ABSTRACT

Title of Dissertation: **DATA CENTRIC CACHE MEASUREMENT USING HARDWARE AND SOFTWARE INSTRUMENTATION**

Bryan R. Buck, Ph.D., 2004

Dissertation Directed By: **Professor Jeffrey K. Hollingsworth, Department of Computer Science**

The speed at which microprocessors can perform computations is increasing faster than the speed of access to main memory, making efficient use of memory caches ever more important. Because of this, information about the cache behavior of applications is valuable for performance tuning. To be most useful to a programmer, this information should be presented in a way that relates it to data structures at the source code level; we will refer to this as data centric cache information. This dissertation examines the problem of how to collect such information. We describe techniques for accomplishing this using hardware performance monitors and software instrumentation. We discuss both performance monitoring features that are present in existing processors and a proposed feature for future designs.

The first technique we describe uses sampling of cache miss addresses, relating them to data structures. We present the results of experiments using an implementation of this technique inside a simulator, which show that it can collect the desired information accurately and with low overhead. We then discuss a tool called Cache Scope that implements this on actual hardware, the Intel Itanium 2 processor.

Experiments with this tool validate that perturbation and overhead can be kept low in a real-world setting. We present examples of tuning the performance of two applications based on data from this tool. By changing only the layout of data structures, we achieved approximately 24% and 19% reductions in running time.

We also describe a technique that uses a proposed hardware feature that provides information about cache evictions to sample eviction addresses. We present results from an implementation of this technique inside a simulator, showing that even though this requires storing considerably more data than sampling cache misses, we are still able to collect information accurate enough to be useful while keeping overhead low. We discuss an example of performance tuning in which we were able to reduce the running time of an application by 8% using information gained from this tool.

DATA CENTRIC CACHE MEASUREMENT USING HARDWARE AND SOFTWARE INSTRUMENTATION

By

Bryan R. Buck

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2004

Advisory Committee:
Professor Jeffrey K. Hollingsworth, Chair
Professor Peter J. Keleher
Professor Alan Sussman
Professor Chau-Wen Tseng
Professor H. Eleanor Kerkham

Dedication

To my parents and to Chelsea, for all their help and support.

Acknowledgements

I would like to thank my advisor, Dr. Jeffrey Hollingsworth, for his help and guidance.

I would also like to thank my fellow students and members of our research group, Chadd Williams, Mustafa Tikir, Ray Chen, I-Hsin Chung, Jeff Odom, and James Waskiewicz for their help.

Table of Contents

List of Figures

List of Tables

Chapter 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 may involve simulation, adding instrumentation code to the application, or the use of hardware performance monitoring features.

One difference between these techniques is the point in time at which they are added to the system or application. Hardware features must be added when the system is designed, whereas software can add instrumentation at any time from when the application is in source code form (by modifying the source code) to after the application has begun execution (using dynamic instrumentation [14]). Because of this, 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, sometimes 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 processor designs include increased support for performance monitoring. Many processors have for some time included a way to count cache misses, and a way to trigger an interrupt when a given number of events (such as cache misses) occur. Some recent processors also 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. The Intel Itanium 2 [3] supports this feature, and reportedly so does the IBM POWER4 [83]. There is still more progress to be made however; as an example, the POWER4 performance monitoring features are largely undocumented, and are not considered supported features of the processor.

This dissertation will consider the problem of how to provide useful feedback to a programmer about the cache behavior of the source code-level data structures in an application. It will present techniques for measuring cache events and relating them to program data structures, using both simulation and hardware performance monitors. The discussion of simulation will mainly be in the context of its use in validating the techniques for use with hardware monitors, and to evaluate future hardware counter designs.

We will begin in Chapter 2 with a discussion of related work and how our work differs from it. In Chapter 3 we will discuss gathering data centric cache information by sampling cache miss addresses. We will present an evaluation of this technique using a simulator, and show that it can be used to collect accurate information with low overhead.

Next, in Chapter 4, we will describe a tool called Cache Scope, which uses a modified version of this technique on real hardware, the Intel Itanium 2. This tool was used to validate the sampling technique in real-world conditions. It was also used to tune the performance of two applications, in order to demonstrate the usefulness of the collected data. The optimized versions of these applications showed reductions in running time of approximately 24% and 19%.

In Chapter 5, we propose a novel hardware feature that would provide information about the addresses of data evicted when a cache misses occurs. We discuss a technique for sampling this eviction information to provide feedback to the user about the interactions of data structures in the cache. We will then describe an implementation of this technique inside a simulator, which we used to show that this technique is feasible in terms of accuracy and overhead. We will also show an example of optimizing an application based the results from this tool, which resulted in an approximately 8% reduction in running time, in order to show the value of the information it provides.

Finally, Chapter 6 will present conclusions and future work.

Chapter 2: Related Work

Many types of instrumentation have been used to measure the performance of the memory hierarchy. These can be thought of as lying along a continuum from hardware techniques that are designed into the system to software techniques that can instrument a program after has begin execution. This chapter first describes some of these instrumentation systems, and then discusses optimizations that have been proposed for improving the use of the memory hierarchy.

2.1 Hardware Instrumentation

An example of hardware support for software instrumentation is the HYPERMON performance monitoring system for the Intel iPSC/2 Hypercube [56]. This system provides hardware support for the collection of software events while keeping perturbation down, by providing an I/O port that software instrumentation can use to record event codes. These codes are then timestamped and read by a node or nodes dedicated to saving or processing the data. Mink et al. [64] describe a similar hybrid software-hardware instrumentation system that includes hardware support for including measurements of resource usage in event records. They also discuss an all-hardware method, using pattern matching on virtual addresses to trigger the storage of events. A monitoring system developed for the INCAS project [86] also uses a hybrid approach, with events generated by software sent to a Test and Measurement Processor that is part of each node. This processor filters or summarizes the data and sends it to a dedicated central test station that presents the information to the user. The IBM RP3 performance monitoring hardware [13] contains support for collecting hardware, rather than software events. Each Processor-Memory Element (PME) in-

cludes a Performance Monitor Chip (PMC), which receives event signals from the other PME elements (with an emphasis on memory events). The data collected can be read by the PME itself or by the I/O subsystem. For both multi- and single-processor systems, the MultiKron board [63, 64] provides a way to add performance monitoring hardware to a system with either an SBus or VME bus. It provides on-board memory to hold events, which are triggered by software. It also provides pins that can be connected to host hardware in order to measure external signals, with the measurements written into memory as part of an event record (sample).

Other systems have used flexibility provided by a hardware system to add instrumentation effectively at the hardware level. ATUM [5] uses the ability to change the microcode in some processors to add instrumentation at the microcode level to store information about memory references. The FlashPoint [59] system uses the fact that the Stanford FLASH multiprocessor [44] implements its memory coherence protocols in software that is executed by a Protocol Processor. The designers observe that the support needed for measuring memory system performance is very similar to the support needed to implement a coherence protocol. Therefore, in a system such as FLASH it is relatively easy to add performance measurement to the code that is normally executed by the Protocol Processor. One thing that distinguishes FlashPoint from other systems discussed here is that it returns data centric information, similar to that returned by MemSpy [58], which will be described below. This allows a user to determine what program objects are causing performance problems.

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 [87], the Compaq Alpha family [25], the UltraSPARC family [49], and the Intel Pentium [2] and Itanium [3, 36, 80] families. All of these can provide cache miss information.

Compaq's DCPI [6] 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 [66] and PCL [8], 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.

2.2 Software Instrumentation

The tools described in this dissertation use software instrumentation to control hardware performance monitors and gather results. Software instrumentation can be inserted any time from when the source code is written (manually by the programmer) to after the program has begun executing. Pablo [68, 73] uses modified Fortran and C compilers to produce a parse tree from source code, and then produces instrumented source code based on the parse tree and information supplied by the user. Sage++ [11]

is a general-purpose system that facilitates the creation of tools that analyze and modify source code. It is a library that can parse Fortran, C, or C++ into an internal representation that can be navigated and altered using library calls. A modified program can then be written out as new source code. Sage++ has been used to implement pC++ [12], an object-parallel extension to C++. ROSE [72] is a tool for building source-to-source preprocessors, which currently reads and produces C++ code (other languages may be supported in the future). It allows a user to read in code as an abstract syntax tree, transform the tree, and write it back out as code. MPTrace [29] is a tool that inserts instrumentation for tracing parallel programs after compilation, by adding new code to the assembly language version of a program that is produced by a compiler.

Many tools have been written to transform programs after compilation and linking. Johnson [40] describes processing a program after linking in order to optimize it, perform profiling, generate performance statistics, and for other uses. FDPR [67] is a tool used to improve the code locality of programs. First, it reads an executable file and places jumps to instrumentation routines at the end of each basic block, in order collect information about how often each block is executed. The instrumented program is then run, and based on the results the original executable file is rewritten again, this time reordering basic blocks in order to improve code locality and reduce branch penalties.

