# The Benchmarker's Guide for CRAY SV1 Systems

**Maynard Brandt, Jeff Brooks, Margaret Cahir, Tom Hewitt, Enrique Lopez-Pineda, Dick Sandness**

**Cray Inc.**

**How to use the features of the CRAY SV1 series computers from a high-level language to improve the performance of application code.**

## 1.0 Introduction

The CRAY SV1 is the latest shared-memory vector processing system from Cray Inc.. The system represents a blend of an old architecture (the CRAY Y-MP) with some new features such as dual vector pipes and vector data caches.

This paper is targeted at users who want to get the highest levels of performance out of their programs on the CRAY SV1 system. The hardware is described, and programming implications that fall out of the architecture and implementation are illustrated.

The programming examples presented here are written in FORTRAN, but the principles described also apply to C and C++.

## 2.0 Hardware Overview

The CRAY SV1 is significantly different from previous Cray vector machines in that it provides a cache for the data resulting from scalar, vector and instruction buffer memory references. Like its predecessors, the SV1 provides high bandwidth to memory for both unit and non-unit stride memory references.

The CRAY SV1 is configured with 4 to 32 CPUs. Each CPU has 2 add and 2 multiply functional units, allowing each CPU to deliver 4 floating point results per CPU clock cycle. With the 300 MHz CPU clock the peak floating point rate per CPU is 1.2 Gflop/s and 38.4 Gflop/s for the system.

The SV1 memory architecture is uniform access, shared central memory. Uniform memory access (UMA) means that the access time for any CPU memory reference to

any location in memory is the same. Another commonly used term for UMA is “flat” memory. Memory capacity for the system ranges from a minimum of 4 GBytes up to a maximum of 32 GBytes.

The CRAY SV1 has two module types, processor and memory. The system must be configured with eight memory modules and one to eight processor modules. Each processor module has four CPUs. A CRAY J90 processor module can be upgraded with a SV1 processor module and the CRAY J90 system can be configured with both processor module types, J90 or SV1.

## 2.1 The Processor

The CRAY SV1 CPU is a custom CMOS. The processor is implemented using two chip types, cpu and cache.

The cpu chip contains the vector and scalar units. Scalar registers, scalar functional units and the instruction buffers reside in the scalar unit while the vector unit contains vector registers and the vector functional units. As in previous Cray vector systems, the CRAY SV1 processor contains 8 vector (V) registers, 8 scalar (S) registers backed by 64 T registers, and 8 address (A) registers backed up by 64 B registers. A parallel job also has access to 8 shared B and 8 shared T registers which are used for low overhead data passing and synchronization between processors.

A vector functional unit contains two pipes each capable of producing a result every CPU clock cycle. This results in a peak rate for a functional unit of two results per clock cycle. The maximum vector length, or VL, is 64. The floating point functional units, add and multiply, combined deliver 4 results per CPU clock cycle and with the CPU clock rate of 300 MHz a peak floating point rate of 1.2 Gflop/s for the processor is achieved. In addition to the add and multiply units, the other vector functional units are reciprocal, integer add, shift, pop/parity/leading zero, bit matrix multiply and logical. The vector units are capable of full chaining and tailgating. Chaining is reading from a V register that is still being written to and tailgating is writing to a V register that is still being read from a prior vector instruction. Scalar floating point operations are executed using the vector functional units. This is different from the Cray J90 which has separate floating point functional units for scalar operations.

To move data between CPU registers and memory via the cache two data paths or ports are provided to the cache. In any given clock cycle two memory requests can be active and consist of two dual-port reads or one dual-port read and one dual-port write. If there are no read requests there can be only one write request active. As mentioned in section 2.0 the processor can access up to 32 Gbytes of memory but an application is limited to a 16 Gbyte address space.

Instructions are decoded by the scalar unit and when vector instructions are encountered they are dispatched to the vector unit which maintains an independent instruction queue. Barring instruction dependency the two units can execute independently.

There are 32 performance counters provided in 4 groups of 8 each. Only one group can be active at a time with software providing user access to the data. The groups are

labeled 0 thru 3 and the collection order based on how useful the performance information is to the user would be is 0, 3, 2 and 1. The hardware performance moniter is covered in more detail in section 6.1.

In addition to the CPU clock there is a system clock which runs at the rate of 100 MHz. When using the CPU instruction to return the count of clock ticks it should be noted that the tick count is generated by the system clock rate.

## 2.2 Cache

The SV1 cache size is 256 KBytes and is 4-way set associative, write allocate, write through with a least recently used (LRU) replacement strategy. The data resulting from vector, scalar and instruction buffer memory references is cached. The cache-line size is 8 bytes, or 1 word long.

The SV1 cache is located between the cpu and the interface to main memory. The interface between cache and the cpu consists of four 64-bit read data paths and two 64-bit write data paths. The same number and types of paths exist between the cache and the interface to memory.

The SV1 cache is 4-way set associative which means that the cache is organized into sets, where each set is composed of 4 separate ways or cache lines. A memory address is mapped in a manner that results in the data represented by that address being placed in one particular cache set. That is, in an N-way set associative cache, an expression such as “modulo (memory_address, cache_size/N)”, will indicate to which set in cache the address maps. Since memory is much larger than cache, each set has many memory addresses that map into it. The four ways allow up to four memory addresses to map into the same cache set. If it becomes necessary to map a new address into a fully allocated cache set, the way for the least recently used address will be used and its data will be overwritten. (Note: direct-mapped cache and fully associative cache may be viewed as special “end” cases of the general N-way set-associative cache, where for direct-mapped cache N equals 1 and for fully-associative cache N equals the number of cache lines.)

The write allocate attribute of the cache requires that the address and data for a cpu write request to memory be mapped and placed into the cache if the request generates a cache miss. Because SV1 has a write-through cache, any value that is updated in cache will always be written through to memory at the same time. This is in contrast to a write back cache where a value is updated in memory when there is no longer room for it in cache, i.e., the value will be written back during a read to another value, which makes reads more expensive. However, write through can mean that more writes to memory are made than are strictly necessary when data is updated several times while remaining cache-resident. The cache contains buffers capable of holding 384 outstanding references. This is sufficient for the cache to handle 6 vector references of stride 1 (384 references = 6 vectors * 64 references per vector).

The one-word cache line size is advantageous for non-unit vector strides as it doesn’t cause the overhead of unnecessary data traffic when referencing memory using larger cache line sizes. It has the disadvantage that a single scalar reference would not bring in

surrounding data, thus potentially inhibiting a scalar code to take advantage of spatial locality. For this reason, scalar references have a prefetch feature, whereby a scalar reference causes 8 words to be brought into cache. These 8 words are determined by addresses which match the reference except for the lower 3 bits. This causes an effect for scalar loads which is similar to having a cache line size of 8 words.

SV1 relies on software for cache coherency between processes that share memory. Cache is invalidated as part of the test-and-set instruction. The test-and-set instruction has been used for processor synchronization in previous generation Cray vector systems. Adding the cache invalidation feature to this instruction allows old Cray binaries to run in parallel on the SV1 with the data cache enabled. The SV1 system libraries for parallel processing have all been modified to invoke the test-and-set when it is necessary to invalidate cache. Therefore users should be able to port codes written for Autotasking, OpenMP, MPI, and PVM without modification for the purposes of cache coherency. SHMEM codes should also port without modification with one exception as noted below.

In the case of the shared-memory parallel programming models, Autotasking and OpenMP, a test-and-set is issued at the beginning and end of parallel regions, at parallel loop iterations and at critical regions (locks and guards). For MPI and PVM programs, there is no shared data from the user's view, and the libraries take care of managing consistency within themselves. Although it is better to maximize the granularity of parallel tasks and minimize synchronization on any architecture, for SV1 there is additional incentive because it is best to avoid cache invalidations.

The SV1 SHMEM library routines that perform cache invalidates are shmem_barrier, shmem_wait and shmem_udcflush. In order to avoid race conditions, a shmem_barrier or shmem_wait is typically issued before remotely updated data is used. This will take care of cache coherency considerations at the same time. Codes that were originally written for the T3D may need to be modified if they make use of the shmem_set_cache_inv and shmem_clear_cache_inv routines. These routines invalidated cache on the T3D and are no-ops on the T3E, but are not supported on the SV1 (i.e., you will get an unsatisfied external message when you try to load). These codes will need to be reworked, probably by replacing each shmem_set/clear_cache_inv call with a shmem_barrier or a shmem_udcflush.

When programs are run on the SV1, but data and instructions are cached by default. A user can turn off data or instruction caching through the use of the "cpu" command. For example:

```
$ /etc/cpu -m ecfoff a.out
```

Turns off cacheing of instruction buffer fetchs for the program (but leaves data caching on).

Performance counter group 0 provides cache hit data. Some characteristics of this data should be noted. Instruction buffer references that generate cache hits will be counted as such but the references will not be counted as memory references. This can indicate that a program is generating a higher cache hit rate for its data than is really the case. Use the

"cpu" command to disable instruction buffer caching to determine this effect on the performance data.

