

# Performance Tuning for the Cray SV1

James Giuliani and David Robertson  
*Science & Technology Support Group*  
*Ohio Supercomputer Center*  
*1224 Kinnear Road*  
*Columbus, OH 43212-1163*  
*USA*

## Abstract

The introduction of cached vector operations and Multi-Stream (MSP) processors in Cray's SV1 architecture will result in new design and performance tuning issues for developers of many scientific applications. In particular, code developed for Y-MP/C90/T90 series machines may require a significant additional tuning effort to achieve efficient performance on the SV1. Following an overview of the relevant SV1 architectural features and their theoretical performance implications, we describe several real-world research applications, which were profiled and tuned on both Cray T90 and SV1 systems at OSC. We analyze performance results in detail, highlighting tuning techniques that prove beneficial on the SV1. Finally, we discuss the insights gained into the process of migrating applications to Cray's new architecture roadmap.

## 1.0 Introduction

It has long been a cornerstone of the Cray philosophy that there is no real substitute for memory (and I/O) bandwidth. Traditional Cray vector machines, such as the Y-MP and T90, achieved very high performance by providing memory sub-systems capable of supplying the CPU with operands essentially as fast as it could use them. However, the cost of such memory has not decreased as rapidly as that of providing additional processing power. This mismatch long ago led developers of microprocessor-based systems - for whom price/performance is relatively more important than performance *per se* - to introduce a hierarchy of memories, with comparatively small, high-speed caches to hide the latencies to a larger main memory.

The SV1 is the first generation of Cray's Scalable Vector (SV) architecture, and represents a move in this same direction. It is a blending of the traditional Cray vector architecture (e.g., segmented functional units and vector registers) with a hierarchical, cache-based memory structure to reduce cost. In addition, it places significant emphasis on parallel vector (or "scalable vector") applications to attain high throughput. Where the Y-MP and T90 systems achieved high performance through hardware innovations, the SV1 relies on a balance of hardware and software.

In this paper we explore some of the implications of these new architectural features for application

developers interested in high performance. In particular, we have in mind the programmer who is familiar with the Y-MP/C90/T90 series and who has experience in optimization for these platforms. Our primary goal is to highlight new performance tuning issues that will arise when porting code developed for these systems to the SV1.

After a brief overview of the SV1 architecture, we shall consider in turn single- and multi-processor performance issues. In each case we discuss a small number of representative examples from user codes, and summarize our findings.

All performance data presented here were obtained under Programming Environment 3.4.

## 2.0 Overview of the SV1 Architecture

As the first in the series of Cray's Scalable Vector systems, the SV1 provides a blend of high performance hardware components with advanced compilers to aid in exploiting the technology. The individual processors in the SV1 are known as Single Stream Processors (SSPs), 32 of which can exist in a single system image. The rev 1b processors run at 300 MHz (3.33 ns clock period) and have dual pipelined floating-point units. Each pipeline can produce a chained add-multiply result per clock period, for a theoretical peak performance of 1.2 GFLOP/sec per SSP. Scalar operations share one of the pipes. As with other Cray systems, the SV1

employs 64-bit single precision. There are eight scalar and eight vector registers and the vector length is 64 words.

While the SV1 is a shared memory machine, each processor does not have a private connection to main memory. Rather, four SSPs are arranged on a single CPU module board, and each module has a pathway to memory which the four SSPs share. The net module-to-memory bandwidth is about 5 GB/sec; however, each SSP can access at most 2.5 GB/sec.

The memory hardware itself relies on commodity DRAM technology to reduce cost and allow large amounts of shared memory. In fact, it is the same memory hardware as used in the Cray J90, despite the fact that the CPU is six times as fast. To offset this gap in performance, each SSP is equipped with a cache for both scalar and vector loads. The cache size is 32 Kwords (256 Kbytes) and is four-way set associative with a Least Recently Used (LRU) replacement algorithm.

This data cache is the primary new architectural feature of the SV1, and its importance can be illustrated by noting that the peak SSP-to-cache bandwidth is 9.6 GB/sec (four words per CP) for reads and 4.8 GB/sec (two words per CP) for writes, almost six times as large as the SSP-to-memory bandwidth. Clearly, effective cache usage will be the key to obtaining peak performance on the SV1.

Another new feature of the SV1, designed to improve both memory and computational performance, is the Multi-Stream Processor (MSP). An MSP consists of four SSPs “ganged” together to act as a single processor with very high throughput. To maximize the bandwidth to memory, an MSP is ideally configured using one SSP from each of four different modules. The net MSP-to-memory bandwidth is then 10 GB/sec. The effective cache size is also increased to 128 Kword (1 Mbyte). In addition, there is a certain amount of integration at the hardware level, including shared B and T registers. This hardware integration will be substantially increased with the advent of the SV2.