Larus and Ball describe techniques used to rewrite executables [47] in the qp and qpt programs. These programs provide basic block profiling, and qpt additionally uses abstract execution [46] to trace a program's data and instruction references. EEL

[48] is a general-purpose library that provides the ability to rewrite executables using a machine- and system-independent interface. It has been implemented on the SPARC architecture. Another general-purpose library that provides the ability to rewrite an executable file is ATOM [81, 84], which is implemented on the Compaq Alpha. One difference between these two systems is that EEL is able to insert instrumentation code inline in an application, whereas in ATOM instrumentation is written as a set of functions in a high-level language (usually C) and calls to the instrumentation code are inserted. Also, ATOM is mostly oriented toward adding instrumentation code only, whereas EEL provides more general functions for altering executables, such as replacing code. Etch [78] is a tool similar to these for machines running Microsoft Windows on the x86 architecture. Because of the environment in which it runs, it must deal with many challenges that similar tools running on RISC architectures do not. For instance, the instruction set is more complex, with instructions of varying lengths, and code and data are not easily distinguished in executable files. Etch allows not only adding instrumentation code to an application, but also rewriting the application in order to optimize it. An example would be reordering instructions in order to improve code locality. BIT [51] is a tool for instrumenting Java bytecodes. It is itself written in Java, and provides functionality similar to ATOM. Because it instruments at the bytecode level, it can be used on any platform with an implementation of the Java Virtual Machine.

Some systems have moved the insertion of instrumentation into a program even later, to when the program is loaded or after it has begun execution. For instance, Bishop [10] describes profiling programs under UNIX by dynamically patch-

ing breakpoint instructions into their images in memory. This allows a controlling application to become aware of when a particular point in the code has been reached. The Paragon Performance Monitoring Environment [75] includes the ability to patch calls to a performance monitoring library into applications that are to be run. These can produce trace information that can then be analyzed. Taking this further, Paradyn [62] uses dynamic instrumentation, which allows instrumentation to be generated, inserted, and removed during the execution of an application. It writes instrumentation code into the address space of the application and patches the application's code to call it at the desired locations, using the debugging interface of the operating system. The code for performing this dynamic instrumentation has been incorporated into the general-purpose Dyninst API library [14]. HP's Caliper [37] uses dynamic instrumentation to profile programs, and also provides an interface for using hardware performance counters. Its dynamic instrumentation is slightly different from Paradyn/Dyninst; instead of patching the target application's code to call the instrumentation, it rewrites whole functions and inserts the instrumentation inline into the new function.

Another option that allows instrumentation to be altered easily at runtime is simulation. Shade [23] performs simulation with instrumentation, mainly oriented toward tracing. It translates code for a target machine into code for the simulation host, with tracing code inline (except specialized code written by the user, which is executed as function calls). The translation is done dynamically, so Shade is able to insert and remove instrumentation while the program executes. The dynamic nature of the translation also allows it to handle even self-modifying code.

2.3 Memory Performance Measurement and Visualization Tools

This section will describe some systems that have been designed with the primary goal of measuring memory hierarchy effects. One such system is Mtool [33], a performance debugging tool that, among other measurements, provides information about the amount of performance lost due to the memory hierarchy. To do this, it first computes an ideal execution time for each basic block in an application, assuming that all memory references will be satisfied by the cache. It then runs an instrumented version of the application that gathers information about the actual execution time of each basic block. The difference between the ideal time and the actual time is then reported as the approximate loss in a given basic block due to the memory system. In contrast to the techniques presented in this dissertation, Mtool does not use any information about the addresses associated with memory stalls, and therefore returns no data centric information.

MemSpy [58] 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.

For purposes of keeping the code- and data-oriented statistics mentioned above, MemSpy separates code and data into bins. A code bin is a single procedure, whereas a data bin is either a single data object or a collection of objects that were all allocated at the same point in the program with the same call path. The authors argue

that such objects generally behave similarly. Using these types of bins, they then define statistical bins, which represent combinations of code and data bins. At each cache miss, the appropriate statistical bin is located and its information is updated. One way this differs from the techniques described in this dissertation is in the use of simulation for the tool itself, whereas in our work simulation is only used when proving techniques that will be used with hardware monitors. In addition, the techniques we will present do not require instrumentation code to take an action at each and every cache miss. MemSpy has also been used with a sampling technique, as described in [57]. 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 tools, which sample individual misses out of the complete stream.

CPROF [50] 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 [19] is another system that allows a user to study memory system interaction. It is used for visualizing memory system protocols under Tempest [74], a library that provides software shared memory and message passing. Tempest 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 [27] is a system that uses software instrumentation to gather a trace of the memory references in an application, which it losslessly compresses. The trace is then used as input to a simulator, along with a description of the memory system parameters to be used (cache size, associativity, etc.). 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. [38] 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. Unlike the simulators and hardware counters used in the work described in this dissertation, 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 inferred and may be imprecise.

Fursin et al. [30] describe a technique for estimating a lower bound on the execution time of scientific applications, and a toolset that implements it. This technique involves modifying code so that it performs the same amount of computation but accesses few memory locations, eliminating most cache misses. The modified code is then profiled to estimate the lower bound.

2.4 Adapting System Behavior Automatically

Other studies have suggested ways for systems to react automatically to information gained by the measurement of memory hierarchy effects. For instance, Glass and Cao [32] describe a virtual memory page replacement algorithm based on the observed pattern of page faults. Their algorithm, SEQ, normally behaves like LRU, but when it detects a series of page faults to contiguous addresses, it switches to MRU-like behavior for that sequence. Cox and Fowler [26] describe an algorithm for detecting data with a migratory access pattern and adapting the coherence protocol to accommodate it. Migratory data is detected by noting cache lines for which, at the time of a write, there are exactly two copies of the cached block in the system, and the processor performing the write is not the same processor that most recently performed a write to that block. For these cache lines, they switch to a strategy in which a read miss migrates the data, by copying it to the local cache and invalidating it on the other processor holding a copy in one transaction.

Bershad et al. [9] describe a method of dynamically reducing conflict misses in a large direct-mapped cache using information provided by an inexpensive piece of hardware called a Cache Miss Lookaside Buffer. Their technique is based on the fact that cache lines on certain sets of pages will map to the same position in the cache. The Cache Miss Lookaside buffer keeps a list of pages on which cache misses occur, associated with the number of misses on each. This can be used to detect when a set of pages that map to the same locations in the cache are causing a large number of misses. All but one of the pages can then be relocated elsewhere in physical memory, eliminating their competition for the same area of the cache.

Another hardware feature that has been proposed as a means of both measuring memory behavior and adapting to it is informing memory operations [35]. An informing memory operation allows an application to detect whether a particular access hit in the cache. The paper proposes two forms of this, one in which operations set a cache condition code that can then be tested, and one in which a cache miss causes a low-overhead trap. The authors propose several uses for this facility, including performance monitoring, adapting the application's execution to tolerate latency, and enforcing cache coherence in software.

2.5 Optimization

Many studies have analyzed ways to improve an application's use of the cache. Their results may be useful in tuning an application after identifying the sources of memory hierarchy performance problems using tools such as those described in this dissertation.

One well-known technique is blocking, or tiling, which has been shown to improve locality in accessing matrices [45, 85]. This is achieved by altering nested loops to work on sub-matrices, rather than a row at a time. Other techniques, including loop interchange, skewing, reversal, fusion, and distribution have also been shown to be useful in improving locality [60, 85]. Lam et al. [45] and Coleman et al. [24] study how reuse in tiled loops is affected by the tile size, and how to choose tile sizes that will lead to good performance. Rivera and Tseng [77] present techniques for the use of tiling in 3D scientific computations.

One problem with tiling is that the full amount of reuse may not be obtained due to conflict misses, which is discussed by Lam et al. [45] and studied in detail by

Temam et al. [82]. Chame et al. [17] examine the factors causing conflict misses, self-interference (interference between items accessed by the same reference) and cross-interference (between items accessed in separate references). They discuss how these are affected by tile size, and present an algorithm for choosing a tile size that will minimize them.

Pingali et al. [70] describe Computation Regrouping, which is a source code level technique for transforming programs to promote temporal locality in memory references, by moving computations involving the same data closer together.

Other studies have suggested changing data layout in addition to or instead of transforming control flow. Methods that have been shown to be useful in eliminating the conflict misses discussed above include padding and alignment [50, 69, 76]. Kandemir et al. present a linear programming approach for optimizing the combination of loop and data transformations [41]. It has also been suggested that array layout should be controllable by the programmer [18]. Shackling [42, 43] is a technique that is similar to tiling, but which uses a data centric approach. Shackling fixes an order in which data structures will be visited, and, based on this, schedules the computations that should be performed when a data item is accessed.

