Performance Characteristics of the Cray X1 and Their Implications for Application Performance Tuning

Hongzhang Shan
Lawrence Berkeley National Laboratory
One Cyclotron Road
Berkeley, CA 94720
hshan@lbl.gov

Erich Strohmaier
Lawrence Berkeley National Laboratory
One Cyclotron Road
Berkeley, CA 94720
estrohmaier@lbl.gov

ABSTRACT
During the last decade the scientific computing community has optimized many applications for execution on superscalar computing platforms. The recent arrival of the Japanese Earth Simulator has revived interest in vector architectures especially in the US. It is important to examine how to port our current scientific applications to the new vector platforms and how to achieve high performance. The success of porting these applications will also influence the acceptance of new vector architectures. In this paper, we first investigate the memory performance characteristics of the Cray X1, a recently released vector platform, and determine the most influential performance factors. Then, we examine how to optimize applications tuned on superscalar platforms for the Cray X1 using its performance characteristics as guidelines. Finally, we evaluate the different types of optimizations used, the effort for their implementations, and whether they provide any performance benefits when ported back to superscalar platforms.

Categories and Subject Descriptors
C.4 [**Performance of Systems**]: Performance Attributes – *optimization;* D.1.3 [**Parallel Programming**]: Performance Characterization and Optimization*;*

General Terms: Algorithms, Measurement, Documentation, Performance

Keywords: Performance optimization, performance characterization, performance measurement, vector processing

1. INTRODUCTION
Superscalar processors have experienced a rapid proliferation in high-performance computing systems during the last decade. A large body of scientific codes has been developed to run on systems with superscalar processors. However, the arrival of the Japanese Earth Simulator has revived interest in vector architectures. The Earth Simulator system has demonstrated sustained performance of almost 65% of peak-performance for a production global atmospheric simulation; while the typical sustained performance of scientific applications on superscalar platforms is below 15% of their peak-performance[1,9]. Recently, the Cray X1 supercomputer was also introduced. It provides exceptional memory bandwidth, low-latency interconnects and vector-processing capabilities. It is therefore important to investigate whether the codes developed on superscalar platforms can be run efficiently on these new vector platforms.

ICS'04, June 26-July 1, 2004, Saint-Malo, France.

Compared with superscalar architectures, the vector architecture has several advantages. First it alleviates the instruction fetching, decoding bottleneck since each vector instruction specifies a large number of operations and thus the vector program needs far fewer instructions. Also, it has a large vector register file and supports stride and gather/scatter memory accesses. Thus, it can tolerate longer memory latency and exploit the memory bandwidth more efficiently. But these architectural advantages can only be realized when there is enough explicit data level parallelism, i.e., the application programmer is able to implement the numerical algorithm in vector-rich subroutines. In addition, the fact that current vector processors run much slower than their contemporary superscalar processors makes it essential to take full advantage of its architectural features in order to achieve better performance on the vector platforms.

Recently, several researchers have compared the application performance on the superscalar platforms and vector platforms [2,3]. But they provide little information for the end users on how to optimize application performance. In the 1980s a large amount of work was done on previous vector-architectures. Wayne and Christopher analyzed how to transform the FORTRAN DO loops to improve the performance on vector architectures [10]. Cheng discussed the importance of vector pipelining and chaining on the IBM 3090 and Cray X-MP [11]. Many of these techniques have been implemented in compilers or hardware on modern vector architectures and today programmers no longer need to perform them manually. We need to examine which optimization techniques are still effective on modern vector platforms. At the same time, both superscalar and vector architectures have changed substantially during the last decade. Due to the increasing gap between the memory speed and CPU speed, the memory performance has become the dominant performance factor on the current computing platforms. In the 1980s a typical vector computer had a flat shared memory often build with SRAM technology and therefore very fast. Today large-scale vector computers have either distributed memory between SMP nodes (Earth-Simulator) or a non-uniform hierarchical shared memory with caches (Cray X1). Also, the Cray X1 includes cache and multi-streaming in its design which is radically different from earlier vector architectures. Whether this innovative design will generate special performance characteristics needs to be investigated. In this paper, we use Apex-Map, a memory

performance probe, to study the performance characteristics of superscalar platform and today's vector platform and are interested in the following questions:

- What are the performance characteristics of modern vector architectures and how do they differ from superscalar architectures?
- Can programs well tuned for superscalar platforms perform well on modern vector architectures without further optimizations?
- If not, can the performance characteristics be used as guidelines to optimize the performance?
- What kinds of optimizations are needed? Is it as simple as adding compiler directives or as complex as restructuring data structures and algorithms?

From our experiments we find that data reuse, the most influential performance factor on the superscalar platform, has only little effect on the vector platform. The total amount of memory accessed also minimally affects the performance. It is still the vector length of data access and memory bank conflicts that affect the performance most. Multi-streaming further helps performance. We also learn that applications developed on superscalar platforms cannot be ported directly to vector platforms and achieve high performance. Adding compiler directives can substantially improve performance in many cases. However, restructuring the programs to increase vector length and to eliminate memory bank conflicts provides the most potential for performance optimization.

The rest of the paper is organized as follows. Section 2 introduces the experimental platforms we used. In Section 3, we introduce Apex-Map, a performance probe based on memory accesses, to analyze the different performance characteristics of superscalar and vector architectures and establish the guidelines to optimize on vector platforms. The applications we selected are briefly described in Section 4. Performance of these applications is analyzed in Section 5, which also examine the programming efforts needed for performance optimization. Finally, Section 6 summarizes our results.

2. EXPERIMENTAL PLATFORMS

For this study we select two platforms, the most commonly used IBM Power4 platform and the newly released Cray X1 platform to represent the superscalar and vector platforms respectively.