A significant limitation of the SV1 is that for multi-SSP applications (whether auto-tasked or MSP) the cache is “software coherent.” This means that at points where the individual caches may have lost coherency, the system simply invalidates the entire cache. As may be expected, compilers tend towards the conservative side when assessing coherency, leading to a proportion of unnecessary cache invalidations. This is presently the most important

performance bottleneck for multi-processor applications.

Cray’s current technology plan is firmly anchored to the concept of the MSP. (The SV2 will consist entirely of MSPs.) This has several significant implications for users seeking to obtain optimal performance from their code. Specifically there are steps that will need to be taken to optimize code for the new platform, done in addition to traditional performance tuning techniques for older Cray systems.

To examine what programming techniques will provide significant performance enhancement when migrating codes to MSPs, it is useful to first consider the performance characteristics on a single SSP. Of particular interest is how applications will scale when exposed to the new memory structure.

### **3.0 Single-Processor Performance Issues**

Some previous Cray systems have had small scalar and instruction caches, but the SV1 is the first Cray system that implements a vector cache. The SV1 cache performs multiple duties, providing support for instructions as well as scalar and vector data. It is “write through” and “write-allocate,” so that each store goes to cache and main memory. For scalar loads the cache has a line size of eight words and behaves like a standard microprocessor cache system. For vector loads, however, the cache has a line width of only one word. This insures that there is no bandwidth penalty for strided vector loads or gather operations. Software pre-fetching is used to increase the apparent performance of vector loads.

There are two key benefits that the cache offers. First, the latency when fetching data from the cache is 4 to 5 times lower than when fetching directly from memory. As we shall see, computational performance is directly connected to memory bandwidth. Vector operations can process significant amounts of data and hold conditions can be reduced if the CPU does not have to wait for data loads.

Second, effective cache utilization reduces contention for main memory. If a load request can be satisfied from cache, a memory transfer does not need to occur and this leaves more bandwidth available for the other the processors on the module board. Also, a load will leave the memory bank inaccessible for whatever the memory cycle time is.

Since the cache is 32 Kwords in size and four-way set-associative, memory locations that are 8192 words apart map to the same four way cache “slot.” If there are more than four loads at this stride, values in the cache will be overwritten, basically causing data to be fetched directly from memory. Called “cache thrashing,” on most systems this results in a significant performance degradation because in addition to the data value desired, an entire cache line is loaded. This causes many more memory reads than would occur if the cache were simply disabled. Since the SV1 has a cache line size of one, there is no such penalty. Performance will simply fall to what would be seen if there was no cache.

### 3.1 Vector Dominated Code

To study the performance characteristics of the SSPs, several user codes and algorithms were examined that would test specific aspects of the hardware. As an illustrative example, consider first the computational kernel shown in Table 1. This code vectorizes well and shows a nice balance among functional units, with 5 multiplies, 3 adds and 2 reciprocals per iteration. The five arrays that appear all have dimension n.

```
do j=1,n
  c(j)=1.0/a(j)+f(j)+(1.0/b(j)+g(j))*e
end do
```

Table 1

Now, for n of order 2000-5000, the vector lengths are reasonably high and all five arrays fit comfortably in the cache. (Recall that the cache is write through, so the assignment in the equation will fill up a cache location in addition to the four arrays on the right hand side.) For problem sizes in this range, then, we expect to see a high proportion of cache hits and good memory - and hence computational - performance. As n approaches 6000-7000, however, the arrays no longer fit entirely in cache and so values will be overwritten before they can be read again. A drop in performance should be seen at this point as data is increasingly fetched from the slower main memory.

Figure 1 shows the measured performance in MFLOP/sec for this code, both with and without the cache, for various values of n. The above expectations are fully realized. For small problem sizes the performance is typically about 2.5 times

higher when the data can be fetched from the cache.

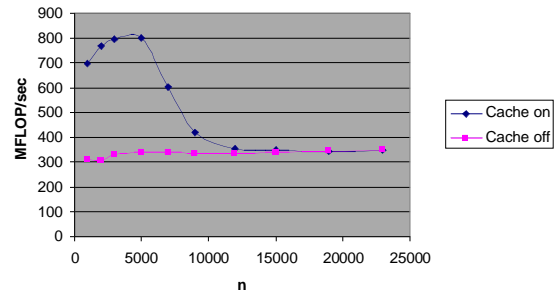


Fig. 1: Code performance

As n is increased through 6000-8000, however, cache use decreases and the performance drops. Eventually it approaches the performance obtained with no cache at all.

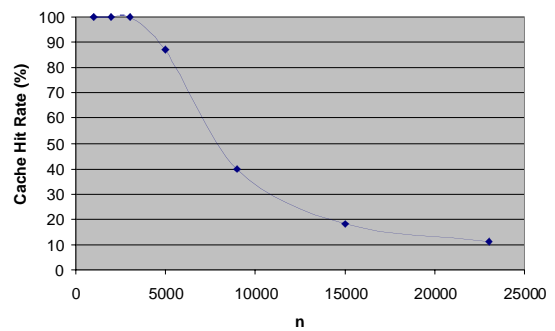


Fig. 2: Cache Hit Rate

Figure 2 shows the measured cache hit rate for this section of the code. This curve can be understood in its essentials by simply assuming about 25-28 Kwords of cache hits in each case; that is, data for the early iterations are found in cache, and after this is exhausted the remainder must come from main memory. The proportion of cache hits thus varies inversely with the problem size, once n is large enough to maximize the cache usage.

