Data Centric Cache Measurement on the Intel Itanium 2 Processor

Bryan R. Buck[1]

Platform Logic
3060 Route 97
Glenwood, MD 21738
buck@cs.umd.edu

Jeffrey K. Hollingsworth

Computer Science Department
University of Maryland
College Park, MD 20742
hollings@cs.umd.edu

Abstract

Processor speed continues to increase faster than the speed of access to main memory, making effective use of memory caches more important. Information about an application's interaction with the cache is therefore critical to performance tuning. To be most useful, tools that measure this information should relate it to the source code level data structures in an application. We describe how to gather such information by using hardware performance counters to sample cache miss addresses, and present a new tool named Cache Scope that does this using the Intel Itanium 2 performance monitors. We present experimental results concerning Cache Scope's accuracy and perturbation of cache behavior. We also describe a case study of using Cache Scope to tune two applications, achieving 24% and 19% reductions in running time.

1 Introduction

Increases in processor speed continue to outpace increases in the speed of access to main memory. Because of this, it is becoming ever more important that applications make effective use of memory caches. Information about an application's interaction with the cache is therefore crucial to tuning its performance. This information can be gathered using a variety of instrumentation techniques that can be broadly categorized as software techniques and techniques that use hardware performance monitoring support.

All-software approaches are more flexible. For instance, a simulator can be made to provide almost any kind of information desired, depending only on the level of detail and fidelity of the simulation. However, simulation can be slow, often prohibitively so. Hardware performance monitors allow data to be gathered with much lower overhead, with the tradeoff that the types of data that can be collected are limited to those the system's designers decided to support.

To be most useful to a programmer in manually tuning an application, information about cache behavior should be presented in a way that relates it to program data structures at the source code level. We refer to this as data centric cache information.

Relating cache information to data structures requires not only counting cache-related events, but also determining the areas of memory that are associated with these events. In the past, this has been difficult to accomplish using hardware monitors, due to limited support for gathering such data. As an example, processors that include support for counting cache misses have often not provided any way to determine the addresses that were being accessed to cause them.

The situation is now changing. Several recent processors include increased support for performance monitoring. Many processors have for some time included ways to count cache misses, and to trigger an interrupt when a given number of events (such as cache misses) occur. Some recent processors provide the ability to determine the address that was accessed to cause a particular cache miss; by triggering an interrupt periodically on a cache miss and reading this information, a tool can sample cache miss addresses. One such processor is the Intel Itanium 2 [3]. This paper presents a tool named Cache Scope that runs under Linux on the Itanium 2 and uses these hardware features to collect data centric cache information.

2 Cache Miss Address Sampling

In order for a tool to relate cache misses to data structures, it must be able to determine the addresses that were accessed to cause those misses. However, running instrumentation code to read and process these addresses every time a cache miss occurs is likely to lead to an unacceptable slowdown in the application being measured.

One solution to this problem is to sample the cache misses. This can be accomplished with the hardware counters on some processors. For instance, many processors provide a way to count cache misses, and a way to cause an interrupt when a hardware counter overflows. By setting an initial value in the counter for cache misses, we can receive an interrupt after a chosen number of misses have occurred. We also need for the processor to identify the address that was being accessed to cause the miss. This is necessary because the interrupt that occurs when a

[1] The work described in this paper was performed while Bryan R. Buck was a student at the University of Maryland.

0-7695-2153-3/04 $20.00 (c)2004 IEEE

cache miss counter overflows is typically not precise. On modern processors with features such as multiple instruction issue and out-of-order execution, the point at which the execution is interrupted could be a considerable distance from the instruction that actually caused the miss. For instance, on the Itanium 2 the program counter could be up to 48 dynamic instructions away in the instruction stream from where the event occurred [3]. Other processor state, such as registers, may also have changed, making it difficult or impossible to reconstruct the effective address accessed by an instruction, even if the correct instruction could be located.

A further argument for sampling is that on many processors that provide the features described above, it is not possible to obtain the address of every cache miss. For instance, on the Intel Itanium 2 [3] and IBM POWER4 [28], a subset of instructions are selected to be followed through the execution pipeline. Detailed information such as cache miss addresses is saved only for these instructions. This is necessary in order to reduce the complexity of the hardware counters.

Given the hardware support described above, it is possible to maintain sampled statistics about the cache misses taking place in an application's data structures. We associate a set of statistics with each variable or dynamically allocated block of memory (or group of related blocks). We then set the hardware counters to generate an interrupt after some chosen number of cache misses. This number should be varied throughout the run, in order to prevent the sampling frequency from being inadvertently synchronized to the access patterns of the application. When the interrupt occurs, an interrupt handler reads the address of the cache miss from the hardware, matches it to the object in memory that contains it, and updates the statistics for that object. After processing the current sample, the entire process is repeated.

The mapping of addresses to objects is performed for program variables by using the debug information in an executable. For dynamically allocated memory, we instrument the memory allocation routines to maintain the information needed to perform the mapping.

If the number of misses sampled for each object is proportional to the total number, then at the end of a run the statistics gathered will provide the programmer with an accurate idea of which program objects are experiencing the worst cache behavior.

3 Itanium 2 Performance Monitoring

The Itanium 2 is a VLIW processor with many features for speculation and for making use of instruction level parallelism [16, 27]. It is an implementation of the IA-64 architecture, which was developed by Intel and Hewlett-Packard.

The Itanium 2 features four 48-bit performance counters that can be set to monitor over a hundred different events. These counters can be configured so that they can be read from user mode or so that they must be read only from privilege level zero (kernel mode). The registers used to configure which events will be counted and to perform other control functions are accessible only in privilege level zero.

The Itanium 2 provides a performance monitor overflow interrupt and writable performance counters, which can be used together to trigger an interrupt after a chosen number of cache misses.

In the Itanium 2, data address support is provided by the Event Address Registers (EARs). These registers provide address and other information about events taking place in the cache and TLB. The Data EAR records information about L1 data cache misses, data TLB misses, and ALAT misses. This information includes the address that was accessed, the instruction that performed the access, and the latency of the miss (defined as the number of cycles the instruction was in flight). The Instruction EAR records similar information about instruction cache and TLB misses. The EAR registers can also be set to monitor only events with a given latency or higher.

