The Benchmarker's Guide to Single-processor Optimization for CRAY T3E Systems

Ed Anderson, Jeff Brooks, and Tom Hewitt
Benchmarking Group, Cray Research

How to use features of the CRAY T3E series processors from a high-level language to improve the performance of application codes

1.0 Introduction

The CRAY T3E series is the second generation of scalable parallel processing systems from Cray Research to be built around a commodity microprocessor. Its fast processors, latency-hiding memory interface, and high-performance interconnect deliver high single-processor performance and industry-leading latencies and bandwidths between processors. The interprocessor network's 3-D torus design helps to minimize the number of hops between nodes and provides excellent scalability up to the maximum configuration of 2048 nodes. With peak performance of 600 Mflops per processor on CRAY T3E systems and 900 Mflops per processor on CRAY T3E-900 systems, the CRAY T3E series is the world's first and only commercially available teraflops system.

Because the strength of the network takes care of scalability, the key to system performance is often the performance of the local node. This paper describes the components of a CRAY T3E node and shows how to utilize them most effectively from a high-level language. Beginning with the microprocessor, we consider functional unit optimizations in Section 3.0 and data alignment issues that affect the performance of the on-chip caches in Section 4.0 At the level of the support circuitry, optimizations for the stream buffers to improve the rate of access to cacheable data are described in Section 5.0, and optimizations using E-register operations for non-cached data are detailed in Section 6.0. Compiler options that implement many of these optimizations are summarized in Section 7.0. We conclude with a series of case studies on the NAS kernels.

2.0 Hardware overview

The CRAY T3E series processors are based on the DEC Alpha 21164 microprocessor, with clock speeds of 300 MHz (3.33 ns) in the CRAY T3E system and 450 MHz (2.22

ns) in the CRAY T3E-900 system. Each node of a CRAY T3E system contains one processor, support circuitry, local memory, and a network router. The network routers in the system are connected in a 3-D torus network. There are four nodes and one I/O controller on a printed circuit board, and the I/O controller is connected to the interprocessor network by connections to each of the four local routers and to external devices by a HiPPI channel.

2.1 The microprocessor

The CRAY T3E microprocessor consists of the following functional components:

Ibox: instruction fetch/decode unit and branch unit

Fbox: floating-point execution unit

Ebox: integer execution unit

Mbox: memory address translation unit

Cbox: cache control and bus interface unit

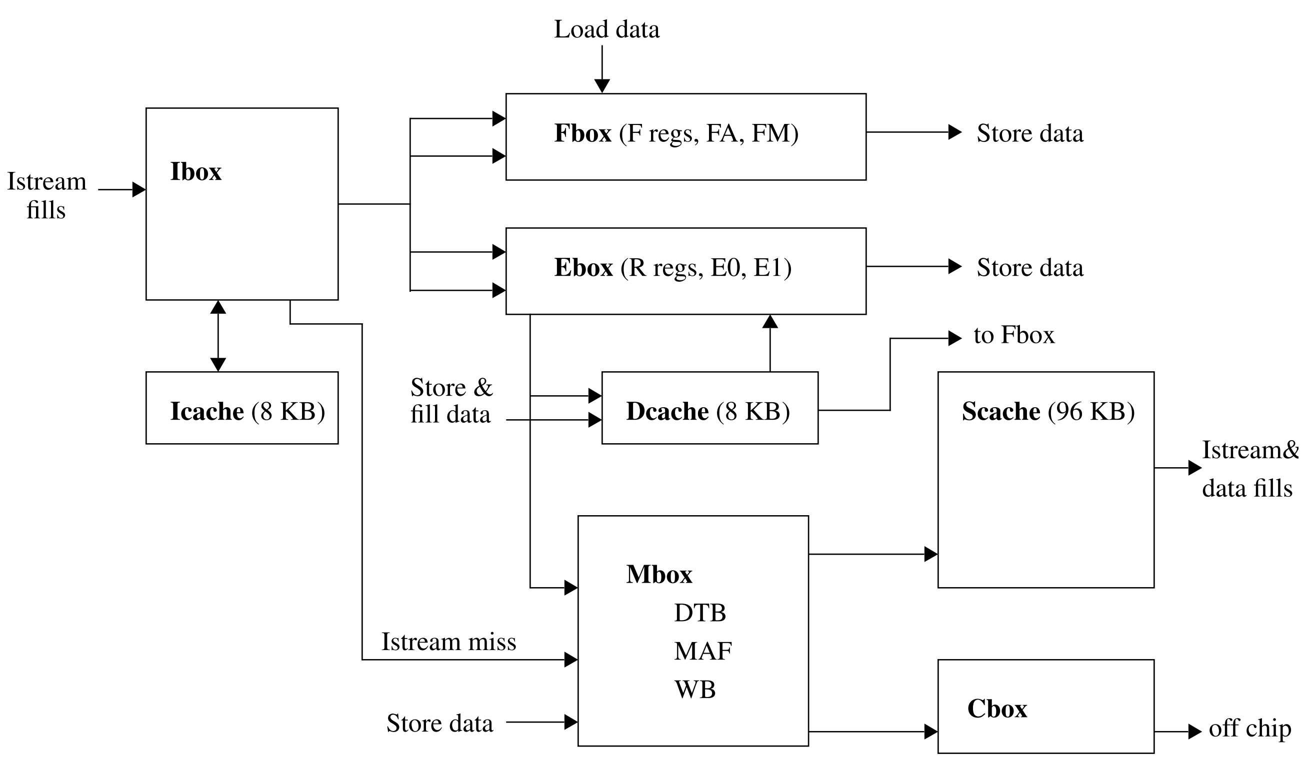
Icache: instruction cache

Dcache: data cache

Scache: second-level cache for data and instructions

Figure 1 is a block diagram showing the flow of data between these components.

FIGURE 1 Block flow diagram of the CRAY T3E (Alpha 21164) microprocessor



The microprocessor can issue up to four instructions per clock period and maintains four concurrent instruction pipelines:

FA: floating-point add pipeline

FM: floating-point multiply pipeline

E0: first integer pipeline, also executes loads or stores

E1: second integer pipeline, also executes loads

Peak performance of the CRAY T3E processor is 1200 MIPS or 600 Mflops, while the peak rate of the CRAY T3E-900 processor is 1800 MIPS or 900 Mflops.

The Mbox controls the Dcache and provides buffering capabilities for load and store operations. It consists of the following components:

DTB: data translation buffer

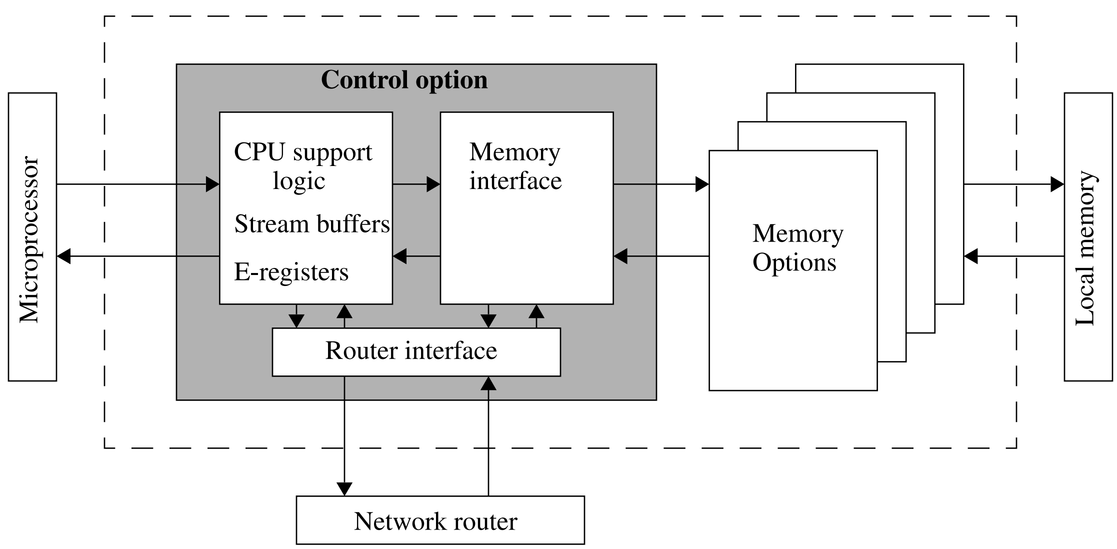
MAF: miss address file

WB: write buffer

Load requests that miss in the Dcache enter one of six 32-byte MAF entries and store requests enter one of six 32-byte WB entries before being sent on to the Scache. Up to four load requests may be merged in a single MAF entry (under certain conditions) and up to four store requests may be merged in a single WB entry. Buffering is needed for latency-hiding because the microprocessor can issue loads and stores at a greater rate than the Scache can supply or accept the data if Scache misses occur.

The Cbox controls the Scache and provides an interface to the support circuitry, which in turn interfaces to the local memory and to other processors *via* the network router (see Figure 2). A cacheable load request that misses in the Scache is sent by the Cbox

FIGURE 2 Support circuitry on a CRAY T3E node



to local memory to retrieve a 64-byte Scache line. Each processor has 64-1024 MB of DRAM organized in eight banks, and Scache lines are spread across the banks, one 64-bit word per bank. Up to two Scache fills from local memory may be in progress at one time, and in the absence of dependencies on the fill data, other computation, including loads and stores to the on-chip caches, may continue while Scache misses are outstanding.

On-chip memory resources include 32 integer and 32 floating point registers, an 8 KB direct-mapped instruction cache (Icache), an 8 KB direct-mapped data cache (Dcache), and a 96 KB, 3-way set associative, write-back, write-allocate secondary cache (Scache), which is used for both data and instructions. Cache lines are 32 bytes long in the Icache and Dcache and 64 bytes long in the Scache. The latency of a Dcache hit is 2 clock periods (CP) while the latency of an Scache hit is 8-10 CP. Access to either the Dcache or the Scache can be sustained at a rate of two loads per clock period or one store per clock period, giving peak on-chip bandwidths of 4.8 GB/sec for loads and 2.4 GB/sec for stores on a CRAY T3E processor, and 7.2 GB/sec for loads and 3.6 GB/sec for stores on a CRAY T3E-900 processor. Table 1 summarizes these peak on-chip rates, which are attainable for floating-point data.

Table 1: Peak on-chip bandwidths

Type of access	CRAY T3E	CRAY T3E-900
Dcache load (2 CP latency)	4800 MB/sec	7200 MB/sec
Scache load (8-10 CP latency)	4800	7200
Dcache or Scache store	2400	3600

2.2 The support circuitry

Memory references that miss in the Scache must go off-chip, where the latencies are much longer. The support circuitry in a CRAY T3E node contains two main mechanisms for hiding the latency and improving the bandwidth of off-chip memory references: the stream buffers and the E-registers. The stream buffer logic implements multi-way stream detection for unit-stride or small-stride accesses to local memory that use cacheable loads and stores. The E-registers were primarily designed for remote memory access, but they can also be used for non-cached access to local memory.

Each CRAY T3E node contains six stream buffers, each containing storage for two consecutive 64-byte Scache lines. A stream is initiated when Scache misses are generated for two consecutive 64-byte blocks of memory. The stream detection hardware maintains a history of the last eight Scache misses, so the two misses do not need to be contiguous in time, and misses to different streams may be interleaved. In addition to the buffering capability of the stream buffers themselves, the memory controllers speculatively fetch and buffer data for the active streams in units of four Scache lines.

Each node also contains a set of memory-mapped external registers known as the E-registers. There are 512 user E-registers and another 128 that are reserved for system use. The E-registers are manipulated via memory-mapped operations. Stores to special memory addresses cause data to be transferred between the E-registers and memory. A large number of these transfers may be in flight simultaneously, so that for large transfer sizes, performance is not limited by memory latencies. Data can also be transferred between the E-registers and the microprocessor registers using ordinary loads and stores to the physical address of the E-registers in non-cached memory space.

