references should exhibit good temporal locality provided we can hold three contiguous columns of X and Y simultaneously in the Scache. In addition, we need to make sure that the writes to AA, DD, RX, and RY do not interfere with X and Y in the Scache.

Since all six arrays are the same size and are accessed at the same rate, we can ensure that they do not interfere with each other if they do not conflict initially. For this example, it is convenient to optimize for only one 4096-word set of the 3-way set associative Scache. One possible alignment of the arrays is shown in Figure 2.

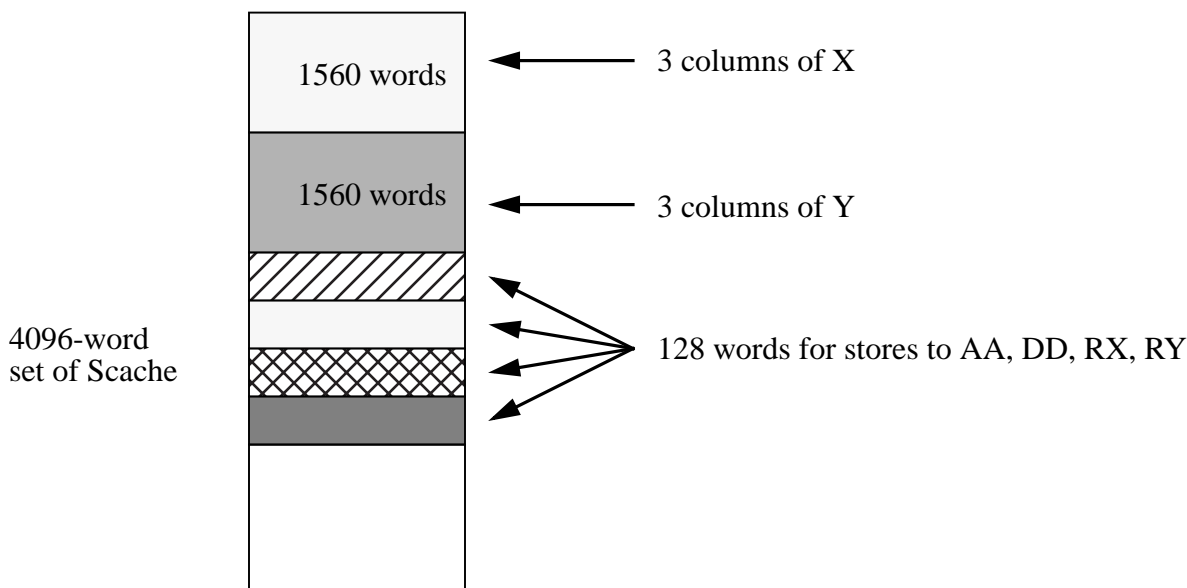


Figure 2: Optimal array alignment in Scache

In order to force this alignment, we placed the arrays in a cache-line aligned common block and inserted padding arrays between them to achieve the desired offsets modulo 4096. We also changed the leading dimension of the arrays from 513 to 520 in order to make it a multiple of 8 words, the Scache line size. The modified Fortran is as follows:

```

REAL*8 AA(520,513), DD(520,513)
REAL*8 X (520,513), Y (520,513)
REAL*8 RX(520,513), RY(520,513)
PARAMETER (IPAD=3576)
COMMON /ARRAYS/ X, P1(IPAD+520*3), Y, P2(IPAD+520*3),
& AA, P3(IPAD+128), DD, P4(IPAD+128), RX, P5(IPAD+128), RY
!DIR$ CACHE_ALIGN /ARRAYS/
DO J = 2,N-1
  DO I = 2,N-1
    {Loop body omitted ...}
  END DO
END DO

```

These changes improved the performance of the kernel by a factor of two, from 156 Mflops to 311 Mflops, when compiled with `f90 -O3,unroll2,pipeline2` on a CRAY T3E-900 system.

3.2 Stream Buffers

Scientific and technical codes often contain large data structures that are accessed as stride-1 or small-stride vectors. There may be little data reuse, but the access pattern is easy to detect and can be predicted far in advance. For this type of data access, a small set of stream buffers which exploit spatial locality alone can take the place of a large, board-level cache, which is designed to exploit both spatial and temporal locality [4].