Finally, Fig. 3 shows the number of memory references per second achieved by the code. Not surprisingly, the computational performance mirrors the memory throughput directly. It is clear that achieving effective cache use will be extremely important when tuning code for the SV1.

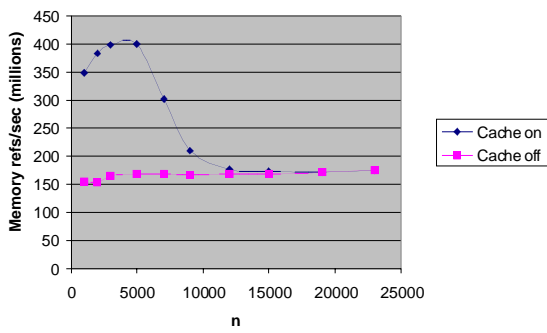


Fig. 3: Memory Throughput

For comparison, this code runs at slightly better than 1200 MFLOP/sec on a T90. Thus under favorable conditions the SSP can achieve about 70% of the T90 performance.

### 3.2 Laplace Equation Solver

Next we consider a code developed to solve Laplace's equation in 2-d space using the method of successive over-relaxation. Two versions of this code will be discussed: one optimized to perform well on vector machines and one optimized for cache-based systems. The vector version was originally tuned for the T90. All computations vectorize well and the vector lengths are relatively high (about 100 for the T90). The cache-friendly version was tuned for microprocessor based systems and emphasizes data reuse rather than vectorization. While this leaves the code with mostly scalar operations, it should take better advantage of the cache.

The problem size chosen was 2001 by 2001, which is large enough to exercise the machine and will allow for some cache re-use. Odd dimension sizes were chosen to avoid memory bank conflicts.

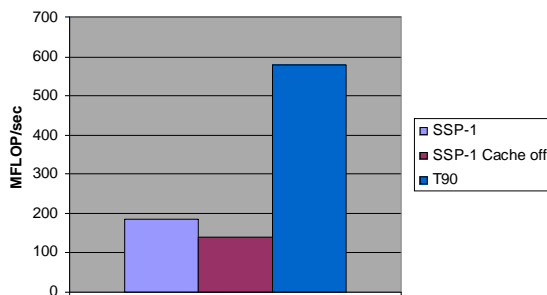


Figure 4: Vector Code Performance

Fig. 4 shows the performance of the vector version measured on the SV1 with cache on and cache off, as well as the T90. As discussed in the previous section, due to the large problem size we would not expect a high cache read hit rate. Indeed, cache

efficiency is around 23% and, as expected, performance on the SV1 is only slightly better than with the data cache disabled.

To explain why the SV1 is only achieving 30% of the T90 performance, we need only look at the memory bandwidth. Fig. 5 shows the memory references per second for the same three cases. The high-bandwidth memory system of the T90 provides almost four times the throughput as that obtained on the SV1. Again, a direct correlation can be seen between memory bandwidth and code performance.

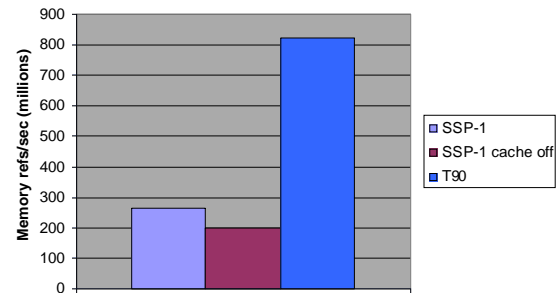


Figure 5: Memory Throughput

Fig. 6 shows the performance of the scalar version on the SV1 and T90. Since the cache is being utilized more efficiently, a substantial performance benefit from the cache can be seen. Where performance for the vector version of the code was approximately 30% compared to the T90, the performance ratio has increased to 80% of the T90. The increase in the performance ratio between the SV1 and the T90 is due to better cache utilization, which yields better memory bandwidth.

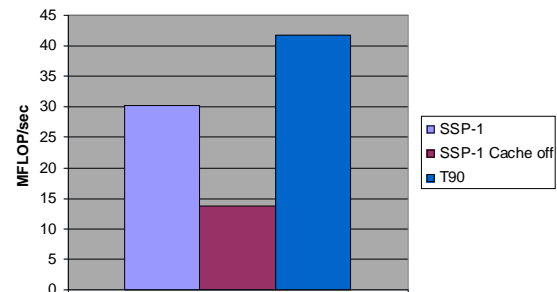


Figure 6: Scalar Code Performance

This level of performance obtained by the vector version - about 30% relative to the T90 - seems to be fairly typical for long vector codes, which will naturally be memory-bandwidth bound. This indicates again that tuning for efficient cache use, e.g., by cache blocking, will tend to give better performance on the SV1 than will the traditional approach of maximizing the vector length.