An external backmap maintains coherence between the caches and local memory during E-register GETs and PUTs, including all remote data accesses. The backmap contains a set of tags that mirrors the contents of the Scache, and all E-register references consult the backmap before referencing local memory. If the data could be in the Scache, the corresponding Scache line is flushed to memory before the E-register operation is allowed to complete. This means there is no need for the user to flush local data from the cache because of a remote GET or PUT.

2.3 Peak and measured bandwidths

The links within a CRAY T3E node have peak transfer rates of 1200 MB/sec. This is based on a peak transfer rate of 128 bits per system clock along the processor bus (or 32bits per system clock along each of four channels between the memory controller chips and the main control chip) and a system clock speed of 75 MHz. The theoretical peak sustainable rate is 80% of this, or 960 MB/sec.

We have not found the peak transfer rates to be a good predictor of single-processor performance, so in Table 2 we report our actual measured bandwidths for some simple local memory transfers. To some extent, measured performance depends on the cleverness of the measurer, so it may be possible to exceed these rates. The cacheable load bandwidth increases significantly from the CRAY T3E to the CRAY T3E-900 due to the addition of a hardware read-ahead feature. We also have observed a degradation in the performance of `shmem_get` or `shmem_put` when used to copy blocks of local memory, apparently as a side-effect of the hardware fixes in the later pass of the chip.

Table 2: Measured unit-stride local memory bandwidths

Type of access	CRAY T3E	CRAY T3E-900
Cacheable load (streams on)	681 MB/sec	875 MB/sec
Cacheable load (streams off)	291	296
E-register GET via benchlib	600	600
Cacheable store (streams on)	294	320
Cacheable store (streams off)	180	187
E-register PUT via benchlib	532	532
Cacheable load/store (streams on)	383	471
Cacheable load/store (streams off)	201	208
E-register GET/PUT via benchlib	591	575
E-register GET/PUT via shmem	765	589

3.0 Functional unit optimization

Although the peak execution rate of the CRAY T3E processor is four instructions per clock period, this rate usually can not be sustained for an entire application because some time must be spent in reading, writing, communicating, or loading data from local memory. However, for certain computationally-intensive segments of an application, the data may reside in cache, and the speed at which we can feed the functional units may be the limiting factor.

The CRAY T3E processor has segmented functional units for floating point multiplication and addition. A floating-point add and an independent floating-point multiply can be issued every clock period, but the result of each operation is not ready for 4 clock periods. This means that we must expose enough functional unit parallelism to the compiler to allow for efficient scheduling of the floating-point operations. In general, this is accomplished by explicit unrolling, directives to the compiler to enable unrolling, or use of hand-optimized library routines.

Another issue in functional unit optimization concerns those operations that do not pipeline. For example, the 64-bit version of the floating point divide requires 22-60 CP before the next divide can be issued. Elementary math functions such as SIN and COS, as well as SQRT and the power function, are implemented as library calls and thus are hidden from the compiler. When these operations occur in a loop, vectorizable versions from libm or libmfastv may supply the necessary functional unit parallelism.

3.1 Loop unrolling

To see the effect of functional unit transit time, we test some simple loops on the CRAY T3E. The following loop does a single floating-point multiply on a scalar variable. The data for this loop can be completely held in registers by the compiler:

```
DO I = 1, 1024
  T = T * T
ENDDO
```

We would expect a floating point multiply result approximately every 4 clock periods. No functional unit pipelining is possible here because the result is used in the next pass of the loop. One result per 4 clock periods equates to 75 Mflops on the CRAY T3E system running at 300 Mhz. The measured result for this loop is 71.5 Mflops.

The following loop could be expected to do much better:

```
DO I = 1, 1024
   T1 = T1 * T1
   T2 = T2 * T2
   T3 = T3 * T3
   T4 = T4 * T4
ENDDO
```

Indeed, measured performance for this loop (compiled with `f90 -O3,unroll2`) is 285 Mflops. The four independent multiplies can fire on successive clock periods, keeping the multiplier busy at all times and achieving near peak performance.

Unrolling inner loops often exposes more functional unit parallelism to the compiler and can dramatically improve performance. In the next example, arrays A, B and C are each of size 256 and are stored consecutively so that all three arrays can reside in the Dcache at the same time. The loop is repeated many times to get a cache-resident performance figure:

```
DO I = 1, 256
   A(I) = B(I) + 2.5 * C(I)
ENDDO
```

When compiled with `f90 -O3`, without any unrolling, the performance rate is about 53 Mflops. When compiled with `f90 -O3,unroll2`, which unrolls by 4, the performance improves to 155 Mflops. We could help the compiler manage the functional unit dependencies by doing our own unrolling; for example, the following version attains 177 Mflops:

```
TEMP1 = 2.5*C(1)
TEMP2 = 2.5*C(2)
TEMP3 = 2.5*C(3)
TEMP4 = 2.5*C(4)
DO I = 1, 252, 4
   A(I) = B(I) + TEMP1
   TEMP1 = 2.5*C(I+4)
   A(I+1) = B(I+1) + TEMP2
   TEMP2 = 2.5*C(I+5)
   A(I+2) = B(I+2) + TEMP3
   TEMP3 = 2.5*C(I+6)
   A(I+3) = B(I+3) + TEMP4
   TEMP4 = 2.5*C(I+7)
END DO
A(I) = B(I) + TEMP1
A(I+1) = B(I+1) + TEMP2
A(I+2) = B(I+2) + TEMP3
A(I+3) = B(I+3) + TEMP4
```

However, the easiest way to increase the functional unit parallelism is simply to unroll more deeply and let the compiler do the work. By adding the following directive before the original loop:

```
!DIR$ UNROLL 8
```

and recompiling with f90 -O3,unroll1, we see our best performance of 181 Mflops.

3.2 Replacing divides with reciprocals

The floating-point divide operation may be a bottleneck in some codes because it can not be pipelined with other divide operations and it has a much longer latency than the floating-point add and multiply. The latency of a 64-bit floating-point divide is 22-60 clock periods, depending on the data. We will assume an average case latency of 32 CP, based on the following experiment: arrays A and B were chosen to fit in the Dcache, the array B was assigned random values between 0 and 1, and the following loop was repeated 1000 times to get an in-cache performance rate:

```
!DIR$ UNROLL 8
      DO I = 1, 512
         A(I) = A(I) / B(I)
      END DO
```

On the 300 MHz CRAY T3E, the performance was 9.3 Mflops, or about 32 CP per divide. If B(I) were replaced by a scalar, the rate would be 115 Mflops because the f90 compiler would move the loop-invariant scalar division out of the loop and replace it with multiplication by a reciprocal, unless directed not to by the `-Oieeeconform` option.

If the divides are reused and the algorithm allows replacing them by a reciprocal, saving the reciprocals in a temporary vector may be beneficial. Consider the case of a square matrix to which we apply a row scaling. The Fortran code is as follows:

```
      DO J = 1, 512
!DIR$ UNROLL 8
         DO I = 1, 512
            A(I,J) = A(I,J) / B(I)
         END DO
      END DO
```

Even with the directive for unrolling by 8, this code manages only 8.5 Mflops. The divide is such a drag on performance that we can actually do better (10.7 Mflops) by interchanging the i and j loops, even though this gives a poor access pattern for the array A. A better solution is to store the reciprocals of B in a temporary vector. The best case is to declare enough space to invert all of B, but we can attain nearly the best case performance by strip-mining by 256. The following code gets 28 Mflops, more than a factor of three improvement over the original:

```
      REAL TMP(256)
      ...
      DO II = 1, N, 256
         NI = MIN(256,N-II+1)
         DO I = 1, NI
            TMP(I) = 1.0 / B(II+I-1)
         END DO
         DO J = 1, N
!DIR$ UNROLL 8
```

```
      DO I = 1, NI
        A(II+I-1,J) = A(II+I-1,J)*TMP(I)
      END DO
    END DO
  END DO
```

3.3 Intrinsic Functions

Intrinsic function evaluations also present a problem of functional unit optimization because single evaluations typically rely on iterative algorithms with little opportunity for pipelining. When the intrinsic function evaluations occur in a loop, an effective technique for optimizing them is "vectorization." This involves separating the intrinsic function calls into a separate loop and calling a "vector" version of the library routine that evaluates several values concurrently. Vectorization is performed automatically by the compiler with the option -O3. It is effective at hiding the 4 CP functional unit latency and often achieves a 3-4 times speedup in the average latency per result. The improvement due to vectorization will typically be less due to the overhead of splitting the loop. The vector intrinsics in libm use the same algorithm as the scalar intrinsics and return bitwise identical results.

For codes that do not require answers that are good to the last bit, faster versions of several intrinsic functions are available in the alternate math library, libmfastv. This library can be substituted for libm by including `-lmfastv` on the load line. Results from libmfastv may differ from those of libm in the last 2 units in the last place (ulps) and may show some differences in handling exceptional values. Functions in libmfastv may be faster than those in libm only for fairly long vectors, on the order of 250 elements. Table 3 compares the average times in clock periods per result for the two libraries for a vector of length 1000.

Table 3: Intrinsic function timings (in clock periods per result)

Routine	libm scalar	libm vector	libmfastv vector
SQRT	86	25	20
1.0/SQRT	86	25	27
ALOG	168	35	38
EXP	93	33	35
SIN	121	110	45
COS	128	128	45
A ** B	838	810	95
EXP(B*LOG(A))	264	73	76

We see from this table that vectorization improves SQRT, EXP, and ALOG in libm and also improves SIN and COS in libmfastv. The real to a real power function is usually one of the slowest functions, but for certain ranges of values it can be replaced with the expression EXP(B*LOG(A)). The libmfastv library makes this substitution after checking the inputs, but if the values are known to be safely away from zero, a manual substitution of the alternate expression would be better yet.

Additional versions of the intrinsic functions that are faster in selected cases are provided in benchlib for the true performance aficionado.

3.4 Example: Livermore Loop 22

Livermore loop 22 is quite short, and most of the time is spent in a call to the EXP math library routine:

```
FW = 1.0
DO 22 K = 1, 101
   Y(K) = U(K) / V(K)
   W(K) = X(K) / ( EXP( Y(K) ) - FW )
ENDDO
```

As coded here, the Livermore Loops program reports 29 Mflops for this loop. Compiling with -O3 to enable vectorization improves the performance to 50 Mflops. This is functionally equivalent to the following version, which calls the benchlib routine EXP_V:

```
FW = 1.0
DO K = 1, 101
   Y(K) = U(K) / V(K)
ENDDO
CALL EXP_V( 101, Y, Y )
DO K = 1, 101
    W(K) = X(K) / (Y(K)-FW)
ENDDO
```

4.0 Cache optimizations

In any cache-based machine, the alignment of data on cache lines and the footprints of competing data segments in the cache affect the performance. The CRAY T3E Programming Environment provides compiler directives for aligning data structures on cache line boundaries and for padding common blocks to avoid some cache conflicts. Other issues, such as bad strides and bad leading dimensions of two-dimensional arrays, may require user intervention. This section gives some examples of these effects.

4.1 Computing cache offsets

A good place to start when optimizing for the processor's data caches is to determine the cache offsets of the main data structures and to look for conflicts. The Dcache offset of an address is obtained from the byte address modulo 8192 (or 8*1024), and the Scache offset into one of the three sets is obtained from the byte address modulo 32768 (or 8*4096). We can not determine which set of the 3-way set associative Scache will hold a particular cache line because the set is chosen randomly when the cache line is loaded. The byte address of a data structure X may be obtained by calling the 'loc' function from a Fortran program with X as an argument. Addresses of program blocks such as COMMON blocks in Fortran can also be obtained from a load map using `f90 -Wl"-m"` or `cc -Wl"-m"`.