In the case of data cache load misses, the processor must track load instructions as they pass through the pipeline in order to determine the information recorded by the Data EAR. The processor can track only one instruction at a time, so not all miss events can be recorded by the Data EAR; while it is tracking one load, all others are ignored. The processor randomizes which load to track, in order not to skew sampling results.

On the Itanium 2, the L1 data cache handles only integer loads, so all floating point loads go to the L2 cache and may be sampled by the Data EAR. As a result, the Data EAR mode that tracks L1 data cache load misses also tracks floating point loads.

4 Linux Monitoring Interface

Access to the performance monitors under Linux is through the "perfmon" kernel interface [4], which is part of the standard Linux kernel for IA-64. A kernel interface is needed because the performance monitors can only be controlled from privilege level zero.

Perfmon virtualizes the counters on a per-process basis. A program can choose between monitoring events system-wide or for a single process. In order to accomplish this, perfmon must be called from the context-switch code, and for this reason it was made a part of the kernel, not an installable device driver.

Perfmon also provides support for randomizing the interval between counter overflows. The user specifies a mask that will be anded with a random number, with the result being added to the number of events that will pass before an overflow.

5 Cache Scope

We implemented a tool named Cache Scope that gathers data-centric cache information using the Itanium 2 performance counters. The tool consists of a program that adds instrumentation code to the application that is to be measured, and an analysis program that allows a user to examine the data that was gathered. These are described below.

5.1 Instrumentation for Sampling Cache Misses

The part of Cache Scope that gathers the data-centric cache information is a program named "cscope." The user gives as parameters to cscope the name of and arguments for an application to be measured, and cscope starts the application and uses the Dyninst API [8] run-time instrumentation library to load code into it that will perform the cache measurement[2]. It also instruments memory allocation functions in order to track dynamic memory allocations.

Optionally, the user can link the application with a library named libcscope, which contains functions that can be called to interact with the instrumentation. For instance, calls are provided to control what part of the execution will be monitored. When an application that uses these calls is run outside of cscope, the calls do nothing.

The instrumentation code uses perfmon to set the Itanium 2 hardware performance monitors to count L1 data cache read misses, L1 data cache reads, L2 cache misses, and Data EAR events. The Data EAR is set to record information about L1 data cache load misses and floating point loads.

The overflow interrupt is enabled for the counter counting Data EAR events (cache misses and floating point loads). The number of Data EAR events between interrupts is controllable by the user, by setting an environment variable before executing the program to be measured. By default, this value is randomly varied throughout the run to ensure a representative sampling of events. When the interrupt occurs, the instrumentation code takes a sample by reading the address that was being accessed, the address of the instruction that caused the event, and its latency. It then updates the data structures for the appropriate memory object (as described below) and restarts the counters. When restarting the counters, if randomization has been enabled, then the instrumentation code uses the randomization feature of perfmon to vary the sampling interval.

For purposes of keeping statistics, memory objects are grouped into equivalence classes, which we refer to as *stat buckets*. Each global or static variable in the program is assigned its own stat bucket. When a block of memory is dynamically allocated, a bucket name is either automatically generated or is supplied by the user, as described below; this name identifies the bucket to which the block is assigned. Different blocks may have the same bucket name, so that multiple blocks are assigned to a single bucket. This is useful when a group of blocks are part of the same data structure, as in a tree or linked list. Automatically assigned names are generated based on the names of the top three functions on the call stack above the memory allocation function that allocated the object. Explicit bucket names are assigned by the user, by replacing the call to an allocation function with a call to a routine in Cache Scope's libcscope library that takes an extra parameter, which is the bucket name to assign to the block.

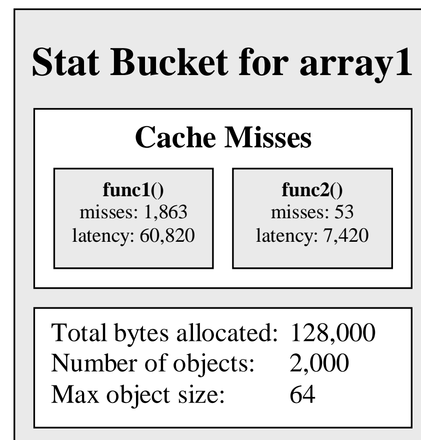

Figure 1: Bucket Data Structure

The information stored in a bucket is illustrated in Figure 1. Most importantly, a bucket keeps a count of cache events (L1 data cache misses and floating point loads) and of the latency associated with them. This information is split up by the functions in which the cache events occurred, adding code centric information to supplement the data centric information we are primarily concerned with. This data is kept in a vector, with an entry for each function in which cache events have occurred for this bucket. The vector data structure was chosen to conserve memory, since the entries do not require pointers to parents or children, as they would in a tree structure. This is necessary because there are potentially a very large number of combinations of memory objects and functions that may occur in a run. The vector is sorted by a unique number that identifies the function associated with each entry, so the entry for a function can be found using a binary search. While faster schemes are certainly

[2] An earlier version of Cache Scope, which was used to collect most of the data for this study, did not use the Dyninst API. Instead, it required the user to link a measurement library with the application and to manually insert calls into the application to start and stop measurement. This version of the tool and the Dyninst version use the same instrumentation code to sample cache events.

possible, the overhead of using this data structure has not been a problem. In addition to the cache information, each bucket also contains various statistics such as the number of objects assigned to the bucket and their sizes.

5.2 Data Analysis Tool

When measurement has completed, Cache Scope writes all the data it collected out to a file in a compact format. This file can be read in by an analysis program called cscope_view. Cscope_view is written in Java, and so is portable to any system for which Java is available.

Figure 2 shows an example of cscope_view's interface, displaying cache events in the application mgrid. The user can view tables of the objects or functions causing the most latency, and can combine the data and code centric data by filtering by function or object. For instance, by choosing a function from the list box on the left when viewing by object, a user can see a table of the data structures experiencing the most latency in the chosen function. Note that the tool presents information in terms of the latency associated with sampled events, rather than simply counts of cache misses. For each object or function, the tool shows the absolute number of cycles of latency, the percentage of total latency, and the latency per event. The absolute value for latency is based on multiplying the sum of the sampled latencies by the sampling frequency. Since not all instructions are tracked by the Data EAR (as described in Section 3), this will be lower than the actual value, but it is useful in comparing runs (for instance, of an unoptimized versus an optimized version of an application).