2.1 IBM Power4 Platform

The Power4 chip contains two 64-bit microprocessor-cores including their L1 caches, a microprocessor interface controller unit, a shared 1.41MB L2 cache, a L3 cache directory, and other controllers. Each processor has eight execution units (2 floating-point units, 2 load/store units, 2 fixed-point units, 1 branch unit, and 1 logic unit), each capable of being issued an instruction each cycle. The floating-point unit can start a fused multiply and add each cycle. Thus, the peak floating point performance for a 1.3 GHz model is 4*1.3G = 5.2 GFLOPS. It can fetch up to eight instructions per cycle and can dispatch and complete instructions at a rate of up to five per cycle.

The Power4 storage hierarchy consists of three levels of cache and the memory subsystem. Each processor has one 64KB instruction cache and one 32KB data cache. The L1 data cache is triple ported capable of two 8-byte read and one 8-byte write per cycle with no blocking. The L1 data cache reloads from L2 are 32-bytes per cycle. This means that the L2 cache can load the equivalent of four double precision floating-point data elements per cycle, which is double the capability of the processors to issue load instructions. The L1 is write-through and the stores are passed to the L2 interface 8 bytes at a time. Cache line size is 128 bytes. The L2 is a store-in cache. In the case where the cache line is not already present in the L2 cache, then it must be loaded from memory, another chip's L2 cache, or the L3 cache to ensure that the L2 cache contains the latest copy of the cache line. Each processor is capable of managing up to eight data cache line requests to the L2 cache (and beyond) at any given time.

The Fabric Controller is responsible for controlling data flow between the L2 and L3 controller for the chip. The bus between L3 controller and L3 cache (on a different chip) is 16-byte wide running at 1/3 the processor frequency. The L3 cache size is 32MB. It is designed to be combined with other L3 caches on the same processor module in pairs or quadruplets to create a larger, address-interleaved L3 cache of 64MB or 128MB. Each processor has three types of buffer caches to speed the process of translating from virtual address to physical address: a translation look-aside buffer (TLB), a segment look aside buffer (SLB), and an effective-to-real address table (ERAT). The page size can be 4KB or 16MB. It supports up to eight prefetch streams. A load/store between floating point register and L1 has a latency of about 4cycles; between registers and L2 it is approximately 14 cycles.

2.2 Cray X1

The basic building block for Cray X1 is the multi-streaming processor (MSP), which consists of four identical single-stream processors (SSP). Each SSP has two 64-bit vector units and one 2-way superscalar unit. The clock frequency for the vector units is 800MHz. Each vector unit is capable of one 64-bit floating-point add and one floating-point multiply operation each cycle. Thus the peak floating-point performance for a SSP is 3.2GFLOPs/s. Each SSP has 32 vector registers holding 64 double-precision words, allowing up to 512 outstanding memory requests to hide latency. Additionally, all vector operations are performed under a bit mask, allowing loop blocks with conditionals to compute without the need for scatter/gather operations. The scalar unit runs at 400MHz with two 16KB caches (instruction cache and data cache).

The four SSPs inside a MSP share a 2-way set associative 2MB data cache, a unique feature for vector architectures. The cache is needed because the memory bandwidth is not large enough to saturate the vector units. The peak memory bandwidth is 25.6 GB/s, i.e. 0.5 Words/FLOP while the cache bandwidth is doubled to 51.2GB/s. The cache here is mainly designed to exploit the temporal locality of scientific applications. An X1 node consists of four MSPs sharing a flat memory through 16 memory controllers (Mchips). Each Mchip is attached to a local memory bank (Mbank), for an aggregate of 200GB/s node bandwidth. To build large configurations, a modified 3D torus interconnect is implemented via specialized routing chips. Finally, the X1 is a globally addressable architecture, with specialized hardware support that allows processors to directly read/write remote memory address in an efficient way.

The X1 programming model is designed to hierarchically leverage parallelism. At the SSP level, vector instructions allow a large number of SIMD operations to execute in a pipeline fashion, thereby tolerating memory latency and allowing for high-sustained

performance. MSP parallelism is achieved by distributing loop iterations across each of the four SSPs. The compiler must therefore generate both vectorizing and multistreaming instructions to effectively utilize the X1. Intra-node parallelism across the four MSPs is explicitly controlled using shared-memory directives such as OpenMP or Pthreads. Finally, traditional message passing via MPI is used for coarse-grain parallelism at the inter-node level. Additionally, the hardware supported globally addressable memory allows efficient implementations of one-sided communication libraries (SHMEM, MPI- 2), as well as implicitly parallel programming languages (UPC and CAF). All X1 experiments reported in this paper were performed on the 128-MSP system running UNICOS/mp 2.3.07 and operated by Oak Ridge National Laboratory.

3. ANALYSIS OF PERFORMANCE CHARACTERISTICS

In this section, we use Apex-Map to investigate the performance characteristics of superscalar architectures and vector architectures. The goal is to find the performance characteristics of the vector architectures that are distinct from the conventional superscalar architectures and set up guidelines for performance tuning on vector platforms.

Apex-Map [4] is a synthetic memory access probe to evaluate different architectures. It has three input parameters: the amount of memory accessed (M), the temporal reuse of data (α), and vector length of data access (L). It assumes that application performance can be characterized by these three architecture independent parameters. Once an application has been characterized with these parameters, Apex-Map generates a non-uniform random access stream based on these parameters to simulate the application's memory access behavior. Therefore the performance the Apex-Map should be close to that of the corresponding application. Such a benchmark can be used as a realistic indicator of achievable application performance and enables the users to directly evaluate a new platform based on their own interests.