Ghosh et al. describe Cache Miss Equations (CME) [31], which allow them to express cache behavior in terms of equations that can be solved to find optimum values for transformations like blocking and padding. Qiao et al. [71] present practical results from applying optimization techniques including blocking and padding to scientific applications, with results consistent with predicted performance gains.

Another approach, which requires some hardware support, is to tolerate cache misses through the use of software-controlled prefetching [16, 65]. Most of the studies described above have operated on data structures such as matrices, in which the data layout is determined at compile time. One advantage of prefetching is that it can more easily be used in the presence of pointers and pointer-based data structures [52, 54]. For instance, Lipasti et al. [52] present a simple heuristic, that the items pointed to by function parameters should be prefetched at the call site for the function. This is based on the assumption that pointers passed into a function are likely to be dereferenced. Luk et al. [54] consider the problem of recursive data structures, and present several schemes for prefetching items in these structures that are likely to be visited in the future. Chilimbi and Hirzel [22] describe dynamic hot data stream prefetching. As an application runs, their system profiles memory accesses to find frequently occurring sequences, and inserts code into the application to detect prefixes of these sequences and prefetch the rest of the stream when they are detected. ADORE [53] is another system that inserts prefetching code at runtime, based on information about cache misses that is gathered using hardware performance counters.

The reordering of data and computation at runtime has also been suggested for reducing cache misses in applications with dynamic memory access patterns [28, 61]. Ding and Kennedy [28] describe locality grouping, which moves interactions involving the same data item together, and dynamic data packing, which relocates data at runtime to place items that are used together into the same cache lines. The authors show that a compiler can perform these transformations automatically, with acceptable overhead. Methods that have been proposed for placing objects when reordering

data at runtime include first-touch ordering, in which items are placed in the order in which they will be first accessed, and the use of space filling curves (for problems in which data items have associated spatial locations, and interact with nearby items) [61].

Chilimbi, Hill, and Larus [21] describe cache-conscious reorganization and cache-conscious data layout, which attempt to place related dynamically allocated structures into the same cache block. They present a system that provides two simple calls that a programmer can use to give a program these capabilities. In another paper, Chilimbi, Davidson, and Larus [20] consider the distinct problem of how to arrange fields within a structure for the best cache reuse. They describe automatic techniques for structure splitting and field reordering.

A different way to reduce cache misses is to eliminate some memory references entirely, by making better use of processor registers, as in [15]. The authors describe a source-to-source translator that replaces array references to the same subscript with references to an automatic variable. This allows a typical compiler's register allocation algorithm to place the value in a register.

Chapter 3: Measuring Cache Misses in Simulation

This chapter will discuss a study of data centric cache measurement using a cache simulator. While the simulator can be used as a tool in its own right, this work will concentrate on using it to evaluate how hardware counters can be used by software instrumentation. This will be done by providing simulated hardware counters, and by running software instrumentation that uses them under the simulator so that we can evaluate the accuracy of the data it gathers and estimate the overhead associated with it.

3.1 Cache Miss Address Sampling

In order for a tool running on real hardware 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. Simply examining the state of the processor when an interrupt occurs due to a cache miss counter overflow is generally not sufficient to accurately determine the addresses associated with the event that caused the interrupt. Due to fea-

tures of modern processors such as pipelining, 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. As an example, 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. For this reason, in order to sample the addresses associated with events, the processor must provide explicit support.

A further argument for sampling is that on some processors that provide the features described above, it may not be possible to obtain the address of every cache miss. For instance, on the Intel Itanium 2 [3] and IBM POWER4 [83], 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, we can collect sampled statistics about the cache misses taking place in an application's data structures. We will present an example of such statistics below in Table 1, which is found in Section 3.3.1. These statistics were gathered by instrumentation code running under the simulator mentioned above. The simulator allows us to keep exact statistics in addition to the sampled statistics, so that the two can be compared in order to evaluate the accuracy of sampling.

In order to measure per-data structure statistics, we associate a count with each object in memory, meaning each variable or dynamically allocated block of memory (or group of related blocks). We then set the hardware counters (which will be simulated in the experiments described in this chapter) to generate an interrupt after some chosen number of cache misses. This number is varied through the run, in order to prevent the sampling frequency from being inadvertently synchronized to the access patterns of the application. When the interrupt occurs, we read the address of the cache miss from the hardware, match it to the object in memory that contains it, and increment its count. 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.

After the execution has completed, or after a representative portion of the execution, we can examine the counts and rank program objects by the number of cache misses caused when accessing each. If the number of misses sampled for each object is proportional to the total number, this will provide the programmer with an accurate idea of which program objects are experiencing the worst cache behavior.

The individual object miss counts described here are similar to the information returned by the tool MemSpy [58]. A major difference between MemSpy and the present work is that MemSpy used a simulator as the primary means to gather information, whereas the simulator described in this chapter is used to demonstrate a low overhead technique for finding memory hierarchy problems using hardware per-

formance counters and software instrumentation. Also, MemSpy used simulation to examine all cache misses; the tool described here attempts to estimate the total cache misses for each object by sampling a subset. As noted in section 2.3, a version of MemSpy using sampling was developed, but the samples used were runs of memory accesses from a full trace. These runs were then provided as input to the cache simulator. This introduces a different kind of error from the technique discussed here, due to lack of knowledge about the state of the cache at the beginning of each run of accesses. The technique described in this dissertation relies on hardware (real or simulated) to provide samples of the cache misses taking place.

3.2 The Simulator

For the study described in this chapter, we implemented the algorithm described above inside a simulator. The simulator runs on the Compaq Alpha processor, and consists of a set of instrumentation code that is inserted into an application to be measured using the ATOM [81, 84] binary rewriting tool. Code is inserted at each load and store instruction in the application, to track memory references and calculate their effects on the simulated cache. Additionally, each basic block is instrumented with code that maintains a virtual cycle count for the execution by adding in a number of cycles for executing that block. The cycle counts do not represent any specific processor, but are meant to model RISC processors in general. The simulator does not model details such as pipelining and multiple instruction issue. Since the virtual cycle count is the only timing data used, slowdown due to the instrumentation for simulation does not affect the results. The cache simulated is a single-level, two-way

set associative data cache. A cache size of 2MB was used for the experiments that will be described below.

The simulator provides a cache miss counter, an interrupt that can be triggered when the counter reaches a chosen value, and the ability to determine the address that was accessed to cause a miss. Additional instrumentation code that runs under the simulator uses these features to perform cache miss address sampling, and uses the sampled addresses to produce information about the number of cache misses caused by each data structure in an application. Since this instrumentation runs under the simulator, it can be timed using the virtual cycle counter, and it affects the simulated cache, making it possible to study overhead and perturbation of the results.

3.3 Experiments

To investigate the accuracy and overhead of gathering data centric cache information by sampling, we ran the cache miss sampling instrumentation we described above under the simulator on a number of applications from the SPEC95 benchmark suite. The applications tested were tomcatv, su2cor, applu, swim, mgrid, compress, and ijpeg. For experiments in which we did not vary the sampling frequency, we used a default value of sampling one in 50,000 cache misses. The following sections show the results of these experiments.

3.3.1 Accuracy of Results

We will first examine the accuracy of the results returned by sampling. Table 1 shows the objects in each application causing the most cache misses, both according to the sampling instrumentation and as determined using exact numbers collected by the simulator. Up to five objects are shown, with objects causing less than 0.1% of

the total misses deleted. Object names that consist of a hexadecimal number represent dynamically allocated blocks of memory (the number is the address).

Application	Variable / Memory Block	Actual		Sampled	
		Rank	%	Rank	%
applu	a	1	22.8	1	23.8
	b	2	22.7	3	21.8
	c	3	22.4	2	22.3
	d	4	17.3	4	17.3
	rsd	5	7.1	5	7.0
compress	orig_text_buffer	1	63.4	1	62.3
	comp_text_buffer	2	35.8	2	35.8
	htab	3	0.7	3	1.9
ijpeg	0x14102e000	1	86.0	1	89.9
	jpeg_compressed_data	2	11.4	2	10.1
	0x14102c000	3	0.4	42	0.0
mgrid	U	1	40.7	2	41.3
	R	2	40.5	1	41.4
	V	3	18.8	3	17.2
su2cor	U	1	58.9	1	58.0
	R	2	6.4	3	5.4
	S	3	6.0	2	6.4
	W2 – sweep	4	3.9	5	3.6
	W1 – intact	5	3.6	4	3.8
swim	VOLD	1	7.7	2	8.1
	CU	2	7.7	10	7.4
	POLD	3	7.7	1	8.5
	UOLD	4	7.7	13	6.9
	P	5	7.7	9	7.4
	CV	7	7.7	5	8.0
	Z	9	7.7	4	8.0
	VNEW	12	7.7	3	8.1
tomcatv	RY	1	22.5	2	22.1
	RX	2	22.5	1	22.9
	AA	3	15.0	3	15.0
	Y	4	10.0	5	10.0
	X	5	10.0	4	10.4

Table 1: Results for Sampling Under Simulator

The “rank” columns show the order of the objects when ranked by number of cache misses. The percent columns show the percentage of all cache misses that were due to the named variable or block of memory. The “actual” pair of columns shows exact values collected at a low level in the simulator, and the “sampled” columns show the values as measured by the sampling algorithm.