### 3.3 Memory Bank Conflicts

Memory bank conflicts arise from the memory cycle time. The charge in DRAM memory must be refreshed after every access, and during this process the bank is inaccessible for a number of CP. If a code accesses the same memory bank too frequently, it will encounter many hold conditions while it waits for the end of the cycle time to read or write another value. Accessing successive memory locations on Cray systems is optimal, as these are distributed across different banks. Generally if all of the banks are visited, enough time has passed that the first bank is ready to be read again. Stepping through memory at a non-unit stride will tend to produce conflicts more readily than stepping at unit stride. The worst case occurs for strides that are a power of 2. This will force the program to hit banks at the highest frequency, causing the longest delays.

Traditional Cray vector computers fetch vector operands directly from main memory, so code performance is directly dependent on the way algorithms step through memory. With the Cray-SV1 a cache now resides between memory and the CPU. This exhibits the behavior of a buffer, which diminishes or hides the effect of memory bank conflicts.

To illustrate this performance characteristic, a small algorithm was developed that allows the stride through memory to be varied. The design is such that up to a stride of 64, the data arrays for the problem fit into cache. Above strides of 64, the arrays no longer

fit in cache completely and a performance drop-off is seen.

Figure 7 shows the code performance as a function of memory stride (and problem size). Three cases are presented: SV1 cache on, SV1 cache off and T90. For each case, a curve showing even memory stride performance and a curve showing odd memory stride performance are drawn. Odd memory strides are shown as they represent the minimum memory bank conflict configuration.

Our discussion of memory stride indicated that strides of powers of two would cause a reduction in performance. This would be caused by the system having to wait for data due memory bank conflicts. This can also be described as a reduction in memory bandwidth since no data is being transferred while the system is waiting to access a memory bank. The curves that designate the odd stride points for the T90 and SV1 with data cache disabled reflect this. In fact, the curves remain basically flat as the stride is increased. Odd strides on the SV1 with cache enabled shows an interesting trend. Initially a large performance benefit is seen, which correlates to problem size fitting completely in cache. This eliminates memory bank conflicts and takes advantage of the higher speed cache. As the stride is increased and the problem size grows, data values are overwritten before their next references occurs. Effectively the data values are fetched directly from memory.

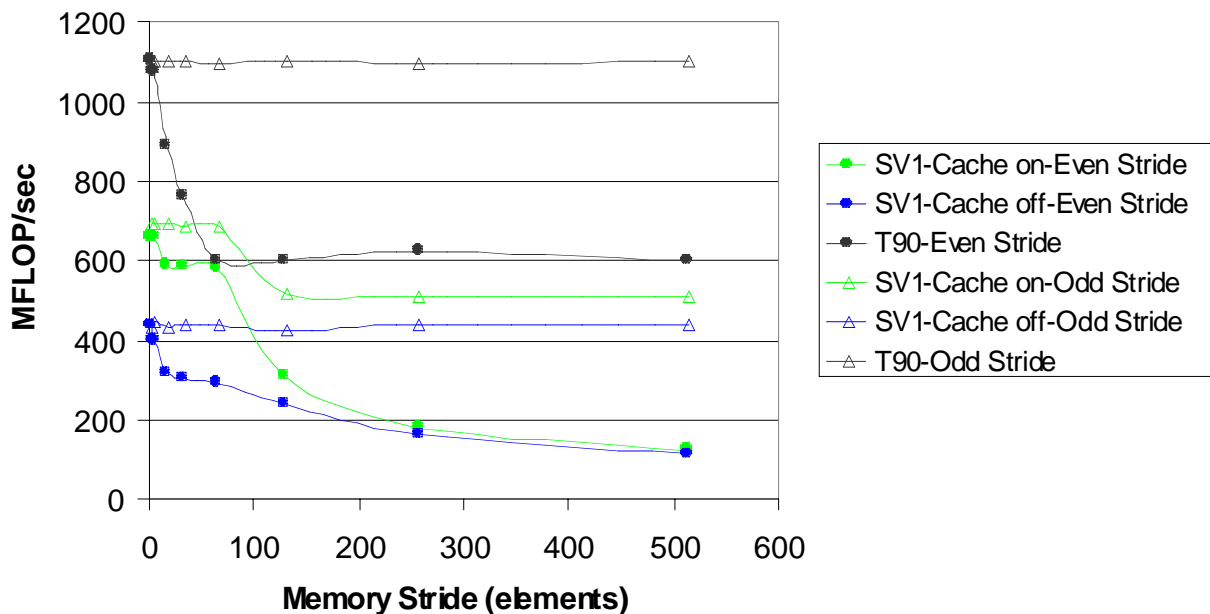


Figure 7: Effects of memory bank conflicts

A discrepancy from this theory occurs, however, as a slight performance increase can be seen over the SV1 cache off curve even after data is no longer being read from the cache. This slight increase in performance represents the cache utilization by program overhead, loop counters, etc. For typical programs some performance benefit should be seen from the SV1 cache, even if it is only caching system overhead instructions.