3.2.1 Description

The control chip in a CRAY T3E node contains a set of six stream buffers, each of which contains two 64-byte buffers. The stream buffers monitor external memory requests from the processor and save the addresses of the last eight cache-line fill requests. When they observe a memory request for the successor to one of the previously-requested lines, they allocate a new stream, replacing the least recently used stream if necessary. This mechanism allows detection of address streams, even when multiple streams are interleaved in time.

For each active stream, the stream buffer logic prefetches the next two consecutive cache lines. Thus, the stream buffers can hold up to 12 cache lines, two for each of up to six streams. Logic on the memory controllers speculatively fetches and buffers data for each active stream in units of four cache lines (at interruptible priority), enhancing use of page-mode accesses in the DRAM. Without the stream buffers, the microprocessor allows a maximum of two outstanding cache line fills, and these will tend to interact poorly with DRAM pages if multiple access streams are interleaved in the application.

3.2.2 Capability

Besides their latency-hiding role in combining requests to a DRAM page, the stream buffers also deliver higher bandwidths to the local memory than cacheable loads without streams. We conducted a series of measurements of simple load/store sequences from local memory in order to assess these effects. These tests measured the performance of assembler-coded subroutines to load or store one or more very long unit-stride vectors to or from local memory, without performing any other operations on the data. The results, shown in Table 3, will be compared with the E-register bandwidths in Section 3.3.

Table 3: Measured local memory bandwidths on the CRAY T3E-900

Type of access	streams off [MB/sec]	streams on [MB/sec]
Cacheable load	299	833
Cacheable store	187	320
Cacheable load and store	208	471

3.2.3 Programming implications

On CRAY T3E systems configured to allow stream usage, the environment variable `SCACHE_D_STREAMS` may be set to 1 or 0 to turn streams on or off. Measuring the performance of an application with streams on and off is a simple test to see if it is taking advantage of streams.

To best exploit stream buffers, codes should maximize stream lengths and generate six or fewer concurrent reference streams. Code optimizations for streams include:

- Splitting loops to limit the number of streams
- Rearranging array dimensions to maximize inner loop trip count
- Rearranging array dimensions to minimize the number of streams
- Grouping statements that use the same streams

The following example from a finite-difference time-domain application illustrates the benefits of loop-splitting:

```

SUBROUTINE FDTD(CAEX,CBEX,CAEY,CBEY,CAEZ,CBEZ,EX,EY,EZ,HX,HY,HZ)
REAL CAEX, CBEX, CAEY, CBEY, CAEZ, CBEZ
REAL EX(129,129,128), EY(129,129,128), EZ(129,129,128)
REAL HX(129,129,128), HY(129,129,128), HZ(129,129,128)
DO J = 2, 127
  DO I = 2, 127
    DO K = 2, 127
      EX(K,I,J) = CAEX*(EX(K,I,J) +
&                  CBEX*(HZ(K,I,J) - HZ(K,I,J-1) +
&                  HY(K,I,J) - HY(K-1,I,J)))
      EY(K,I,J) = CAEY*(EY(K,I,J) +
&                  CBEY*(HX(K-1,I,J) - HX(K,I,J) +
&                  HZ(K,I-1,J) - HZ(K,I,J)))
      EZ(K,I,J) = CAEZ*(EZ(K,I,J) +
&                  CBEZ*(HX(K,I,J-1) - HX(K,I,J) +
&                  HY(K,I,J) - HY(K,I-1,J)))
    END DO
  END DO
END DO
RETURN
END

```

Although there are only six arrays, there are more than six streams because the difference equations access well-separated columns j and $j-1$ of the arrays HX and HZ. We have at least eight input streams: one stream for each of the arrays EX, EY, EZ, and HY, and two streams for each of HX and HZ. This is a good candidate for loop splitting because the trip counts are high, there are long sequences of unit-stride references, and the arrays are large enough that they can not already reside in the Scache. The inner loop may be split by a compiler option, by placing a SPLIT directive before the loop, or by hand. The effects of different compiling options and splitting strategies on a single processor of the CRAY T3E-900 are shown in Table 4.