Figure 2: Cscope_view Interface

Cscope_view is also able to provide information about the non-cache-related statistics that are kept by the instrumentation code, such as the number of allocated memory objects that belong to a given bucket, and the size of those objects. This can be useful in tuning cache performance, as will be seen in the examples in Sections 7.1 and 7.2.

6 Experiments

We ran a series of experiments in which we used Cache Scope to measure the cache misses in a set of applications from the SPEC CPU2000 benchmark suite.

The applications used in the experiments were wupwise, swim, mgrid, applu, gcc, mesa, art, mcf, equake, crafty, ammp, parser, gap, and twolf. They were compiled using gcc 3.3.3. We ran each application a number of times while sampling cache misses at different rates, in order to examine the effect of varying this parameter. The rates given are averages; actual number of events between samples was randomly varied throughout the run, using the randomization feature of the perfmon kernel interface, which was described in Section 4. For tests in which we did not vary the sampling frequency, we chose 1 in 32K events as our default rate. Since the hardware counters cannot provide information about all cache events, this study did not directly examine the accuracy of sampling at the rates used. For information about accuracy, see the authors' paper from SC2000 [9], which describes results gathered under a simulator that allowed a comparison of sampled data with exact information.

We also ran tests in which we did not sample cache misses, but used the hardware counters to gather various overall statistics to be compared with the runs in which sampling was performed. The only statistics gathered in these runs were those that could be measured with almost no overhead, by starting the counters at the beginning of execution and reading their values at the end, with no interrupts while the applications were running.

The results presented are averages over three runs of each application. The following sections describe the data obtained from these experiments.

6.1 Perturbation of Results

Figure 3 shows the increase in L2 cache misses seen in each application we tested when sampling at the rates shown in the legend. This increase is over the number of cache misses observed when no sampling was performed. Striped bars represent negative values with the absolute value shown. Note that the scale of the y axis is logarithmic. We are concerned primarily with the L2 cache because most optimization will probably be for this level. This is because of the fact that the L1 data cache does not store floating point values, and the penalty for going to the L2 cache is small, as low as five cycles for an integer load and seven cycles for a floating point load [21]. These, combined with the fact that the L1 cache is only 16KB while the L2 is 256KB, make the L2 cache more significant to performance. As an example of this, we saw in our experiments variations in L1 cache misses between runs that did not translate into significant variations in running time.

The increase in L2 cache misses for most applications was relatively small except at the two highest sam-

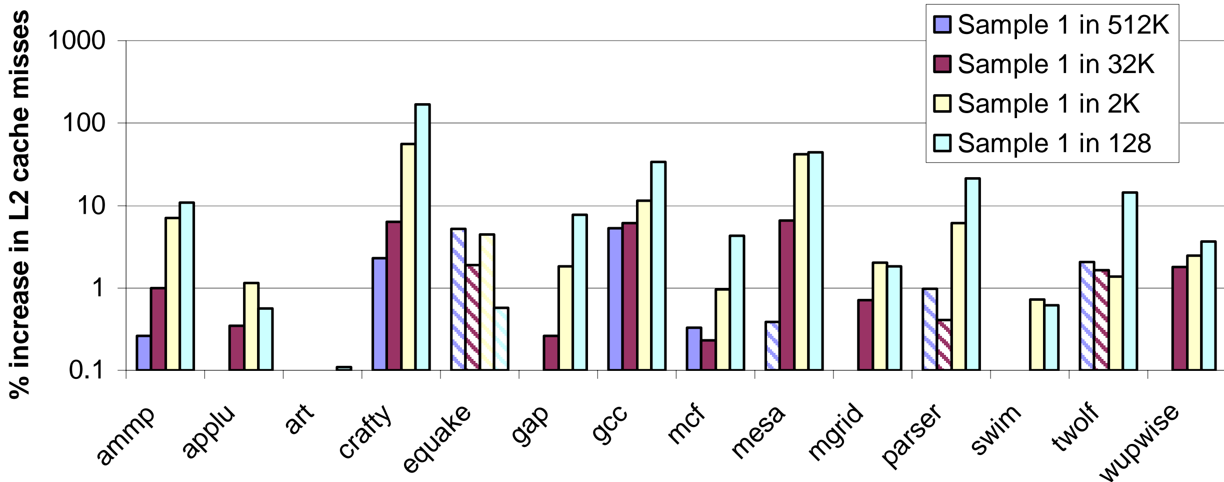

Figure 3: Increase in L2 Cache Misses

pling frequencies, and as will be discussed below, some applications actually showed a decrease in cache misses in the sampled runs.

When sampling 1 in 512K events, the highest increase in misses was seen in gcc, which had an approximately 5.3% increase. One feature of this application that differentiates it from most of the others is that it frequently allocates and deallocates memory in the heap. Therefore, we suspected that the instrumentation code that maintains the map of dynamically allocated memory may be the cause of the cache disruption. In order to test this possibility, we reran gcc with the code that maintains the dynamic memory map, but without doing any sampling. These runs produced an average increase in L2 cache misses of 6.9%, which was very close to and actually slightly higher than the 5.3% we saw when sampling. Therefore, we conclude that the increase in cache misses is primarily due to the code for maintaining the map of dynamically allocated memory. After gcc, the next highest increase in L2 misses was seen with crafty, with a 2.3% increase.

At a sampling frequency of 1 in 32K events, the highest increases in cache misses are seen in gcc, with a 6.1% increase, crafty with 6.3%, and mesa, with 6.5%.

As we increase the sampling frequency, we see increases in cache misses as high as 168%, seen when running crafty while sampling 1 in 128 cache misses. This shows that increasing the sampling rate does not necessarily lead to increased accuracy, due to the instrumentation code significantly affecting cache behavior.

As noted above, some applications showed a small decrease in cache misses when running with sampling as compared to runs without. The largest of these was seen in equake, which showed a decrease in L2 cache misses of 5.3% when sampling 1 in 512K events. This is likely due to the fact that the instrumentation code allocates memory, which can affect the position of memory blocks allocated by the application. It was observed by Jalby and Lemuet [18] that for a set of applications they examined running on the Itanium 2, factors such as the starting addresses of arrays had a significant effect on cache behavior. In Section 7.1, we present data from equake and how the data was used to perform optimizations; while we were doing this work, we found equake to be particularly sensitive to this phenomenon.