Generally, the results were indicative of the actual number of cache misses occurring due to references to each object. For almost all applications, sampling ranked the objects in order by the number of actual cache misses, except when the difference in total cache misses caused by two or more objects was small (less than one percent).

The largest error in the percent of cache misses estimated by sampling was seen in ijpeg, where the memory block with the base address 0x14102e000 caused 3.9% fewer cache misses than estimated. Since this object caused 86% of all cache misses, this difference did not affect the ranking of the objects. The next largest error was the estimate for the array V in mgrid, which differed from the actual value by 1.5%; all other errors for the objects shown were smaller than this. We can conclude that for these applications, sampling at the rate used, one in 50,000 cache misses, provided information that was accurate enough to be useful.

3.3.2 Perturbation of Results

Another aspect of accuracy is how the instrumentation code itself affects the values being measured. In the case of sampling cache misses, we are interested in how many cache misses are being caused by the instrumentation code rather than the original application. Figure 1 shows the percent increase in cache misses for each

application when run with sampling at the frequencies given in the legend. Note that the scale of the y axis, showing the percentage, is logarithmic.

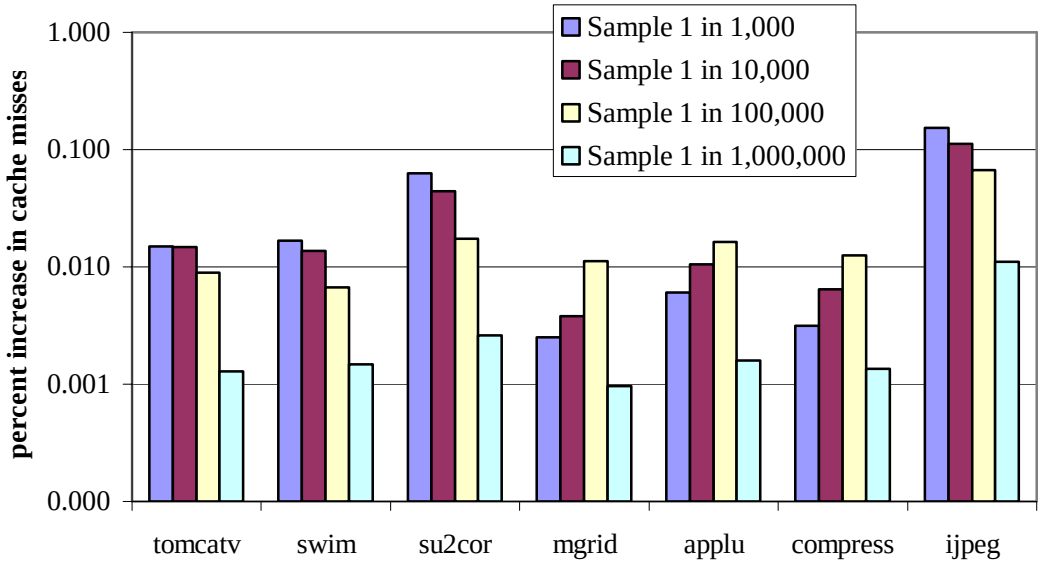

Figure 1: Increase in Cache Misses Due to Instrumentation (Simulator)

The results shown were gathered by running the sampling code under the simulator along with the application, and comparing the results with runs of the application alone under the simulator, with no sampling. For all runs with and without instrumentation, the applications were allowed to execute for the same number of application instructions (this was made possible by the simulator). Operating system code is not included in the simulator, so the cache effects of kernel code for context switches and the delivery of signals for sampling was not modeled.

The increase in cache misses was very low for all applications. The largest increase, when sampling one in every 1,000 cache misses while running ijpeg, was approximately 0.15%. The larger increase in ijpeg relative to the other applications is due to the fact that ijpeg normally has a lower cache miss rate, only 144 misses per million cycles, and therefore a smaller absolute number of additional cache misses are

required to cause a larger percent increase. For comparison, the application with the next lowest miss rate is compress, with 361 misses per million cycles, followed by mgrid, with 6,827.

Interestingly, for mgrid, applu, and compress, the number of additional cache misses goes up as the sampling frequency goes down, until we reach a sampling frequency of one miss per million. This effect is likely due to the frequency with which the instrumentation code runs; when it is run often, the data it uses is kept in the cache, whereas when it is run less frequently, the data is more likely to be evicted from the cache before the next time the instrumentation code is run to sample a miss. At extremely low sampling frequencies, this effect becomes unimportant. Factors that affect whether or not this phenomenon will occur with a given application include the cache miss rate, the size of the memory object map used by the instrumentation, and frequency with which cache misses land in the same set of objects.

3.3.3 Instrumentation Overhead

Figure 2 shows the increase in running time due to instrumentation code when sampling with each of the frequencies shown in the legend, for each application tested. The increase is shown as the percent increase over an uninstrumented run of the same application. These times are in terms of the virtual cycle count maintained by the simulator. Again, the scale of the y axis is logarithmic.

The values shown include the time spent executing instrumentation for sampling (in virtual cycles), plus a cost for receiving each interrupt signal that triggers the instrumentation to take a sample. For this value, we used results obtained experimentally on an SGI Octane workstation with 175Mhz processors. We used the perform-

ance counter support in the Irix operating system to cause an interrupt after a chosen number of cache misses, which we varied. The cost measured was approximately 50 microseconds per interrupt, or 8,800 cycles. While the clock speed of the processors used for this test was relatively slow, the number of cycles it takes to handle an interrupt likely has not significantly changed on newer processors.

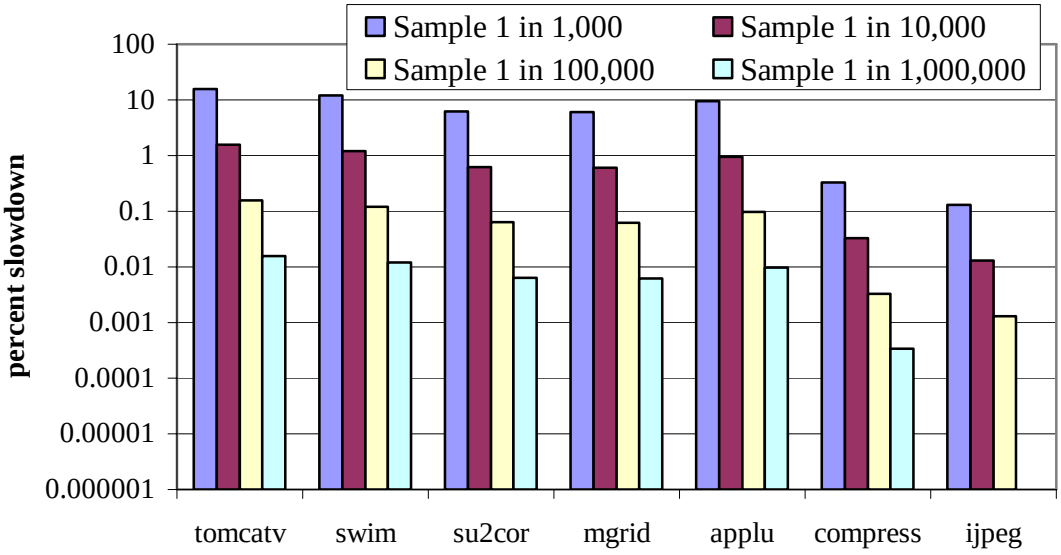

Figure 2: Instrumentation Overhead (Simulator)

The chart shows that the overhead is low unless sampling is performed too often. At a sampling rate of one in 10,000 misses, which was shown to be sufficient in Section 3.3.1, the highest slowdown was 1.6%, for tomcatv. The overhead increases almost linearly as we increase the sampling frequency. At a sampling frequency of one in 1,000 misses, the overhead becomes significant, with the highest overhead being approximately 16%, again for tomcatv. However, even this slowdown may be acceptable, depending on the application.