Table 4: Effect of loop splitting on performance of FDTD

Compiling options	Speed in Mflop/s
f90 -O3,unroll2	39.2
f90 -O3,unroll2,split2	64.9
Manual split: {ex, ey}, {ez}, with f90 -O3,unroll2	74.0

3.3 E-registers

As mentioned in Section 2, all remote communication and synchronization in the CRAY T3E is done between E-registers and memory. E-registers provide two primary benefits over a more conventional load/store mechanism for accessing global memory: they extend the physical address space of the microprocessor to allow a very large, directly addressable memory, and they dramatically increase the degree of pipelining attainable for global memory requests. They also provide efficient non-unit-stride memory access and gather/scatter capabilities.

3.3.1 Description

E-registers employ a novel address translation mechanism. The microprocessor implements a cacheable memory space and a non-cacheable I/O space, distinguished by the most significant bit of the physical address. Local memory loads and stores use cacheable memory space. Address translation takes place on the processor in the usual fashion, and physical addresses are passed to the memory. Memory-mapped registers, including the E-registers, use noncacheable I/O space.

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 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. The global operations to read memory into E-registers or write E-registers to memory are called **Gets** and **Puts**, respectively. Gets and Puts are triggered via I/O space stores, in which the **address** supplies the command and target/source E-register, and the **data** bus supplies the global memory address. Virtual-to-physical address translation thus occurs outside the processor in the system logic. Complete details are provided elsewhere [7].

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 read-after-write hazard in the E-registers.

Since there are a large number of E-registers (512 user + 128 system), Gets and Puts may be highly pipelined. A large number of Gets can be issued, for example, before the first result is accessed. The processor 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. A variant of Gets and Puts moves a series of eight 64-bit words separated by an arbitrary stride between memory and E-registers. Thus, row accesses in Fortran, for example, can be fetched into contiguous E-registers using strided Gets. The resulting blocks of E-registers can then be loaded “broadside” into the processor, making significantly more efficient

use of the bus than would be possible with normal cache line fills, in which 7/8 of the loaded data would not be used. Single-word Gets and Puts are also provided, which can be used for gather/scatter operations.

When E-register operations are used to transfer data to/from local memory, the cache is bypassed. Non-cached data access may be preferred for copying blocks of data that are too large to fit in the cache and for any large-stride or irregular access patterns that would bring in unwanted data if loaded a cache line at a time. The use of E-register operations for local data access can be suggested to the compiler by means of the loop-level `CACHE_BYPASS` directive, or it can be done directly by calls to Shmem library routines, specifying the local processor number instead of a remote processor number in the argument list.

3.3.2 Capability

We demonstrate the advantages of selected E-register operations on local data by comparing their performance to that of cacheable loads and stores for the following types of data motion:

1. Copying a vector x to a vector y, both stride 1
2. Copying a vector x with stride 31 to a vector y with stride 1
3. Initializing a vector x to a constant value
4. Gathering a randomly-distributed set of 1000 elements from a 100000-element vector

The library routine `SHMEM_IGET` was used for the stride-1 copy operation, but the others were implemented using a `CACHE_BYPASS` directive on the basic Fortran loop. For example, the random gather operation is directed to use E-registers as follows:

```
!DIR$ CACHE_BYPASS A, X
DO I = 1, NGATH
  X(I) = A(INDX(I))
END DO
```

Table 5 shows the results for a single processor of a CRAY T3E-900 system.

Table 5: Comparison of cached and non-cached operations for local data motion

Type of data motion	Cached load/store performance [MB/s]	E-register performance [MB/s]
Stride-1 copy	471	646
Stride-31 to stride-1 copy	70	537
Initialize to constant	320	529
Random gather	59	339

The performance of both strided gets and gather operations through the E-registers can be affected by bank conflicts in the local memory system. The local memory is distributed across eight banks, and, when a cacheable load request misses in the Scache, each bank supplies one word of the 8-word Scache line during the fill. However, the pattern of access for strided gets and gather operations through the E-registers may not be evenly distributed among the banks. Strides that are multiples

of 2, 4, or 8 are concentrated in 4, 2, or 1 of the eight banks and are returned at a reduced rate. Figure 3 compares the bandwidth of the operation $y(1:n) = x(1:n:k)$ for different strides k using the `CACHE_BYPASS` directive for E-register access and plain Fortran for access through the cache.