6.2 Instrumentation Overhead

Figure 4 shows the percent increase in running time for each application when sampling at the frequencies shown in the legend. This increase is over the running time of the application with no sampling instrumentation. The scale of the y axis is logarithmic, with striped bars representing negative values with the absolute value shown. The overhead measured includes all instrumentation, both for sampling cache miss addresses and for tracking dynamic memory allocations.

For the two lowest sampling frequencies tested, the overhead was acceptable for all applications. Looking at the higher of these frequencies, sampling 1 in 32K events, the overhead was less than 1% for ten out of the fourteen applications tested. The remaining applications were ammp and swim with overheads between 1 and 2%, gcc with an overhead of approximately 2.3%, and equake with an overhead of approximately 7.7%.

One reason for the higher overhead in equake appears to be the number of memory objects it allocates. More memory objects lead to a larger data structure that the instrumentation code must search through in order to map an address to a stat bucket. Equake allocates 1,335,667 objects during its run, while the application with the next highest number of allocations, twolf, allocates 575,418. Eight of the thirteen applications allocate approximately 1,000 or fewer objects.

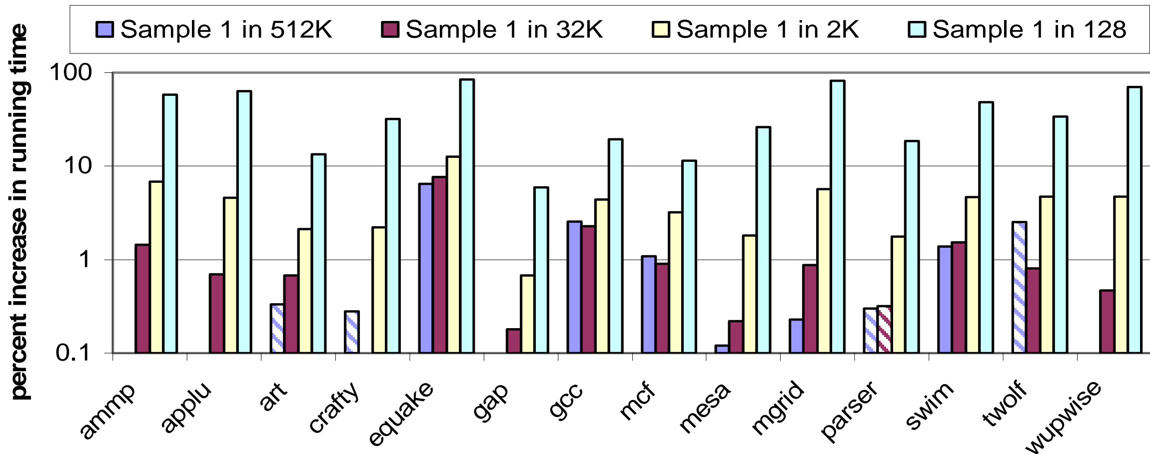

Figure 4: Instrumentation Overhead

7 Case Studies

This section will examine two example applications from the SPEC CPU2000 benchmarks, equake and twolf. We used Cache Scope to analyze these applications and to tune their performance by improving their utilization of the cache. We will follow this analysis and optimization step by step, to demonstrate how the tool allowed us to quickly locate where the applications were losing performance due to poor cache utilization, and to determine how data structures could be changed to enable better cache use.

We were able to achieve significant gains in performance for both applications, showing the usefulness of the information provided by the tool. This was accomplished in a short time; one day for equake, and a few days for twolf. The programmer optimizing the applications (one of the paper's authors) had no prior familiarity with the source code of either application, and relied entirely on the cache tool to determine what data structures and code to focus on for tuning. Furthermore, the optimizations we will describe consisted almost entirely of changes to the data structures in the applications, with few changes to the code.

7.1 Equake

We will first examine equake. This is an application that simulates seismic wave propagation in large valleys, and was written by David R. O'Hallaron and Loukas F. Kallivokas [6, 15]. It takes as input a description of a valley and seismic event, and computes a history of the ground motion. It performs the computations on an unstructured mesh using a finite element method.

We ran this application with Cache Scope to measure its cache behavior. The first interesting piece of information this provided was that the hit ratio of loads in the L1 data cache is only about 64% (this is the ratio of L1 data cache hits to loads that are eligible to be cached in the L1 data cache). This low hit ratio suggests that cache behavior could be a performance problem in this application, although it is important to remember that on Itanium 2 the L1 cache is not used for floating point loads.

Another piece of information that the tool can provide is the average latency of a cache miss (or floating point load). For equake, this is 21.3 cycles. This is high relative to most of the other applications tested. Average latency ranged from 6.5 cycles for mgrid to 73.2 cycles for mcf; however, only three of the applications tested had average latencies greater than equake's. This is another indication that the application may be losing performance due to poor cache utilization.

Next we will look at which data structures in the application are causing the most latency. As noted in Section 5.2, Cache Scope returns information in terms of latency rather than the number of cache misses. It is important to note that the latency does not indicate the number of cycles the processor is stalled waiting for a load. The Itanium 2 performance monitor defines the latency of a load instruction as the number of cycles the instruction is in flight. Multiple loads may be outstanding at any given time, and instruction level parallelism may allow the processor to continue executing other instructions while waiting for data.

Table 1 shows the stat buckets in the application that cause the most latency, sorted by latency. As described in Section 5.1, a stat bucket represents a data structure in memory. It can be a global or static variable, a block of dynamically allocated memory, or a number of related blocks of dynamically allocated memory. For this run, we allowed the tool to automatically group dynamically allocated memory into stat buckets. When this option is selected, Cache Scope will generate a stat bucket name for each allocated block of memory based on the names of the last three functions on the call stack above the actual memory allocation function. Blocks that have the same

stat bucket name are grouped together. In this example, there are two automatically named buckets, mem_init-main and readpackafile-main.