3.3.4 Simulation Overhead

This section discusses the overhead in actual wall clock time that is incurred by running an application under the cache simulator. This is distinct from the overhead discussed in Section 3.3.3, which is the overhead of the instrumentation code that performs sampling, as measured in virtual cycles by the simulator. Figure 3 shows the slowdown of each application when running under the simulator, in units of normalized execution time of the application running natively outside the simulator. This means that a value of ten on the y axis ("slowdown") indicates ten times the execution time of the program running outside the simulator.

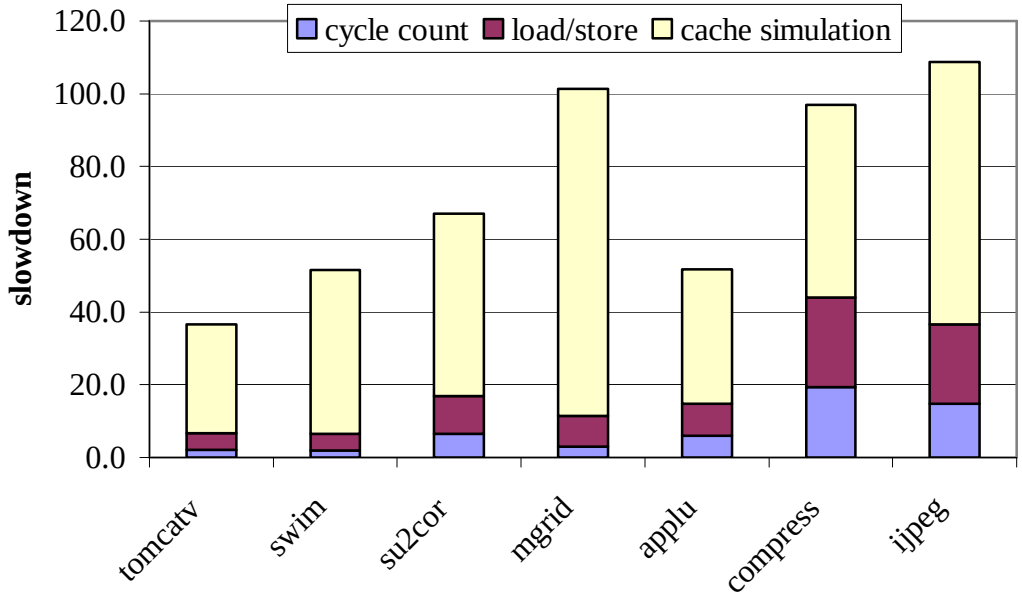

Figure 3: Slowdown Due to Simulation

The execution time for each application is split up into three categories. The "cycle count" portion represents the slowdown due to the code that maintains the virtual cycle count. "Load/store" represents the slowdown due to having instrumentation at all load and store instructions. This is measured as the time taken to jump to and return from an empty function at each load and store. The "cache simulation"

time represents the time added by performing the cache simulation inside the function called at each load and store.

For all applications, the slowdown due to running under the simulator is large, ranging from approximately a 37 times slowdown for tomcatv to approximately 109 times for ijpeg. The largest part of this overhead is cache simulation time, which accounts for from a 30 times slowdown for tomcatv to a 90 times slowdown for mgrid. This is followed by the load/store time (a 5 to 25 times slowdown), and then time maintaining the virtual cycle count (a 2 to 20 times slowdown). The cache simulation time could possibly be reduced by optimizing the simulation code, but the load/store and virtual cycle count times are mostly due to the overhead of jumping to instrumentation code, and cannot be reduced without using a different method of instrumentation. This is a strong argument for why hardware support for data centric cache information is needed. With hardware support, a tool incurs only the overheads talked about in Section 3.3.3, and not the much higher overhead of simulation.

3.4 Conclusions

In this chapter, we have discussed how to use hardware support for sampling cache miss addresses to collect information about the behavior of data structures at the source code level. We then described an implementation of this technique that runs under a cache simulator. Our experiments using this simulator showed that sampling can provide accurate information with low overhead.

We also discussed the overhead of the cache simulator. The applications we tested showed slowdowns of up to 109 times when run under the simulator. From

this, we conclude that using hardware support is highly desirable for gathering data centric cache information, in order to avoid the high overhead of simulation.

Chapter 4: Measuring Cache Misses Using Hardware Monitors

This chapter describes a tool named Cache Scope that uses the hardware performance monitors on the Intel Itanium 2 processor to gather data centric cache information. The instrumentation code used by Cache Scope is a modified and expanded version of the cache miss sampling code described in Chapter 3. One major addition is that it also gathers code centric information, which can be combined with the information about data structures. For instance, it is possible to examine where in the code the cache misses for a particular memory object took place. The following sections describe the Itanium 2 and its performance monitoring features, the implementation of cache miss address sampling in Cache Scope, the results of a set of experiments using Cache Scope, and examples of tuning applications based on information from the tool.

4.1 Intel Itanium 2 Performance Monitoring

The experiments described in this chapter were performed on a two-processor system with Intel Itanium 2 processors. The Itanium 2 is a VLIW processor with many features for speculation and for making use of instruction level parallelism [36, 80]. It is an implementation of the IA-64 architecture, which was developed jointly by Intel and Hewlett-Packard.

The IA-64 architecture specifies basic performance monitoring support, and provides for extending it with features specific to each implementation. The Itanium 2 provides substantial performance monitoring abilities beyond those mandated by the IA-64 architecture, including features that allow the sampling of cache miss addresses [3].

4.1.1 PMC and PMD Registers

Every IA-64 implementation must provide at least four performance monitor register pairs. Each pair consists of a Performance Monitor Control register (PMC) and a Performance Monitor Data Register (PMD). These are labeled PMC/PMD_4 through PMC/PMD_7. The PMC register in a pair specifies what event will be counted, under what conditions it will be counted (for instance, in what privilege levels), and whether or not interrupts will be generated when the value being counted overflows. These registers are only accessible in privilege level zero (the highest). The PMD registers contain the actual values; they are writable only at privilege level zero, but are readable by code running at other levels. Optionally, setting a bit in another special register, the Processor Status Register (PSR), causes the PMDs to return zero when read from a non-zero privilege level, effectively making reading the PMDs a privileged operation. This can also be done on a per counter pair basis. In the experiments described in this dissertation, all reading and writing of performance monitor registers is performed in the kernel at privilege level zero.

On the Itanium 2, the PMD registers are 48 bits wide. When a PMD register is read, the top 16 bits are copied from the high bit of the 48 bit value, so that it appears sign extended.

All IA-64 implementations also include four additional PMC registers that contain overflow status bits. These are labeled PMC_0 through PMC_3. When a counter overflows, a corresponding bit in one of these four registers is set to one. PMC_0 also contains a “freeze” bit (labeled PMC_0.fr), which freezes all counting when set. This will be discussed below in connection with counter overflow interrupts.

The IA-64 architecture specifies only two events, which are: instructions retired, which counts all instructions that execute fully; and processor clock cycles. The Itanium 2 adds over 100 events beyond these. The events used by the instrumentation described in this dissertation are: L1 data cache read misses, L1 data cache reads, L2 data cache misses, and a special event that counts L1 data cache read misses and allows the measurement code to determine instruction and data addresses associated with the miss.

4.1.2 Performance Monitor Overflow Interrupt

In order to sample information about cache miss events, it is necessary for instrumentation code to be notified periodically that an event has occurred. The IA-64 architecture specifies a performance monitor overflow interrupt that can be used for this purpose. Since the PMD registers are writable, by setting an initial value in a PMD, it can be made to overflow after a chosen number of events have occurred.

Interrupts can be enabled or disabled independently for each PMD by setting a bit in the corresponding PMC (the overflow interrupt bit, PMC_n.oi). As mentioned earlier, when a PMD overflows, a corresponding bit in one of PMC_0 through PMC_3 is set to indicate the overflow. If the overflow interrupt (PMC_n.oi) bit is set, the processor then sets the freeze bit, PMC_0.fr, and raises a performance monitor interrupt. Setting the freeze bit stops the counters, so that the values read by the interrupt handler do not reflect an event caused by the interrupt handler itself.

4.1.3 Event Addresses

The Itanium 2 processor features a set of registers named the Event Address Registers (EARs) that provide addresses and other information related to events tak-

ing place in the cache and TLB. They can record instruction and data addresses for data cache load misses and data TLB misses, and instruction addresses for instruction cache and instruction TLB misses. For cache misses, they can also record the number of cycles the fetch was in flight (the latency of the miss). They are configured through the Instruction Event Address Configuration register, PMC_{10}, and the Data Event Address Configuration Register, PMC_{11}. The options that can be set using these registers include what events to monitor, such as cache vs. TLB events; what privilege modes to monitor in; whether to allow user mode code to read the register; and a minimum latency value for the counted events (for example, if monitoring L1 data cache load misses, only misses with a latency equal to or higher than the minimum latency value will be monitored). The data values are read from a set of PMDs, PMD_0 and PMD_1 for instruction events and PMD_2, PMD_3, and PMD_{17} for data events. These can only be read when PMC_0.fr is set, freezing the performance counters.