Looking at the curves that represent the even stride data, the impact of memory bank conflicts is apparent. Comparing the curves representing the SV1 runs with cache on and cache off show the significant benefit of the cache when memory bank conflicts are present. A small initial performance penalty is seen, but then the curve levels out until a stride of 64 is reached. This is due to the cache hiding the memory bank conflicts. At this point, the problem size begins to exceed the size of the cache and data begins to be fetched directly from main memory. The cache has the characteristic that it hides the negative effects of memory bank conflicts, so long as the problem size fits partially in cache.

### 3.4 Summary of SSP Performance Analysis

Examining Cray's new Scalable Vector architecture roadmap, there are several new features that must be considered when designing applications. Most significant is maximizing memory bandwidth for vector operations. On the SV1, the primary tool for maximizing memory bandwidth is efficient cache utilization.

Basic cache optimization rules indicate that it is best to structure applications so that data fits in cache as much as possible. On the SV1, calculations must first be vectorized. The algorithm should then be examined to find methods to maximize data reuse. It is also desirable to access data with small, preferably unit memory strides. Due to the physical size of the cache, avoid strides near 8192. This will thrash the cache and reduce potential benefits.

Thrashing the cache with vector operations on the SV1, however, does not impose a performance penalty that would reduce performance below computing without the cache. An acceptable cache hit rate seems to be in the range of 40% to 60%, depending on the application.

Significant performance gains are also seen when memory strides are unit or odd. Even with the cache, some memory references will usually need to be satisfied from main memory. Due to the slower

memory sub-system of the SV1 it is imperative to avoid even memory strides.

Looking at just the SSP performance, experiences with a variety of user codes suggest that typical performance is 30-70% of a T90 for codes that vectorize well. Performance values will tend to the lower end of this range for codes with long vector lengths or codes requiring significant memory bandwidth. Better performance is seen with codes that have a shorter vector length and can keep more data in cache.

Of these two points, the cache issue is more significant. Since there is almost a one to one correlation between overall code performance and memory performance, any cache hits will speed up the program. Pre-fetching is one potential optimization that the compiler can use to improve cache hit rate. Problems with exploitable temporal locality (data reuse) also run very well.

For codes that are dominated by scalar operations, SSP performance is typically 60-80% of the T90. This is in large part due to the scalar cache design.

### 4.0 Multi-Processor Performance Issues

The Cray compilers will generate code suitable for execution on an MSP when given the option

```
-Ostreamn
```

where  $n$  can be 0-3. At present the resulting code implements a loop-level parallelism similar to what may be accomplished using OpenMP or auto-tasking. For example, when a generic DO loop is encountered the iterations are partitioned over the individual SSPs that make up the MSP:

do j=1,1000	<b>SSP0: j=1,250</b>
a(j)=b(j)*2.71828	<b>SSP1: j=251,500</b>
end do	<b>SSP2: j=501,750</b>
	<b>SSP3: j=751,1000</b>

Each SSP determines on its own the range of iterations it will handle; there is no master-slave concept.

In essence multi-streaming serves as a pipe/register/cache multiplier. The MSP is effectively an eight-pipe processor with 128 Kwords (1 Mbyte) of cache. This increase in cache size can result in super-linear

speedups, compared to single-SSP execution, for appropriately sized problems.

Multi-streaming has certain performance advantages relative to traditional auto-tasking. The MSP has optimal access to memory bandwidth, and in addition there is a tighter integration at the hardware level and reduced execution overhead. These advantages should become more pronounced in the SV2.

Like auto-tasking, multi-streaming also opens new opportunities for performance tuning. For example, loops that would ordinarily not vectorize (e.g., most outer loops) can still stream. This can lead to additional considerations when choosing, e.g., loop nest orderings.

In this section we discuss issues that may arise when optimizing code for execution on an MSP.

#### 4.1 Laplace Equation Solver

We shall begin by revisiting the Laplace’s equation solver discussed previously. Recall that its SSP performance was about 30% of the T90 due to limited cache utilization. Most memory requests, therefore, had to be satisfied directly from main memory.

Table 2 shows the loopmark listing obtained when the code was compiled for execution on an MSP. Only the main computational loop is shown, and additional annotations have been added to show the effects of multi-streaming. The compiler performs three additional optimizations when compiling this code for an MSP. First, it recognizes two loops as candidates for streaming. The loop starting at line 20 and the loop starting at line 30, both marked with an “S” in the 2<sup>nd</sup> column, are streamed. The other additional optimization is that the outer loop at line 20 is vectorized. Note that the reduction operation in

the middle loop (the calculation of the global maximum) does not currently stream.