Stat Bucket	Latency		
	Cycles (billions)	%	Per Event
mem_init-main	97.18	95.6%	24.0
Exc	2.10	2.1%	5.4
<stack>	1.12	1.1%	5.0
readpackfile-main	0.68	0.7%	102.1
<unknown>	0.55	0.5%	5.0

Table 1: Data Structure Statistics for Equake

The "Cycles" column shows an estimate of the absolute number of cycles of latency caused by objects assigned to the given stat bucket, in billions. This is computed by multiplying the recorded cycles for each sample by the sampling frequency. The "%" column shows the percentage of all latency in the application that was caused by the stat bucket. "Per Event" shows the average latency for the cache events caused by the bucket.

The <stack> bucket represents the stack. The tool includes an option to create a separate bucket for the stack frame of each function, but this creates additional overhead and was not used in this run. <unknown> represents all memory that is not associated with a known object. This includes such objects as memory used by runtime libraries that do not have debug information. As mentioned above, the remaining two buckets are automatically named, indicating that they were allocated from the functions mem_init and readpackfile, both of which were called from main.

The most important bucket is obviously mem_init-main, which causes 95.6% of the latency in the application. Looking at the function mem_init, we see that it uses malloc to allocate a large number of arrays. It would be useful to break this down further, by the individual arrays or groups of related arrays. As described in Section 5.1, the user can explicitly name a bucket by providing a string name to a special allocation function. We did this for the memory allocation calls in mem_init.

Table 2 shows the results of re-running the tool with explicitly named buckets. The buckets with names beginning with "heap_" represent arrays or groups of arrays that were dynamically allocated with explicit bucket names. Those shown are all allocated by the function mem_init. Most of these are parts of a set of two- and three-dimensional matrices, where each matrix is made up of a group of independently allocated blocks of memory. The reason they are allocated in this way is to allow their sizes to be determined at runtime, while still making them easy to use in C. An example of this is the array "disp," which is declared as "double ***disp." The elements of disp are then allocated as in the abbreviated code from equake shown in Figure 5.

Stat Bucket	Latency		
	Cycles (billions)	%	Per Event
heap_K_2	23.67	35.4%	118.5
heap_disp_3	18.41	27.5%	10.7
heap_K_3	7.49	11.2%	5.4
heap_disp_2	3.47	5.2%	22.8
Exc	2.13	3.2%	5.5
heap_M_2	1.53	2.3%	15.2
heap_C_2	1.49	2.2%	14.7
<stack>	1.12	1.7%	5.0
heap_M_1	0.95	1.4%	23.9
heap_K_1	0.93	1.4%	80.0

Table 2: Data Structure Statistics in Equake

The advantage of this is that the matrix can be accessed using the syntax disp[i][j][k]. The only way to use this syntax without the multiple indirections is to declare the size statically.

The numbers at the end of the names in Table 2 show which matrix dimension each stat bucket is associated with. For the first two dimensions, these are arrays of

```
double ***disp;

/* Displacement array disp[3][ARCHnodes][3] */

disp = (double ***) malloc(3 * sizeof(double **));

for (i = 0; i < 3; i++) {
       disp[i] =
         (double **) malloc(ARCHnodes * sizeof(double *));

       for (j = 0; j < ARCHnodes; j++) {
              disp[i][j] =
                (double *) malloc(3 * sizeof(double));
       }
}
```

Figure 5: Memory Allocation in Equake

pointers to the arrays that make up the next dimension. For the third dimension, the arrays contain the actual data. The second dimension of K, heap_K_2, causes 35.4% of the total latency. The third dimension of the arrays disp and K, heap_disp_3 and heap_K_3, together cause approximately 38.7% of the total latency. Using the code features of our tool shows that the vast majority of these misses take place in the function smvp. Nearly 100% of the latency when accessing heap_K_2, 69.6% percent when accessing heap_disp_3, and 99.8% when accessing heap_K_3 take place in this function. Smvp computes a matrix vector product, and contains a loop that iterates over the matrices.

One potential problem here is that the size of the individual arrays that make up these buckets is very small. Heap_K_2 contains arrays of three pointers, while heap_K_3 and heap_disp_3 contain arrays of three doubles. Therefore, each of these arrays is only 24 bytes long. This can easily be seen using the cscope_view tool, which can show statistics about the sizes of the blocks of memory that make up each stat bucket. When malloc creates a block of memory, it reserves some memory before and after the block for its own internal data structures. Since the L2/L3 data cache line size on the Itanium 2 is 128 bytes, we could pack 5 of the arrays that make up heap_K_2, heap_K_3, and heap_disp_3 into a single cache line, but this does not happen due to the overhead of malloc. To improve this situation, we modified the program to allocate one large contiguous array to hold all the pointers or data for each dimension matrix, and then set the pointer arrays to point into them. This modified code is shown in Figure 6.

This change decreases L1 cache misses in the application by 57%, L2 cache misses by 30%, and running time by 10%. The results of re-running Cache Scope on the new version of the application are shown in Table 3.

Stat Bucket	Latency		
	Cycles (billions)	%	Per Event
heap_K_3	14.89	49.4%	22.5
heap_disp_3	7.43	24.7%	9.1
heap_K_2	1.32	4.4%	52.4
mem_init-main	1.17	3.9%	15.8
heap_disp_2	1.11	3.7%	35.8
Exc	1.01	3.4%	5.5
heap_C_2	0.64	2.1%	13.4
<stack>	0.53	1.8%	5.0

Table 3: Results for Optimized Equake

The absolute amount of estimated latency for heap_K_2 is reduced by approximately 94%, and for heap_disp_3 it is reduced by 40%. The latency for heap_K_3 has almost doubled, but this is more than made up for by the gains in the other two buckets. Note that this optimization not only improves latency, but lowers the required bandwidth to memory as well, since more of each cache line fetched is useful data, rather than overhead bytes used by malloc for its internal data structures.

The arrangement of K and disp each into two pointer arrays (for example, heap_K_1 and heap_K_2) and a data array (heap_K_3) continues to be a source of latency. The heap_K_2 bucket is causing 4.4% of the latency in the application, and heap_disp_2 is two places below it with 3.7%. These misses could easily be avoided by eliminating the need for those arrays entirely. If we are willing to accept statically sized matrices, we could simply declare disp and K as three-dimensional arrays.