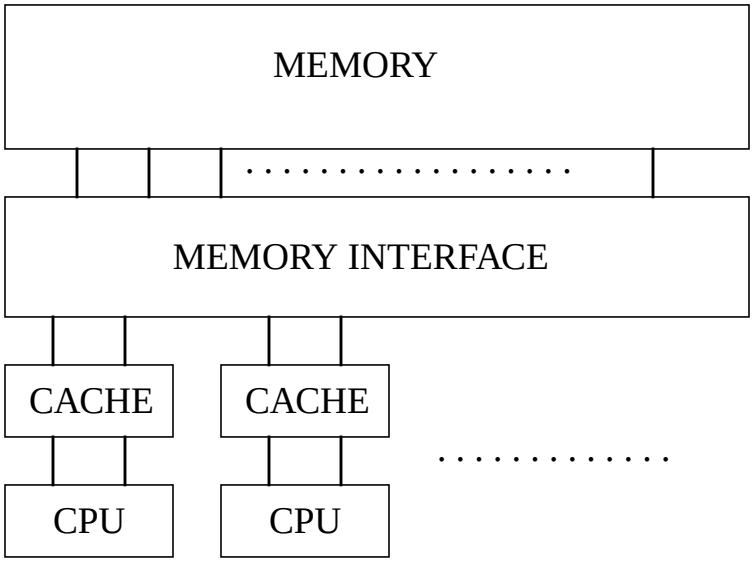
In Table 1 some typical processor latencies are listed.

**Table 1: Cray SV1 Processor Latencies**

| Operation | Time (300Mhz clocks) |
|---|---|
| V reg - cache | 25 |
| S reg - cache | 22 |
| V reg - memory | 109 |
| FP Add unit | 8 |
| FP Multiply unit | 9 |
| FP Reciprocal unit | 16 |
| Jump | 6 |

### 2.3 Peak and Measured Bandwidths

This section reviews the memory architecture and the peak and sustained performance rates between the CPU and Cache plus CPU and Memory. The following figure represents the relationship between CPUs, their caches and main memory.



Memory modules are the building blocks of the SV1 main memory, the type of which determines the memory density, the number of banks and the bank busy time. A single SV1 cabinet will have 8 memory modules, for 8x8 backplanes. A 4x4 backplane, with 4 memory modules, was available on J90 systems and can be upgraded with SV1 CPUs.

Central memory is divided into 8 sections and each processor module has an independent path into each memory section. The 4 CPUs and the I/O on a processor module share these eight paths. Each of the eight paths is capable of sending one request to memory per memory subsystem clock period (100 Mhz.) and receiving read data from memory at the same rate. The resulting theoretical peak bandwidth is 6.4 Gbytes/s per module. This compares to the measured STREAMs rate of 5 Gbytes/s for 4 processors on a common module. If the 4 processors are located on separate modules, then the measured STREAMs rate is 9.7 Gbytes/s, which is approximately 4 times the individual processor measured rate.

The memory modules can be one of two types which are named mem128 or mem512. The type number indicates the size of the module in millions(1024) of 64 bit words. Each memory section has eight subsections configured with eight banks plus eight pseudo banks. This bank configuration is also known as a pseudo bank pair. A bank is labeled pseudo because it shares a data path with its corresponding bank in the subsection. Every CPU data path or port, thru the cache and memory interface, has an independent path to each memory section.

The peak rate between a CPU and memory can be viewed from the perspective of the CPU's ability to generate requests, and from the memory's capability to satisfy requests. Since a CPU can generate 4 memory requests per *CPU clock* cycle, 4 read or 2 read and 2 write, the resulting peak CPU to cache rate is 9.6 Gbyte/s. If we assume cache is turned off the peak CPU to memory rate is 4 memory requests *per memory subsystem clock period* or 3.2 Gbyte/s. The maximum *sustained* bandwidth of a single processor to memory is 2.5 Gbyes/s as seen in the STREAM benchmark results.

From the memory's perspective, bank or pseudo bank pair can transfer 8 bytes of data every N system clock cycles. N is called the bank or pseudo bank pair busy time. The following formula calculates the peak rate achievable by a given memory configuration.

*Gbyte/s = number of banks / bank busy cycles * system clock * 8 *.001*

For a system configured with mem512 modules the memory rate would be:

*Gbyte/s = 512 / 6 *100 * 8 *.001 or 68.3*

A summary of the configuration characteristics and peak rates for SV1 memory module types is presented in Table 2.

**Table 2: SV1 Memory configuration and peak rates**

| | SV1 Configuration and Peak Rates | |
|---|---|---|
| Memory module type | mem128 | mem512 |
| Number of CPUs | 32 | 32 |
| Number of memory sections | 8 | 8 |
| Number of memory subsections | 8 | 8 |
| Number pseudo bank pairs | 512 | 512 |
| Pseudo bank pair busy clocks | 8 | 6 |
| Memory size Gbyte | 8 | 32 |
| CPU to cache rate, Gbyte/s | 9.6 | 9.6 |
| CPU to memory rate, Gbyte/s | 3.2 | 3.2 |
| Memory rate Gbyte/s | 51.2 | 68.3 |

The *sustainable* rate between the CPU and cache or memory is defined as the rate measured by a program whose performance limit is determined by this capability.

The sustainable rate for CPU reads from cache is 5.1 Gbyte/s. This rate is most likely to occur with vector reduction algorithms that fit in the cache.

To measure the sustainable rate between the CPU and memory the STREAM benchmark was used. Because of its memory reference patterns the performance of the benchmark is not affected by the cache. In Table 3 results for the benchmark are presented representing different processor counts as well as processor assignments to modules.

**Table 3: CPU to Memory rate measured by the STREAM SUM Test**

| Stream SUM test Gbyte/s between the CPUs and Memory | | | |
|---|---|---|---|
| Number | Number cpu | Memory Module Type | |
| CPUs | Modules | mem512 | mem128 |
| 1 | 1 | 2.52 | 2.51 |
| 2 | 1 | 4.71 | 4.68 |
| 2 | 2 | 5.00 | 4.97 |

**Table 3: CPU to Memory rate measured by the STREAM SUM Test**

| Stream SUM test Gbyte/s between the CPUs and Memory | | | |
|---|---|---|---|
| Number | Number cpu | Memory Module Type | |
| CPUs | Modules | mem512 | mem128 |
| 4 | 1 | 5.08 | 5.07 |
| 4 | 2 | 9.25 | 9.10 |
| 4 | 4 | 9.89 | 9.76 |
| 8 | 2 | 9.95 | 9.74 |
| 8 | 4 | 16.76 | 15.00 |
| 8 | 8 | 18.89 | 18.26 |
| 12 | 4 | 14.22 | 13.51 |
| 12 | 6 | 21.72 | 19.58 |
| 12 | 8 | 22.93 | 20.62 |
| 16 | 4 | 17.69 | 16.30 |
| 16 | 8 | 25.04 | 21.66 |
| 20 | 8 | 24.85 | 21.65 |
| 24 | 8 | 24.98 | 21.84 |
| 28 | 8 | 25.37 | 21.83 |
| 32 | 8 | 25.44 | 21.93 |

## 3.0 Single Processor Programming Implications

In this section, we will discuss performance programming implications for the Cray SV1 system.

### 3.1 Vectorization

Vectorized constructs perform up to a factor of 20X faster compared with non-vector constructs. On non-cached Cray vector systems, performance on vector constructs generally increased as a predictable function of vector length. This is not always the case on the Cray SV1, however, as long vectors can lead to a reduction in data cache efficiency. In general, it is better to used blocked algorithms (similar to those commonly used on

microprocessors) balancing the vector length against any potential data reuse that can be exploited via the data cache.

The Cray cf90 compiler will generate a loop mark listing showing which loops in a program vectorize or parallelize. This can be enabled with the *-rm* option under f90. Loops with a very large number of lines (100s) may require the *-Oaggress* optimization flag on the compiler as this allows for larger internal tables for compiler analysis.

Vectorization inhibitors within DO loops include:

- CALL statements
- I/O statements
- Backward branches
- Statement numbers with references from outside the loop
- References to character variables
- Non-vector external functions
- RETURN, STOP, or PAUSE statements
- dependencies

Many of these can be addressed through slight modifications of the source code.

### 3.1.1 Dependencies

Dependencies may be real or potential, depending upon data sets in many cases. For example, the compiler will not cleanly vectorize the following loop (NOTE: the compiler will mark this loop as “vectorized”, but the method used has extra overhead to check for repeated index values):

```
do i = 1, n
  A(index(i)) = A(index(i)) + b(i)
enddo
```

In this case, there is a potential dependency on the array **A**. A dependency exists if the index array has repeated values within a vector length (64 elements). If, for example, the programmer knows that **index**(i) is monotonically increasing, or **index**(i) is unique for each value of i, then no dependency exists and the programmer can assert so through a compiler directive:

```
!dir$ ivdep
      do i = 1, n
        A(index(i)) = A(index(i)) + b(i)
      enddo
```

Adding this directive to a “safe” loop can improve performance by up to a factor of 10.