Fig. 8 shows the overall floating-point performance of the program. Note that the MSP performance is more than a factor of four greater than that of the SSP. This is due to the larger effective cache of the MSP and the resulting increase in memory throughput.

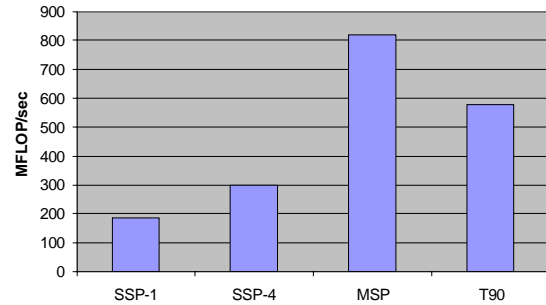


Figure 8: Floating-Point Performance

The auto-tasked version of this program did not scale as well, due to the fact that the compiler did not efficiently utilize the cache. Scalability of this version can be improved using appropriate compiler directives, but we will not discuss this here.

Fig. 9 shows the memory performance of the code with data from the MSP run included. The two significant trends from this data show that the effective memory bandwidth of the MSP job has surpassed the T90 and noticeably increased over the SSP run. Since an MSP is created from one SSP on four separate module boards, the connection to memory is optimized and the highest bandwidth can be seen. The larger effective cache reduces memory contention and accounts for the additional memory bandwidth gains. Again the strong correlation

```

18      1 -----          do it=1,itmax
19      1 -----          dumax=0.0
20  S--1v -----          do j=2,jm1
21  S  12v -----          do i=2,im1
22  S  12v -----          du(i,j)=0.25*(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1))-u(i,j)
23  S  12v ----->          enddo
24  S--1v ----->          enddo
25      12 -----          do j=2,jm1
26      12v -----          do i=2,im1
27      12v -----          dumax=max(dumax,abs(du(i,j)))
28      12v ----->          enddo
29      12 ----->          enddo
30  S--12 -----          do j=2,jm1
31  S  12v -----          do i=2,im1
32  S  12v -----          u(i,j)=u(i,j)+du(i,j)
33  S  12v ----->          enddo
34  S--12 ----->          enddo
35      1 -----          write (1,*) it,dumax
36      1 ----->          enddo

```

between memory bandwidth and processor performance is apparent.

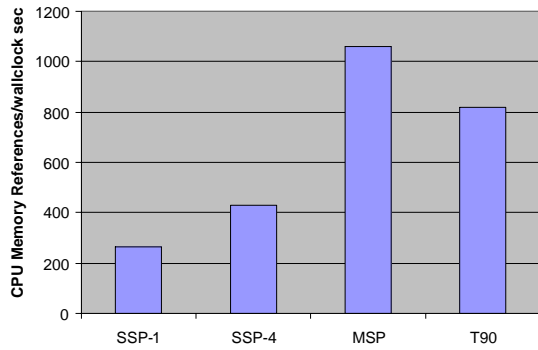


Fig. 9: Memory Throughput

Fig. 10 shows select Hold Issue Conditions for the program. High wait conditions on the vector functional units shows that the program is getting good vector performance. This is true for all cases considered.

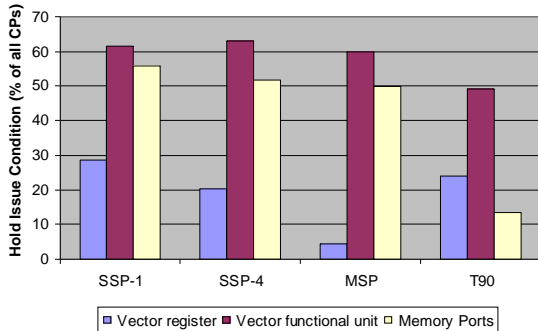


Fig. 10: Hold-Issue Conditions

## 4.2 Nonlinear Wave Equation Solver

The final program we shall discuss solves the coupled set of quasi-steady state equations to determine frequency and growth rate of pressure oscillations within a chamber. This code was originally tuned for the Cray Y-MP and later for the T90. It is loop-intensive, but has a relatively short average vector length of about 30.

This code originally went through two stages of optimization. In the first stage the various profiling tools available on Cray platforms were used to

identify the most significant loops and optimize them for vectorization and tasking. The second step was to insure that memory bank conflicts were minimized. The code that resulted after the first round of optimizations will be referred to as V1 and the code that resulted after the second round of optimizations will be referred to as V2.

These are traditional optimization techniques designed to obtain maximum single processor performance on a vector system. A third optimization step was later performed to parallelize the remaining serial loops. The loops were modified to either eliminate serial dependencies or to move them to separate loops so that additional code would be available for tasking or streaming. The final code after this optimization stage will be referred to as V3.