Table 4 shows the results of making this change. Note that the latency for K is significantly less than the

```
double ***disp;

double *disp_3;
double **disp_2;
double ***disp_1;

disp_3 = malloc(3 * ARCHnodes * 3 * sizeof(double));

disp_2 = malloc(ARCHnodes * 3 * sizeof(double *));

disp_1 = dctl_mallocn(3 * sizeof(double **));

disp = disp_1;

for (i = 0; i < 3; i++) {
       disp[i] = &disp_2[i*ARCHnodes];

       for (j = 0; j < ARCHnodes; j++) {
       disp[i][j] = &disp_3[i*ARCHnodes*3 + j*3];
       }
}
```

Figure 6: Modified Memory Allocation in Equake

latency for heap_K_3, where the actual data for the array was previously stored. This is probably because eliminating the pointers in heap_K_1 and heap_K_2 freed a large amount of space in the L2 cache that could then be used for the actual data. In addition, the compiler is more likely to be able to prefetch data, since the location of the data is computed rather than read from pointers. All of this is also true for the other main array, disp.

Stat Bucket	Latency		
	Cycles (billions)	%	Per Event
K	6.84	53.4%	21.8
disp	3.50	27.3%	8.9
Exc	0.37	2.9%	5.5
heap_M_2	0.31	2.4%	13.7
heap_C_2	0.30	2.3%	13.3
<stack>	0.27	2.1%	5.1
heap_C23_2	0.20	1.5%	14.7
heap_V23_2	0.15	1.2%	13.6
<unknown>	0.14	1.1%	5.2
heap_M23_2	0.13	1.0%	13.5

Table 4: Results for Second Optimization of Equake

Overall, this version of the application shows an 80% reduction in L1 cache misses, a 46% reduction in L2 cache misses, and a 24% reduction in running time over the original, unoptimized application. This required changing only the layout of data structures and basically no change to the code of the application other than in the initialization code.

7.2 Twolf

The second example program we will look at is twolf. This is a placement and routing package for creating the lithography artwork for microchip manufacturing [15, 26], which uses simulated annealing to arrive at a result.

Using Cache Scope, we find that the L1 data cache hit ratio of this application is about 74%, which is fairly low, although not as low as our previous example. The average latency is 21.9 cycles, slightly larger than equake. These are an indication that poor cache utilization may be a performance problem.

Table 5 shows the stat buckets causing the most latency in the application. All of the stat buckets shown are automatically named. The safe_malloc function that appears in the bucket names is used wherever twolf allocates memory. It simply calls malloc and checks that the return value is not NULL; therefore the functions we are interested in are those that call safe_malloc. The majority of cache misses were caused by memory allocated by a small set of functions: readcell, initialize_rows, findcostf, and parser. To get more useful information about specific data structures in this application, we must manually name the blocks of memory that are allocated by these functions. Most of these blocks are allocated as space to hold a particular C struct; the easiest and most useful way to name them is after the name of the structure.

Stat Bucket	Latency		
	Cycles (billions)	%	Per Event
Safe_malloc-readcell-main	193.33	62.0%	30.1
Safe_malloc-initialize_rows-main	35.75	11.5%	14.9
Safe_malloc-parser-readcell	33.25	10.7%	23.4
Safe_malloc-findcostf-controlf	27.65	8.9%	21.0
<unknown>	7.40	2.4%	6.9

Table 5: Cache Misses in Twolf

Table 6 shows the results of re-running the tool, after altering memory allocation calls to provide a name for the stat bucket with which the memory should be associated. We have again used the convention that the named buckets begin with "heap_" to show that they are dynamically allocated memory.

Stat Bucket	Latency		
	Cycles (billions)	%	Per Event
heap_NBOX	137.55	42.9%	28.1
heap_rows_element	44.42	13.8%	8.4
heap_DBOX	23.81	7.4%	22.0
heap_TEBOX	19.58	6.1%	26.7
heap_CBOX	16.64	5.2%	27.5
heap_TIBOX	14.30	4.5%	45.6
heap_BINBOX	13.71	4.3%	15.5
<unknown>	7.46	2.3%	7.0
heap_cell	6.32	2.0%	30.4
heap_netarray	3.33	1.0%	8.8

Table 6: Cache Misses in Twolf with Named Buckets

Structure	Size	Number Allocated
BINBOX	24	224,352
CBOX	48	2,724
DBOX	96	1,920
NBOX	48	16,255
TEBOX	40	17,893
TIBOX	16	2,724

Table 7: Structures in Twolf

At the top of the list is a cluster of blocks allocated to hold C structs named NBOX, DBOX, TEBOX, CBOX, TIBOX, and BINBOX. These are all small structures. The cscope_view program can provide statistics about the size of the objects in a stat bucket and how many of them were

allocated by the application. Table 7 shows this information for the structures causing the most latency.

Some of these structures are smaller than the Itanium L1 data cache line size of 64 bytes, and all are smaller than the L2/L3 cache line size of 128 bytes. Therefore, more than one structure could be packed into an L1 or L2 cache line, but this is probably not happening due to the memory that malloc reserves for its own data structures before and after an allocated block.

This problem cannot be solved as easily as it could with equake, since these structures are not allocated all at one time during program initialization. Instead, they are allocated and freed individually at various times. Also, they are not always traversed in a single order. One feature we can make use of, however, is that many of the structures contain pointers to other structures. It is likely that if structure A points to structure B, then B will be accessed soon after A (because the program followed the pointer).

The method we chose to optimize the placement of the structures is similar to the cache-conscious memory allocation described by Chilimbi et al. [11]. We wrote a specialized memory allocator for the small structures used by twolf. It has two main features intended to reduce the cache problems revealed by Cache Scope. First, it can place small structures directly next to each other in memory. Unlike most malloc implementations, it does not reserve memory before or after each block for its own use; all overhead memory is located elsewhere. Second, it uses information about which structures point to others. When memory is allocated, the caller can specify a "hint," which is the address of a structure that either will point to the one being allocated, or be pointed to by it. In the majority of places in the code where the small structures in question are allocated, this information is readily available. The memory allocator tries to allocate the new structure in the same L1 data cache line as the "hint" address. If this is not possible, it tries to allocate it in the same L2 cache line. If this also cannot be done, it simply tries to find a location in memory that will not conflict in the cache with the cache line containing the hint address. This allocator is used for the structures shown in Table 7 that are smaller than 64 bytes, as well as another similar structure, CHANGRDBOX. These structures were chosen for this optimization because of their small size, the amount of latency they cause, and the availability of hint addresses.