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. There is a special event that the performance monitor can count called DATA_EAR_EVENTS, which counts the number of events tracked by the Data EAR. By counting this event and enabling an interrupt on overflow, an interrupt handler can be certain that the Data EAR values it reads are associated with the last event tracked by the Data EAR. If the Data EAR is set to track data cache load

misses, then the DATA_EAR_EVENTS event is a random subset of all data cache load misses, allowing sampling of the addresses associated with the misses.

One other important fact to note is that the Data EAR mode that tracks L1 data cache load misses also tracks floating point loads. On the Itanium 2, the L1 data cache handles only integer data, so all floating point loads go to the L2 cache and may be sampled by the Data EAR.

4.2 Linux IA-64 Performance 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. Such a kernel interface is necessary in order to use the performance monitors from user-level code, since the performance monitor control registers are accessible only from privilege level zero. Perfmon's interface is a single kernel call named perfmonctl.

In order to make the perfmon interface portable to different IA-64 implementations, it provides only a thin layer over the hardware registers, with its main purpose being to allow user-level code to read and write them. The idea is to provide an implementation-independent way to access the implementation-dependent performance monitoring features. This is possible because the IA-64 architecture specifies the basic framework for the performance counters. For instance, putting a certain value in PMC_4 will allow a user to count a certain event on the Itanium 2, but may count some other event on another IA-64 processor. Perfmon would provide only a way to write a value into PMC_4, and does not have any knowledge of the meaning of the value.

Another function of perfmon is to virtualize 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 (this is actually accomplished by subtracting it from the initial value set in the counter).

In order to make the perfmonctl call easier to use, the perfmon project has also produced a library named libpfm, which simplifies the task of programming the counters. Given a set of events to count, libpfm determines which registers can be used to count them, and returns the values that should be written into the hardware registers to do so. It does not set the registers itself; that is done by calling perfmonctl. The events are specified by names passed to the library as strings.

Libpfm is made up of two layers, a processor-independent layer that handles the basic functionality specified in the IA-64 architecture, and a processor-dependent layer that handles the functionality specific to a particular processor. There are processor-dependent layers for the Itanium and Itanium 2 processors. The Itanium 2 support includes features for setting up the Instruction and Data EARs, which are used in the work described below to capture cache miss addresses.

4.3 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 set of instrumentation code that is added to an application to be measured, and an analysis

program that allows a user to examine the data that was gathered. These are described below.

4.3.1 Instrumentation for Sampling Cache Misses

The part of Cache Scope that collects data centric cache information about an application is implemented in a library named libdcache_tool. To measure an application, the user links it with this library, and inserts into the application a call to an initialization function, dctl_initialize. Optionally, the user can also insert calls to the functions dctl_start_measurement and dctl_stop_measurement, to control what part of the execution will be monitored.

In order to track dynamic memory allocations, calls to functions such as malloc must be replaced with equivalent calls supplied by Cache Scope (for instance, dctl_malloc in the case of malloc). This is normally done by adding preprocessor flags when compiling the program, for instance "-Dmalloc=dctl_malloc."

There is also a partially completed version of the instrumentation code that uses the Dyninst API [14] dynamic instrumentation library to insert all necessary calls into the application at runtime. This version was not used in the work described in this chapter because Dyninst did not yet support the IA-64 architecture when the work was done. A completed version of Cache Scope that uses Dyninst would eliminate the requirement to link with the tool library and to manually insert library calls.

The instrumentation code uses libpfm and 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 interrupt on overflow bit is set for the counter counting Data EAR events (cache misses). 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. When the interrupt occurs, the instrumentation code reads the address of the instruction that caused the cache miss, the address that was being accessed, and the latency of the instruction. It then updates the data structures for the appropriate memory object (as described below) and restarts the counters. When restarting the counters, it uses the randomization feature of perfmon to control the number of Data EAR events between samples, to ensure a representative sampling of events.

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 libdcache_tool library that takes an extra parameter, which is the bucket name to assign to the block. Typically, a user would run the tool first without explicitly naming blocks, and then based on what functions were shown to be allocating

blocks with performance problems, he or she would add explicit names to memory allocation calls in those functions, to differentiate them from each other. The map of memory locations to stat buckets is maintained in an interval tree that is implemented using a red-black tree, in order to provide range queries and efficient insertions, deletions, and lookups. The locations of variables are derived from the debug information in the executable to be measured. The locations of dynamically allocated memory are obtained by the Cache Scope memory allocation functions mentioned above.

The information stored in a bucket is illustrated in Figure 4. A bucket keeps a count of cache events (L1 data cache misses or floating point loads) that have taken place when accessing the objects associated with the bucket, and the sum of the latencies of these events. 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 misses 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 object and function 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 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.

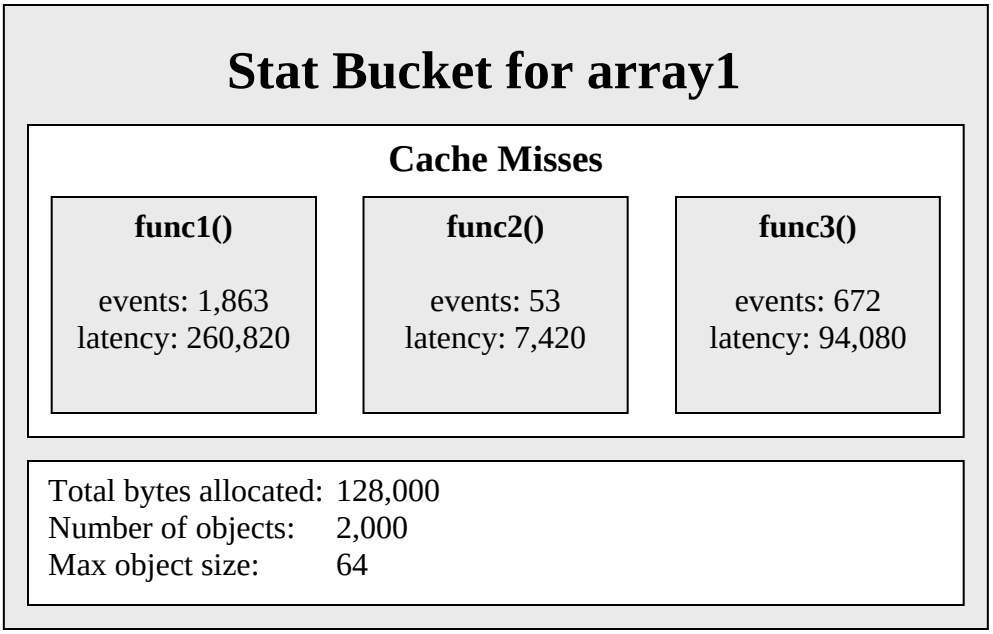

Figure 4: Stat Bucket Data Structure

When an interrupt occurs, first the data address of the associated cache miss or floating point load is mapped to a bucket using the interval tree. The interval tree contains a number that identifies the bucket. This number is used as an index into an array of bucket structures, and within the desired bucket structure is a field that is the vector of function information. In this vector, functions are identified by a unique numeric identifier. To find the unique identifier to look for in the vector, the instruction address associated with the cache miss is looked up in a map of addresses to functions. This map is implemented as a sorted list of functions with their associated address ranges, which can be queried using a binary search. The information for this map comes from the debug information in the executable being measured. The map contains the unique identifier for the function, which is then used in the binary search through the vector of function information found earlier. This returns the structure that represents the data for cache events that take place in the given bucket and func-

tion. Finally, the counts in this structure are updated with the information for the event.

4.3.2 Data Analysis Tool

When measurement is finished, 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 DView. DView is written in Java, and so is portable to any system for which Java is available.

DView provides a simple set of commands for examining the data. Figure 5 shows a sample session in which the tool is used to examine the cache events in the application mgrid. There are commands that produce tables of the objects or functions causing the most latency, as well as commands that can combine the data centric and code centric data. For instance, a user can produce a table of the functions causing the most latency when accessing a certain data structure, or the data structures experiencing the most latency in a given function. Note that the tool presents information in terms of the latency associated with sampled events, rather than simply counts of cache misses.