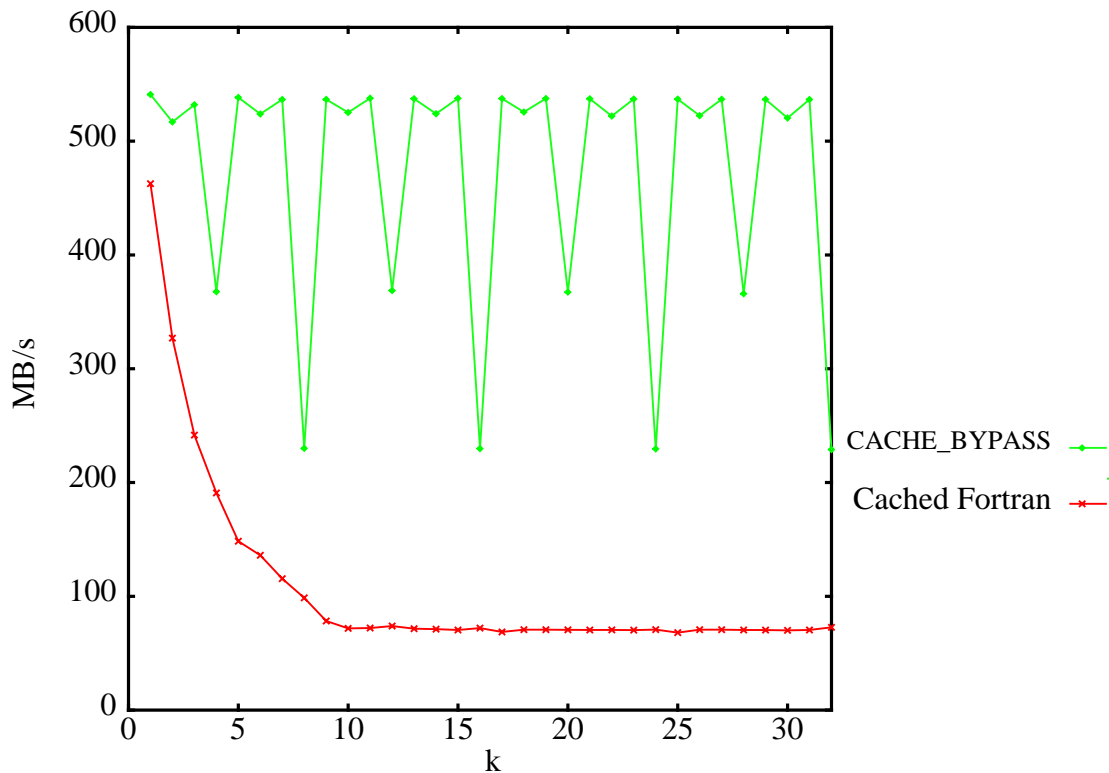


Figure 3: Performance of strided copy operation

3.3.3 Programming implications

By allowing highly-pipelined requests, supporting efficient non-unit-stride memory accesses, and allowing the on-chip caches to be bypassed, E-registers enhance performance when no locality is available. E-register optimizations are difficult for a compiler to perform automatically, however. This is because the compiler has little knowledge of the temporal locality between basic code blocks. If the data to be loaded were in the S-cache, then accessing that data via E-registers would be sub-optimal because the cache-backmap would first have to flush the data from S-cache to memory. Thus, E-register optimizations are reserved for programmer intervention through either compiler directives or library routines.

Two examples of E-register usage will be shown: a local transpose and a sparse matrix-vector multiplication. In the first example, only data motion is performed, as data is copied from memory to E-registers and back to memory without entering the processor. In the second example, data is gathered from memory into E-registers, loaded into the processor registers for use in a sparse dot product operation, put back to E-registers, and then stored to memory. Special library routines are available at each stage of this process for the true performance aficionado [1].

Experiments with the local transpose showed it was usually faster to copy rows of the source matrix to columns of the target array. The version with the loop-level directive is as follows:

```

DO J = 1, M
!DIR$ CACHE_BYPASS A, B
DO I = 1, N
B(I,J) = A(J,I)
END DO
END DO

```

This form of the transpose assumes that a row of the matrix A can be accessed with a favorable stride. If the leading dimensions of A and B were multiples of 4, it would be better to copy A to B by diagonals in order to use an odd stride. Table 6 shows the performance of copying a 1025-by-1025 matrix A to a 1025-by-1025 array B on a single processor of a CRAY T3E-900 system. Bypassing the cache causes a huge improvement, from 100 MB/s to 439 MB/s. Our best result, obtained with lower-level one-sided communication routines, is included for comparison.