For example, in Fortran one could say

```
      REAL X(129), Y(129)
      IX = LOC(X)
      IY = LOC(Y)
      WRITE(*,1) 'D', 'X', MOD(IX,8*1024), MOD(IX/8,1024)
      WRITE(*,1) 'S', 'X', MOD(IX,8*4096), MOD(IX/8,4096)
      WRITE(*,1) 'D', 'Y', MOD(IY,8*1024), MOD(IY/8,1024)
      WRITE(*,1) 'S', 'Y', MOD(IY,8*4096), MOD(IY/8,4096)
    1 FORMAT(A1,'cache offset of ',A1,' = ',I6,' bytes (',I5,' words)')
      END
```

The C equivalent is

```
#include <stdio.h>
main()
{
      long ix, iy;
      double x[129], y[129];
      ix = (long) x;
      iy = (long) y;
      printf("Dcache offset of X = %6i bytes (%5i words)\n",
         ix % (8*1024), (ix >> 3) % 1024);
      printf("Scache offset of X = %6i bytes (%5i words)\n",
         ix % (8*4096), (ix >> 3) % 4096);
      printf("Dcache offset of Y = %6i bytes (%5i words)\n",
         iy % (8*1024), (iy >> 3) % 1024);
      printf("Scache offset of Y = %6i bytes (%5i words)\n",
         iy % (8*4096), (iy >> 3) % 4096);
}
```

The Fortran version produces the output

```
Dcache offset of X =  2696 bytes (  337 words)
Scache offset of X = 27272 bytes ( 3409 words)
Dcache offset of Y =  1664 bytes (  208 words)
Scache offset of Y = 26240 bytes ( 3280 words)
```

We see that X and Y do not conflict in either the Dcache or Scache in this example, because their cache offsets are 129 words apart, the same as their length.

4.2 Aligning on a cache-line boundary

Several mechanisms are available for forcing data to be allocated on a cache-line boundary.

- From Fortran, the CACHE_ALIGN directive is provided:

```
      REAL A(M,K), B(K,N), C(M,N)
      COMMON /ACOM/ C
CDIR$ CACHE_ALIGN A, B
CDIR$ CACHE_ALIGN /ACOM/
```

- From C, alignment is specified with a #pragma:

```
#pragma _CRI cache_align flags
int flags[4096];
```

- From CAM (Cray Assembler for the MPP), a block of data is declared in a program section, or psect, which can be specified to begin on a cache-line boundary in one of two ways:

```
.psect  my_data,cache,data
```

  or

```
.psect  my_data,6,data
```

  where an integer k in the range [0,9] in the second field of the instruction aligns the psect on a 2^k byte boundary (i.e., the last k bits of the address are zero).

- From cld, directives control the alignment of blocks, including Fortran common blocks. The default alignment is on an 8-byte boundary; we want to change this to a 64-byte boundary. Most users do not call cld directly, so we show the syntax from the f90 or cc command lines:

```
f90 -Wl"-Dallocate(alignsz)=64" foo.f
```

  or

```
cc -Wl,"-Dallocate(alignsz)=64" foo.c
```

  The value shown is in bytes; values may be specified in 64-bit words, 16-bit parcels, bytes, or bits, but must be a power of two and must be at least a 64-bit word (see the cld man page). The following values are equivalent: `8w, 32p, 64, 64b,` and `512i`. cld allows alignment on any power of two boundary up to the size of a DRAM page.

4.3 Example of cache-line alignment benefits

To illustrate the potential benefits of cache-line alignment, we consider the case of copying an 8-by-N array to another 8-by-N array, in which both arrays have been deliberately misaligned so that each 8-element column spans two Scache lines, instead of one:

```
      INTEGER LDX, N
      PARAMETER (LDX=64, N=1000)
      REAL MBYTES, SECS, T1, T2
      REAL A(LDX,N), B(LDX,N)
      COMMON /AB/ BAD(4), A, B
!DIR$ CACHE_ALIGN /AB/
      DATA A / 64000*1.0 /
      CLKSPD = CLOCKTICK()
      T1 = RTC()
      CALL COPYAB( 8, N, A, LDX, B, LDX )
      T2 = RTC()
      MBYTES = 8.0*REAL( 16*N ) / 1.E+6
      SECS = (T2-T1)*CLKSPD
      WRITE(*,*) `8-BY-N BLOCK COPY = `, MBYTES/SECS,' MB/SEC'
      END
      SUBROUTINE COPYAB( M, N, A, LDA, B, LDB )
      INTEGER N, LDA, LDB
      REAL A(LDA,N), B(LDB,N)
      DO J = 1, N
         DO I = 1, M
            B(I,J) = A(I,J)
         END DO
      END DO
      RETURN
      END
```

The subroutine CLOCKTICK, which returns the speed in seconds of one clock period in the microprocessor, is implemented in C:

```
#include <unistd.h>
float CLOCKTICK(void) {
    long            sysconf(int request);
    float           p;

    p = (float) sysconf(_SC_CRAY_CPCYCLE);      /* cycle time in
                                                 * picoseconds */
    p = p * 1.0e-12;            /* cycle time in seconds */
    return (p);
}
```

When compiled with f90 -O3,unroll2, the 8-by-N block matrix copy runs at 104 MB/sec, counting each byte copied once on the read and once on the write as in the STREAM benchmark.

If we now align the arrays A and B on an 8-word Scache-line boundary by removing the array BAD in the common block declaration:

```
COMMON /AB/ A(LDX,N), B(LDX,N)
```

then the performance improves by nearly a factor of two, to 195 MB/sec. However, using cacheable loads and stores leads to inefficient use of the caches in this example because only a small portion of the arrays A and B remain in cache on completion. As we will see in Section 6.0, use of E-registers for memory-to-memory copies like this can be significantly faster.

4.4 Choosing a good leading dimension

Fixing the alignment of the first element of a two-dimensional array won't help if the leading dimension is not a multiple of the cache line size. In the example of the preceding section, if the declaration of LDX is changed from 64 to 65, then the performance falls to 106 MB/sec with or without BAD in the common block. This is because seven of every eight columns of A and B are unaligned and span two cache lines instead of one.

When a multi-dimensional array is accessed in sub-blocks, it may be beneficial to declare its leading dimensions to be multiples of the Scache line size, which is 8 for 64-bit real data, 4 for 64-bit complex data, 16 for 32-bit real data, and 8 for 32-bit complex data. However, leading dimensions that are multiples of 2 or 4 or 8 may result in inefficient access to rows of the array when E-register methods are used because of bank conflicts (recall from Section 2.0 that the local memory has 8 banks). What constitutes a "good" leading dimension is likely to depend on how the data is used.

5.0 Stream buffer optimizations

Since the greatest bandwidth from local memory to the processor is via the stream buffers, many of the optimization techniques for the CRAY T3E processor involve making efficient use of streams. When streams are enabled, they are detected automatically by the hardware from the pattern of references to local memory. Misses to two consecutive 64-byte Scache blocks are required to initiate a stream, and there is a limit of six streams. As a result, techniques to optimize for streams focus on creating longer patterns of unit-stride memory references and on limiting the number of active streams to six or fewer. The CRAY T3E Fortran Optimization Guide devotes several sections to this subject and includes examples on

- 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

Two additional techniques to be discussed in this section are

- Combining streams with temporary vectors
- Prefetching streams into the cache

Of these optimization techniques, only loop splitting can be performed automatically. The command line options to control loop splitting are:

`-O split0`: disable all loop splitting

`-O split1`: enable loop splitting only for loops marked by a SPLIT directive

`-O split2`: enable loop splitting for all inner loops

When loop splitting is enabled, the compiler will split the loop only if it can be done without changing the computation. Loops that are split are also strip-mined by 256 in order to allow for cache reuse between the resulting smaller loops.

5.1 Counting the streams

The first step in optimizing for the stream buffers is to determine how many streams are in use. The following guidelines may be helpful:

1. Loading instructions the first time through a loop consumes one stream, but this can usually be ignored if there are many iterations because the instructions will subsequently reside in the Icache.
2. Each vector that is loaded from memory is a stream.
3. A vector that is stored generates a stream if the data must first be brought into the Scache due to the write-allocate/write-back nature of the Scache.

For example, the loop

```
DO I = 1, 10000
   C(I) = A(I) + B(I)
END DO
```

involves 3 streams, one each for loading A and B, and one for loading C as it is brought into the Scache to be updated.

A more complicated example is the following code to implement part of a pentadiagonal matrix times vector product:

```
 DO I = 101, 9900
    Y(I) = A(I,1)*X(I-100) + A(I,2)*X(I-1) + A(I,3)*X(I) +
&          A(I,4)*X(I+1) + A(I,5)*X(I+100)
 END DO
```

Although we are accessing the vector X at several different points, the range of values from X(I-100) to X(I+100) is not that large, so we can probably assume that the previous values are still in the cache. Asymptotically, then, only X(I+100) is reading new values, so we have 7 streams: one for Y, one for X, and one for each of the five columns of A.

5.2 Splitting loops to limit the number of streams

To supplement the examples of the Fortran Optimization Guide, we present the following example from an actual application:

```
 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
```

At first glance the six arrays seem an ideal fit for the six stream buffers. But by accessing the well-separated columns j and j-1 of the arrays HX and HZ, we have at least eight input streams, three of which are also output streams. 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 effects of different compiling options are shown in the following table:

Table 4: Compiler optimization of FDTD

f90 -O3	36.8 Mflops
f90 -O3,unroll2	37.1
f90 -O3,split2	49.1
f90 -O3,split2,unroll2	52.2

Just directing the compiler to split the innermost loop nets a 40% improvement. Now, of course, is where it gets interesting.

Splitting the loop manually may give different results. Since the variables EX, EY, and EZ are independent, we can put them in three separate loops, or in two loops with any combination of EX, EY, and EZ in one loop and the remaining array in the other loop. We eliminated the combination of EX and EZ as this would result in seven streams. Using the compiling options f90 -O3,unroll2, we observed the following performance:

Table 5: Manual splitting of FDTD

Split into {EX},{EY},{EZ}	49.4 Mflops
Split into {EX}, {EY,EZ}	41.0
Split into {EX,EY}, {EZ}	59.9

5.3 Combining streams with temporary vectors

An alternative strategy that works nearly as well is to condense several streams into a temporary vector. Continuing with the example of the previous section, we show how to do this for the subroutine fdtd:

```
 SUBROUTINE FDTD(CAEX,CBEX,CAEY,CBEY,CAEZ,CBEZ,EX,EY,EZ,HX,HY,HZ)
 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)
 REAL TMP(3,128)
 DO J = 2, 127
    DO I = 2, 127
       DO K = 2, 126, 2
          TMP(1,K) = HZ(K,I,J) - HZ(K,I,J-1)
          TMP(2,K) = HX(K-1,I,J) - HX(K,I,J) +
&                    HZ(K,I-1,J) - HZ(K,I,J)
          TMP(3,K) = HX(K,I,J-1) - HX(K,I,J)
          TMP(1,K+1) = HZ(K+1,I,J) - HZ(K+1,I,J-1)
          TMP(2,K+1) = HX(K,I,J) - HX(K+1,I,J) +
&                      HZ(K+1,I-1,J) - HZ(K+1,I,J)
          TMP(3,K+1) = HX(K+1,I,J-1) - HX(K+1,I,J)
       END DO
       DO K = 2, 127
          EX(K,I,J) = CAEX*(EX(K,I,J) +
&                     CBEX*(TMP(1,K) + HY(K,I,J) - HY(K-1,I,J)))
          EY(K,I,J) = CAEY*(EY(K,I,J) +
&                     CBEY*(TMP(2,K)))
          EZ(K,I,J) = CAEZ*(EZ(K,I,J) +
&                     CBEZ*(TMP(3,K) + HY(K,I,J) - HY(K,I-1,J)))
       END DO
    END DO
 END DO
```

```
RETURN
END
```

The first loop over k contains five streams, consisting of the array tmp and two columns each of HZ and HX. The second k loop also contains five streams: three for the arrays EX, EY, and EZ, one for HY, and one for TMP (which may still be in the cache). This version returns 57.6 Mflops. This is a lot more work than just splitting the loop, but it may be the only option if the inner loop variables all depend on each other.