Apex-Map uses indexed accesses to simulate memory behavior. The starting addresses are aligned by length L and generated by a random power distribution function controlled by the parameter α. Once an address is accessed, the following L addresses will also be accessed. The range of memory accessed is within size M. The value of α varies between 0 and 1. Zero indicates that maximum reuse (only one data item will be repeatedly accessed) while one means uniform random access. The value of L or M is between 1 and ∞ (or the maximum memory available). Following is the kernel code for Apex-Map:

```
for  (j=0; j < Length Of the Index Array/4; j ++) {
    for  (k = 0; k < L; k++) {
        tmp += data[index[j*4]+k];
        tmp1 += data[index[j*4]+1+k];
        tmp2 += data[index[j*4]+2+k];
        tmp3 += data[index[j*4]+3+k];
    }
}
```

The index array is computed using a power distribution function with parameter α, L and M. The loop is manually unrolled four times because some compilers cannot optimize the loop well. Now let's examine how the performance are affected by these three parameters.

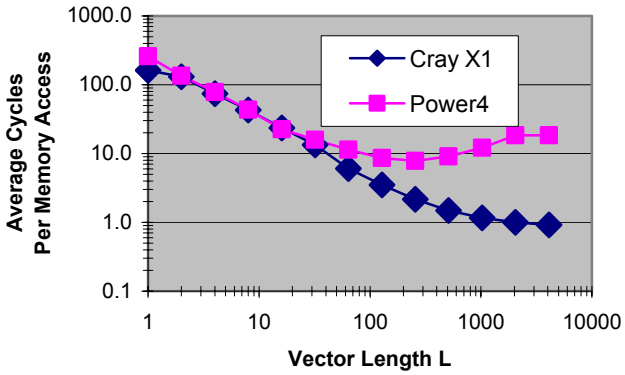

Figure 1. Effect of Vector Length L (M = 512 MB, α= 1.0).

The first experiment evaluates the effect of the parameter L. The output of Apex-Map is the average number of cycles needed for one memory access. The accesses to the index-array used to store the random addresses are not counted as memory addresses, only the accesses to the main data array. On Cray X1, the size of L is equal to the vector length. When L is small, the startup cost of the vector operation cannot be amortized well, leading to an average of 160 cycles per memory access for vector length 1 (see Fig. 1). With the increase of the vector length, the average access time reduces substantially. On average it even takes less than 1 cycle when the vector length reaches 4096. On the Power4 platform, the average number of cycles per memory access needed starts from around 250 for vector length 1 and goes down with increasing L. However, beyond a vector-length of 256, the average access times start to increase again. Contrary to the Cray X1 platform, the longer vector length begins to hurt performance. By examining hardware counters on the Power4, we found that the number of loads per TLB miss drops almost four times compared with the version without loop unrolling, which causes the average access times to be flat for longer vectors. The possible reason is that the prefectching of the unrolling version becomes too aggressive for longer vectors. The results of this experiment clearly indicate that using long vectors is critical to achieve high performance on Cray X1 and the longer the better; on the Power4 platform, only certain vector length helps to improve the performance.

The next experiment examines the effect of the parameter M, the amount of memory accessed. The average cycles needed when α=1 (uniform random access) and L = 64 are shown in Fig. 2. On the Cray X1, the average cycles only change slightly when the data set size approaches the cache size from about 5 cycles to 6 cycles. Otherwise, the number of cycles needed is almost stable. However, on the Power4 platform, the number of average cycles increases significantly from 1 cycle for 32KB memory size to more than 11 cycles for 512MB. The memory access on the Cray X1 is similar to uniform memory access. But the performance on the Power4 platform is very sensitive to the parameter M. When M is small, the data fit in cache and one needs only a few cycles per memory access; when M becomes larger, most of the data resides in main memory and cause the average cycles per memory access to increase significantly. The Cray X1 hides the memory latency much better than the Power4 platform.

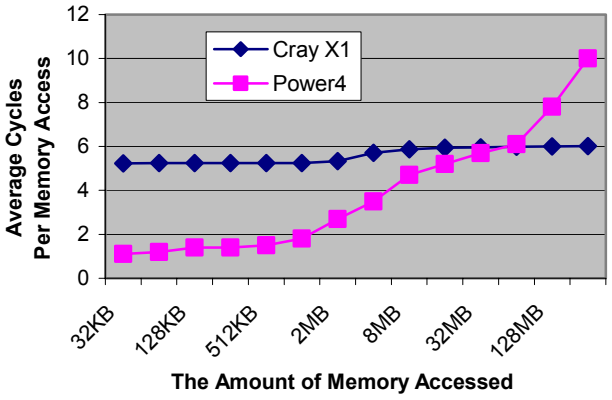

Figure 2. The effect of the amount of memory accessed (a = 1.0, L = 64).

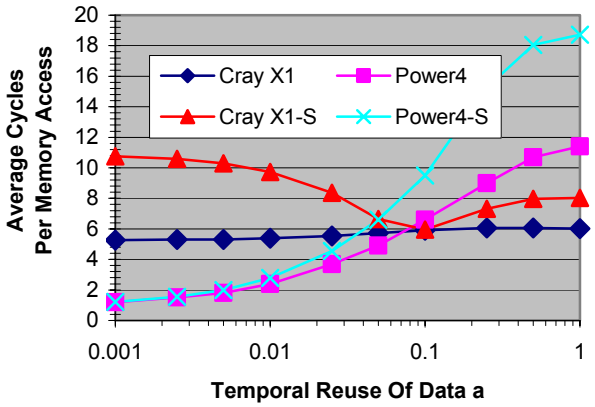

Figure 3. Effect of Temporal Reuse α (M = 512 MB, L = 64). Smaller α value represents higher temporal data reuse. α = 1 represents uniform random distribution. The results of the original code are labeled as Cray X1, and Power4 respectively. The results after switching the loops are labeled as Cray X1-S and Power4-S.