Note that if this strategy is successful in placing structures that are used together in the same cache block, it will not only improve latency but also lower the required bandwidth to memory, because we will again be eliminating malloc overhead memory that would have been fetched with the data.

Running the application with this memory allocator results in a 57% decrease in L1 data cache misses, a 26% decrease in L2 misses, and an 11% reduction in running time. Table 8 shows the results of running the tool on this version of the program. The total latency and latency per event for most stat buckets is down significantly from the unoptimized version. For example, for heap_NBOX, the estimated latency is down approximately 50%, and the latency per event is down from 28.1 cycles to only 13.8 cycles. The latency for heap_CBOX is up almost 30%, but this is more than made up for by the decreases in other data structures.

Stat Bucket	Latency		
	Cycles (billions)	%	Per Event
heap_NBOX	67.93	38.0%	13.8
heap_CBOX	21.59	12.1%	60.3
heap_TEBOX	15.27	8.5%	23.8
heap_DBOX	11.93	6.7%	12.1
heap_tmp_rows_element	8.07	4.5%	20.6
<unknown>	7.62	4.3%	7.1
heap_rows_element	6.03	3.4%	20.3
heap_BINBOX	5.13	2.9%	9.9
heap_cell	4.92	2.8%	23.7

Table 8: Cache Misses in Twolf with Specialized Memory Allocator

Below the stat buckets for the structures we have been discussing, the stat bucket that causes the next highest latency is heap_tmp_rows_element. The objects associated with this stat bucket are allocated and used in the same way as those in heap_rows_element, so we will look at them both. These data structures are similar to the ones we saw in equake, in that they implement a variably sized two-dimensional array as an array of pointers to single-dimensional arrays (the arrays of pointers are named "tmp_rows" and "rows"). The arrays containing the actual data hold a small number of elements of type char; the statistics kept by Cache Scope show that these arrays are 18 bytes long when running on the problem size used for our experiments (this can also easily be seen by examining the source code and input). Since several of these would fit in a cache line, we could gain some spatial locality by allocating them as one large array, like we did for the matrices in equake. We would then set the pointers in tmp_rows and rows to point into this array.

Making this change reduces L1 data cache misses by 33%, L2 cache misses by 29%, and running time by 16% versus the original version of the application. If we are willing to accept a compiled-in limit for the largest problem size we can run the application on, we could also simply make tmp_rows and rows into statically sized two-dimensional arrays, eliminating the need for indirection. This change gives us further slight improvements. L1 data cache misses are reduced by 36%, L2 cache misses are reduced by 35%, and running time is reduced by 19% over the unoptimized version of the application.

8 Related Work

Most modern processors include some kind of performance monitoring counters on-chip. These typically provide low-level information about resource utilization such as cache hit and miss information, stalls, and integer and floating point instructions executed. Examples include the MIPS R10000 [29], the Compaq Alpha family [12], the UltraSPARC family [19], the Intel Pentium [2] and Itanium [3, 16, 27] families. All of these can provide cache miss information.

Compaq's DCPI [5] runs on Alpha processors and uses hardware counters and the ability to determine the instruction that caused a counted event to provide per-instruction event counts. On Alpha processors that use out-of-order execution, this requires extra hardware support called ProfileMe. This provides the ability to sample instructions. The processor periodically tags an instruction to be sampled, which causes it to save detailed information about its execution. Afterward, it generates an interrupt, at which time an interrupt handler can read the saved information.

Libraries are often used to simplify the use of hardware monitors, and in some cases to provide an API that is as similar as possible across processors. These include PAPI [24] and PCL [7], both of which run on multiple platforms. Perfmon [4] provides access to the Itanium family performance counters on Linux. PMAPI [1] is a library for using the POWER family performance counters on AIX.

A number of tools have been written with the primary goal of measuring memory hierarchy effects. Mtool [14] provides information about the amount of performance lost due to the memory hierarchy by computing an ideal execution time for each basic block in an application and comparing this with actual measurements.

MemSpy [23] is a tool for identifying memory system bottlenecks. It provides both data- and code-oriented information, and allows a user to view statistics related to particular code and data object combinations. MemSpy uses simulation to collect its data, allowing it to track detailed information about the reasons for which cache misses take place. For instance, a cache miss may be a cold miss or due to an earlier replacement.

MemSpy has also been used with a sampling technique, as described in [22]. The authors modified MemSpy to simulate only a set of evenly spaced strings of runs from the full trace of memory references, and found that this technique provided accuracy to within 0.3% of the actual cache miss rate for the cache size and applications they tested. This differs from the sampling performed by our tool, which samples individual misses out of the complete stream.

CPROF [20] is a cache profiling system somewhat similar to MemSpy. It uses simulation to collect detailed information about cache misses. It is able to precisely classify misses as compulsory, capacity, or conflict misses, and to identify the data structure and source code line associated with each miss.

StormWatch [10] is another system that allows a user to study memory system interaction. It is used for visualizing memory system protocols under Tempest [25], a library that provides software shared memory and message passing. It allows for selectable user-defined protocols, which can be application-specific. StormWatch runs using a trace of protocol activity, which is easy to generate since the protocols are implemented in software. The goal of StormWatch is to allow a user to select and tune a memory system protocol to match the communication patterns of an application.

SIGMA [13] is a system that uses software instrumentation to gather a trace of the memory references in an application, and uses the trace as input to a simulator. The user can also try different layouts of objects in memory by providing instructions on how to transform the addresses in the trace to reflect the new layout. The results of the simulation can then be examined using a set of analysis tools.

Itzkowitz et al. [17] describe a set of extensions to the Sun ONE Studio compilers and performance tools that use hardware counters to gather information about the behavior of the memory system. These extensions can show event counts on a per-instruction basis, and can also present them in a data-centric way by showing aggregated counts for structure types and elements. The UltraSPARC-III processors used by this tool do not provide information about the instruction and data addresses associated with an event, so the reported values are a best guess based on the instruction pointer when an interrupt occurs.

9 Conclusions

In this paper, we have described Cache Scope, an implementation of data centric cache measurement on the Intel Itanium 2 processor. This tool samples cache miss addresses using the Itanium 2's performance monitoring features, and uses this address information to keep statistics about the cache behavior of the data structures in an application. Cache Scope is unique in using the latency of cache misses as its metric, rather than the number of misses. This is made possible by the fact that the Itanium 2 hardware performance monitors can provide this information. We found this to be extremely useful; the Itanium 2 has three levels of cache, and therefore the number of L1 data cache misses alone does not necessarily determine how a data structure's cache behavior is affecting performance.