Table 6: Performance of local transpose of 1025 x 1025 array

Method	Rate of transfer (MB/s)
Fortran	100
CACHE_BYPASS	439
Library E-register routines	544

Our sparse matrix-vector multiplication example is taken from the NAS Parallel Benchmark (NPB) program CG [2]. The Class A problem size was run on two processors, producing a sparse matrix with 7000 rows. The dominant portion of the program is an indexed dot product:

```

DO J=MINROW,MAXROW
Y(J) = 0.
!DIR$ CACHE_BYPASS X, Y
DO K = COLSTR(J), COLSTR(J+1) - 1
Y(J) = Y(J) + A(K)*X(ROWIDX(K))
END DO
END DO

```

This portion of the program was instrumented, and the performance was measured for three different versions, as shown in Table 7. Most of the improvement was realized without extensive code changes using the CACHE_BYPASS directive. The difference between cached Fortran and CACHE_BYPASS is not as dramatic as for the random gather example of Section 3.3.2, because the 7000-element vector A is small enough to get some reuse out of the 12288-word Scache. The last line shows our best result with library routines.

Table 7: Performance of sparse matrix-vector kernel from CG

Method	Performance per PE [Mflop/s]
Fortran	17
CACHE_BYPASS	32
Library E-register routines	39

4 Overall Single-Processor Performance

In this section, we examine the NAS Kernel benchmark test. When originally compiled with zero changes and run on the CRAY T3E (300 MHz), we get the results shown in the first column of Table 8 on a single processor.

Table 8: NAS kernels with zero changes (Mflop/s, CRAY T3E 300MHz)

Kernel	f90	-O3	-O3 -apad	-O3, unroll2 -apad	-O3, unroll2, pipeline2 -apad	-O3, unroll2, pipeline2 -apad -lmfastv
MXM	84.8	84.8	92.3	137.3	172.6	174.2
CFFT2D	21.2	21.2	21.2	21.3	21.9	23.1
CHOLSKY	19.7	19.5	19.5	20.4	26.4	26.4
BTRIX	49.3	49.3	47.8	47.9	47.8	48.0
GMTRY	67.9	72.3	73.1	73.1	73.1	73.0
EMIT	61.5	139.3	139.9	189.5	219.4	246.4
VPENTA	4.2	4.3	26.5	26.5	26.4	26.4

Performance is improved by over a factor of 2X overall through compiler options alone. Kernels GMTRY and EMIT improve by using the -O3 flag. This is due primarily to the intrinsic functions in these routines which are vectorized with the -O3 flag. Kernels MXM, and VPENTA speed up with the common block padding feature which optimizes array locations for the Scache. Turning on unrolling helps MXM and EMIT substantially and software pipelining improves MXM and EMIT yet again. Finally, linking with the fast math intrinsic substantially improves the EMIT kernel, confirming the importance of intrinsic functions to that kernel. We use -O3,unroll2,pipeline2 -apad as our optimal set of compiler options and link to the faster math intrinsic library, -lmfastv.

Using the T3E Apprentice and PAT performance profiling tools, each of the codes was analyzed, and optimized through a variety of techniques, including additional cache alignment and array padding, cache bypassing via E-registers, loop interchanging, loop index permuting, loop splitting, and substitution of code with calls to math libraries. The resulting performance is shown in Table 9.

Table 9: Modified NAS kernels (CRAY T3E 300MHz)

Kernel	Original [Mflop/s]	Optimized [Mflop/s]
MXM	174.2	409.1
CFFT2D	23.1	156.4
CHOLSKY	26.4	48.2
BTRIX	48.0	81.4
GMTRY	73.0	172.3
EMIT	246.4	246.4
VPENTA	26.4	88.8

5 Multi-PE Performance

The primary features enhancing scalability in the CRAY T3E are the E-registers and the interconnection network. E-registers allow up to several hundred memory references to be outstanding concurrently, providing excellent latency-hiding capabilities.

5.1 Hardware Features

Figure 4 shows the effect of pipelining on global memory bandwidth. This test loads an array of 16K entries (128 KB) from a PE 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.