Finally, let's look at the effect of the data reuse α (Fig. 3). The temporal reuse of data has little effect on the average cycles per memory access on Cray X1 (Series: Cray-X1) while it significantly affects the performance on Power4 (Series: Power4). The number of average cycles per memory access increases from 1.5 cycles to 11.5 cycles when α changes from 0.001 to 1.0. At the same time, on the Cray X1, the average number of cycles increases only from 5.3 to 6. The Cray X1 has a large register file and each SSP allows a maximum of 512 outstanding load requests. This large pipeline of memory operations reduces the effect of temporal reuse of the data on the performance. The Power4 only allows eight outstanding memory requests and its memory performance is therefore substantially influenced by the value of α.

For nested loops, the Cray X1 prefers the longer loops to be the inner loop to achieve high efficiency. If we switch loop *j* and loop *k* in the above code, the new performance data are shown in Fig. 3 as Series Cray X1-S and Power4-S. We would expect that the number of average cycles per memory access is smaller when the data reuse is higher (smaller α value). This is exactly the case on the Power4 platform. To our surprise, on the Cray X1 platform, the number of cycles to load the data reaches its maximum when the temporal reuse reaches the highest point at α=0.001. This abnormal behavior is the consequence of memory bank conflicts. When accessing the data in the same memory bank, the second operation must wait until the first operation has finished and the memory bank access logic has refreshed. The pipeline of memory access for vector loads cannot work efficiently in this case. If the temporal reuse of data is high, the chance of memory bank conflicts in our programs becomes also high. And the performance begins to suffer from the memory bank conflicts. This is an artificial effect of our program. But it does reflect the effect of memory bank conflicts on the vector platform.

By studying the effects of different parameters on memory performance, we have found that the most influential parameter on the Cray X1 platform is the vector length. Memory bank conflict can also significantly hurt the performance. But the memory size accessed and temporal locality, which highly affect the memory performance on the Power4 platform, only have little effect on the Cray X1. Therefore, the application tuning for vector architectures should focus on increasing the vector length and avoiding memory bank conflicts.

Optimizing the inter-node communication is not the focus of this paper since it is common on both platforms for programs written in MPI programming model. In this paper, we are interested in the performance characteristics on vector platforms that are different from superscalar platforms. Also the applications we selected are already well tuned for inter node communications on superscalar platforms.

4. APPLICATIONS

We start with applications that are well tuned for cache-based superscalar platforms. Three applications, 1-D FFT, Radix Sorting, and Nbody are originally from the SPLASH2 suites. We rewrote them in the MPI programming model and tuned them on the Origin 2000 platform [5,6]. Another application we used is matrix multiplication.

FFT is a double precision complex 1-D version of the radix-$\sqrt{n}$ six-step FFT algorithm described in [7], which is optimized to minimize the inter-process communications. The *n*-point data set is arranged in the form of a $\sqrt{n} * \sqrt{n}$ matrix, and the matrix is partitioned among the processors in blocks of $\sqrt{n}/p$ continuous rows each. The whole FFT structure is as follows: (i) transpose matrix, (ii) perform 1-D FFTs individually on local rows of size

$\sqrt{n}$ each, (iii) multiply the elements of the resulting complex matrix by the corresponded roots of unity, (iv) transpose matrix, (v) perform 1-D FFTs individually on local rows, (vi) transpose matrix.

Radix sorts a series of integer keys in ascending order using the radix algorithm. The radix size used determines the number of iterations. For each iteration, it computes the histograms first and then moves the keys according to the histograms. There are two main arrays working as source and destination alternatively. The source data are read sequentially while the destination data are written scattered.

Nbody simulates the interaction of a system of bodies in three dimensions over a number of time steps, using a hierarchical N-body method. There are three main stages, building an oct-tree to represent the distribution of the bodies, browsing the oct-tree and calculating the gravitational force, updating the body positions and velocities. At the beginning of each time step, a process has to exchange information with other processes to build the tree. Then it will browse the tree from the top for each body and compute the gravitational force along the path. The data accesses are scattered and involve many pointer-chasing operations. Force calculation is the most time-consuming stage.

Matrix-Multiplication (MM) computes the product of two matrices, which is a very common basic operation in scientific computations.

5. PERFORMANCE TUNING ON CRAY X1

The sequential performance of the original version on both platforms is shown in Fig. 4. On the Cray X1, the codes run in MSP mode. The data set sizes for Nbody, FFT, Radix, and MM are 2 million bodies, 16 million data points, 256 million integers, and 2048*2048 matrices respectively. In all cases, the execution time on the Cray X1 is clearly much longer than that on Power4. Note that the Y-axis is in log scale! The compiler on Cray X1 is conservative and many loops cannot be vectorized or multistreamed directly. By adding compiler directives wherever suitable to instruct the compiler to generate both vectorizing and multistreaming instructions, the performance can be significantly improved. The new results are also shown in Fig. 4. The vendor provides a compiling tool to indicate whether the loops can be vectorized or not. The compiler is conservative when analyzing the data independence between loop iterations. The compiler directives can be used to explicitly instruct the compiler the data independence so that the loop can be vectorized.