The benchmarking group has also addressed four indirect update cases where an index list is used. A set of routines has been developed which will vectorize four flavors of indirect update loops by removing any vector dependency regarding the storing index. The routines will exceed the performance of Fortran 90 generated code provided the

storing index changes infrequently compared to the number of times the indirect update loops are executed. The four cases covered are:

- *ind_update.f* will vectorize the loop:

```
do j=1,N
   A( I( j ) ) = A( I( j ) ) + B( j )
enddo
```

- *ind_vpv.f* will vectorize the loop:

```
do j=1,N
   A( I( j ) ) = A( I( j ) ) + B( K( j ) )
enddo
```

- *ind_vpvu.f* will vectorize the loop:

```
do j=1,N
   A( I( j ) ) = A( I( j ) ) + B( j ) * C( K( j ) )
enddo
```

- *ind_vpvv.f* will vectorize the loop:

```
do j=1,N
   A( I( j ) ) = A( I( j ) ) + B( K( j ) ) * C( L( j ) )
enddo
```

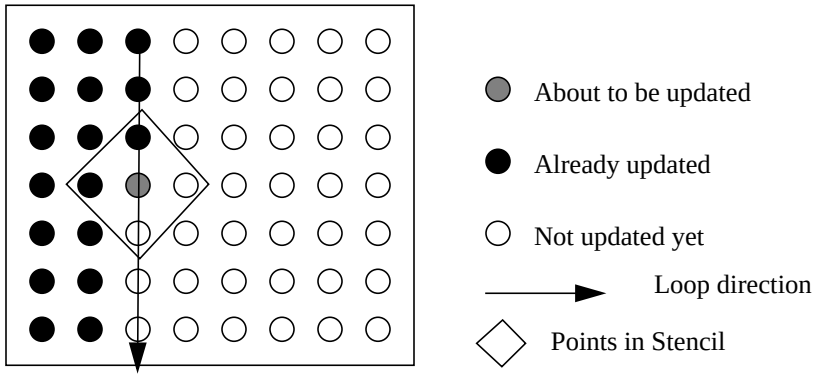
The above routines are available from the benchmarking group upon request.

In many cases, the dependency is real but can be programmed around with some loop restructuring. For example, in the following loop there is a vector dependency in both the inner and outer loops:

```
do j = 1, n
 do i = 1, m
 temp = .25*(x(i,j-1)+x(i-1,j)
*           + x(i+1,j)+x(i,j+1))-x(i,j)
 x(i,j) = x(i,j) + omega * temp
 if (abs(temp).gt.err1) err1=abs(temp)
 enddo
enddo
```

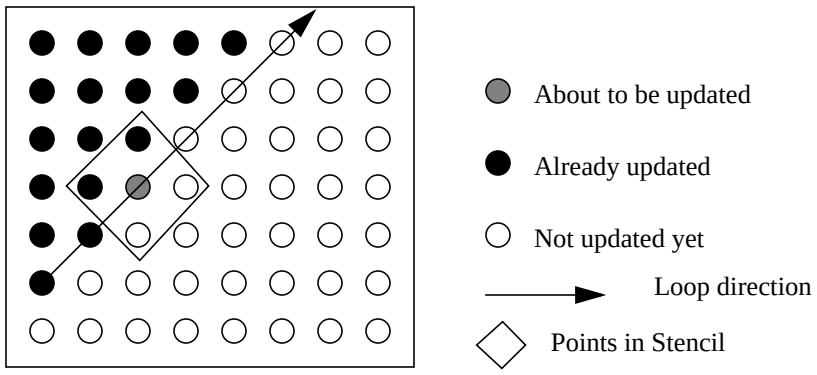
In this case, x(i,j) is defined using the north, south, east, and west neighbors. In the current form of the loop, x(i-1,j) is needed to update x(i,j) which creates a dependency in

the i direction. Switching the loops gives us a similar problem in the j direction. This loop runs at about 20 Mflop/s on the Cray SV1. The stencil and resulting dependency is shown in the following figure:




A solution to this problem is to change the vectors to run diagonally through the matrix X. This can be coded as follows:

```
        do jd=2,n+m
!dir$ ivdep
        do j=max(1,jd-m),min(n,jd-1)
        i = jd - j
        temp = .25*(x(i,j-1) + x(i-1,j)
     *           + x(i+1,j) + x(i,j+1)) - x(i,j)
        x(i,j) = x(i,j) + omega * temp
        if (abs(temp) .gt. err1) err1 = abs(temp)
        enddo
        enddo
```




Note that the leading dimension of the matrix should now be even because the stride is LDA-1. This loop now vectorizes and runs at over 260 Mflop/s on a 1000x1000 grid.

Cache hit rates run at about 50% because two of the four stencil points lie in the recently updated vector and hence are in cache.

## 3.2 The Data Cache

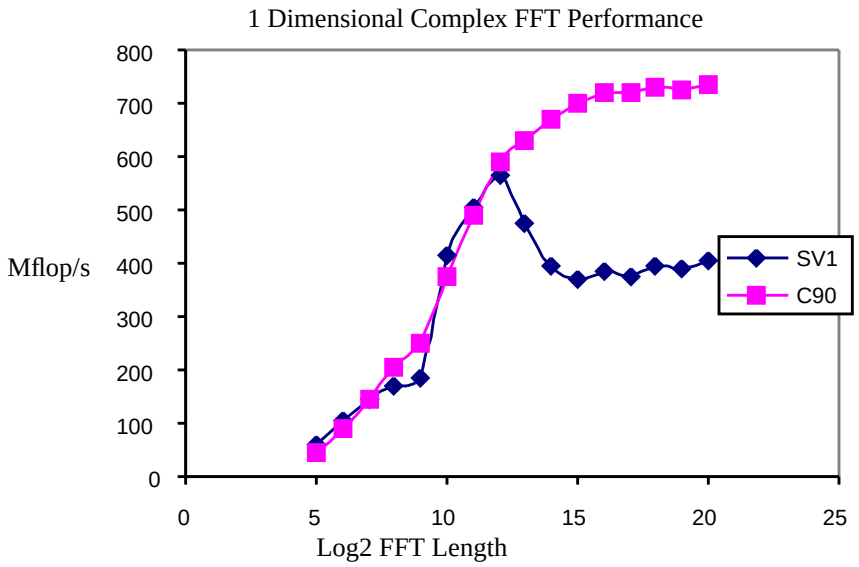
The most important departure of the SV1 from previous CRAY vector machines, is the addition of a data/instruction cache. The data cache can deliver operands to the functional units at a rate 2 to 3 times greater than the main memory and this can result in a 2 to 3 times faster code if the data cache is utilized well.

The SV1 cache differs from those associated with mainstream microprocessors in several important ways:

1. Cache line width: The cache-line width for vector references is a single word (8 Bytes). Unlike most microprocessors, contiguous memory references are not required in order to achieve full cache/memory bandwidth.
2. Bandwidth: The SV1 data cache has very high memory bandwidth (up to 4 words per clock period).
3. Size: 256KB is small by modern standards. You may need smaller blocking factors than on many microprocessor-based systems.
4. Write-through: This means that any memory stores go all the way to main memory. As system memory bandwidth is often a limiting resource it pays to minimize unnecessary data stores.
5. Non-Coherence: Due to the need to be binary compatible with the Cray J90, the Cray SV1 cache, does not maintain coherency with the other processors. This means that if another processor performs a store to memory, that the users cache does not obtain the new data value. We bear the cost when we perform multiprocessor synchronization, at which time the cache is invalidated. The implication is that better parallel performance is achieved with larger granularity.

Data in the SV1 cache is much closer to the processor than data in main memory. A scalar loop running with data already in the cache can run very much faster than if the data is in memory only. Scalar cache-misses load eight words, i.e. they act like the cache-line size was 8 words (64 Bytes). For this reason scalar stride one loops (for non-cache resident data) will run faster than general stride loops.

The data cache tends to increase performance when temporal locality exists in an algorithm. Consider a single-dimensional, complex fft of length N. Because an fft has Nlog2(N) floating point operations and only 2N data, significant temporal re-use exists. The following graph shows the performance of such an fft on the SV1 system (cached vector system) and a Cray C90 (non-cached vector system). As expected, the non-cached C90 runs faster with larger problems. The SV1 performance is fastest at N=4096 and then declines due to cache size.




In this case, longer ffts can be “blocked” using an alternative formulation such as the four-step method. In general, many of the same blocking techniques used with cache-based microprocessors will work well with the SV1 system.