An example of the modifications performed in obtaining code version V3 is given in Tables 3 and 4. Table 3 shows a section of the code V2 in which two

1	ANUM = 0.0
1	ADEN = 0.0
1v -----	DO 131 j=1,Njj
123 -----	DO 131 i=1,Nii
123v -----	DO 131 k=1,Nkk
123v	ANUM=ANUM+conjG(GDn(i,j,k))*HDn(i,j,k)
123v	& +conjG(GVz(i,j,k))*HVz(i,j,k)
123v	& +conjG(GVr(i,j,k))*HVR(i,j,k)
123v	& +conjG(GVa(i,j,k))*HVa(i,j,k)
123v	ADEN=ADEN+conjG(XDn(i,j,k))*XDn(i,j,k)
123v	& +conjG(XVz(i,j,k))*XVz(i,j,k)
123v	& +conjG(XVr(i,j,k))*XVr(i,j,k)
123v	& +conjG(XVa(i,j,k))*XVa(i,j,k)
123v =====>	131 CONTINUE

Table 3: Version V2 after compilation for multi-streaming

global sums are computed. The inner loop vectorizes nicely and maximum performance is obtained on a machine like the T90, where sufficient memory bandwidth is available to sustain the calculational throughput. The loop does not stream, however, due to the global reduction operation. Table 4 shows the modified code in V3, with dummy arrays introduced to hold the partial sums and the global reduction moved outside the loop. In this case the partial sum calculations both stream and vectorize. Note that the number of vector calculations has actually decreased in passing from V2 to V3.

The computational performance of the three versions of this code is presented in Fig. 11. The original version of the code, V1, has good performance relative to the T90 because memory bank conflicts are throttling the T90 and the SV1 cache is helping to curtail the effect of this bottleneck.

Eliminating the memory bank conflicts in the second round of optimizations, code version V2, dramatically improves performance on the T90. While a factor-of-3 performance increase was seen on the T90, only a factor of about 2 was seen on the SV1. This indicates that the SV1 cache was helping to hide the memory bank conflicts in V1. When they were eliminated in V2, there was less of a performance increase to be seen.

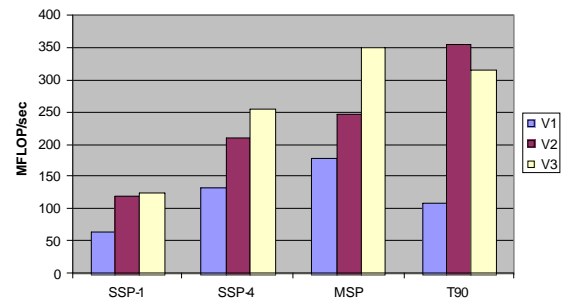


Fig. 11: Code Performance

While single processor performance was maximized on the T90, profiling tools indicated that there were still loops with serial dependencies that were hurting program scalability. Removing the serial dependencies added extra overhead to the program, which hurt single processor performance on the T90 as MFLOP performance dropped from V2 to V3. It significantly improved multiprocessor performance

on the SV1, however, under both tasking and streaming.

Figures 12 and 13 show once again the strong correlation between memory performance and processing performance. Fig. 12 contains the MFLOP data for the SV1. For each code version, the auto-tasked and streamed MFLOP values are normalized against the SSP run. This is to show the relative speedup gained by each version in passing from the single- to multi-processor execution.

```

M-- 12  -----      DO 131 j=1,Njj
M   123  -----      DO 131 i=1,Nii
M   123v  -----     DO 131 k=1,Nkk
M   123v
M   123v      &      tmp1(i,j,k)=conjG(GDn(i,j,k))*HDn(i,j,k)
M   123v      &      +conjG(GVz(i,j,k))*HVz(i,j,k)
M   123v      &      +conjG(GVr(i,j,k))*HVR(i,j,k)
M   123v      &      +conjG(GVa(i,j,k))*HVa(i,j,k)
M   123v      tmp2(i,j,k)=conjG(XDn(i,j,k))*XDn(i,j,k)
M   123v      &      +conjG(XVz(i,j,k))*XVz(i,j,k)
M   123v      &      +conjG(XVr(i,j,k))*XVr(i,j,k)
M   123v      &      +conjG(XVa(i,j,k))*XVa(i,j,k)
M-- 123v =====> 131  CONTINUE
      1      ANUM=SUM(tmp1)
      1      ADEN=SUM(tmp2)