5.4 Prefetching streams into the cache

Another technique for minimizing the number of streams is to prefetch one or more streams into the cache. This technique may not be as effective as loop splitting in complicated cases involving many streams because cache conflicts between the streams may cause the prefetched data to be removed from the cache before it can be used. However, it is useful for getting data into the cache that might not be recognized as a stream because it is accessed in a non-uniform way, such as across rows or in reverse order.

The Benchlib subroutine for cache prefetching on the CRAY T3E has the interface

```
INTEGER POINTERS(2,NUM_POINTERS)
CALL PREF3M( NUM_POINTERS, POINTERS )
```

where **NUM_POINTERS** is the number of address ranges and for i=1,**NUM_POINTERS**, the address range **POINTERS**(1,i) to **POINTERS**(2,i) will be loaded into the cache. There is no restriction on the number of regions to be prefetched, and the sizes of the different regions need not be the same. The subroutine does not check to see if it is possible for all the requested data to coexist in the cache. At least two consecutive cache lines are loaded for each address range.

For example, the following subroutine performs a local transpose and copy of two arrays that are small enough to fit in cache:

```
SUBROUTINE ORIG(FLD1,FLD2)
PARAMETER (LONPPE=16,LEVS=32,LONFD=LONPPE*LEVS)
REAL FLD1(LONFD),FLD2(LONFD)
COMMON /ZZ/ WRK1,WRK2
REAL WRK1(LONPPE,LEVS),WRK2(LONPPE,LEVS)
DO I=1,LONPPE
  ICOL=LEVS*(I-1)
  DO J=1,LEVS
    FLD1(ICOL+J)=WRK1(I,J)
    FLD2(ICOL+J)=WRK2(I,J)
  ENDDO
ENDDO
RETURN
END
```

The first access of the arrays **WRK1** and **WRK2** is slow because the access pattern is across rows, preventing use of the streams. We can bring these arrays into the cache by prefetching them before their first use, as follows:

```
INTEGER P(2)
...
P(1)=LOC(W)
P(2)=LOC(W(1024))
CALL PREF3M(1,P)
```

This improves the performance from 41 microseconds down to 28 microseconds (we do not quote a transfer rate in megabytes per second because the store data is still in the Scache). Alternative techniques for transposing local arrays using E-register operations will be described in section 6.0.

5.5 Turning streams on and off

All the results in this report were obtained on a CRAY T3E system using Programming Environment 3.0 with streams enabled. However, maintaining data coherence of the memory streams was problematic in early releases of the CRAY T3E hardware and software. In the worst case, use of streams could cause the system to hang and require a reboot. As a result, stream use was disabled by default on many early systems.

For the single-processor results of this paper, stream safety was generally not a concern, so streams were enabled by setting an environment variable:

```
setenv SCACHE_D_STREAMS 1
```

Conversely, streams can be disabled by

```
setenv SCACHE_D_STREAMS 0
```

A simple test to see if streams are on is to compile and run the following Fortran program:

```
INTEGER NMAX, N
PARAMETER (NMAX = 501*1024)
REAL CLKSPD
REAL X(NMAX), Y(NMAX), Z(NMAX), ALPHA, PAD(512)
REAL CLOCKTICK
EXTERNAL CLOCKTICK
DATA X / NMAX*1.0 /, Y / NMAX*2.0 /, ALPHA / 3.0 /
COMMON / ACOM / X, PAD, Y, Z
CLKSPD = CLOCKTICK()
N = 500*1024
WRITE(*,*) `SAXPY performance for n = `, n
DO IOFF = 1, 8
   T0 = RTC()
```

```
   CALL SAXPY( N, ALPHA, X(IOFF), 1, Y(IOFF), 1 )
   T1 = RTC()
   OPS = REAL(2*N) / 1.E+6
   TIMS = ( T1-T0 )*CLKSPD
   RATE = OPS / TIMS
   WRITE(*,*) `   ioff = `, ioff, `, `, rate, ` Mflops'
END DO
END
```

If the output shows at least 48 Mflops on a CRAY T3E, or 57 Mflops on a CRAY T3E-900, then the streams are on. If the streams are off, the performance will be about 27 Mflops.

Further information about using hardware streams is available from the `intro_streams` and `streams_guide` man pages.

6.0 E-register optimizations

Applications that move large amounts of data or that access data with large or non-constant strides may not benefit from the optimizations for the caches and stream buffers described in the preceding sections. For these cases, non-cached data access using E-register operations may be more efficient. In this section, we show how to make use of the E-registers by means of the CACHE_BYPASS directives, shmem library routines, or benchlib.

Following some general usage notes in section 6.1, the three major E-register access techniques are described in sections 6.2, 6.3, and 6.4. In particular, section 6.4 serves as a user's guide to the E-register routines in benchlib, which are not documented anywhere else. In section 6.5, we give examples of some common applications that can be optimized by two or more of the E-register methods. These examples include a simple block copy, a matrix transpose, initialization of a large data area, and a gather operation. In section 6.6, we give some further examples to illustrate unique applications of benchlib. These include an 8-by-n matrix copy, an in-place transpose, selective cache-line flushing, and direct access to the E-register data area from user codes.

6.1 Usage notes

Some general warnings about the user E-registers should be noted. The same set of E-registers is used by the compiler, the benchlib subroutines, and the external communication routines in MPI, PVM, and SHMEM. Data in the E-registers are not preserved across calls to any of these routines, so it is necessary to separate different uses of the E-registers in time. Also, the benchlib routines may require some synchronization to maintain coherence of the local memory when E-register accesses are mixed with cacheable loads and stores.

The use of non-cached data access for an array has the side-effect of removing any lines of that array that may already be in the cache. This could make the program slower by increasing the amount of memory traffic or by requiring the array to be reloaded if it is used again. Since the compiler can not tell where in memory an array will reside at a particular point in time, E-register optimizations always require some programmer intervention, in the form of directives or calls to library routines.

User programs that call benchlib routines or that use the E-register memory space explicitly should specify one of the following directives in the calling routine:

From Fortran:

```
!DIR$ USES_EREGS
```

From C:

```
#pragma uses_eregs
```

From CAM:

```
.uses_eregs
```

These directives cause a bit to be set in the a.out header, indicating that the program uses E-registers. Other E-register optimizations (such as the CACHE_BYPASS directives) also set this bit. Whether or not having this bit set has any side effects, such as disabling streams, is site-dependent.

6.2 CACHE_BYPASS directives

A new feature of Programming Environment 3.0 is a directive that suggests to the compiler that one or more variables may benefit from non-cached data access. The format of the directive is as follows:

From Fortran:

```
!DIR$ CACHE_BYPASS var1, var2
```

From C:

```
#pragma _CRI cache_bypass var1, var2
```

The directive can only be applied to loop constructs: “do” loops in Fortran and “for”, “while” and “do while” loops in C/C++. The directive has a loop scope, and the named variables must have a base type that is 64 bits. In Fortran 90, the supported types are INTEGER(KIND=8), REAL(KIND=8), and LOGICAL(KIND=8), and in C the supported types are long and double. Arrays and pointers to arrays of these base types may be named in the directive.

The compiler may ignore the directive if it determines that it can not generate efficient code for the indicated loop. One of the requirements is that the loop must be an inner loop; another is that it must be “vectorizable”. The compiler directive IVDEP may be used to help the compiler get past ambiguous data dependencies.

Use of the CACHE_BYPASS directives enables the bit in the a.out header to indicate that the program uses E-registers.

6.3 SHMEM

The SHMEM library is a collection of logically shared, distributed memory access routines that operate on remote and local memory. It includes routines for remote data transfer, work-shared broadcast and reduction, barrier synchronization, and atomic memory operations. The routines for remote data transfer can be used for local data transfers if the remote processor number is the same as the local processor number. Separate interfaces are provided for access from Fortran or C; a capitalized version of the name serves as the Fortran entry point and assumes all arguments are passed by address, and a lower-case version of the name serves as the C entry point and assumes the scalar arguments are passed by value.

The interfaces to several of the SHMEM routines used in the examples of the following sections are as follows:

`shmem_get(target,source,length,pe)`

Copy `length` words of data from `source` (on processor `pe`) to `target` (on the local processor, `my_pe`).

`shmem_iget(target,source,target_inc,source_inc,length,pe)`

Copy `length` words of data from `source` (on `pe`) with stride `source_inc` to `target` with stride `target_inc`.

`shmem_ixget(target,source,source_index,length,pe)`

Gather `length` words of data from `source` (on `pe`) with index `source_index` to `target`.

`shmem_ixput(target,source,target_index,length,pe)`

Scatter `length` words of data from `source` (on `pe`) to `target` with index `target_index`.

The `source` and `target` arrays are assumed to contain 64-bit data; separate interfaces are provided for 32-bit data. The indexing of the `source` and `target` arrays in `shmem_ixget` and `shmem_ixput` is zero-based, which means that `source` and `target` are assumed to have been declared as `source(0:n-1)` and `target(0:n-1)`, respectively. For local data transfers, `pe` is equal to `my_pe`, and `shmem_get` is the same as `shmem_put`.

Further details about the contents of the SHMEM library are provided on the `intro_shmem` man page.

6.4 Benchlib

Benchlib for the CRAY T3E includes a collection of E-register routines that perform one-sided data transfers, from memory to E-registers or from E-registers to memory. These one-sided transfers give the user the greatest amount of control over his or her use of E-registers at the expense of some of the SHMEM library's ease of use. Because only 480 E-registers are available for user data, the maximum length of an E-register transfer in the benchlib routines is generally 480. The benchlib routines initiate data transfers but do not wait for them to complete, giving the user the option of performing some other computation while the transfer is going on or of waiting for the transfer to complete. Only one set of interfaces is provided -- a call-by-address interface aimed at Fortran users. Benchlib is available to any interested CRAY T3E user, but it is not a supported product.

To clarify our discussion of local memory transfers, we will define the following four classes of operations:

LOAD: a transfer of data from memory to a register

STORE: a transfer of data from a register to memory

GET: a transfer of data from memory to an E-register

PUT: a transfer of data from an E-register to memory

LOADs and STOREs have data types associated with them, depending on whether they involve an integer or floating point register. GETs and PUTs are typeless because they only move bits from memory to memory. To distinguish LOADs and STOREs to cacheable memory space from LOADs and STOREs to E-register data, we will sometimes refer to the former as cacheable LOADs and STOREs.

6.4.1 64-bit benchlib routines

The interfaces to the 64-bit E-register routines in benchlib are as follows:

```
CALL LGETV( source, source_inc, length )
CALL LGETVO( source, source_inc, length, offset )
```

Get `length` words from `source` with stride `source_inc` to E-registers, beginning with user E-register 0 (LGETV) or user E-register `offset` (LGETVO).

```
CALL LPUTV( target, target_inc, length )
CALL LPUTVO( target, target_inc, length, offset )
```

Put `length` words from E-registers, begnning with user E-register 0 (LPUTV) or user E-register `offset` (LPUTVO) to `target` with stride `target_inc`, where `offset` must be a multiple of 8.

```
CALL LSETV( target, target_inc, length, value )
```

Set `length` words of the array `target` with stride `target_inc` to `value`.

```
CALL LGATH( source, source_index, length )
```

Gather `length` words from `source` with index `source_index` to E-registers.

```
CALL LSCAT( target, target_index, length )
```

Scatter `length` words from E-registers to `target` with index `target_index`.

As in the case of the SHMEM library, the indexing of the `source` and `target` arrays in LGATH and LSCAT is zero-based. In LGETV, LPUTV, LGATH, and LSCAT, the `length` argument may be at most 480 because there are only 480 user E-registers. In LGETVO and LPUTVO, the `length` argument may be at most 480-`offset` in order to stay within the E-register space, where $0 \le \texttt{offset} < 480$.

Two additional benchlib routines are provided for GETs and PUTs of 8-word contiguous data blocks separated by an arbitrary stride:

CALL LGETV8(source, source_inc, length)

Get `length` 8-word blocks of data from `source` with a stride of `source_inc` between blocks into E-registers.