### 3.3 Memory layout

The Cray SV1 is a real memory machine and programs occupy a contiguous space in memory. One benefit of this design is that programmers do not have to be concerned with the performance bottlenecks associated with the page tables of demand paged virtual memory systems. Random accesses to tables as large as the entire memory will perform well.

Like previous machines, the SV1 obtains high system memory bandwidth, by interleaving memory banks. The SV1 contains 512 pseudo-bank pairs. A long odd-strided memory reference will cycle through all of these banks. In addition, the cache is banked 8 ways. The main effect from the programmers standpoint is that memory/cache bandwidth is stride dependent. As a general rule:

Odd strides: These are optimal, any odd stride (positive or negative) should be as good as any other. All strides within a loop should match in order to minimize cache footprint conflict effects.

Even strides: Factor the stride, and look for powers of two If there is only a single factor of two, the access will only be slowed slightly (say 20%). If it contains a factor of four, the best you can get is half, speed. Multiples of eight run at 1/4 speed.

The following code fragment is from the X-Ray crystallography program SHELXL. As written here, it performs at 153 Mflop/s on the Cray SV1:

```
REAL A(N),B(N),C(N),D(N),E(N)
```

```
      M=(N/4)*4
      DO 1 I=1,M,4
      A(I)=A(I)+C(I)*D(I)
      B(I)=B(I)+C(I)*E(I)
      A(I+1)=A(I+1)+C(I+1)*D(I+1)
      B(I+1)=B(I+1)+C(I+1)*E(I+1)
      A(I+2)=A(I+2)+C(I+2)*D(I+2)
      B(I+2)=B(I+2)+C(I+2)*E(I+2)
      A(I+3)=A(I+3)+C(I+3)*D(I+3)
      B(I+3)=B(I+3)+C(I+3)*E(I+3)
1     CONTINUE
      IF(M+1.GT.N)GOTO 2
      A(M+1)=A(M+1)+C(M+1)*D(M+1)
      B(M+1)=B(M+1)+C(M+1)*E(M+1)
      IF(M+2.GT.N)GOTO 2
      A(M+2)=A(M+2)+C(M+2)*D(M+2)
      B(M+2)=B(M+2)+C(M+2)*E(M+2)
      IF(M+3.NE.N)GOTO 2
      A(N)=A(N)+C(N)*D(N)
      B(N)=B(N)+C(N)*E(N)
2     RETURN
      END
```

Upon close examination, we determine that this is a simple loop that has been unrolled by 4. This was a common technique to speed up programs on non-vector systems although today the unrolling task is better handled automatically by compilers. Unrolling the loop by 4 causes the following problems on the SV1:

- The loop still vectorizes, but we have introduced a stride of 4
- We have reduced the effective vector length from N to N/4

We simplify the code and get rid of these two problems by re-rolling the loop as follows:

```
      DO 1 I=1,M
        A(I)=A(I)+C(I)*D(I)
        B(I)=B(I)+C(I)*E(I)
1     CONTINUE
```

The resulting code improves to 357 Mflop/s.

### 3.4 Minimizing Stores

The SV1 cache policy is write-allocate and write-through. This means that any store will consume memory bandwidth and cache footprint. In many cases, stores can be reduced through unrolling and a technique called outer loop vectorization. We use the following matrix-vector multiply kernel to illustrate the techniques. A matrix vector

multiply of size N has 2*N^2 floating point operations and N^2 data. From an algorithm perspective, a minimum of one memory operation will be required for every two floating point operations giving us a maximum computational intensity of 2 (2 flops per memory operation).

To inhibit full compiler optimization, the loop is compiled as follows:

```
f90 -Onopattern,nointerchange -rm mxv.f
```

```
                subroutine mxv(a,lda,n,b,x)
                real a(lda,n), b(lda), x(lda)
1-----<         do j = 1, n
1 Vr--<           do i = 1, n
1 Vr              x(i) = x(i) + a(i,j)  *b(j)
1 Vr-->           enddo
1----->         enddo
                return
                end
```

From the listing file, we see that the inner loop is vectorized (V) and unrolled (r). Note that the unrolling here is on a vector chime basis (64-elements), not on a iteration by iteration basis as illustrated in the last section. For example, if the compiler tells us that a vector loop was unrolled by 2, it means two vector chimes (or 128 elements) are processed before loop iterations are incriminated.

From HPM, we are able to determine that this loop runs at 250 Mflop/s is requesting operands at the rate of 374 Mwords/s (3 words for every 2 flops, or a computational intensity of 2/3). Of this 374 Mwords/s, 250 Mwords/s is satisfied by main memory and the remaining 124 Mwords/s is satisfied from the data cache.

Since x(i) is updated for each pass of j, we can unroll the j loop into the inner loop and reduce the number of times x(i) is updated. For example, if we unroll by 4 times, we should reduce loads and stores to X by a factor of 4:

```
                subroutine mxv1(a,lda,n,b,x)
                real a(lda,n), b(lda), x(lda)
1----<          do j = 1, n, 4
1 V--<          do i = 1, n
1 V             x(i) = x(i) + a(i,j)  *b(j)
1 V         1             + a(i,j+1)*b(j)
1 V         1             + a(i,j+2)*b(j)
1 V         1             + a(i,j+3)*b(j)
1 V-->          enddo
1---->          enddo
                return
                end
```

Performance has improved from 250 Mflop/s to 301 Mflop/s. In addition, overall bandwidth consumed has dropped from 374 Mwords/s to 226 Mwords/s (38 Mwords/s from cache and 188 Mwords/s from main memory)

Another alternative is to switch the loop ordering and go to a dot-product formulation. This eliminates vector store traffic in the inner loop:

```
                subroutine mxv(a,lda,n,b,x)
                real a(lda,n), b(lda), x(lda)
1-----<         do i = 1, n
1       cdir$ prefer vector
1 Vr---<          do j = 1, n
1 Vr                x(i) = x(i) + a(i,j)  *b(j)
1 Vr--->          enddo
1----->         enddo
                return
                end
```

This formulation runs at 298 Mflop/s consuming 306 Mwords/s of total bandwidth (153 Mwords/s from cache and 153 Mwords/s from memory). In this case, loads to b(j) are cached and loads to a(i,j) are not. We have reached our algorithmic ideal ratio of 2 flops for every memory operation. Dot products, however, are not ideal on vector systems due to the final vector reduction operation where a vector register of operands is collapsed down to a single scalar value.

The ideal algorithm would hold 64-elements of X in a vector register until all updates are complete. The 64-elements of the completed vector X would be written to memory. This technique is called “outer-loop vectorization” this can be achieved by writing the loop in a dot-product formulation and then inserting a *cdir$ prefer vector* directive on the outer loop:

```
                subroutine mxv(a,lda,n,b,x)
                real a(lda,n), b(lda), x(lda)
          cdir$ prefer vector
V-----<         do i = 1, n
V r---<           do j = 1, n
V r               x(i) = x(i) + a(i,j)  *b(j)
V r--->           enddo
V----->         enddo
                return
                end
```

This loop now runs at 395 Mflop/s consuming only 201 Mwords/s of total bandwidth (199 Mwords/s from memory and 2 Mwords/s from cache). Re-use has moved from cache into a vector register. Note that in many cases the compiler will choose this formulation automatically.

The SV1 memory is capable of delivering operands at the rate of 2.5 GBytes/sec (312 Mwords/s). Since 2 flops are computed for every word of memory bandwidth, this algorithm has a peak potential rate of 624 Mflop/s (312 Mwords/s * 2) on the SV1. When coded in assembly language, vector loads to A can be very carefully scheduled to achieve maximum bandwidth:

```
      subroutine mxv(a,lda,n,b,x)
      real a(lda,n), b(lda), x(lda)
      CALL SGEMV ('n', N, N, 1., A(1,1), LDA,
     $            B(1), 1, 1., X(1), 1)
      return
      end
```

The libsci code runs this problem at 600 Mflop/s, consuming 305 Mwords/s of memory bandwidth (3 Mwords/s from cache and 302 Mwords/s from memory)

## 3.5 Choosing a Loop Ordering

When presented with a nest of loops, the compiler can make choices about which loop to vectorize and which loop to parallelize or multi-stream based on stride and vector length. Sometimes, incomplete information is presented to the compiler which can result in a sub-optimal choice. In addition, the compiler does not make loop order choices based on temporal locality considerations. Consider the following loop nest:

```
       real a(69,92,115)
        do k = 2,115
          do j = 2,92
             do i = 2,69
                c(i,j,k) = 2.5*(a(i,j,k)-a(i,j,k-1))
     $                    *(b(i,j,k)-b(i,j,k-1))
             enddo
          enddo
        enddo
```

In this case, the compiler vectorized the k loop because it has the largest iteration count. Because array A has an even middle dimension and because K is the last index, this results in an even-strided vectorized loop. The code runs at 109 Mflop/s.

```
 25.  V------<       do k = 2,115
 26.  V 2----<         do j = 2,92
 27.  V 2 3--<            do i = 2,69
 28.  V 2 3                  c(i,j,k)=
 29.  V 2 3-->            enddo
 30.  V 2---->         enddo
 31.  V------>       enddo
```

The j loop (2nd dimension) will give us the longest vector length that is not a multiple of 2 so it makes sense to choose it for the innermost loop. Next, we see that a temporal re-use opportunity exists on the K index because both K and K-1 are referenced for arrays A and B. For this reason we choose K to be the next loop in the nest. This leaves the i loop as the outermost. In addition, we can force the compiler to unroll the K loop into the J loop with an unroll directive which facilitates register re-use opportunities. Finally, we use the "prefer vector" directive to tell the compiler to vectorize the J loop. The resulting code looks like this:

```
      real a(69,92,115)
      do i = 2,69
!dir$ unroll(4)
        do k = 2,115
!dir$ prefer vector
           do j = 2,92
              c(i,j,k) = 2.5*(a(i,j,k)-a(i,j,k-1))
$                        *(b(i,j,k)-b(i,j,k-1))
           enddo
        enddo
      enddo
```

Performance improves to 247 Mflop/s.