DView 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 4.5.1 and 4.5.2.

```
DView 1.0
                  Application: mgrid_base.ia64_linux
                Total latency:            2,751,228
              Total L1 misses:            2,412,080
              Total L2 misses:        2,402,079,498
               Sampling scale:               32,768.50
            Estimated latency:       90,153,614,718
              Average latency:                 6.50

Command? objects
Objects by latency:

Object                    Latency  %Latency   %Events      LPer
All                        90,154    100.0%    100.0%       6.5
cmn_x_u__                  56,661     62.8%     64.8%       6.3
cmn_x_r__                  30,487     33.8%     33.4%       6.6
cmn_x_v__                   2,640      2.9%      1.5%      12.9
<UNKNOWN>                     162      0.2%      0.1%       8.2
cmn_x_a__                     118      0.1%      0.1%       8.0

Latency values in millions.

Command? quit
```

Figure 5: DView Sample Session

4.4 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. One goal of this study is to validate the conclusions we made from the simulation results in the previous chapter by running the sampling technique on hardware. For instance, running on hardware will allow us to measure the overhead of the instrumentation code and how the code affects cache misses in a real-world setting. Another goal is to examine how varying the sampling rate affects these values. This information is likely to differ from what was observed under the simulator, due to the fact that we will be sampling not only L1 data cache misses but also all floating point loads.

The applications used in the experiments were wupwise, swim, mgrid, applu, gcc, mesa, art, mcf, equake, crafty, ammp, parser, gap, and twolf. They were com-

piled 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; the actual number of events between samples was randomly varied throughout the run. For tests in which we did not vary the sampling frequency, we chose one in 32K as our default rate.

We also ran tests in which we did not sample cache misses, but did use 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, without any requiring any 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.

4.4.1 Perturbation of Results

Figure 6 shows the increase in L2 cache misses seen in each application we tested when sampling at the rates shown in the legend, 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. Table 2 shows the absolute number of cache misses (in billions) with no sampling and when sampling at the rates shown. We are concerned primarily with the L2 cache for several reasons. First, the L1 cache on the Itanium 2 handles only integer loads. Second, 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 [55]. Third, the L1 cache is

only 16KB while the L2 is 256KB. For these reasons, most optimization on the Itanium 2 will likely target the L2 cache.

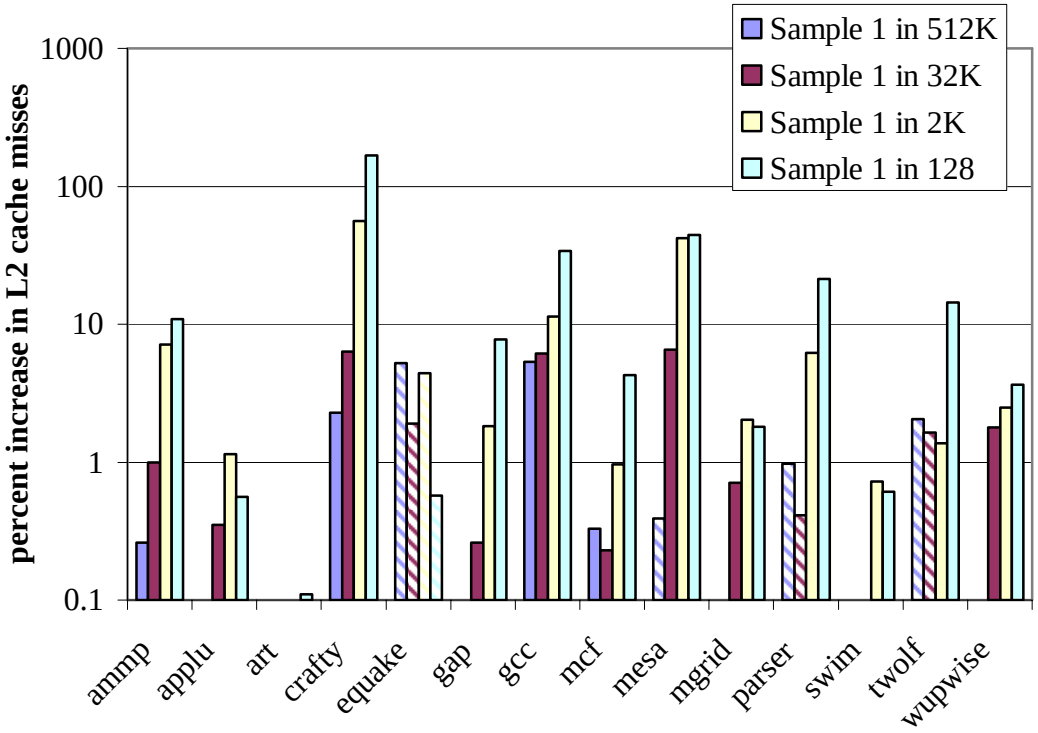

Figure 6: Increase in L2 Cache Misses on Itanium 2

Application	L2 Cache Misses (Billions) at Sampling Rate				
	None	512K	32K	2K	128
ammp	3.76	3.77	3.80	4.03	4.18
applu	2.11	2.11	2.12	2.14	2.12
art	2.97	2.97	2.97	2.98	2.98
crafty	0.11	0.11	0.12	0.17	0.29
equake	2.01	1.90	1.97	1.92	1.99
gap	0.55	0.55	0.55	0.56	0.59
gcc	0.52	0.55	0.56	0.58	0.70
mcf	6.01	6.03	6.03	6.07	6.27
mesa	0.14	0.14	0.15	0.19	0.20
mgrid	2.39	2.39	2.40	2.43	2.43
parser	1.71	1.70	1.71	1.82	2.08
swim	6.88	6.88	6.89	6.93	6.92
twolf	4.79	4.69	4.71	4.85	5.48
wupwise	0.68	0.68	0.70	0.70	0.71

Table 2: L2 Cache Misses on Itanium 2 in Billions

The increase in L2 cache misses for most applications was relatively small except at the two highest sampling frequencies, and as will be discussed below, some applications actually showed a decrease in cache misses in the sampled runs. It is important to note that the sampling frequencies used here cannot be directly compared to those used in the simulations discussed in Chapter 3. The major reason for this is that the event used is different. Whereas in the simulator the interrupt was triggered by a specified number of cache misses, on the Itanium 2, L1 data cache misses and floating point loads both contribute to the count that triggers the interrupt. In addition, under the simulator we were only able to collect statistics for a limited portion of each run, due to the high overhead in running time of the simulator. We therefore needed to use a high sampling frequency in order to extract as much information as possible in this limited period.

When sampling one 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 might suspect 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 increases in L2 misses were seen with crafty, with a 2.3% increase.

At a sampling frequency of one 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 further increase the sampling frequency, we see increases in cache misses as high as 168%, seen when running crafty while sampling one 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 one 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 [39] 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 4.5.1, 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. This may also account for some of the increase in cache misses seen when running gcc with the memory allocation tracking instrumentation.

4.4.2 Instrumentation Overhead

Figure 7 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.

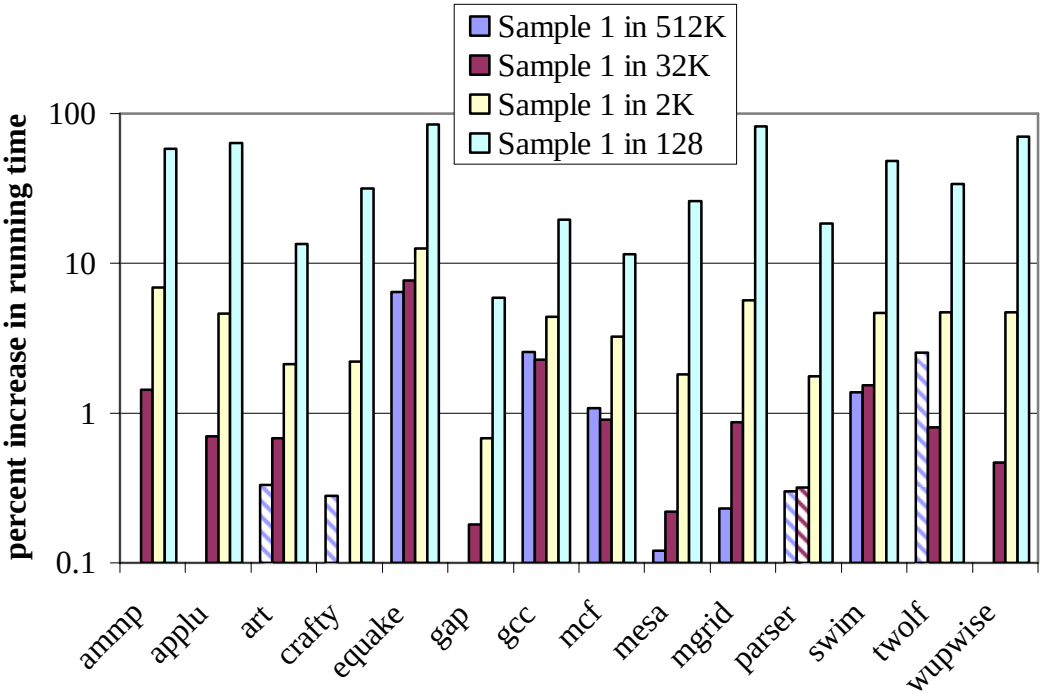

Figure 7: Instrumentation Overhead (Itanium 2)

For the two lowest sampling frequencies tested, the overhead was acceptable for all applications. Looking at the higher of these frequencies, sampling one 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. Including both variables and dynamically allocated blocks of memory, Equake declares or allocates 1,335,667 objects during its run, while the application with the next highest number of allocations, twolf, declares or allocates 575,418. Eight of the thirteen applications declare or allocate approximately 1,000 or fewer objects.