With proper compiler directives, the Cray X1 now performs better for FFT and especially the matrix multiplication but still worse for nbody and radix. The average vector lengths for nbody, FFT, and radix are 3.71, 9.8, and 1 respectively (obtained using the PAT tool on Cray X1). They are far less than the vector register length of 64. This partially explains the lower efficiency of the Cray X1 compared to the Power4 for these kernels. In case of the matrix multiplication the vector length exceeds the vector register length of 64. This leads to a substantially higher efficiency of the Cray X1. Many applications, which have been optimized for execution on superscalar platforms, can however not be directly ported to the Cray X1 platform and achieve good performance without further optimization.

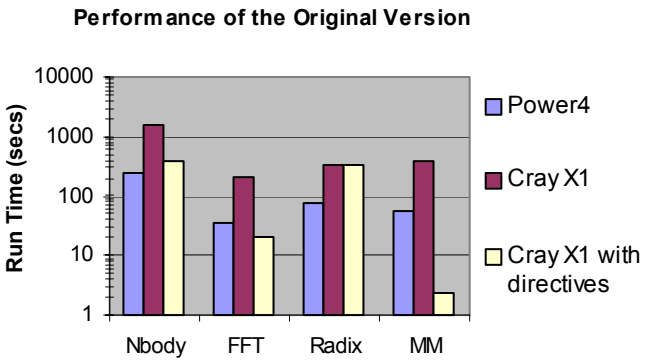

Figure 4. The Sequential Performance of the Original Code Version on Power4 (5.2 GFlops/ peak) and Cray X1 (12.8 GFlop/s peak).

As we have discussed in Section 3, the performance of vector machines is highly affected by the average vector length and memory bank conflicts. Now let's examine for each individual application whether increasing the average vector length and eliminating memory bank conflicts can optimize the performance to a substantial extent. The codes with proper compiler directives are called base version. Increasing the vector length is mainly carried out by restructuring the code to exploit the data parallelism across loops, which is difficult for the current compiler to identify automatically.

5.1 FFT

Fig. 5 presents the performance of the base version on both the Cray X1 and Power4 platforms. The Cray X1 performs better than the Power4 platform, especially for large number of processors. The average vector length is 9.8. In order to take full advantage of the vector units, we need to increase the average vector length. The approach is to perform the FFTs on 64 local rows simultaneously instead of on only one row each time so that the local computation can be fully vectorized. Unfortunately, though the average vector length has been increased to almost 64, the run time (Cray X1-vec) adversely becomes much longer.

One possible reason for the performance drop is the memory bank conflicts since the row length $\sqrt{n}$ is a power of two. In order to verify our guess, we pad each row with additional space so that the row length is no longer a power of two. The padding results, labeled as Cray X1-vec-pad in Fig. 5, indicate that the memory bank conflict is indeed the reason for the performance loss. After vectorizing and padding, the code can now deliver 16 times better performance on the Cray X1 than on the Power4 platform for the sequential case and at least five times better performance for large number of processors. Actually, we find that even the base version can benefit significantly from padding. By simply padding the matrix for the base version, the performance on the Cray X1 (Cray X1-pad) has been increased four times for the sequential case.

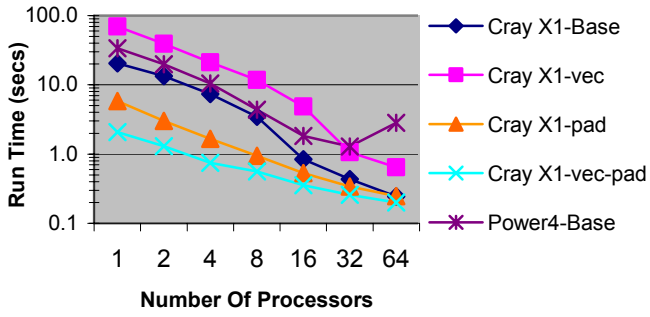

Figure 5. The Performance of FFT on Power4 and Cray X1.

5.2 Radix

Radix sort includes three phases: browsing the local data to compute the local histogram, communicating among all the processors to compute global histogram, and reassigning the data based on the global histogram. However, none of these three phases in the base version could be directly vectorized because of loop dependencies. For example, following is the code used for the first phase:

```
For (I = 0; I < N; I++) {
      key_val = key_from[I] & bb;
      key_val = key_val >> shift;
      bucket[key_val]++;
}
```

N is the number of data. *Key_from* is the source data array. *Bb* and *shift* are used to compute which buckets the data belongs to. *Bucket* is used to record the local histogram. The computation of the value for the *bucket* prevents the loop from parallelized for vector units on the Cray X1. We see in Fig. 6 that the base version runs much slower on Cray X1. For the sequential case, the run time is almost 4 times worse. This is mainly due to the slower CPU speed and the smaller cache size on Cray X1 in addition to the short average vector length. Both platforms can achieve almost linear speedups when the number of processors is less than 32. However, when the number of processors reaches 64, Cray X1 becomes better since the communication switch currently installed on the Power4 platform is not as efficient as that on Cray X1.

In order to vectorize the code, we borrowed the "virtual processor" concept from [8]. Each element of the vector register is viewed as a virtual processor. Each virtual processor is assigned a portion of the data and a set of independent buckets so that it can work exactly as a processor in the base version. The corresponding code for the first phase is changed to following:

```
For (j=0; j< VL; j++) {
  For (I=0; I < N / VL; I++) {
      key_val = key_from[j*N/VL+I] & bb;
      key_val = key_val >> shift;
      bucket_size[j*radix+key_val]++;
  }
}
```

VL is the number of virtual processors. The register size on Cray X1 is 64 elements and there are four SSPs in each MSP processor. Therefore, there are total 64*4 virtual processors available. The performance for the vectorized code is shown in Fig. 6, labeled as Cray X1-vec-64. Now the average vector length becomes 64. The performance is better than the base version but still significantly worse compared with its performance on the Power4. The problem is also related to memory bank conflicts. Using 256 as the number of virtual processors causes significant memory bank conflicts and substantially hurt the performance. Instead, if we use 63*4 as the number of virtual processors, the run time, labeled as Cray X1-vec-63, is 8 – 36 times better than its base version and 3 – 9 times better than Power4.