CALL LPUTV8(target, target_inc, length)

Put `length` 8-word blocks of data from E-registers to `target` with a stride of `target_inc` between blocks.

Moving data in 8-word chunks is of interest because it makes best use of the eight interleaved memory banks in the local memory of a CRAY T3E node. The maximum value for `length` in LGETV8 and LPUTV8 is 60.

6.4.2 Synchronizing E-register operations

Because E-register memory transfers are performed asynchronously, it is necessary to wait for pending transfers to complete if GETs and PUTs to the same memory locations are close in time. This is also true if cacheable loads occur close to a PUT or cacheable stores occur close to a GET. The benchlib function LPUTP is provided for this purpose. The syntax for using LPUTP from Fortran is as follows:

```
       INTEGER*8 LPUTP
      ...
123   IF ( LPUTP().NE.0 ) GOTO 123
```

A synchronization of this form should be used in the following situations:

1. If a STORE or PUT is followed by a GET of the same memory location, or a PUT is followed by a LOAD of the same memory location, a call to LPUTP should be inserted after the STORE or PUT. This ensures that the memory location has been updated with the store data before we read it.

2. If a GET is followed by a STORE or PUT to the same memory location, or a LOAD is followed by a PUT to the same memory location, a call to LPUTP should be inserted after the LOAD or GET to ensure that we have read the load data before it is changed.

The hardware will handle synchronization of a STORE followed by a LOAD of the same location, or a LOAD followed by a STORE.

It is good practice to insert a call to LPUTP after the last of a series of E-register operations to independent memory locations, even if the transferred data will not be used for a long time. It is not necessary to insert calls to LPUTP to avoid race conditions within the E-registers themselves. Each E-register has an empty/full bit that is set to empty when a GET is initiated and full when the GET completes, and a PUT of an empty E-register is blocked until it is marked as full.

6.4.3 32-bit Benchlib routines

Versions of most of the benchlib routines are also available for which `source`, `target`, `source_index`, or `target_index` are arrays of 32-bit data. The half-precision GET operation loads each 32-bit data element into the first half of a 64-bit E-register. If the E-register data were then referenced using a pointer to an array of 32-bit data elements, it would appear as if only every other element in the 32-bit array had been initialized (see the example of section 6.6.4). In the following list, we indicate the 32-bit arguments by prepending their names with the letter “h” (for “half-precision”):

```
CALL HGETV( hsource, source_inc, length )
CALL HGETVO( hsource, source_inc, length, offset )

CALL HPUTV( htarget, target_inc, length )
CALL HPUTVO( htarget, target_inc, length, offset )

CALL HSETV( htarget, target_inc, length, hvalue )

CALL LGATHH( source, hsource_index, length )
CALL LSCATH( target, htarget_index, length )

CALL HGATH( hsource, source_index, length )
CALL HSCAT( htarget, target_index, length )

CALL HGATHH( hsource, hsource_index, length )
CALL HSCATH( htarget, htarget_index, length )
```

6.5 Common applications

In this section, we provide examples of some basic data transfers that benefit from non-cached data access. Each example begins with a clean cache and the data to be accessed in local memory. Performance results for comparison are given in MB/sec (megabytes per second), counting each byte once on a GET and again on the corresponding PUT.

6.5.1 Block copy

The simplest example of an operation that should avoid the cache is a block copy in which the data being copied can not all fit in the cache. When cacheable loads and stores are used for the operation `y = x`, the number of words moved is $3n$, where n is the length of the vectors: n to read x, n to load y into the cache so it can be updated, and n to write y to memory when its cache lines are replaced. A non-cached algorithm would move only $2n$ words: n to read x and n to write y.

We measured a cached Fortran implementation of the block copy with $n = 100000$ at 401 MB/sec. The E-register implementations are as follows:

CACHE_BYPASS: 593 MB/sec

```
!DIR$ CACHE_BYPASS X, Y
      DO I = 1, N
         Y(I) = x(I)
      END DO
```

SHMEM library: 775 MB/sec

```
CALL SHMEM_GET(Y,X,N,MY_PE)
```

Benchlib: 592 MB/sec

```
    DO I = 1, N, 480
       NI = MIN( 480, N-I+1 )
       CALL LGETV( X(I), 1, NI )
       CALL LPUTV( Y(I), 1, NI )
    END DO
123 IF( LPUTP().NE.0 ) GO TO  123
```

6.5.2 Matrix transpose

A variation on the block copy that involves non-unit stride accesses is a local matrix transpose, $B \leftarrow A^T$. In this case, several implementations are possible: columns of the array A could be copied to rows of the array B, rows of A could be copied to columns of B, or diagonals of A could be copied to diagonals of B. We observed different performance for the different E-register methods for each implementation, and the best case depended on the leading dimensions of the arrays A and B. For instance, copying diagonals to diagonals was better when the leading dimensions were powers of two, but copying rows to columns was usually the best otherwise.

We measured the cached Fortran performance for copying a 1025 x 1025 matrix A to a matrix B of the same size at 92 MB./sec. The E-register versions are as follows:

CACHE_BYPASS: 410 MB/sec

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

SHMEM library: 418 MB/sec

```
DO J = 1, N
   CALL SHMEM_IGET( B(1,J), A(J,1), 1, LDA, N, MYPE )
END DO
```

Benchlib: 501 MB/sec

```
DO J = 1, N, 480
   JN = MIN( N-J+1, 480 )
   DO I = 1, M
      CALL LGETV( A(I,J), LDA, JN )
      CALL LPUTV( B(J,I), 1, JN )
```

```
         END DO
      END DO
123 IF( LPUTP().NE.0 ) GO TO 123
```

6.5.3 Data initialization

Another application of E-registers is initializing a data area to a constant value. The use of cacheable stores may be inefficient because the data is first loaded into the secondary cache, then modified for write-back later. Half of the data motion is eliminated by not copying the data to be modified from memory into the cache. This is probably the easiest opportunity for optimization to recognize in an application code.

We measured the cached Fortran performance for initalizing a 64000-element array A at 256 MB/sec. The E-register versions shown below are over two times faster:

CACHE_BYPASS: 527 MB/sec

```
!DIR$ CACHE_BYPASS A
      DO I = 1, N
         A(I) = 0.0
      END DO
```

Benchlib: 527 MB/sec

```
      SUM = 0.
      CALL LSETV( A, 1, N, SUM )
  123 IF( LPUTP().NE.0 ) GO TO 123
```

6.5.4 Gather/scatter

A powerful feature of the E-registers is their ability to do non-cached gather and scatter operations. Cacheable LOADs and STOREs tend to waste a lot of bandwidth in irregular access patterns because often only one word of each eight-word Scache line is used. The more random the access pattern, the less likely it is that any reuse will be made of an Scache line while it is in the cache.

The performance benefit of using a non-cached gather or scatter depends a great deal on the size of the operation, how random it is, and whether the reuse, if any, of the indexed vector occurs in a small enough time interval. Therefore, to illustrate the different techniques, we will use an artifical example of gathering 1000 randomly chosen elements from a 100,000 element vector. With an average of one word chosen from every 100 vector elements, there is very little cache reuse, and an implementation using cacheable loads and stores runs at 56 MB/sec, counting only the floating point data read and written. The E-register versions are as follows:

CACHE_BYPASS: 302 MB/sec

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

SHMEM library: 252 MB/sec

```
MYPE = MY_PE()
CALL SHMEM_IXGET( X, A(0), INDEX, NGATH, MYPE )
```

Benchlib: 355 MB/sec

```
    DO I = 1, NGATH, 480
       NI = MIN( 480, NGATH-I+1 )
       CALL LGATH( A(0), INDEX(I), NI )
       CALL LPUTV( X(I), 1, NI )
    END DO
123 IF( LPUTP().NE.0 ) GO TO 123
```

An application that uses the gather operation is the multiplication of a sparse matrix by a dense vector. We will defer this example to section 6.6.5 to show how to use data out of the E-registers.

6.6 Uncommon applications

A few of the features of benchlib are not reproducible using CACHE_BYPASS or SHMEM. In this section, we include a few examples of E-register optimizations for which our only version makes use of benchlib routines. In particular, the in-place transpose, which uses the E-register space for temporary storage during a swap, and the examples that operate on the E-register space directly following a GET make use of the one-sided nature of the E-register routines in benchlib.

6.6.1 8-by-n block copy

In Section 4.3, we gave an example of an 8-by-n block copy that ran at 195 MB/sec using cacheable loads and stores if the source and target arrays were both Scache-line aligned and had leading dimensions that were multiples of the cache line size. We can get better performance from benchlib using the routines LGETV8 and LPUTV8:

```
    DO J = 1, N, 60
       JN = MIN( N-J+1, 60 )
       CALL LGETV8( A(1,J), LDX, JN )
       CALL LPUTV8( B(1,J), LDX, JN )
    END DO
123 IF( LPUTP().NE.0 ) GO TO 123
```

This version achieves 586 MB/sec, a factor of three improvement over the best case implementation using cacheable loads and stores, and it is insensitive to the leading dimensions of A and B and their initial cache-line offsets.

6.6.2 In-place transpose

The E-register space can serve as temporary storage during a swap by loading one vector into the E-registers starting with user E-register 0 and another into the E-registers at some other offset, such as 240. The benchlib routines LGETVO and LPUTVO are the only library interface that give the user control over which E-registers will be used. The simplest example of a swap is a row swap, as in the LINPACK benchmark when a row interchange is required to implement partial pivoting in the Gaussian elimination algorithm. A more interesting example is an in-place transpose, in which a row must be exchanged with a column or a diagonal with another diagonal. An example of an in-place transpose using benchlib is as follows:

```
    DO J = 1, N-1
       DO I = 1, N-J, 240
          NI = MIN( (N-J)-I+1, 240 )
          CALL LGETVO( A(J+I,J), 1, NI, 0 )
          CALL LGETVO( A(J,J+I), LDA, NI, 240 )
          CALL LPUTVO( A(J,J+I), LDA, NI, 0 )
          CALL LPUTVO( A(J+I,J), 1, NI, 240 )
       END DO
    END DO
123 IF( LPUTP().NE.0 ) GO TO 123
```

Note that the inner loop is strip-mined by 240 because half the E-registers are being used for each vector. If A is a 1025 x 1025 matrix, this transpose runs at the rate of 456 MB/sec, compared to 121 MB/sec from a Fortran version using cacheable loads and stores.

6.6.3 Flushing the cache

Recall from section 2.2 that the external backmap maintains coherence of the caches and the local memory during an E-register operation by removing from the caches any line that could match the address of an E-register GET or PUT. We can exploit this feature to implement selective cache-line flushing with the one-sided E-register routines in benchlib. User-directed flushing of local cache data is not required for correctness on the CRAY T3E, so library routines that did this on the CRAY T3D, such as `shmem_udcflush` and `shmem_udcflush_line`, have been converted to null operations for the CRAY T3E. However, it is useful to be able to flush the Scache for timing purposes.