We used Cache Scope to analyze two example applications, equake and twolf, and optimized them based on the results. Especially in the case of twolf, we were able to reduce the observed latency per event, due to op-

timizing for multiple levels of cache. We were able to achieve a 24% reduction in running time for equake, and an almost 19% reduction in running time for twolf. This was done in a short time (a few days), by a programmer who was not previously familiar with the code of either application. In both cases, the improvements were gained mainly by changing data structures rather than code. This demonstrates how Cache Scope allows a programmer to quickly identify the source of lost performance due to poor cache utilization. The optimizations used could not easily have been performed by a compiler, showing the value of a tool that provides this kind of feedback to a programmer.

Cache Scope is available for download at http://www.dyninst.org/cachescope.

Acknowledgments

This work was supported in part by DOE Grants DE-FG02-93ER25176, DE-CFC02-01ER25489, and DE-FG02-01ER25510.

References

1. *AIX 5L Version 5.2 Performance Tools Guide and Reference*. IBM, IBM Order Number SC23-4859-01, 2003.
2. *IA-32 Intel Architecture Software Developer's Manual, Volume 1:Basic Architecture*. Intel, Intel Order Number 253665, 2004.
3. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*. Intel, Intel Order Number 251110-002, 2003.
4. Perfmon project web site, HP, 2003. http://www.hpl.hp.com/research/linux/perfmon/
5. Anderson, J., Berc, L., Chrysos, G., Dean, J., Ghemawat, S., Hicks, J., Leung, S.-T., Licktenberg, M., Vandevoorder, M., Walkdspurger, C.A. and Weihl, W.E., Transparent, Low-Overhead Profiling on Modern Processors. In *Proceedings of the Workshop on Profile and Feedback-Directed Compilation*, (Paris, France, 1998).
6. Bao, H., Bielak, J., Ghattas, O., Kallivokas, L.F., O'Hallaron, D.R., Schewchuk, J.R. and Xu, J. Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers. *Computer Methods in Applied Mechanics and Engineering*, *152* (1-2). 85-102.
7. Berrendorf, R., Ziegler, H. and Mohr, B. PCL - The Performance Counter Library, Research Centre Juelich GmbH, 2003. http://www.fz-juelich.de/zam/PCL/
8. Buck, B.R. and Hollingsworth, J.K. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, *14* (4). 317-329.
9. Buck, B.R. and Hollingsworth, J.K., Using Hardware Performance Monitors to Isolate Memory Bottlenecks. In *Proceedings of SC2000*, (Dallas, TX, 2000).
10. Chilimbi, T.M., Ball, T., Eick, S.G. and Larus, J.R., StormWatch: A Tool for Visualizing Memory System Protocols. In *Proceedings of Supercomputing '95*, (San Diego, CA, 1995).
11. Chilimbi, T.M., Hill, M.D. and Larus, J.R., Cache-Conscious Structure Layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (Atlanta, GA, 1999), 1-12.
12. Compaq Computer Corporation *Alpha Architecture Handbook (Version 4)*, 1998.
13. De Rose, L., Ekanadham, K. and Hollingsworth, J.K., SIGMA: A Simulator Infrastructure to Guide Memory Analysis. In *Proceedings of SC2002*, (Baltimore, MD, 2002).
14. Goldberg, A.J. and Hennessy, J.L. MTOOL: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications. *IEEE Transactions on Parallel and Distributed Systems*, *4* (1). 28-40.
15. Henning, J.L. SPEC CPU2000: Measuring CPU Performane in the New Millenium. *Computer*, *33* (7). 28-35.
16. Huck, J., Morris, D., Ross, J., Knies, A., Mulder, H. and Zahir, R. Introducing the IA-64 Architecture. *IEEE Micro*, *20* (5). 12-23.
17. Itzkowitz, M., Wylie, B.J.N., Aoki, C. and Kosche, N., Memory Profiling using Hardware Counters. In *Proceedings of SC2003*, (Phoenix, AZ, 2003).
18. Jalby, W. and Lemuet, C., Exploring and Optimizing Itanium2 Cache Performance for Scientific Computing. In *Proceedings of the Second Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology*, (Istanbul, Turkey, 2002).
19. Lauterbach, G. and Horel, T. UltraSPARC-III: Designing Third Generation 64-Bit Performance. *IEEE Micro*, *19* (3). 73-85.
20. Lebeck, A.R. and Wood, D.A. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer*, *27* (9). 15-26.
21. Lyon, T., Delano, E., McNairy, C. and Mulla, D., Data Cache Design Considerations for the Itanium 2 Processor. In *Proceedings of the International Conference on Computer Design*, (Freiburg, Germany, 2002), 356-363.
22. Martonosi, M., Gupta, A. and Anderson, T., Effectiveness of Trace Sampling for Performance Debugging Tools. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, (1993).

23. Martonosi, M., Gupta, A. and Anderson, T., MemSpy: Analyzing Memory System Bottlenecks in Programs. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, (Newport, Rhode Island, 1992), 1-12.
24. Mucci, P.J., Browne, S., Deane, C. and Ho, G., PAPI: A Portable Interface to Hardware Performance Counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, (Monterey, CA, 1999).
25. Reinhardt, S.K., Larus, J.R. and Wood, D.A., Typhoon and Tempest: User-Level Shared Memory. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, (1994).
26. Sechen, C. and Sangiovanni-Vincentelli, A. The TimberWolf Placement and Routing Package. *IEEE Journal of Solid-State Circuits*, *20* (2). 432-439.
27. Sharangpani, H. and Arora, K. Itanium Processor Microarchitecture. *IEEE Micro*, *20* (5). 24-43.
28. Tendler, J.M., Dodson, J.S., J. S. Fields, J., Le, H. and Sinharoy, B. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, *46* (1). 5-26.
29. Zagha, M., Larson, B., Turner, S. and Itzkowitz, M., Performance Analysis Using the MIPS R10000 Performance Counters. In *Proceedings of Supercomputing '96*, (Pittsburgh, PA, 1996).