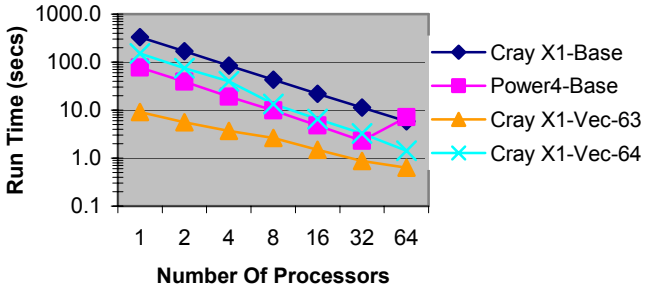

Figure. 6. The Performance of Radix on Power4 and Cray X1.

5.2 Nbody

The performance of the nbody application is shown in Fig. 7. Its base version also runs faster on the Power4 platform than on the Cray X1. There are three phases in each time step of a simulation. But, the main computation lies in the force calculation phase. In this phase, each body traverses the oct-tree starting from the root. If the distance between the body and a visited node is large enough, the whole subtree rooted at it, will be approximated by that node. Otherwise, the body will visit all the children of the node, computing their effects individually and recursively. Therefore, the path through which a body traverses the oct-tree is dynamically decided and different from each other. Because of the complexity and irregularity of the code, we have not been able to reform the code to enable the compiler to vectorize this whole process.

A partial solution is to separate this process into two stages: finding all the nodes affecting the body first (browsing stage) and computing the force effects later (computing stage). In this way, we can easily vectorize the second stage and leave only the first stage not vectorized. But the first stage is not intrinsically serial. It could be efficiently multi-streamed. The performance of the new version is shown in Fig labeled as Cray X1-opt. The performance improvement is limited to 20%. The average vector length is increased from 3.71 to 27.40. However, The code still runs slower on the Cray X1. By further analysis we find that the browsing stage is very time consuming and dominates the tree traversing process. For example, using the optimized version, the combined time of these two stages within a one-step simulation for 2 million bodies is 130 seconds. By removing the computing stage, the run time is only reduced to 122 seconds. The pointer-chasing operation on the Cray X1 platform is very expensive. It needs several instructions to form a 64-bit address.

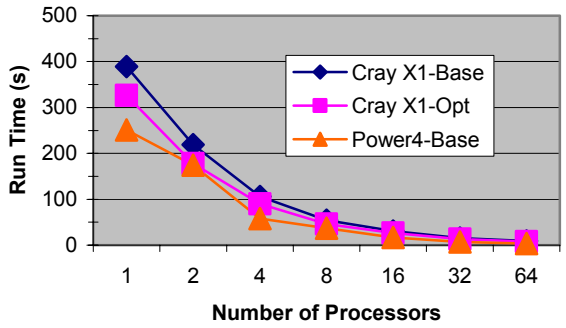

Figure 7. The Performance of Nbody on Power4 and Cray X1.

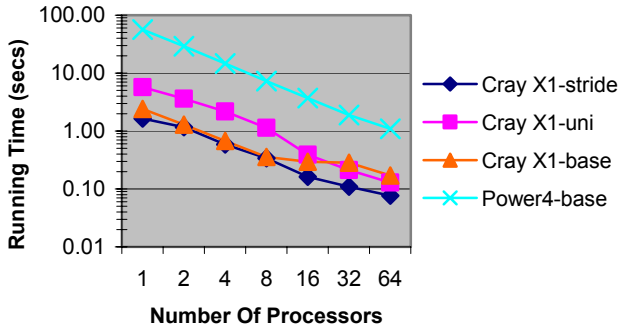

Figure 8. The Performance of Matrix Multiplication on Power4 and Cray X1.

5.4 Matrix Multiplication (MM)

The base version we used is a blocked matrix multiplication. On the cache-based superscalar platforms, block algorithms are often exploited to achieve better reuse of data in local memory. We select 16 as the block size on Power4 platform and 256 on Cray X1 individually. The run times are presented in Fig. 8. The Cray X1 runs 26 times faster for the sequential case and at least six times faster for the multiprocessor cases. The average vector length reaches 64. The advantages of vector units for such regular, vector-rich applications have been fully demonstrated.

Compared with non-blocked code, blocked code is usually much more difficult to develop and understand. It also introduces an artificial parameter, block size, that has nothing to do with the algorithm and has to be tuned on each specific platform. Therefore, we like to examine how the non-blocked code works on the Cray X1 platform since it prefers longer vectors. Suppose the matrix sizes for a and b are *M*N* and *N*K* respectively and the matrices are stored in one-dimensional arrays. There are two naïve ways to implement the matrix multiplication: stride and uni-stride:

Stride Implementation:

```
For (i=0; i < M; i ++) {
      For (j = 0; j < K; j++) {
         tmp = 0;
         For (k = 0; k < N; k++) {
                tmp += a[i*N+k] * b[k*K+j];
         }
         c[i*K+j] = tmp;
      }
}
```

Uni-stride Implementation:

```
For (i=0; i < M; i ++) {
      For (k = 0; k < N; k++) {
         For (j = 0; j < K; j++) {
                c[I*K+j] += a[i*N+k] * b[k*K+j];
         }
      }
}
```

The main difference between these two implementations is the order of the second and third loop. Both versions can achieve the same average vector length 64 on the Cray X1. And we expect the uni-stride version to deliver better performance. This is true on the cache-based Power4 platform. Surprisingly, the stride version performs at least two times better than the uni-stride version and also better than the blocked version. If we unroll the most outer loop four times, the stride version can deliver even slightly better performance than the vendor-provided *dgemm* function in the *sci* library. By the performance tools on the Cray X1 (PAT), we find that there are actually no vector load references with stride bigger than 2. The main difference between these two versions is the number of load/store references. The uni-stride version has M times more store references and eight time more load references since each time a c element has changed, it has to stored first and read back later. By examining the assembler code, we find that the characteristics of stride access have been changed due to vector processing. Fig. 9 illustrates this effect.