A good general rule of thumb is to vectorize on the longest odd dimension, and then look for temporal re-use opportunities for the next level loop.

### 3.6 Fast Intrinsics:

The Cray math libraries provide a set of vectorized intrinsic functions that are accurate to the last ULP (Unit in the Least significant Place). In most cases, the majority of the time in these functions is spent ensuring that the last bit or two are correct. For many applications, accuracy to the last 2 bits is sufficient and a significant performance advantage can be realized. A set of faster intrinsics functions is available from the benchmarking group at Cray. The performance of these functions is shown in Table 4.

**Table 4:**

| Intrinsic Function Performance (300 Mhz clocks per result) | | |
|---|---|---|
| Function | libm.a | benchlib |
| ALOG | 16 | 5 |
| ATAN | 17 | 13 |

**Table 4:**

| Intrinsic Function Performance (300 Mhz clocks per result) | | |
|---|---|---|
| Function | libm.a | benchlib |
| COS | 19 | 7 |
| EXP | 12 | 5 |
| SIN | 19 | 7 |
| SQRT | 8 | 6 |

The following example is Livermore Loop number 22 which calls the EXP function:

```
fw = 1.0
do 22 k = 1, 101
  y(k) = u(k) / v(k)
  w(k) = x(k) / (exp(y(k)) - fw)
enddo
```

The default performance of this loop is 254 Mflop/s and improves to 449 Mflop/s with the libbnch intrinsics (this software is available upon request from the authors).

## 4.0 Parallel Programming

In this section, we discuss the various parallel programming models that are available on the Cray SV1.

### 4.1 Multi-Streaming Processor and Streaming

The Cray SV1 is the first system from Cray that features a Multi Streaming Processor (MSP). An MSP is composed of 4 processors, chosen in such a way that each processor is on a unique CPU module (if possible). In the hardware section, we pointed out that a single processor can sustain about 2.5 Gbytes/s of main memory bandwidth. This CPU placement allows an MSP to sustain four times this, or 10.0 Gbytes/s of main memory bandwidth.The processors are gang-scheduled by UNICOS, and are tied tightly to the requesting program.

Although it may seem that parallel processing on an MSP is similar to other shared-memory directive-based models (Autotasking, OpenMP), it is different in some key areas. For example, when Autotasking on 4 CPUs, a program will use all the processors only in parallel regions and only when they are available (there are idle CPUs in the system). An MSP, on the other hand, will lock 4 CPUs onto a job through both serial and parallel sections of a program, regardless of the amount of user code that is actually parallelized. Autotasking was designed so that CPU time would not be wasted. It would

allow a program to efficiently soak up any idle cycles that might be available. An MSP was designed for performance. Because 4 CPUs are gang-scheduled, lower overhead methods can be used to synchronize CPUs and hence finer granularity parallelism can be exploited.

A second component of a Multi-Streaming Processor is the streaming option in the compiler itself. To generate multi-streamed code for running on an MSP, use the following compiler option:

```
f90 -Ostream[0-3] file.f
```

The compiler will automatically stream inner vector loops, and parallel outer loops where available. The parallel code differs from Autotasking code as follows:

- Static scheduling. All parallel constructs are divided into 4 equal parts by the streaming compiler. Autotasking defaulted to guided scheduling for inner loops and to single-iteration scheduling for outer loops.
- Shared B registers are used for synchronization. This wasn't possible with Autotasking because gang-scheduling is required to make use of this hardware and this was generally at odds with the design goals of Autotasking.
- Inner vector loops are streamed by default. Under Autotasking, an extra option was required because this usually wasn't desirable.
- High level parallelism (across the subroutine level) cannot be exploited by the streaming compiler. Under Autotasking, user directives could be used to parallelize code across subroutine boundaries.

The loop-mark listing (*-rm*) will show which loops streamed.

Consider the following loop (Livermore Loop number 7):

```
DO 7 k = 1, 1000
     X(k) = U(k) + R*(Z(k)) + R*Y(k)) +
     T*(U(k+3) + R*(U(k+2) + R*U(k+1)) +
     T*(U(k+6) + Q*(U(k+5) + Q*U(k+4))))
CONTINUE
```

This loop runs at 665 Mflop/s on a single processor of an SV1. When this loop is streamed and run on an MSP, the first processor takes iterations 1-250, the second processor takes 251-500, and so on. The performance on an MSP of the streaming code is 1018 Mflop/s, only 1.5 times faster than the single processor.

The main reason for the disappointing performance is the software cache-coherence on the SV1. The Livermore loops are run with many repetitions and when multi-streamed the cache is invalidated each time the loop is completed. Because the cache-line width on the SV1 is only one word, this cache invalidation is usually not necessary. If we manually edit the assembler language and pull out the test and set instruction responsible for invalidating the cache, performance increases to 1542 Mflop/s. Unfortunately, understanding when it is safe to do this is beyond the scope of the compiler. The next-generation vector system from Cray will have hardware cache coherence so this invalidation step will go away.

Long-vector codes tend to stream well on the SV1, in part because they do not use the cache well to begin with and so the extra cache invalidations do not hurt. Codes which are dominated by nested loops can also stream well.

## 4.2 Autotasking/OpenMP

Autotasking and OpenMP are available on the Cray SV1. As mentioned above, the data caches on the SV1 are not hardware coherent. The SV1 is also upwardly compatible with the Cray J90. Parallel binaries from the J90 are able to run with the cache enabled on the Cray SV1 due to the design of the test-and-set instruction. On the J90, this instruction was used whenever it was necessary to synchronize processors in a parallel program. On the SV1, a side-effect of this instruction is to also invalidate the caches, allowing J90 binaries to run in parallel with cache enabled.

Because of this invalidation issue, the best performance can be obtained by synchronizing as little as possible. Consider the following example (from the Tomcatv SPEC 95 benchmark):

```
      DO     60    J = 2,511
        DO     50    I = 2,511
          XX = X(I+1,J)-X(I-1,J)
          YX = Y(I+1,J)-Y(I-1,J)
          XY = X(I,J+1)-X(I,J-1)
          YY = Y(I,J+1)-Y(I,J-1)
          A  = 0.25D0  * (XY*XY+YY*YY)
          B  = 0.25D0  * (XX*XX+YX*YX)
          C  = 0.125D0 * (XX*XY+YX*YY)
          AA(I,J) = -B
          DD(I,J) = B+B+A*REL
          PXX = X(I+1,J)-2.D0*X(I,J)+X(I-1,J)
          QXX = Y(I+1,J)-2.D0*Y(I,J)+Y(I-1,J)
          PYY = X(I,J+1)-2.D0*X(I,J)+X(I,J-1)
          QYY = Y(I,J+1)-2.D0*Y(I,J)+Y(I,J-1)
          PXY = X(I+1,J+1)-X(I+1,J-1)-X(I-1,J+1)+X(I-1,J-1)
          QXY = Y(I+1,J+1)-Y(I+1,J-1)-Y(I-1,J+1)+Y(I-1,J-1)
C
C     CALCULATE RESIDUALS
C
          RX(I,J)   = A*PXX+B*PYY-C*PXY
          RY(I,J)   = A*QXX+B*QYY-C*QXY
   50     CONTINUE
   60   CONTINUE
```

The inner loop consists of two 9-point stencils. Eight of the nine stencil points hit in the data cache provided that we can hold three columns of array X and Y in the data cache.

Note that the outer loop works across columns of these matrices. On a single processor, this code runs at 502 Mflop/s.