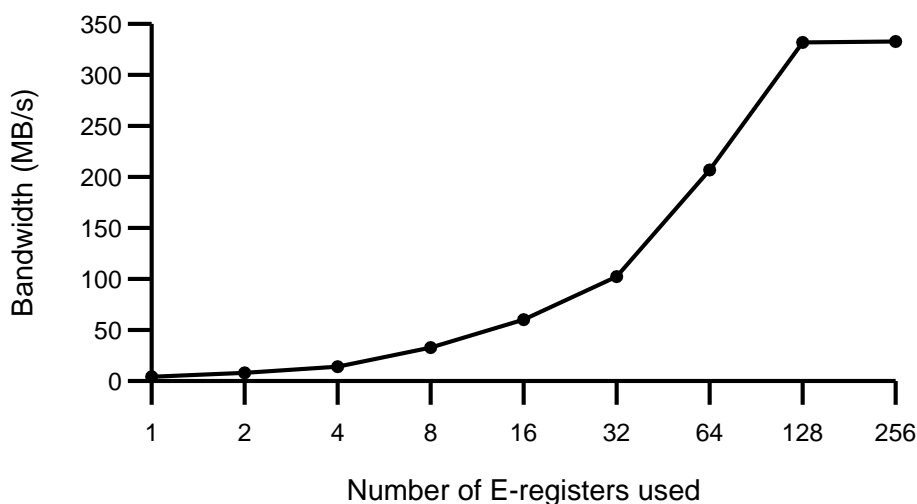


Figure 4: Effect of pipelining in the memory

Realized Get bandwidth increases with the number of E-registers used. Using 8 E-registers, the realized bandwidth is less than 50 MB/s. Bandwidth increases to over 300 MB/s (actually limited by network-related system logic at the remote PE) using 128 E-registers, at which point all of the network latency is hidden, and additional E-registers are not needed.

The CRAY T3E network latency is nominally 1 microsecond. From various message passing libraries, however, the effective latency can be much larger due to overhead associated with buffering and with deadlock detection. The library message passing mechanism uses the E-registers to implement transfers directly from memory to memory. These transfers do not use or go through the data cache of the participating processors. Table 10 shows the network latency for various message passing libraries.

Table 10: Network latency for message passing libraries

Library	Network latency [microseconds]	Bandwidth [Mbyte/s]
sma (Shmem)	1	350
PVM	11	150
MPI	14	260

The sma (shared memory access) library is a simple one-sided communication library available on SGI/Cray products. This library has minimal latency and maximal asymptotic bandwidth. The PVM and MPI message passing libraries have similar asymptotic bandwidths, but the latencies are more than an order of magnitude greater than those for the sma library. In the next section we will see how this latency affects application performance.

5.2 Benchmark Results

Various latency hiding features enable both faster single processor speeds and utilizing a large number of processors. With the low latency network, programs containing significant communication are able to run efficiently with more than 1000 processors. The techniques used to optimize the single PE performance of these benchmark codes include exploiting the cache bandwidth and stream buffers, using vectorized intrinsic functions, and selectively employing E-registers for local data manipulation. The techniques used for multi-PE performance include using E-registers for inter-PE communication and using E-registers for local transposes.

In this section, we present the results of the NAS Parallel Benchmarks (NPB) on the Class C problem size on from 64 to 1024 processors of a CRAY T3E-900 system. The eight NPB programs have sufficient degrees of freedom to exploit thousands of processors, but have varying amounts of communication. The amount of communication varies from essentially none in program EP, to a substantial and significant amount in program FT to a dominating amount for large PE counts in program CG.

The performance per PE for the NPB programs is shown in Table 11. Because operation counts are not available for class C version 1 programs, version 2 operation counts were used. The actual operation counts for this implementation ranges from 50% to 115% of the “benchmark flops”.

Table 11: PB, version 1, Class C, performance per PE. Results are in Mflop/s.

PEs	EP	MG	CG	FT	IS	LU	SP	BT
64	340	110	30	130	20	250	100	140
128	340	110	25	125	15	250	90	130
256	340	100	15	130	15	220	80	120
512	340	90	15	110	15	200	70	110
1024	340	85	15	70	10	160	60	100

Several programs have unique features which affect their computation rate. Program EP has no communication and hence it demonstrates constant performance per PE with increasing numbers of PEs. Each PE starts with several random number seeds and produces and histograms its generated random numbers. The random number generation uses the intrinsic functions ALOG and SQRT. The compiler generates “vector” intrinsics which allow pipelining of data through the function algorithm. This computation is done almost entirely in registers and in the data cache and hence memory latency is hidden or eliminated. Program EP therefore demonstrates latency-hiding at the software or algorithm level.