In order to compute $c_{0,0}$, we need to load the first row of matrix A and the first column of matrix B. Accessing $a_{0,0} \sim a_{0,n-1}$ is continuous access. However, visiting $b_{0,0} \sim b_{0,n-1}$ is stride accesses. Because of the vector effect, $c_{0,0} \sim c_{0,VL-1}$ are computed simultaneously. Therefore $b_{i,0} \sim b_{i,VL-1}$ should also be accessed at the same time. The result is that the stride access is replaced by continuous access. We only need to load matrix A and C one time unlike the uni-stride version where the matrix C has to be accessed many times.

In summary, the tuning results of FFT and Radix indicate that increasing the vector length and avoiding memory bank conflicts at the same time indeed improve the performance on the vector machine substantially. This conclusion is also validated by the negative nbody result. If a code cannot be vectorized well, there is no way to expect good performance from vector machines. Matrix multiplication tells us that vector processing may benefit from some access patterns that we try to avoid on cache-based superscalar architectures.

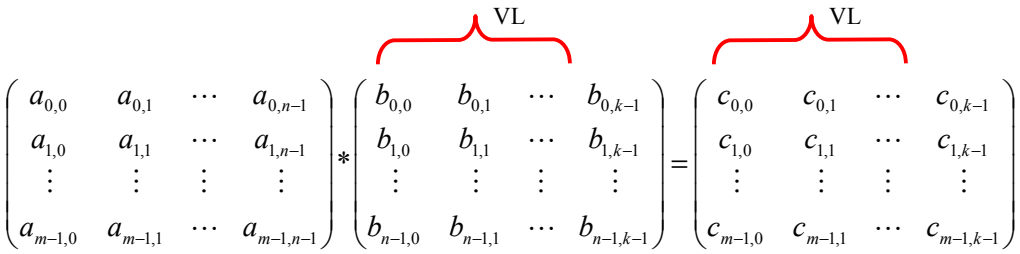

Figure 9. Memory Access for Stride Matrix Multiplication.

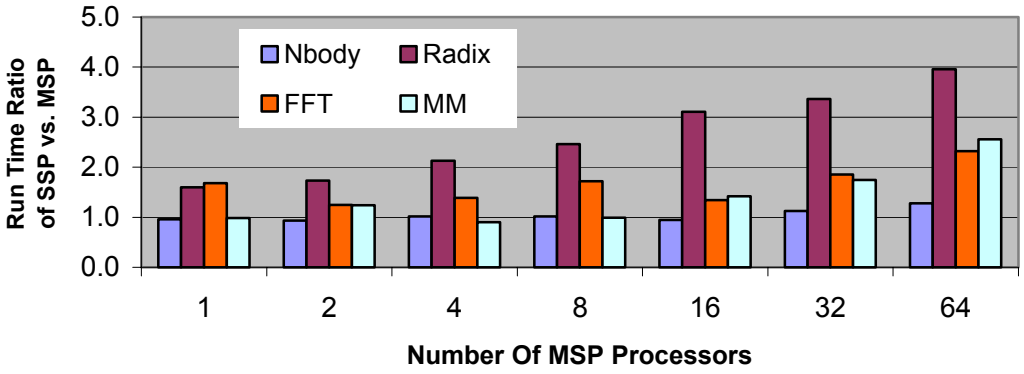

Figure 10. The Run time Ratio of SSP vs. MSP on Cray X1.

5.5 MSP vs. SSP

On the Cray X1, each MSP processor contains four identical SSPs that share a 2-way set associative 2MB cache. Programming in MSP mode is similar to developing codes on SMP-based platforms using the OpenMP model inside a SMP node and the MPI programming model across the nodes. But the compiler on the Cray X1 can automatically distribute loop iterations across each of the four SSPs and generate either SMP or MSP executable codes from the same source code. It does not require the programmer to provide explicit OpenMP directives so the programmer can freely select either MSP mode or SSP mode based on the application characteristics to achieve better performance without modifying the source code.

Fig. 10 presents the run time ratio of SSP over MSP using the best version we presented above. The number of processors on the X-axis is the number of MSP processors. Each MSP processor contains four identical SSP processors. We find that only for a few cases, the SSP mode performs slightly better than the MSP mode. In most cases, its performance falls behind that of the MSPs, especially when the number of processors goes up. In the worst case, it causes a four-time performance slowdown. The main reason for the performance drop is that the SSP mode will increase the number of messages each processor has to process and reduce the size of each message.

5.6 Optimization Impact on the Power4 platform

It is interesting to examine whether the optimizations that are beneficial on Cray X1 are portable to the Power4 platform or if they are unique to the Cray X1. The sequential results for the base version and optimized version on Power4 are exhibited in Fig. 11. The run times of the optimized version are either similar to their base version (Nbody, FFT, Radix) or substantially longer.