The following subroutine will remove from the Scache any 64-byte blocks that include the indicated range of addresses, from `a(1)` to `a(1+(length-1)*stride)` with stride `stride`:

```
SUBROUTINE FLUSHLINES( A, STRIDE, LENGTH )
REAL A(*)
INTEGER*8 STRIDE, LENGTH
DO I = 1, LENGTH, 480
   ILEN = MIN(480,LENGTH-I+1)
   CALL LGETV(A((I-1)*STRIDE+1),STRIDE,ILEN)
END DO
RETURN
END
```

It can be invoked as follows:

```
ILEN = (N+7)/8
CALL FLUSHLINES(A,8,ILEN)
```

6.6.4 Accessing the address of the E-registers

Benchlib is unique among library interfaces to the E-registers in that it leaves information in the E-registers for use by subsequent operations. If the subsequent operation is one of the PUT routines from benchlib, then the user need not know the location of the E-registers in memory -- it is computed internally. However, if the subsequent operation is a library routine that expects an address, such as a libsci routine, then the user may need this address from a high-level language. It is provided by the benchlib routine LOCOFE. The following example illustrates its usage:

```
REAL(KIND=8) X(10), Y(10), UEREGS(0:479)
POINTER (IAME,UEREGS)
IAME = LOCOFE()
DO I = 1, 10
   X(I) = REAL(I)
   Y(I) = 1.0/REAL(I)
END DO
CALL LGETV( X, 1, 10 )
TEN = SDOT( 10, UEREGS(0), 1, Y, 1 )
WRITE(*,*) `RESULT OF SDOT = `,TEN
END
```

When the 32-bit E-register routines are used, only the first half of each 64-bit E-register receives a result. An equivalent example using 32-bit data is as follows:

```
REAL(KIND=4) X(10), Y(10), UEREGS(0:959)
POINTER (IAME,UEREGS)
IAME = LOCOFE()
DO I = 1, 10
   X(I) = REAL(I)
   Y(I) = 1.0/REAL(I)
END DO
CALL HGETV( X, 1, 10 )
TEN = HDOT( 10, UEREGS(0), 2, Y, 1 )
WRITE(*,*) `RESULT OF HDOT = `,TEN
END
```

6.6.5 Vector operations using E-register data

Although libsci routines may be passed the address of the E-registers, they treat their arguments as if they were in local memory or the Scache. Several subroutines for basic vector operations are available in benchlib that have been optimized for the case where one of the arguments is an address in the E-registers. The subroutine interfaces are as follows:

```
CALL VEADD (N,C,E,X,S)
CALL VEMLT (N,C,E,X,S)
CALL VESUBC(N,C,E,X,S)
CALL VESUBE(N,C,E,X,S)
```

where

N is the number of elements

C is a REAL(KIND=8) array of dimension (N), scheduled as if in Scache

E is a REAL(KIND=8) array of dimension (N), scheduled as if in E-registers

X is a REAL(KIND=8) array of dimension (N) that contains the result on exit

S is a REAL(KIND=8) array of dimension (8), used for workspace

and the operations performed are

VEADD:	X = C + E
VEMLT:	X = C * E
VESUBC:	X = E - C
VESUBE:	X = C - E

There is also one function subprogram:

```
dot = VEDOT (N,C,E,S)
```

which computes the REAL(KIND=8) dot product of C and E:

$$\mathrm{dot} = \sum_{i=1}^{N} C(i) \cdot E(i)$$

An example in which these subroutines may be used is in optimizing a sparse matrix-vector multiply. Typically, only the nonzeros of a sparse matrix are stored, along with some indexing information that specifies the row and column in which each element resides. One such format uses three arrays:

A, the nonzero elements, stored sequentially by rows

COLIDX, an integer array containing the column index of each element of A

ROWSTR, an integer array of order NROWS containing an index to the start of each row in A and COLIDX

This is the format used by the Conjugate Gradient (CG) benchmark from the NAS Parallel Benchmark suite.

In the Class A problem, the sparse matrix has 7000 rows, and in the Class B problem, the matrix has 18,750 rows. The standard Fortran implementation attains 14.5 Mflops/PE on four processors for the Class A problem, and 8.0 Mflops/PE on 16 processors for the Class B problem. We see a drop in performance because the dense vector in the larger problem is too big to fit in the cache. Most of the time in the CG benchmark is spent in the sparse matrix-vector multiplication.

We can optimize the sparse matrix-vector multiplication for E-register usage by adding a CACHE_BYPASS directive on the inner loop:

```
      DO 200 I = 1, NROWS
         Y(I) = 0.
!DIR$ CACHE_BYPASS X
         DO K = ROWSTR(I), ROWSTR(I+1) - 1
            Y(I) = Y(I) + A(K) * X (COLIDX(K))
         END DO
  200 CONTINUE
```

This one-line change improves the performance to 30 Mflops/PE on both the Class A and Class B problems.

Further optimizations are possible by using a 32-bit index vector for the column indices and by using the benchlib routines that operate on data in the E-registers. In this case we will use the benchlib routine LGATHH to gather a sparse row of the 64-bit array X with a 32-bit index vector into a segment of the E-registers, and the subroutine VEDOT to form the dot product of a row of the sparse matrix with the gathered vector, which is still in the E-registers. The resulting code is as follows:

```
      REAL(KIND=8) UEREGS(0:479), STMP(8)
      POINTER (L_EREG,UEREGS)
      L_EREG = LOCOFE()
      ...
      DO 200 I = 1, NROWS
        YTMP=0.
        DO K1=ROWSTR(I),ROWSTR(I+1)-1, 480
           K2= MIN(K1+479, ROWSTR(I+1)-1)
           CALL LGATHH(X(0),COLIDX(K1),K2-K1+1)
           YTMP=YTMP+VEDOT(K2+1-K1,A(K1),UEREGS(0),STMP)
        ENDDO
        Y(I)=YTMP
  200 CONTINUE
```

This version attains 35 Mflops/PE for both the Class A and Class B problems.

7.0 Compiling for performance

This section covers the CRAY F90 Version 3.0 compiler and the options and directives that can affect code performance. Many optimizations are turned on by default, but in most cases additional performance can be obtained by enabling more advanced features.

7.1 Compiler Options

This section covers several key compiler options that are available under version 3.0 of the CF90 programming environment.

7.1.1 Common Block Padding: -a pad[n]

In many cases, performance can be improved by padding local static storage and common blocks to reduce intra-array cache conflicts. This option is ordinarily prevented for common blocks by the Fortran standard, which specifies that data areas in common must be consecutive. The `-a pad` option overrides the standard behavior and allows the compiler to pad between arrays in common. To use this option safely, you must ensure that common blocks are identically declared throughout the program and that no arrays within common blocks make out-of-bounds references. All files must be compiled with `-a pad`, and the same version of the compiler should be used because the padding algorithm may change.

If n is not specified, a fixed amount of padding is added after each array. The size of this padding is dependent on the array sizes, and the compiler makes its best attempt to choose a value to minimize cache conflicts between arrays. If n is specified, padding is added in 8-byte words according to the value specified.

Typically, programs which will benefit most from this option tend to have arrays that are sized as large powers of 2.

7.1.2 General Optimizations: -O[0123]

This option specifies general levels of scalar and vector optimization. In particular, -O3 enables intrinsic function vectorization. The default general optimization level is 2.

7.1.3 Unrolling: -Ounroll[012]

Turning on automatic loop unrolling almost always improves performance because more functional unit parallelism is exposed to the scheduling phase of the compiler. In addition, unrolling permits loads and stores to “merge” effectively in the MAF (miss address file) entries in the EV5 processor. The default level is 0. Level 1 performs no automatic unrolling, but merely interprets unrolling compiler directives. Level 2 performs automatic unrolling, usually by 4, on loops which are likely to benefit as determined by the compiler.

7.1.4 Pipelining: -Opipeline[0123]

Software pipelining is similar to loop unrolling in many respects. Again, the idea is to increase functional unit utilization by presenting independent operations to the hardware. The numbers refer to different levels of pipelining, with 0 being the default.

Formally, a software pipeline is created by identifying a repetitive pattern which would appear while scheduling an infinite number of loop iterations. This scheduling pattern is then converted into a new loop. Software pipelining is likely to improve the performance of parallel and vector loops, even after the loop has been unrolled, unless the resulting software pipeline needs more registers than available on the processor. In such cases the software pipeliner leaves the loop in its original form.

7.1.5 Loops Splitting: -Osplit[012]

Because the T3E memory system is designed with a system of six stream buffers instead of a board level cache, it is important to try to limit the number of data streams to six or fewer in any given loop. For this reason, it is often beneficial to split the loop. Level 0 is the default and performs no loop splitting. Level 1 will only interpret user compiler directives and Level 2 will automatically split loops. Automatic loop splitting frequently slows down codes overall because the compiler assumes that all streams are satisfied from memory and not the data cache. Since memory reference patterns are often satisfied from the Scache, they do not always require the use of a stream buffer and hence may not need splitting. Determining where data coming from in any given loop is very context dependent and can be readily determined by a programmer, but not a program (i.e. a compiler). For this reason, the compiler directive tends to be much more useful than the automatic loop splitting option.

7.1.6 Vectorization: -Ovector[0123]

This option determines the general level of vectorization transformations. At the moment, this includes substituting intrinsic functions such as EXP, ALOG, and SQRT with vectorized equivalents. The default is vector2.

7.1.7 Scalar Optimization: -Oscalar[0123]

Specifies the various levels of scalar optimization. The default is scalar2.

7.1.8 Inlining: -Oinline[0123], inlinefrom=source

Specifies the various levels of inlining. The default is inline0. The inlinefrom=source argument specifies a file or directory that contains procedures for inline code expansion.

7.1.9 Using 32-bit Reals and Integers: -s default32

This option switches the default size of REALS and INTEGERS to 32-bits. The use of 32-bit data instead of 64-bit data will improve the memory system performance by reducing the amount of data that must be transferred, but it may not work with library routines if they expect 64-bit operands.

7.1.10 Loading Fast Math: -Wl”-lmfastv”

This option directs the loader to use the faster (but not fully IEEE compliant) math intrinsic libraries.

7.2 Compiler Directives

Table 6 provides a summary of compiler directives that are useful for code optimization purposes.

Table 6: f90 compiler directives

Compiler Directive	Purpose
`!dir$ unroll n`	Unroll loop by *n*
`!dir$ cache_align var1` `!dir$ cache_align /com1/`	Align variable or common block on Scache boundary
`!dir$ cache_bypass var1,var2`	Use E-register mechanism for accessing var1, var2.
`!dir$ split`	Split loop for streams utilization.
`!dir$ ivdep`	Loop is vectorizable (in the case of intrinsic functions or splitting optimizations).

8.0 Case study: the NAS benchmark kernels

In this section, we examine the NAS Kernel benchmark test. When originally compiled with zero changes and run on the Cray T3E system, we get the results shown in Table 7 on a single processor.

Table 7: NAS kernels with zero changes

PROGRAM	f90	-O3	-O3 -apad	-O3, unroll2 -apad	-O3, unroll2, pipeline2 -apad	-O3, unroll2, pipeline2 -apad -lmfastv	-O3, unroll2, pipeline2, split2 -apad -lmfastv
MXM	84.8	84.8	92.3	137.3	172.6	174.2	174.5
CFFT2D	21.2	21.2	21.2	21.3	21.9	23.1	23.1
CHOLSKY	19.7	19.5	19.5	20.4	26.4	26.4	26.3
BTRIX	49.3	49.3	47.8	47.9	47.8	48.0	31.5
GMTRY	67.9	72.3	73.1	73.1	73.1	73.0	72.5
EMIT	61.5	139.3	139.9	189.5	219.4	246.4	247.3
VPENTA	4.2	4.3	26.5	26.5	26.4	26.4	8.66
Total	19.1	19.4	35.9	37.4	40.0	41.0	28.36

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. 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.

The automatic loop-splitting feature was less than successful, however. Kernels BTRIX and VPENTA slowed down substantially and none of the kernels improved with automatic loop splitting.

We use -O3,unroll2,pipeline2 -apad as our optimal set of compiler options linking to the fast math intrinsics. Next, we look at the seven kernels in detail.

8.1 Kernel 1: MXM

As implemented from FORTRAN, MXM (matrix multiply) is unrolled by 4 and is running at about 175 Mflops. Since this subroutine exists in an optimized library, we could simply call SGEMM. The CF90 3.0 compiler supports pattern recognition and should be able to recognize and replace this kernel automatically. The compiler doesn't recognize this unrolled variant of SGEMM:

```
      C = 0.0
      DO 110 J = 1, M, 4
        DO 110 K = 1, N
          DO 110 I = 1, L
            C(I,K) = C(I,K) + A(I,J) * B(J,K)
     $             + A(I,J+1) * B(J+1,K) + A(I,J+2) * B(J+2,K)
     $             + A(I,J+3) * B(J+3,K)
110   CONTINUE
```

Making the change to the simplified loop:

```
      C = 0.0
      DO 110 J = 1, M
        DO 110 K = 1, N
          DO 110 I = 1, L
            C(I,K) = C(I,K) + A(I,J) * B(J,K)
110   CONTINUE
```

gets converted to a call to SGEMM.