Program MG has irregular “all-to-all” communication. The data for each communication is buffered and sent to the respective target. This buffering eliminates short messages and allows better scaling. A lower latency network would allow the same performance with less or no buffering. The computation portion of this program involves three dimensional stencils. The proper coding of these stencils allows the data in adjacent rows, columns and planes to be reused in the data cache.

Programs CG and SP will be addressed in more detail in Section 5.3. Program FT performs a 3D FFT. The basic kernel of this program is a library call to a single (stride 1) FFT. In order to manipulate stride one data, the PEs perform inter-PE and in-place transposes. The in-place transposes have strides which are the dimensions of the problem, 512. This is a low performance stride for most cache and memory designs, so E-registers are used to manipulate this data. These functions allow the data movement. In the final implementation, the strided in-place transposes perform at over 400 Mbyte/s.

The programs LU, SP and BT rely on local transposes to line up data for stride one access. Programs SP and BT also use inter-PE transposes to move and reorient data among columns and rows of processors. The dependency on transpose performance is very common in 3D data structures. Further reduction in latencies and penalties for non-unit strides and further enhancing latency hiding would greatly simplify programming of these kinds of problems.

Figure 5 shows the performance of the NAS parallel benchmarks relative to the number of processors. The effective performance per PE is still reasonable for 1024 PE. For most of the programs, the processors are running at from 100 to 200 Mflop/s. Programs LU and BT have a system performance of over 100 Gflop/s when run with 1024 PEs. This is significant in light of the fact that both of these programs are doing a large amount of data movement between processors.

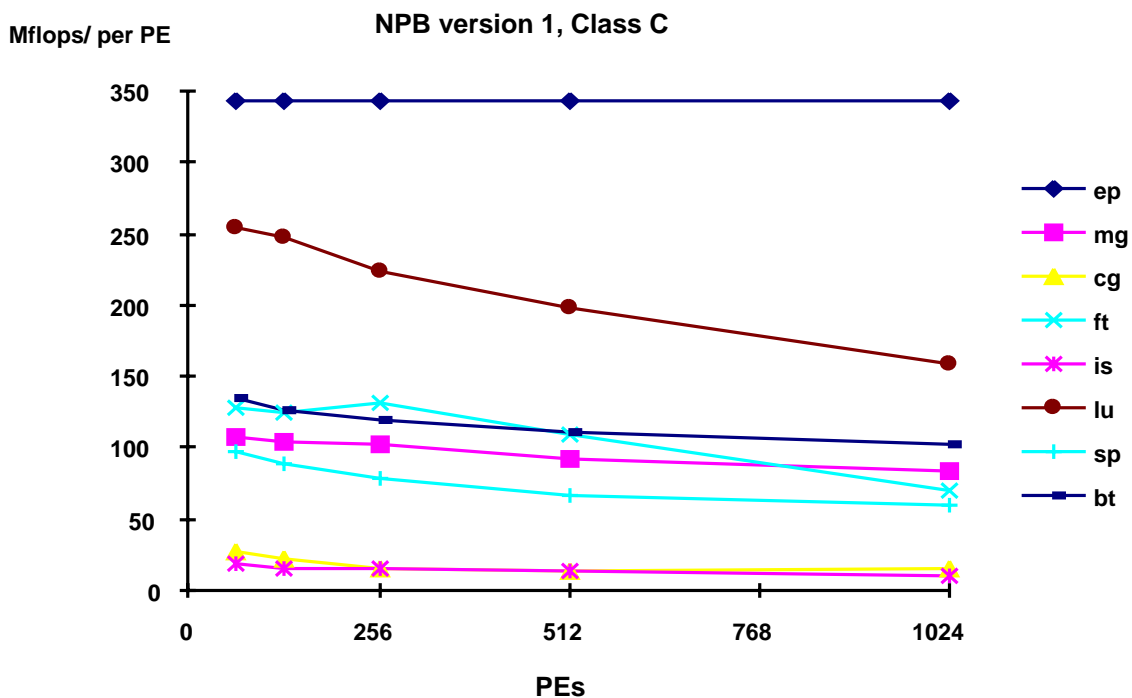


Figure 5: NPB performance relative to number of PEs.