With default Autotasking, the outer loop is parallelized as follows:

```
105.  1 P------<         DO     60    J = 2,511
106.  1 P        C
107.  1 P V----<           DO     50    I = 2,511
108.  1 P V                  XX = X(I+1,J)-X(I-1,J)
109.  1 P V                  YX = Y(I+1,J)-Y(I-1,J)
110.  1 P V                  XY = X(I,J+1)-X(I,J-1)

            LOOP BODY OMMITTED

128.  1 P V      C
129.  1 P V---->    50     CONTINUE
130.  1 P------->   60   CONTINUE  <- invalidate here
```

On 4 CPUs, we see 1475 Mflop/s, or only a 2.9x speedup from Autotasking. This is because Autotasking hands out iterations one at a time to available processors, and issues the necessary test-and-set after every iteration. This all but assures that 3 columns of X and Y will NOT be held in cache by any available processor. In this case, parallel processing has destroyed the high level of cache re-use we enjoyed on a single processor.

Static scheduling provides a solution to this problem. If we break the iteration space up into 4 contiguous pieces (for 4 processors), we will only synchronize (and invalidate) once at the end of the parallel region. Each processor will handle a contiguous chunk of the outer iteration space and hence will likely have the required 3 columns of X and Y cached for performance. This can be accomplished with the *numchunks* scheduling directive in autotasking:

```
          !cmic$ do all autoscope numchunks(4)
105.  1 P------<         DO     60    J = 2,N-1
106.  1 P        C
107.  1 P V----<           DO     50    I = 2,N-1
108.  1 P V                  XX = X(I+1,J)-X(I-1,J)
109.  1 P V                  YX = Y(I+1,J)-Y(I-1,J)
110.  1 P V                  XY = X(I,J+1)-X(I,J-1)
            LOOP BODY OMMITTED
128.  1 P V      C
129.  1 P V---->    50     CONTINUE
130.  1 P------->   60   CONTINUE  <- invalidate once
```

Autotasking didn’t use static scheduling because gang scheduling is required to make it work well. Fortunately, we can run an autotasking binary on a gang-scheduled MSP by using the following command (the environment variable NCPUS has been set to 4):

```
/etc/cpu -a 1 a.out
```

This improves performance to 1850 Mflop/s, or a 3.7X improvement.

If we compile the kernel with the *-Ostream2* option, the outer loop is multi-streamed. This breaks up the outer loop into the same contiguous pieces as above, but with lower overhead streaming primitives:

```
105.  1 M------<         DO     60    J = 2,N-1
106.  1 M        C
107.  1 M V----<           DO     50    I = 2,N-1
108.  1 M V                  XX = X(I+1,J)-X(I-1,J)

         Loop body omitted…

129.  1 M V---->    50     CONTINUE
130.  1 M------>    60   CONTINUE
```

In this case, our speedup is almost perfect. The streamed loop nest runs at 2005 Mflop/s, or 3.99 times faster than the single processor code.

Keep the following rules of thumb in mind when writing code using Autotasking or OpenMP:

- Be aware than any parallel loop will generate cache invalidate instructions
- Push parallel regions as far outward as possible.
- Consider static scheduling by using the numchunks directive.
- Static scheduling works especially well on an MSP, due to the gang scheduling.

### 4.3 Message Passing

Message passing programs tend to exhibit coarse granularity parallelism, hence they can work very well on the SV1. In fact, our implementation of the linpack benchmark is written entirely in message passing.

The SV1 supports two modes of MPI. When running with *-nt* (number of tasks), the MPI library uses shared memory for communication. The *-np* option on mpirun will use the much slower TCP/IP communication. When communication latency is an issue, lower overhead shmem calls can be substituted for MPI sends and receives. Performance of the SV1 message passing software is shown in Table 5.

**Table 5: SV1 Message Passing Performance**

| Library | Latency (microsec.) | Bandwidth (Mbyte/sec) |
|---|---|---|
| mpi -np | 501 | 23 |
| mpi -nt | 75 | 793 |

**Table 5: SV1 Message Passing Performance**

| Library | Latency (microsec.) | Bandwidth (Mbyte/sec) |
|---|---|---|
| shmem | 2 | 2320 |

## 5.0 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 SV1 system, we get the results shown in Table 6 on a single processor.

**Table 6: NAS kernels with zero changes**

| PROGRAM | f90 -O1 Mflop/s | f90 -O2 Mflop/s | f90 -O3 Mflop/s |
|---|---|---|---|
| MXM | 500.2 | 697.2 | 590.6 |
| CFFT2D | 75.4 | 87.6 | 86.7 |
| CHOLSKY | 84.9 | 108.4 | 107.9 |
| BTRIX | 185.3 | 215.4 | 183.7 |
| GMTRY | 87.9 | 325.8 | 325.8 |
| EMIT | 442.2 | 493.6 | 491.9 |
| VPENTA | 57.8 | 63.2 | 62.2 |
| Total | 112.2 | 144.1 | 139.4 |

Simple compilation gives fair to good performance for the kernels, with the compiler doing a good job of doing library substitutions and loop optimizations in a few of the kernels. All of the performance improvement with simple compilation is achieved with the *-O2* flag, with some degradation occurring with the *-O3* flag. This degradation is due to the compiler's tasking optimization techniques. In particular, in optimizing code for parallel computation, the compiler will attempt to interchange loops as it deems necessary for improved parallelism. In some cases, this may adversely effect single CPU performance, as is clearly seen in the MXM and BTRIX kernels. One way to improve the performance when using *-O3* is to add the option: *-Onointerchange*. If maximum single CPU performance is the main goal, one really wants to just specify a minimum tasking level, with a maximum vectorization level. This is best accomplished by using: *-Ovector3*. This will default to a tasking level of 1 (*task1*) which will not optimize for parallel performance (but will interpret any tasking directives), and will allow the compiler to do loop interchange if it finds it important for vector optimization.

The following sections give a deeper analysis of each of the seven kernels. Analysis is followed using the *-O2* results. Parallel performance optimization will also be presented, aimed particularly at obtaining maximum advantage of the MSP, especially insuring outer-loop cache reuse.

### 5.1 Kernel 1: MXM

As implemented from FORTRAN, MXM (matrix multiply) is unrolled by 4 and is running at about 697 Mflop/s.

Matrix multiply exists as part of Cray's Scientific Library, so we could simply call the matrix multiply (SGEMM) routine itself. Furthermore, the CF90 3.0 compiler supports pattern recognition, so it normally recognizes and replaces this kernel automatically. However, 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
```

If we make 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
```

the compiler automatically converts it to a call to SGEMM.

This compiler substitution can be quickly verified by generating a listing (-r2), by running the code (which now sits at 961 Mflop/s) or by looking for the entry point in the object file:

```
$ nm -g mxm.o
  mxm.o:
          288 T MXM
              U SGEMMX@
```

Performance for the MXM kernel is also enhanced in parallel with the library substitution. The library routine SGEMM performs in parallel, running in 3.6 Glop/s when run with 4 CPUs. Furthermore, when the code is run using MSP (by using the /etc/cpu command as follows: /etc/cpu -a 1), the performance for this kernel reaches 3.7 Gflop/s on 1 MSP processor.

### 5.2 Kernel 2: CFFT2D

This kernel performs a series of complex Fast Fourier Transforms (FFTs) on a 2-dimensional 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)
```

Optimization for this kernel starts by re-dimensioning array **X**, in order to avoid memory bank conflicts caused by the dimensions of array **X**:

```
      PARAMETER( M = 128, N = 256, LDX = 128, M1 = 129)
      COMPLEX X( M1, N )
```

This eliminates bank conflicts and improves the performance of the code from 87 to 199 Mflop/s.

Second, we can again turn to the math libraries to improve our performance. The routine **CCFFT2D** from libsci performs a two-dimensional complex to complex fft on an array of data. This call is equivalent to the pair of calls made by the kernel (**CFFT2D1** and **CFFT2D2**). The calls then become:

```
      PARAMETER( M = 128, N = 256, LDX = 128, M1 = 129)
      COMPLEX X( M1, N )
      real table(100+2*(N+M)),work(512*N)

      ...
c perform forward ffts
      CALL CCFFT2D (1, M, N, 1.0, X, M1, X, M1, TABLE, WORK, 0)
c perform inverse
      CALL CCFFT2D (-1, M, N, 1.0, X, M1, X, M1, TABLE, WORK, 0)
```

This simple substitution, although requiring an extra workspace array, brings the overall single CPU performance of this kernel to 463 Mflop/s. This library routine also runs in parallel, giving 1.2 Gflop/s in performance when ran on 4 CPUs.

### 5.3 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 251 independent systems, each with 4 right-hand-sides. The performance analysis tools tell 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, all of which vectorize. 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 108.4 Mflop/s to 197.8 Mflop/s.

Furthermore, since the outer K loop is from 0 to 3, we can unwind this loop into the inner loops “fattening” them up a bit. We accomplish this by using f90 array syntax as follows:

```
        DO 7 K = 0, N
          DO 8 L = 0, NMAT
            B(L,0:3,K) = B(L,0:3,K) * A(L,0,K)
 8        CONTINUE
          DO 7 JJ = 1, MIN (M, N-K)
            DO 7 L = 0, NMAT
              B(L,0:3,K+JJ) = B(L,0:3,K+JJ) - A(L,-JJ,K+JJ) * B(L,0:3,K)
 7      CONTINUE
C
        DO 6 K = N, 0, -1
```

```
        DO 9 L = 0, NMAT
          B(L,0:3,K) = B(L,0:3,K) * A(L,0,K)
 9      CONTINUE
      DO 6 JJ = 1, MIN (M, K)
          DO 6 L = 0, NMAT
            B(L,0:3,K-JJ) = B(L,0:3,K-JJ) - A(L,-JJ,K) * B(L,0:3,K)
 6    CONTINUE
```

This brings the overall single CPU performance to 242 Mflop/s.

In order to run this kernel in parallel, we want to strip-mine the **NMAT** loops in order to get vector loops working over one-quarter of the iteration space. In doing so, we are able to distribute work to 4 CPUs with a strip-mined vector length of 63:

```
      parameter(ncpus=4)
      lstride = (nmat + 1 + ncpus -1)/ncpus
!mic$ do all autoscope
      do ll = 0,nmat,lstride
      ltop = min(nmat, ll+lstride-1)
      DO 1 J = 0, N
        I0 = MAX ( -M, -J )
        DO 2 I = I0, -1
          DO 3 JJ = I0 - I, -1
            DO 3 L = ll, ltop
3           A(L,I,J) = A(L,I,J) - A(L,JJ,I+J) * A(L,I+JJ,J)

          DO 2 L = ll, ltop
2           A(L,I,J) = A(L,I,J) * A(L,0,I+J)
        DO 4 L = ll, ltop
4         EPSS(L) = EPS * A(L,0,J)
        DO 5 JJ = I0, -1
          DO 5 L = ll, ltop
5           A(L,0,J) = A(L,0,J) - A(L,JJ,J) ** 2
        DO 1 L = ll, ltop
1         A(L,0,J) = 1. / SQRT ( ABS (EPSS(L) + A(L,0,J)) )

        DO 7 K = 0, N
          DO 8 L = ll, ltop
8           B(L,0:3,K) = B(L,0:3,K) * A(L,0,K)
          DO 7 JJ = 1, MIN (M, N-K)
            DO 7 L = ll, ltop
7             B(L,0:3,K+JJ) = B(L,0:3,K+JJ)-A(L,-JJ,K+JJ)*B(L,0:3,K)
C
        DO 6 K = N, 0, -1
```

```
          DO 9 L = ll, ltop
9           B(L,0:3,K) = B(L,0:3,K) * A(L,0,K)
          DO 6 JJ = 1, MIN (M, K)
            DO 6 L = ll, ltop
6             B(L,0:3,K-JJ) = B(L,0:3,K-JJ) - A(L,-JJ,K) * B(L,0:3,K)
        enddo
```

This parallel version brings the MSP/tasking performance to 618 Mflop/s.

### 5.4 Kernel 4: BTRIX

Kernel BTRIX is a vectorized block tri-diagonal solver. According to the performance tools, 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. We 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
```

Matching strides for all arrays minimizes the cache interference between these arrays and increases the overall performance of this kernel to 248 Mflop/s. Finally, we attempt to “fatten up” the inner loops by unrolling the small loops of length 5 into the inner loop as follows:

```
      DO 100 J   = JS,JE
C
      IF(J.EQ.JS) GO TO 4
          DO 3 L = LS,LE
cdir$ unroll(5)
      DO 3 M = 1,5
cdir$ unroll(5)
        DO 3 N = 1,5
            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 L = LS,LE
cdir$ unroll(5)
        DO 200 M = 1,5
            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
```

By unwinding the short loops, these statements become part of longer vector loops, bringing the single CPU performance of this kernel to 260 Mflop/s.

For parallel performance, we looked at the way the routine is being called:

```
        DO 120 K = 1, KD
          CALL COPY (NB, BX, B)
          CALL BTRIX (JS, JE, LS, LE, K, B)
120   CONTINUE
```

and notice that each call the values for **K** are independent. We call the routine in parallel, maximizing granularity:

```
!mic$ do all autoscope private(k,b) shared(js,je,ls,le,bx) numchunks(4)
        DO 120 K = 1, KD
          CALL COPY (NB, BX, B)
          CALL BTRIX (JS, JE, LS, LE, K, B)
120   CONTINUE
```

When running with 4 CPUs, this kernel now gives a performance of 924 Mflop/s.

### 5.5 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 can also be done using the **SGETRF** routine from LAPACK. **SGETRF** uses a block algorithm which is much better suited for cache reuse. However, our particular kernel does no pivoting, while the LAPACK routine provides a decomposed matrix in pivoted row order. Hence, it is best to rewrite the above construct using a similar blocking algorithm for maximum efficiency. We implement a block algorithm as follows (courtesy of Ed Anderson):

```
      PARAMETER (NW=100, NB=5, LDR=NW*NB)
      REAL               ONE, ZERO
      PARAMETER          ( ONE = 1.0E+0, ZERO = 0.0E+0 )
         ...
C  GAUSS ELIMINATION
C
      MB = 64
      DO II = 1, MATDIM-MB+1, MB
         I2 = MIN(MATDIM,II+MB-1)
         IB = I2-II+1
         DO I = II, I2
            RMATRX(I,I) = 1. / RMATRX(I,I)
            DO J = I+1, MATDIM
               RMATRX(J,I) = RMATRX(J,I) * RMATRX(I,I)
            END DO
            IF( I2-I.GT.0 )
     &      CALL SGER( MATDIM-I, I2-I, -ONE, RMATRX(I+1,I), 1,
     &         RMATRX(I,I+1), LDR, RMATRX(I+1,I+1), LDR )
         END DO

         IF( II+IB.LE.MATDIM ) THEN
            CALL STRSM( 'Left', 'Lower', 'NoTranspose', 'Unit', IB,
     &                  MATDIM-II-IB+1, ONE, RMATRX(II,II), LDR,
     &                  RMATRX(II,II+IB), LDR )
            CALL SGEMM( 'NoTranspose', 'NoTranspose', MATDIM-II-IB+1,
     &                  MATDIM-II-IB+1, IB, -ONE, RMATRX(II+IB,II),
     &                  LDR, RMATRX(II,II+IB), LDR, ONE,
     &                  RMATRX(II+IB,II+IB), LDR )
```

```
         END IF
       END DO
```

With this construct, single CPU performance increases to 646 Mflop/s overall.

Furthermore, there is a considerable amount of work in the intrinsic functions LOG and complex EXP. We can take advantage of special libraries aimed at performing faster than the library-supplied routines for these functions. By linking with the *benchlib* routines, we improve the performance of this kernel to 721 Mflop/s. When this code is run on 4 CPUs, the resulting parallel performance is 1.7 Gflop/s.

## 5.6 Kernel 6: EMIT

According to the performance tools, 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
```

Although the standard libraries give a respectable 494 Mflop/s, if we link with the *benchlib* routines, we are able to improve the performance of this kernel to 835 Mflop/s overall. Running this code in parallel on 4 CPUs gives a performance of 1.7 Gflop/s.

This parallel performance can be improved by dividing the parallel work from the **DO 6** loop into 4 chunks (i.e. static scheduling):

```
!mic$ do all autoscope numchunks(4)
        DO 6 K = 1, NWALL(L)
```

This improves the 4 CPU performance to 2.5 Gflop/s.

### 5.7 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 here, the inner loop on K is completely independent. 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 memory conflicts. Thus, by re-dimensioning the leading dimension (NJA) of the arrays to 129, we eliminate such conflicts and improve the performance of this kernel from 62 Mflop/s to 285 Mflop/s.

Finally, for parallel optimization, we introduce a strip-mine loop based on the inner loops (which have a vector length of 128). As with CHOLSKY, this maximizes granularity and minimizes synchronization. The code, for our 4 processor case study looks as follows (all inner loops, not shown below, are now of length 32):