This happens automatically via the compiler and you can quickly verify this by generating a listing (-r2) , by running the code (which now sits at 340.7 Mflops) or by looking for the entry point in the object file:

```
$ nm -g mxm.o
  mxm.o:
```

```
0 T MXM
 U SGEMMX@
```

The initialization of array **C** to zero occurs outside the triple loop which means that the substitution could probably be improved. By modifying the loop so that **C** is initialized within the triply nested loop, we obtain a library substitution that also initializes **C,** and performance improves to 381.0 Mflops.

```
      DO 110 K = 1, N
        DO I = 1, L
         C(I,K) = 0.
        ENDDO
      DO 110 J = 1, M
            DO 110 I = 1, L
               C(I,K) = C(I,K) + A(I,J) * B(J,K)
110   CONTINUE
```

Finally, we check address alignment of arrays **A**, **B**, and **C** which live together in the same common block. The "-apad" option has already been applied which ensures that arrays start a good multiple of 8 (an Scache line) apart, however we discover that the common block itself is not on an 8-word boundary. We can ensure that all common blocks start on an 8-word boundary with the "align" loader directive:

```
$ f90 -Wl"-Dallocate(alignsz)=64b" mxm.o
```

Final performance of the MXM kernel is 409.1 Mflops.

8.2 Kernel 2: CFFT2D

This kernel performs a series of complex ffts on a 2D matrix of size 128x256. Two subroutines are used to perform this operation (W1 and W2 are trigs arrays set up in the program for the ffts):

```
        PARAMETER( M = 128, N = 256, LDX = 128)
        COMPLEX X( M, N )
        ...
c perform forward ffts on columns
        CALL CFFT2D1 (1, M, LDX, N, X, W1, IP)
c perform forward ffts on rows
        CALL CFFT2D2 (1, M, LDX, N, X, W2, IP)
c perform inverse ffts on rows
        CALL CFFT2D2 (-1, M, LDX, N, X, W2, IP)
c perform inverse ffts on columns
        CALL CFFT2D1 (-1, M, LDX, N, X, W1, IP)
```

Again, we can turn to the math libraries to improve our performance. The routine **CCFFT** from libsci performs a single complex to complex fft on a vector of data. In order to process a series of columns (equivalent to **CFFT2D1** in this example), we will need to call **CCFFT** in a loop for each column of X.

Processing the ffts row-wise is more difficult because **CCFFT** only works with stride-1 data. We need to perform a transpose of **X** into a new array **XT**, and then process ffts again column-wise. The following code segment is equivalent:

```
      PARAMETER( M = 128, N = 256)
      COMPLEX X( M, N ), XT ( N, M )
      ...
c perform forward ffts on columns
      do ii = 1, n
       call ccfft(1,m,1.0,x(1,ii),x(1,ii),w1,work,0)
      enddo
c transpose the data
      do ii = 1, n
        do jj = 1, m
          xt(ii,jj)   =   x(jj,ii)
        enddo
      enddo
c perform forward ffts on rows (columns in xt)
      do ii = 1, m
       call ccfft(1,n,1.0,xt(1,ii),xt(1,ii),w2,work,0)
      enddo
c perform inverse ffts on rows (columns in xt)
      do ii = 1, m
       call ccfft(-1,n,1.0,xt(1,ii),xt(1,ii),w2,work,0)
      enddo
c transpose data back to x
      do jj = 1, m
        do ii = 1, n
          x(jj,ii)     = xt(ii,jj)
        enddo
      enddo
c perform inverse ffts on columns
      do ii = 1, n
       call ccfft(-1,m,1.0,x(1,ii),x(1,ii),w1,work,0)
      enddo
```

The performance of this kernel increases from 23.1 Mflops to 91.1 Mflops.

The 2D transpose routine is written in FORTRAN. A more efficient implementation can be constructed using E-registers to handle the strided data motion. This code:

```
  do ii = 1, n
    do jj = 1, m
      xt(ii,jj)   =   x(jj,ii)
    enddo
  enddo
```

Can be replaced with this:

```
  call cmplx_transpose(m,n,x,m1,xt,n)
```

An all FORTRAN version of the routine appears below:

```
      subroutine cmplx_transpose(n1,n2,a,lda,b,ldb)
      real*8 a(2,lda,n2)
      real*8 b(2,ldb,n1)
!dir$ unroll 4
      do j=1,n2
!dir$ cache_bypass b,a
        do i=1,n1
          b(1,j,i) = a(1,i,j)
          b(2,j,i) = a(2,i,j)
        enddo
      enddo
      return
      end
```

We make use of the cache_bypass feature of the CF90 3.0 compiler to perform this transpose using E-registers, which are less sensitive to bad strides. The cache_bypass directive does not currently support type complex, so we code using real arrays. The outer loop is unrolled by 4 in an attempt to use all 8 banks on data writes. Performance increases from 91 Mflops to 148 Mflops with the efficient transpose.

We can also use benchlib routines to accomplish the same task. The complex array to be transposed has a leading dimension which is a large power of two (common in ffts). For this reason, a straightforward E-register lgetv followed by an lputv is not optimal due to the stride. Instead, an algorithm is generated which gathers 4 columns of the source array and uses a call to lputv8 to write things out as 4 rows. The working routine is shown here.

```
subroutine cmplx_transpose(n1,n2,a,lda,b,ldb)
complex*16 a(lda,n2)
complex*16 b(ldb,n1)
integer*4 ind(0:479),itab(0:7)
do i=0,3
  itab(2*i+0)=0 +2*lda*i
  itab(2*i+1)=1 +2*lda*i
enddo
```

```
do i=0,480-1
  ind(i)=2*(i/8)+itab(and(i,7))
enddo
do j1=1,n1,(480/8)
  jlen=min((480/8),n1+1-j1)
  do i =1,n2,4
    call lgathh(a(j1,i),ind,  8*jlen)
    call lputv8(b(i,j1),2*ldb,  jlen)
  enddo
enddo
do while (lputp().ne.0)
end do
return
end
```

Overall performance of this kernel increases to 156.4 Mflops. The transpose itself is running at over 400 MBytes/second.

8.3 Kernel 3: CHOLSKY

This kernel performs a Cholesky decomposition and solve. Here, NMAT=250 is the number of independent systems, NRHS = 3 is the number of right-hand-sides. Since the loops count from zero, this means we have 250 independent systems, each with 4 right-hand-sides. Apprentice tells us that most of the time is spent in the forward-backsolve stage of the algorithm, which is currently written to vectorize over the number of systems:

```
      DO 6 I = 0, NRHS
        DO 7 K = 0, N
          DO 8 L = 0, NMAT
            B(I,L,K) = B(I,L,K) * A(L,0,K)
 8        CONTINUE
          DO 7 JJ = 1, MIN (M, N-K)
            DO 7 L = 0, NMAT
              B(I,L,K+JJ) = B(I,L,K+JJ) - A(L,-JJ,K+JJ) * B(I,L,K)
 7      CONTINUE
C
        DO 6 K = N, 0, -1
          DO 9 L = 0, NMAT
            B(I,L,K) = B(I,L,K) * A(L,0,K)
 9        CONTINUE
        DO 6 JJ = 1, MIN (M, K)
            DO 6 L = 0, NMAT
              B(I,L,K-JJ) = B(I,L,K-JJ) - A(L,-JJ,K) * B(I,L,K)
 6      CONTINUE
```

All the inner loops run over the number of systems, which are independent and offer little re-use. The loops are simple enough such that the hardware streams should be very effective, however. The B array is problematic, however, since the inner loops run over the second dimension. This is easily changed via the “flipper” utility, permuting the indices of the B matrix. This improves performance from 26.4 Mflops to 48.2 Mflops.

8.4 Kernel 4: BTRIX

Kernel BTRIX is a vectorized block tri-diagonal solver. According to Apprentice, there are several loop constructs in this kernel which take a significant percentage of the total time. We should point out that there is an outer-loop in this subroutine that runs over J and that the L index represent the independent systems. As such, most loops run over L as this allows for effective vectorization. We start by looking at the two largest time users:

```
      DO 100 J   = JS,JE
C    Original Code:
      DO 3 M = 1,5
        DO 3 N = 1,5
          DO 3 L = LS,LE
            B(M,N,J,L) = B(M,N,J,L) - A(M,1,J,L)*B(1,N,J-1,L)
     $           - A(M,2,J,L)*B(2,N,J-1,L) - A(M,3,J,L)*B(3,N,J-1,L)
     $           - A(M,4,J,L)*B(4,N,J-1,L) - A(M,5,J,L)*B(5,N,J-1,L)
    3 CONTINUE
      ...
      Other code omitted...
  100 CONTINUE
      DO 200 J = JEM1,JS,-1
        DO 200 M = 1,5
          DO 200 L = LS,LE
            S(J,K,L,M) = S(J,K,L,M) - B(M,1,J,L)*S(J+1,K,L,1)
     $           - B(M,2,J,L)*S(J+1,K,L,2) - B(M,3,J,L)*S(J+1,K,L,3)
     $           - B(M,4,J,L)*S(J+1,K,L,4) - B(M,5,J,L)*S(J+1,K,L,5)
  200 CONTINUE
```

Arrays A and B are 5 X 5 in the first two dimensions. We have a nice vector loop on L which is independent for each iteration. This is a plus for functional unit parallelism, but a minus in terms of cache re-use. The J loop provides some opportunity for temporal re-use. Firstly, we will attempt to keep the current loop structure, but permute the array indices so that L is in the first dimension and J is the last dimension (a 4123 permutation in flipper). We do this for the A, B, and C arrays (C is a similar array in another less important loop). The S array has a K index which is a constant passed into the routine so we move K to the last position and give S a 3412 permutation:

```
      DO 100 J   = JS,JE
C
      IF(J.EQ.JS) GO TO 4
```

```
      DO 3 M = 1,5
        DO 3 N = 1,5
          DO 3 L = LS,LE
            B(L,M,N,J) = B(L,M,N,J) - A(L,M,1,J)*B(L,1,N,J-1)
     $           - A(L,M,2,J)*B(L,2,N,J-1) - A(L,M,3,J)*B(L,3,N,J-1)
     $           - A(L,M,4,J)*B(L,4,N,J-1) - A(L,M,5,J)*B(L,5,N,J-1)
    3 CONTINUE
      ...
      Other code omitted.
  100 CONTINUE
      DO 200 J = JEM1,JS,-1
        DO 200 M = 1,5
          DO 200 L = LS,LE
            S(L,M,J,K) = S(L,M,J,K) - B(L,M,1,J)*S(L,1,J+1,K)
     $           - B(L,M,2,J)*S(L,2,J+1,K) - B(L,M,3,J)*S(L,3,J+1,K)
     $           - B(L,M,4,J)*S(L,4,J+1,K) - B(L,M,5,J)*S(L,5,J+1,K)
  200 CONTINUE
```

This increases the overall performance of BTRIX from 48 Mflops to 68 Mflops.

The other approach to take is to focus on the temporal re-use that is available on the J dimension. Currently, the J loop is the outermost loop of the routine which is probably not the optimal position to capture this potential cache reuse. Our second strategy is to push the vector loop L to the outermost loop position. There are several 1-dimensional vectors on L that can be demoted to scalars (not shown here) and we also perform the necessary array dimension permutations:

```
      DO 999 L = LS,LE
      DO 100 J   = JS,JE
      IF(J.EQ.JS) GO TO 4
C  31 % of the time
C  19 Mflops
c     DO 3 L = LS,LE
        DO 3 N = 1,5
          DO 3 M = 1,5
            B(M,N,J,L) = B(M,N,J,L) - A(M,1,J,L)*B(1,N,J-1,L)
     $           - A(M,2,J,L)*B(2,N,J-1,L) - A(M,3,J,L)*B(3,N,J-1,L)
     $           - A(M,4,J,L)*B(4,N,J-1,L) - A(M,5,J,L)*B(5,N,J-1,L)
    3 CONTINUE
C
   ... some code omitted.
   100 CONTINUE

c     DO 200 L = LS,LE
        DO 200 J = JEM1,JS,-1
```

```
            DO 200 M = 1,5
              S(M,J,L,K) = S(M,J,L,K) - B(M,1,J,L)*S(1,J+1,L,K)
     $             - B(M,2,J,L)*S(2,J+1,L,K) - B(M,3,J,L)*S(3,J+1,L,K)
     $             - B(M,4,J,L)*S(4,J+1,L,K) - B(M,5,J,L)*S(5,J+1,L,K)
  200 CONTINUE
  999 CONTINUE
```

Performance of this kernel increases to 81.4 Mflops.