5.3 The Effect of a Low Latency Network

Many applications require a low latency communication network in order to maintain high system efficiency. We have investigated the effect of network latency on NPB programs CG and SP. For this investigation we used the class A size programs in order to amplify the effect.

Program CG uses a block decomposition. With this implementation, most of the messages range in size from 800 to 3500 words for the 16 PE case and from 50 to 800 words for the 256 PE case. This implementation was chosen because it has a minimal amount of communication, though not maximum message lengths. For all PE count cases, there are also collective sums related to dot products. These sums involve only single word messages and are very sensitive to latencies.

Program SP uses a three dimensional block decomposition. Inter-PE transposes are at the heart of the communication scheme. These transposes have a constant amount of communication for more than 1 PE, but the message sizes decrease with increasing number of PEs. The message size is 80 words for 16 PEs and 20 words for 256 PEs. For all cases, the transfers are very sensitive to network latency.

We can show how much effect the network latency has on the performance of programs CG and SP by increasing the latency. We programmed the communication routines such that the effective latency ranged from 1 microsecond to 25 microseconds and measured the time spent in these portions of the programs. The results are shown in Table 12 and Table 13.

Table 12: Program CG performance relative to message latency. Results are percent of time spent in communication routines

PEs	Message Lengths [Bytes]	Latency in microseconds				
		1	2	5	10	25
16	6000 - 30000	5	5	5	7	7
32	3200 - 30000	12	12	12	10	12
64	1600 - 30000	17	18	19	20	24
128	8000 - 3000	30	31	32	34	40
256	400 - 3000	46	48	50	53	60

Table 13: Program SP performance relative to message latency. Results are percent of time spent in communication routines

PEs	Message Lengths [Bytes]	Latency in microseconds				
		1	2	5	10	25
16	640, 640	29	30	34	40	53
32	640, 320	34	36	41	48	62
64	320, 320	41	43	48	56	69
128	320, 160	52	54	60	66	78
256	160, 160	61	64	69	75	84

We see from Table 12 that for program CG, the low latency keeps the program relatively computation-bound. A higher latency, such as the 10 microsecond latency for PVM or MPI, would make this program computation bound at a much lower PE count. For program CG, the low latency network allows the program to use a more efficient algorithm which minimizes the amount of communication.

Program SP is already nearly communication bound with 1 microsecond latency. We see from Table 13 that a further reduced network latency would speed up the performance of this program with even a modest number of PEs.

For program SP, a further reduced latency could also simplify the transpose routines. The current implementation of the transpose routines use buffers of 5 row segments for communication. If the network latency were sufficiently reduced, then this buffering step could be eliminated and the algorithm would be significantly simplified.

We can conclude, from the example of program SP, that further reduced latencies could simplify programming. Reduced network latencies can eliminate the need for buffering of messages and hence reduce coding. Another supporting example is the use of E-registers for local memory strided data. The use of these registers can reduce the need for local memory transposes and hence again simplify coding.

References

- [1] Anderson, E., J. Brooks, and T. Hewitt, "The Benchmarkers' Guide to Single-processor Optimization for CRAY T3E Systems," Cray Research, June 1997. Available at the URL <http://www.cray.com/products/systems/crayt3e/benchmark.ps>
- [2] Bailey, D. H., J. T. Barton, T. A. Lasinski, and H. D. Simon, eds: "The NAS Parallel Benchmarks," NASA Technical Memorandum 103863, NASA Ames Research Center, Moffett Field, CA, 94035-1000, July 1993.
- [3] Berry, M., C. Grassl, and V. Krishna, "Blocked Data Distribution for the Conjugate Gradient Algorithm on the CRAY T3D," Cray Research, 1994.
- [4] 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.
- [5] Scott, S., "The GigaRing Channel," **IEEE Micro**, pp 27-34, February 1996.
- [6] Scott, S. and G. Thorson, "The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus," **HOT Interconnects IV**, Stanford University, August 1996.
- [7] Scott, S. L. "Synchronization and communication in the T3E Multiprocessor," **Proc. Seventh International Conference on Architectural Support for Programming Languages and Operating Systems**, pp. 26-36, October 1996.