```

Table 4: Version V3 after compilation for multi-streaming

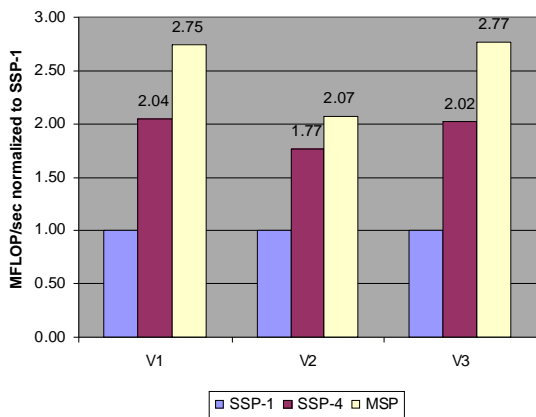


Fig. 12: FP performance

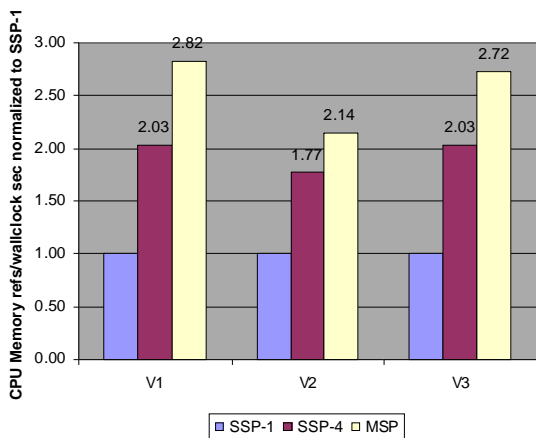


Fig. 13: Memory throughput

Tasking and streaming brought good performance improvements with V1, which had high memory bank conflicts. This stems from the fact that they were providing more cache and memory bandwidth. Eliminating the memory bank conflicts was mostly a single processor optimization, so the speedups seen for V2 are not as high as those seen in V1. The final optimization step was aimed at improving scalability, and we do see a return to better speedup numbers. Fig. 13 shows CPU memory references per wallclock second in the same format as Fig. 12. For each code version, the auto-tasked and streamed memory references/second are normalized against the single processor run. It is not so much the data presented in Fig. 13 that is important, as is how the values compare to those in Fig. 12: the increases in CPU performance match almost exactly the increases in memory performance. This reinforces again our observation from the SSP analysis that CPU performance is dependent on memory performance. This holds true for MSP codes as well.

### 4.3 Summary

It has been our experience that with basic compiler options, MSP performance is typically as good as or better than that of a single T90 processor. For long-vector code, MSP execution can approach, and even sometimes exceed, a factor-of-four speedup over a single SSP due to the larger effective cache.

Loop nest optimizations appropriate for the Y-MP/T90 generally seem to still be best. That is, in most cases one should optimize a loop nest as though for a single processor, then allow the compiler to stream the outer loop. The most important consideration when choosing loop nest ordering is effective cache use, as exhibited by the direct correlation between memory throughput and floating-point performance.

In general, users will want to perform one additional tuning step and minimize serial code in loops. This has the general effect of reducing MFLOP/cpu sec performance, but increasing MFLOP/wall-clock sec performance. Constructs that currently inhibit streaming include I/O statements, data dependencies and global reduction operations (both logical and arithmetical).

## 5.0 Conclusions

Cached vector operations and Multi-Stream processors in Cray’s Scalable Vector architecture result in new performance tuning issues for developers. Our experience indicates that while most traditional optimization techniques remain relevant for the SV1, some additional techniques are required to take advantage of the cache and multiprocessor characteristics of the new architecture.

Primary tuning techniques for systems such as the Cray Y-MP and T90 focus on maximizing single processor vector performance. For the most part this means insuring that calculations vectorize, and working to maximize the vector length. Avoidance of memory bank conflicts was also crucial so that the high memory bandwidth available on these systems was not compromised.

All of these techniques remain important for the SV1 — it is still very much a vector machine. In addition, however, programmers will want to give careful attention to effective cache use, to achieve high memory throughput. Standard techniques such as cache blocking will have significant benefits. While vectorization should not be sacrificed for better cache performance, once vectorized the algorithm should be examined to find methods to maximize data reuse.

This is likely to be more important than working to maximize the vector length.

It is also desirable to access data with small, preferably unit, memory strides. Due to the physical size of the cache, strides near 8192 will thrash the cache and reduce potential benefits. In addition, using memory strides that are unit or odd will improve bandwidth to memory by minimizing bank conflicts. While the cache will act as a buffer and hide some of the effect of memory bank conflicts, the slower memory sub-system of the SV1 is very sensitive to this hardware characteristic.

Compiling programs to run on MSPs also gives improved memory performance. By building the MSP from SSPs on separate module boards, four independent pipes are opened to memory. MSPs also have four times the effective cache of an SSP, which helps reduce memory conflicts.

For optimal MSP performance, additional tuning is required in addition to the single processor optimizations. Specifically, the code should be examined to maximize the amount of work that can be performed in parallel. This may involve splitting up loops with serial dependencies, so that the serial work is isolated and other tasks may be streamed. While these techniques will generally decrease single processor MFLOP/sec, they can have significant benefits when running streamed code.

All of these techniques should be relevant for the SV2 as well; it will have the same sort of memory hierarchy and will consist entirely of MSPs. Tuning code effectively for the SV1 should thus prepare it to run well as the Scalable Vector series hardware and compilers continue to evolve.