8.5 Kernel 5: GMTRY

Kernel GMTRY performs Gaussian elimination in its most time-consuming loop. About 90% of the time is spent in Gaussian elimination of a 500x500 matrix:

```
C  GAUSS ELIMINATION
C
      DO 8 I = 1, MATDIM
        RMATRX(I,I) = 1. / RMATRX(I,I)
        DO 8 J = I+1, MATDIM
          RMATRX(J,I) = RMATRX(J,I) * RMATRX(I,I)
          DO 8 K = I+1, MATDIM
            RMATRX(J,K) = RMATRX(J,K) - RMATRX(J,I) * RMATRX(I,K)
8     CONTINUE
```

The inner two loops represent a rank-1 update, similar to the algorithm used in LINPACK without pivoting. Since the inner loop on K causes a strided reference pattern to RMATRX, a simple thing to do from FORTRAN is to interchange the J and K loops. Close examination of the listing file proves this is unnecessary, however, as CF90 3.0 has replaced the inner two loops with a call to SGER, a BLAS-2 rank-1 update routine. The manual replacement would look like this:

```
C  GAUSS ELIMINATION
C
      DO 8 I = 1, MATDIM
        RMATRX(I,I) = 1. / RMATRX(I,I)
      DO J = 1, MATDIM - I
         RMATRX(I+J,I) = RMATRX(I+J,I)*RMATRX(I,I)
      END DO
      CALL SGER (MATDIM-I, MATDIM-I, -1., RMATRX(I+1,I), 1,
     1   RMATRX(I,I+1), 500, RMATRX(I+1,I+1), 500)
    8 CONTINUE
```

This method for solving matrices is not ideal for a micro-processor. The blocked algorithms in LAPACK are much better suited for cache reuse. This kernel does no pivoting, so there are details to take care of outside of the solver since LAPACK will provided a decomposed matrix in pivoted row order. Alternatively, we can grab the source code for SGETRF and its subroutines from Netlib and modify it so it does no pivoting. If we replace the entire loop nest will a call to SGETRF, performance increases to 164.3 Mflops overall:

```
 CALL SGETRFNOPIV(MATDIM, MATDIM, RMATRX, MATDIM, IPIV, INFO)
```

SGETRF depends highly on SGEMM for performance. SGEMM performs best with Scache aligned matrices with leading dimensions which are a multiple of 8. The LDA used in the matrix multiplies inside SGETRF are the same as the LDA passed in the 4th parameter. In this problem, LDA=MATDIM is 500. If we bump that up to 504, and try again, overall performance increases to 172.3 Mflops.

There is a considerable amount of work in the intrinsic functions LOG and complex EXP. Checking the compiler listing shows that the -O3 option on the CF90 compiler has vectorized the inner-loop intrinsics so there is little else to be done.

8.6 Kernel 6: EMIT

According to apprentice, a majority of the time in the EMIT kernel is spent in intrinsic functions, particularly in ALOG. The most time consuming loop is shown below:

```
      COMPLEX DUM1, EXPZ(NVM), EXPMZ(NVM), WALL, EXPWKL, EXPMWK
      DO 6 K = 1, NWALL(L)
         EXPWKL = CEXP (WALL(K,L) * PIDP)
         EXPMWK = 1. / EXPWKL
         SPS = 0.
         DO 4 I = 1, NV
           DUM1 = EXPZ(I) * EXPMWK - EXPWKL * EXPMZ(I)
           PS(I) = GAMMA(I) * LOG (REAL(DUM1) ** 2 +
     &                AIMAG(DUM1) ** 2 + SIG2)
           SPS = SPS + PS(I)
4        CONTINUE
         PSI(K) = AIMAG(WALL(K,L) * CONJG (UUPSTR + CMPLX (0., U0)))
     &              - SPS * 0.25 / PI
6       CONTINUE
```

Compiler options are able to extract 246.4 Mflops from this kernel. The intrinsic function LOG in the inner loop is vectorized, and we have linked to the faster vector intrinsics. There is very little else to do here.

8.7 Kernel 7: VPENTA

The VPENTA kernel inverts 3 pentadiagonals simultaneously. There are two double-nested loops where most of the time is spent. For simplicity, we show only one here:

```
     PARAMETER (NJA=128, NJB=128, JL=1, JU=128, KL=1, KU=128)
     COMMON /ARRAYS/ A(NJA,NJB), B(NJA,NJB), C(NJA,NJB), D(NJA,NJB),
    $ E(NJA,NJB), F(NJA,NJB,3), X(NJA,NJB), Y(NJA,NJB)
      . . .
```

```
      DO 3 J = JL+2,JU-2
        DO 11 K = KL,KU
          RLD2 = A(J,K)
          RLD1 = B(J,K) - RLD2*X(J-2,K)
          RLD = C(J,K) - (RLD2*Y(J-2,K) + RLD1*X(J-1,K))
          RLDI = 1./RLD
         F(J,K,1) = (F(J,K,1) - RLD2*F(J-2,K,1) - RLD1*F(J-1,K,1))*RLDI
         F(J,K,2) = (F(J,K,2) - RLD2*F(J-2,K,2) - RLD1*F(J-1,K,2))*RLDI
         F(J,K,3) = (F(J,K,3) - RLD2*F(J-2,K,3) - RLD1*F(J-1,K,3))*RLDI
          X(J,K) = (D(J,K) - RLD1*Y(J-1,K))*RLDI
          Y(J,K) = E(J,K)*RLDI
11      CONTINUE
3     CONTINUE
```

As implemented for PVP systems, the inner loop on K is completely independent. Unfortunately, this also implies little re-use. Arrays A, B, C, D, E, X, Y, and F are all referenced in the inner loop on the second dimension, so all memory access patterns are strided references, in this case stride 128. Also note that all arrays are dimensioned 128x128 which means terrible cache alignment. This explains the dramatic improvement we saw with the -apad option in CF90 3.0.

Again we have a choice of inner loops. The K loop is independent but the J loop offers temporal re-use on the J dimension for arrays X, Y, and F. In this case, the vector loop is only 128 iterations long which means that we can hold several columns of F, X, and Y in the cache concurrently and take advantage of the re-use in the J loop with the existing loop ordering. For this reason, we elect to keep the current loop structure, but make the necessary loop index permutations:

```
      DO 3 J = JL+2,JU-2
        DO 11 K = KL,KU
          RLD2 = A(K,J)
          RLD1 = B(K,J) - RLD2*X(K,J-2)
          RLD = C(K,J) - (RLD2*Y(K,J-2) + RLD1*X(K,J-1))
          RLDI = 1./RLD
         F(1,K,J) = (F(1,K,J) - RLD2*F(1,K,J-2) - RLD1*F(1,K,J-1))*RLDI
         F(2,K,J) = (F(2,K,J) - RLD2*F(2,K,J-2) - RLD1*F(2,K,J-1))*RLDI
         F(3,K,J) = (F(3,K,J) - RLD2*F(3,K,J-2) - RLD1*F(3,K,J-1))*RLDI
          X(K,J) = (D(K,J) - RLD1*Y(K,J-1))*RLDI
          Y(K,J) = E(K,J)*RLDI
11      CONTINUE
3     CONTINUE
```

Performance has increased from 26.4 Mflops to 68.7 Mflops.

With our modified loop ordering, we have more than 6 streams. In particular, we have streams for arrays A, B, C, D, E, X, Y and F. We notice that X and Y are used together everywhere in the routine, so we fuse the two arrays as a new array XY with a leading dimension of 2. Performance increases to 74.4 Mflops. Finally, we also notice that arrays A, B, C, D and E are always used together so we fuse the 5 arrays into 1 (array AE) with leading dimension of 5:

```
      DO 3 J = JL+2,JU-2
        DO 11 K = KL,KU
          RLD2 = AE(1,K,J)
          RLD1 = AE(2,K,J) - RLD2*XY(1,K,J-2)
          RLD = AE(3,K,J) - (RLD2*XY(2,K,J-2) + RLD1*XY(1,K,J-1))
          RLDI = 1./RLD
         F(1,K,J) = (F(1,K,J) - RLD2*F(1,K,J-2) - RLD1*F(1,K,J-1))*RLDI
         F(2,K,J) = (F(2,K,J) - RLD2*F(2,K,J-2) - RLD1*F(2,K,J-1))*RLDI
         F(3,K,J) = (F(3,K,J) - RLD2*F(3,K,J-2) - RLD1*F(3,K,J-1))*RLDI
          XY(1,K,J) = (AE(4,K,J) - RLD1*XY(2,K,J-1))*RLDI
          XY(2,K,J) = AE(5,K,J)*RLDI
11      CONTINUE
3     CONTINUE
```

Performance improves to 88.8 Mflops. A final thing to try would be to work with the array sizes and padding so that we ensure that we take advantage of the temporal re-use in the J dimension. This was attempted, but we could not do better than the automatic array padding provided by the compiler.

8.8 Summary

The following table summarizes the improvements that were made for the 7 NAS Kernels.

Table 8: Modified NAS kernels

Kernel	Orig (Mflops)	Optimized (Mflops)
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
Total	41	125

9.0 Performance Tools

In this section, we describe some of the tools that can be used to analyze and improve single-PE performance of code on the Cray T3E.

9.1 Apprentice

Apprentice is very useful for initially determining where time is spent in a code. To compile a code with Apprentice, use the -Ta option when compiling, and the -l app option when loading:

```
% f90 -eA file.f
% mppldr -lapp file.o
% a.out
% apprentice app.rif
```

Using the point-and-click interface, the intensive loops in the program can be identified. Intrinsic functions show up separately, so a loop with a lot of SQRT calls in it may not show up as a big time user, but SQRT may show up as a separate routine.

If you select SQRT, and look at the call sites under the display menu, you can determine which loops called it.

The "observations" selection under the display menu is also useful as it will show Mflops for any individual section of code selected.

9.2 pat

The performance analysis tool (PAT) provides a low-overhead method for estimating the amount of time spent in functions, determining load balance across processing elements (PEs), generating and viewing trace files, timing individual calls to routines, and displaying hardware performance counter information. PAT uses the UNICOS/mk profile system call to periodically sample the program counter to generate an execution-time profile of a user's program and uses the EV5 performance counters to gather performance information and instruction counts.

You can use the PAT tool to analyze a C, C++, or Fortran 90 executable file by simply relinking the executable with the PAT library (libpat.a). It is not necessary to recompile.

9.3 Flipper

Flipper is a perl script available from the benchmarking group. It allows you to permute the indices of an array around quickly in a subroutine or function. This is often necessary in order to get stride-1 memory access on key inner loops or in reducing the number of memory reference streams. The syntax is:

Usage: flipper [-v] [-o order] [-i permute] variable filename

-v	verbose mode

-o order	the order of variable (default: 2)
-i permute	the permuted sequence of indices (default: 21)
variable	the name of the array whose indices you want swapped
filename	the name of the input source file

For example, to convert array A(M, N, 3, 3, 2) to A(2, 3, 3, M, N), we would invoke flipper with a 53412 permutation:

```
%  flipper -o 5 -i 53412 A file.f > newfile.f
```

Flipper will choke if there are references to A with fewer than the full number of indices. For example, in this case, it would not like:

```
DO I = 1, M*N*3*3*2
   A(I) = 0.0
ENDDO
```