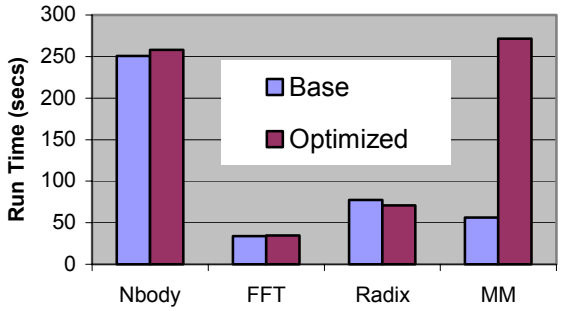

Figure 11. Effects of Cray X1 Optimizations on Power4.

The optimizations for these four are unique to the Cray X1 and cannot be ported back to the Power4 platform. This is different from the results reported from [12]. The vectorized code runs 10~20% faster there when ported back to Power4 platforms. The reason is that the optimized code brought more loop level parallelism in the leaf subroutines, and the Power 4 had more useful flops to do rather than branches. Therefore, the performance impact of vectorized code on Power4 is closely related with application characteristics.

6. CONCLUSIONS

In this paper, we have investigated the performance characteristics of the Cray X1 using Apex-Map, a memory access probe. We find that the most important factors, which affect the performance on this platform, are the average vector length and memory bank conflicts. The memory size accessed and data reuse have a much smaller effect. Using these characteristics as a guide for performance-tuning, we are able to substantially improve the performance of several applications. The performance of FFT has been improved 10 times for the sequential case and 1.2 times for the 64-processor case. When the number of processors is high, the transpose in the FFT starts to dominate the execution time, leading the performance to drop. Radix has been improved 36 times for the sequential case and 9 times for the 64-processor case. However, for nbody, the performance gain is only 20%. This limited improvement is mainly due to unsuccessful vectorization. For such dynamic, irregular applications, the code becomes more complex and irregular; vectorizing the code becomes challenging. For matrix multiplication, we use a strided version, which is intentionally avoided on Power4 platforms, to improve the performance 1.5 ~ 2.2 times. The optimized version can achieve 85% of the peak performance on a single SMP node and even runs slightly faster than vendor provided *dgemm* library routine.

Adding compiler directives can significantly improve the performance because in many cases the compiler cannot perform an adequate dependency analysis on its own. The compiler directives can explicitly inform the compiler of the data independence between loop iterations. However, adding directives itself cannot deliver optimal performance since directives do not help to exploit the data parallelism across loops or functions. For all our four applications, we have to substantially change the code to change data structures or algorithms in order to help the compiler to find the parallelism. For FFT, we have padded the data structure to eliminate the memory bank conflicts and change the loop structure to increase the average vector length. For Radix, we use the virtual processor concept to reorganize the code. In nbody, we separate the tree browsing and force calculation operations. For matrix multiplication, we use strided access. Unfortunately, we find that these optimizations on Cray X1 cannot be ported back to Power4 platform as well as some other applications.

7. ACKNOWLEDGMENTS

The authors thank ORNL for providing access to the Cray X1 and Power4 platforms. All authors are supported by the Office of Advanced Scientific Computing Research in the U.S. DOE Office of Science under contract DE-AC03-76SF00098. The authors would also like to thank James Schwarzmeier for his valuable discussion.

8. REFERENCES

[1] Gordon Bell and Jim Gray, *What's Next in High-Performance Computing?* Communications of the ACM, February 2002, Vol. 45 No. 2.

[2] T.H.Dunigan, M.R. Fahey, J.B.White, P.H. Worley, *Early Evaluation of the Cray X1*, SC2003, Phoenix, AZ. Nov. 2003.

[3] Leonid Oliker, Rupak Biswas, Julian Borrill, Andrew Canning, Jonathan Crater, M. Jahed Djomehri, Hongzhang Shan, and David Skinner, "A Performance Evaluation of the Cray X1 for Scientific Applications", 6th International Meeting on High-Performance Computing for Computational Science, Valencia, Spain, June 28-30, 2004.

[4] Apex-Map, Application Performance Characterization – Memory Access Probe. See http://ftg.lbl.gov.

[5] Hongzhang Shan, Jaswinder Pal Singh, *Comparison of Message Passing, SHMEM and Cache-coherent Shared Address Space Programming Models on the SGI Origin 2000*, International Conference of Supercomputing, May 1999, Rhodes Island.

[6] Hongzhang Shan, Leonid Oliker, Rupak Biswas, Jaswinder Pal Singh, *Comparing Three Programming Models for Adaptive Applications on SGI Origin 2000*, Supercomputing, 2000, Dallas, Texas.

[7] David H. Bailey, *FFTs in External or Hierarchical Memories*, Journal of Supercomputing, 4:23-25, 1990.

[8] Marco Zagha, Guy E. Blelloch, *Radix Sort For Vector Multiprocessors*, SC1991, Albuquerque, New Mexico.

[9] S. Shingu et al, *A 26.58 Tflops Global Atmospheric Simulation With The Spectral Transform Method On the Earth Simulator*, in Proceeding SC2002, Baltimore, MD, 2002.

[10] Wayne R. Cowell and Christopher P. Thompson, *Transforming FORTRAN DO loops to improve performance on vector architectures*, ACM Transactions on Mathematical Software, Vol.2 Issue 4, Pages 324-353, 1987.

[11] H. Cheng, *Vector Pipelining, Chaining, and Speed on the IBM 3090 and Cray X-MP*, IEEE Computers, 22(9):31-46, sep 1989.

[12] Hoffman, Forrest M., Trey White, and Mariana Vertenstein. June 25, 2003. *Vectorizing the CLM: Progress and Plans*. Land Model Working Group Meeting at the Community Climate System Model (CCSM) Annual Meeting, Breckenridge, Colorado. http://climate.ornl.gov/clm/presentations/20030625/