```
      J = JL
      kstride = (kku-kkl+1 + ncpu-1)/ncpu
!mic$ do all shared (A, B, C, D, E, F, JL, JU, KKL, KKU)
!mic$*       shared (KSTRIDE, X, Y)
!mic$*       private (J, JX, K, KK, KL, KU, RLD, RLD1, RLD2, RLDI)
      do kk = kkl,kku,kstride
      kl = kk
      ku = min(kku,kk+kstride-1)
```

(rest of code omitted)

The resulting parallel performance for this kernel is now 640 Mflop/s.

### 5.8 Summary

Table 7 summarizes the improvements that were made for the 7 NAS Kernels. For comparison purposes, we have included the single processor optimized Cray T90 results. Using tasking plus the MSP, the SV1 averages 1.6 Gflop/s across these kernels verses about 1.1 Gflop/s for a single-processor T90.

**Table 7: Modified NAS kernels**

| Kernel | SV1 (1cpu) Original (Mflop/s) | SV1 (1cpu) Optimized (Mflop/s) | T-90 (1cpu) Optimized (Mflop/s) | SV1 (1 MSP) Optimized (Mflop/s) |
|---|---|---|---|---|
| MXM | 697 | 961 | 1574 | 3747 |
| CFFT2D | 88 | 463 | 1339 | 1220 |
| CHOLSKY | 108 | 242 | 674 | 618 |
| BTRIX | 215 | 260 | 430 | 924 |
| GMTRY | 326 | 711 | 1195 | 1699 |
| EMIT | 494 | 835 | 1369 | 2517 |
| VPENTA | 63 | 285 | 1086 | 640 |
| Average | 284 | 537 | 1096 | 1624 |

## 6.0 Performance Tools

In this section, we describe some of the tools that can be used to analyze and improve performance of code on the Cray SV1.

### 6.1 Hardware Performance Monitor (hpm)

The hardware supports 32 counters in 4 groups of 8 each. The groups are labeled 0, 1, 2, and 4 with group 0 usually selected by UNICOS as the default.

Group 0 provides floating point operation and memory reference data. The data represents both scalar and vector operations. To determine the vector and scalar components for the floating point data collect group 3 data. Also, group 3 provides data for the other vector functional units. Note that a divide in FORTRAN will result in a reciprocal approximation operation and 3 multiplies. If a program is doing a large number of divides the multiply data should be adjusted accordingly. The group 0 cache data is a count of the read requests which result in cache hits. Write requests which result in cache hits are not counted. Instruction buffer memory references which result in cache hits are counted but they are not counted as memory references. Because of this, programs generating a large amount of instruction buffer references might appear to have a much higher cache hit rate for data references than is really the case. One way to check

this is to multiply the instruction buffer reference count by 32 and add this value to the memory reference count. Using the new memory reference count calculate the cache hit ratio and compare it to the original. A second way is to use the cpu command to turn of instruction buffer caching off and run the program to see the effect on the cache hit ratio. A command example is: *cpu -m ecfoff a.out.*

Group 2 data provides the detailed information about the cpu memory references reported in group 0. CPU read and write data is reported along with scalar and vector data. Also, a memory conflict count is reported. A large conflict count could indicate a large power of 2 stride or bank conflicts with vector gather / scatter instructions.

Group 1 data rarely provides any useful data to the application user or developer.

## 6.2 Code Profiling: profview

This utility processes the data generated by the ***prof*** command, and reports information about the execution of each program module. This information identifies the segments of the program which are using the most time, helping the user in focusing optimization efforts on these particular areas of the code. Information processed is based on grouping of addresses of instructions executed, allowing for detailed analysis of time-consuming routines, even at the loop level. Particularly useful information from ***prof*** is the display of library entry points (data which is not provided by either **Perftrace** or **Flowtrace**), as this can help the user in identifying heavy usage of intrinsic routines; routines which could have a faster version available (for example, in *benchlib*; see section 3.6).

Usage of this tool is as follows:

```
f90 -G1 -lprof -o prog.exe prog.F
prog.exe
prof -x prog.exe > prog.prof
profview prog.prof
```

Although the compilation with debugging turned on (*-G1*) is not necessary, it is helpful in identifying code segments within a program module. In the above example, the Fortran program *prof.F* is compiled and linked to the profiling library. Execution of the program *prog.exe* produces a profiling work-file, named *prof.data*. This work-file is combined with the executable by the ***prof*** utility, storing the output to be used for interactive execution by ***profview.*** The information presented by ***profview*** can be displayed graphically or in text line mode. Time-consuming portions of the code are readily identified by module name and symbol name (if debugging is enabled) as displayed or listed. Each module is presented with percentage of activity based on counts for instructions as executed.

This tool can also be used to identify potential inlining candidates with very little overhead (as compared to ***perfview*** and ***flowview***). This is done by analyzing the entry points and starting points of a routine. If a particular routine shows a high percentage of activity (hit counts) in the entry point and the start of the executable code for the routine (marked as *E.name* and *P.name* respectively), then this routine could be a good candidate for inlining.

Furthermore, ***profview*** is capable of capturing interactively the statistics of a running process (if loaded with the *libprof.a* library). This capability allows for analysis to be done without waiting for the code to finish. This is done by providing the process ID number (PID) at the graphical interface *Capture Running Process* file menu option, or by issuing the following:

```
prof -p PID a.out > prof.x ; profview prof.x
```

## 6.3 Flowtracing: flowview

This tool is used to obtain information on program flow, number of calls per subroutine, total time, and time per subroutine. It is useful in identifying subroutines which are called excessively and identifying the calling tree for the program.

The ***flowview*** tool processes **Flowtrace** information to display timings and other information about procedure calls such as inlining factor, number of calls, and average time per call. **Flowtracing** can be done for an entire code as follows:

```
f90 -ef -o prog.exe prog.F
prog.exe
flowview
```

Execution of *prog.exe* produces a file called *flow.data* which is used by ***flowview*** to present the information graphically or in text line mode.

**Flowtracing** can be done at the routine level by including the directive:

```
!DIR$ FLOW
```

Also, one can use **Flowtracing** on blocks of code using the **FLOWMARK** subroutine. By surrounding the particular program block of interest with calls to **FLOWMARK** **Flowtrace** will treat it as if it were a subroutine.

***Flowview*** can also be used to capture a running process, which becomes particularly important if a particular routine is called often (which may indicate that it is a good candidate for inlining), as this will slow down the execution of the program considerably. In order to capture a running process, one can do it from the graphical interface *Capture Running Process* file menu option or by using the ***flodump*** command as follows (PID is the process ID number, obtained by issuing a *as* command):

```
flodump -p PID -e | flowview
```

***Flodump*** can also be used to produce **Flowtrace** output from an abnormally terminated program.

## 6.4 Flowtracing with performance counters: perfview

The ***perfview*** utility reports hardware performance statistics of the execution of a program as gathered by **Perftrace**. This information uses the hardware performance monitor (*hpm*) on individual routines within the program, as well as **Flowtrace** information.

This information includes, but is not limited to, time, Mflop/s, cache use, number of calls, and inline factor.

This tool is important in identifying how well a particular routine (or code segment) is performing, regardless of time spent executing. **Perftrace** works by using the **Flowtrace** compilation and linking to the *perf* library as follows:

```
f90 -ef -lperf -o prog.exe prog.F
prog.exe
perfview
```

After execution of *prog.exe*, a file **perf.data** is created which is used by ***perfview*** to present the information graphically, or in text line mode.

As with **Flowtrace**, Perftrace allows for selective analysis of specific blocks of code by using the **!DIR$ FLOW** compiler directive or by using calls to **FLOWMARK**.

Information can be accumulated for the various *hpm* counters by using the environment variable **PERF_GROUP**, thus allowing a complete analysis by perfview as follows:

```
env PERF_GROUP=0 PERF_DATA=group0.raw prog.exe
env PERF_GROUP=1 PERF_DATA=group1.raw prog.exe
env PERF_GROUP=2 PERF_DATA=group2.raw prog.exe
env PERF_GROUP=3 PERF_DATA=group3.raw prog.exe
cat group*raw > perf.total
perfview perf.total
```

Perfview can also be used to capture a running process; this is done by providing the process ID number (PID) at the graphical interface *Capture Running Process* file menu option, or by using the underlying command *perfdmp:*

```
perfdmp -p PID -e | perfview
```

***Perfdmp*** can also be used to produce **Perftrace** output from an abnormally terminated program.

## 6.5 Makefile Generator: fmaker

A good makefile is essential for optimization activities because it allows for easy incremental builds and compiler option manipulation on a routine by routine basis. ***Fmaker*** is a csh script available from the benchmarking group, which is used to split out all of the Fortran subroutines from a file and produce a *makefile* to use for compiling and loading. ***Fmaker*** will change the case of the files split to lowercase.

This tool produces a very simplistic *makefile* without many comments, and very explicit targets and rules. The resulting *makefile* is meant to assist the user in simplifying compi-

lation when working on optimizing a program. The user will need to edit the *makefile* in order to obtain the necessary compiler options and link to the proper libraries as needed.

The syntax for this utility is as follows:

Usage: fmaker [-m makefile] [-o command] [-s] [-b] [-c] [-r] [-help] files

-m makefile makefile is the name of the makefile generated by the fmaker command (Makefile by default).

-o command command is where the resulting executable is placed (a.out by default).

-s strip flag (passed directly to the fsplit(1) program) will strip columns 73+ and all trailing blanks.

-b indicates that the SHELL makefile variable should be set to the Bourne shell (/bin/sh).

-c indicates that the SHELL makefile variable should be set to the C shell (/bin/sh).

-r retain the case of the split files (as fsplit does).

-help produces help screen.

## 6.6 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 not work 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
```