4.5 Tuning Using Data Centric Cache Information

Previous sections have discussed how data centric cache information can be measured using sampling, the hardware features that enable this, and statistics about the overhead and accuracy of our implementation of sampling in simulation and on the Itanium 2. We will next examine the question of the value of the information provided by the tool.

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.

As we will describe, 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 (the author of this dissertation) 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.

4.5.1 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 [7, 34]. 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 reads 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 event. For equake, this is 21.3 cycles. This is high relative to most of the other applications tested. This value 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 may also indicate that cache performance is being lost 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 4.3.2, Cache Scope returns information in terms of

latency rather than the number of cache misses. It is important to note that 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. Nevertheless, using latency as the main metric allows the tool to return more useful information than a simple count of L1 data cache misses would, since it takes into account the effects of all three levels of cache. Another point to note is that the latency values returned by Cache Scope are based only on the subset of events that are tracked by the Data EAR, and as such may be an underestimate. Not all events are tracked by the Data EAR because it can track only one load instruction at a time (see Section 4.1.3).

Table 3 shows the stat buckets in the application that cause the most latency, sorted by latency. As described in Section 4.3.1, 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. In this example, there are two automatically named buckets, mem_init-main and readpackafile-main.

Stat Bucket	Latency (millions)	% Latency	% Events	Latency/ Event
mem_init-main	97,184	95.6%	84.8%	24.0
Exc	2,098	2.1%	8.1%	5.4
<STACK>	1,115	1.1%	4.6%	5.0
readpackfile-main	679	0.7%	0.1%	102.1
<UNKNOWN>	548	0.5%	2.3%	5.0

Table 3: Data Structure Statistics in Equake

The “Latency” column shows an estimate of the absolute number of cycles of latency caused by objects assigned to the given stat bucket, in millions. The “% Latency” column shows the percentage of all latency in the application that was caused by the stat bucket. “% Events” shows the percent of all sampled events that were caused by the bucket. Finally, “Latency/Event” shows the average latency for the cache misses caused by the bucket.

The <STACK> bucket represents the stack. The tool is able 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, and any memory that was dynamically allocated by a library that was not processed by the tool. 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 4.3.1, dynamically allocated memory can be explicitly assigned to a stat bucket named by the user. This is done by using a special call provided by Cache Scope to allocate the memory. For instance, the user may replace calls to malloc(size)

with calls to dctl_mallocn(size, name). We did this for the memory allocation calls in mem_init.

Table 4 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 them are parts of a set of three-dimensional matrices, where each matrix is made up of a group of independently allocated blocks of memory. The reason they are split up 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 allocated as in the abbreviated code from equake shown in Figure 8.

Stat Bucket	Latency (millions)	% Latency	% Events	Latency/ Event
heap_K_2	23,673	35.4%	4.2%	118.5
heap_disp_3	18,407	27.5%	36.1%	10.7
heap_K_3	7,487	11.2%	28.9%	5.4
heap_disp_2	3,473	5.2%	3.2%	22.8
Exc	2,134	3.2%	8.1%	5.5
heap_M_2	1,533	2.3%	2.1%	15.2
heap_C_2	1,489	2.2%	2.1%	14.7
<STACK>	1,124	1.7%	4.7%	5.0
heap_M_1	952	1.4%	0.8%	23.9
heap_K_1	929	1.4%	0.2%	80.0

Table 4: Data Structure Statistics in Equake with Named Buckets

```
/* 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 8: Memory Allocation 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 4 show the matrix dimension with which each stat bucket is associated. For the first two dimensions, these are arrays of 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. Almost 100% of the latency in heap_K_2, 69.6% percent of the latency when accessing heap_disp_3, and 99.8% of the latency 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 DView 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. A way to improve the situation would be 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 is shown in the code in Figure 9.

```
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 9: Modified Memory Allocation in Equake

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 5.

Stat Bucket	Latency	% Latency	% Events	Latency/ Event
heap_K_3	14,886	49.4%	31.6%	22.5
heap_disp_3	7,433	24.7%	38.7%	9.1
heap_K_2	1,316	4.4%	1.2%	52.4
mem_init-main	1,174	3.9%	3.5%	15.8
heap_disp_2	1,111	3.7%	1.5%	35.8
Exc	1,013	3.4%	8.8%	5.5
heap_C_2	639	2.1%	2.3%	13.4
<STACK>	531	1.8%	5.0%	5.0

Table 5: Data Structure Statistics in 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 6 shows the results of making this change. Note that the latency for K is significantly less than the 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.

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.

Stat Bucket	Latency (millions)	% Latency	% Events	Latency/ Event
K	6,840	53.4%	32.7%	21.8
disp	3,503	27.3%	40.9%	8.9
Exc	370	2.9%	7.0%	5.5
heap_M_2	307	2.4%	2.3%	13.7
heap_C_2	295	2.3%	2.3%	13.3
<STACK>	270	2.1%	5.5%	5.1
heap_C23_2	196	1.5%	1.4%	14.7
heap_V23_2	151	1.2%	1.2%	13.6
<UNKNOWN>	137	1.1%	2.7%	5.2
heap_M23_2	127	1.0%	1.0%	13.5

Table 6: Data Structure Statistics in Second Optimized Equake

4.5.2 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 [34, 79], 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 may be an indication that poor cache utilization is a performance problem for this application.

Table 7 shows the stat buckets causing the most cache misses and 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	% Latency	% Events	Latency/ Event
safe_malloc-readcell-main	193,333	62.0%	45.0%	30.1
safe_malloc-initialize_rows-main	35,749	11.5%	16.8%	14.9
safe_malloc-parser-readcell	33,247	10.7%	9.9%	23.4
safe_malloc-findcostf-controlf	27,651	8.9%	9.2%	21.0
<UNKNOWN>	7,397	2.4%	7.6%	6.9

Table 7: Cache Misses in Twolf

Table 8 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	% Latency	% Events	Latency/ Event
heap_NBOX	137,553	42.9%	25.7%	28.1
heap_rows_element	44,420	13.8%	27.8%	8.4
heap_DBOX	23,808	7.4%	5.7%	22.0
heap_TEBOX	19,578	6.1%	3.8%	26.7
heap_CBOX	16,644	5.2%	3.2%	27.5
heap_TIBOX	14,300	4.5%	1.6%	45.6
heap_BINBOX	13,709	4.3%	4.6%	15.5
<UNKNOWN>	7,464	2.3%	5.6%	7.0
heap_cell	6,322	2.0%	1.1%	30.4
heap_netarray	3,334	1.0%	2.0%	8.8

Table 8: Cache Misses in Twolf with Named Buckets

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 DView 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 9 shows this information for the structures causing the most latency.

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 9: Structures in Twolf

One thing to note is that some of these 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/L3 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 all 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. [21]. 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 im-

plementations, 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. 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. 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, as was the case with the optimizations for equake, lower the required bandwidth to memory by not fetching memory used internally by malloc.

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 10 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.

The stat bucket that causes the next highest latency below the ones we have been discussing 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.

Stat Bucket	Latency	% Latency	% Events	Latency/ Event
heap_NBOX	67,925	38.0%	38.5%	13.8
heap_CBOX	21,592	12.1%	2.8%	60.3
heap_TEBOX	15,266	8.5%	5.0%	23.8
heap_DBOX	11,927	6.7%	7.7%	12.1
heap_tmp_rows_element	8,068	4.5%	3.1%	20.6
<UNKNOWN>	7,615	4.3%	8.3%	7.1
heap_rows_element	6,028	3.4%	2.3%	20.3
heap_BINBOX	5,131	2.9%	4.1%	9.9
heap_cell	4,918	2.8%	1.6%	23.7

Table 10: Cache Misses in Twolf with Specialized Memory Allocator

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.

4.6 Conclusions

In this chapter, we have described Cache Scope, an implementation of data centric cache measurement on the Intel Itanium 2 processor. One goal of this tool was to verify that cache miss address sampling is practical on a real system, as predicted by the simulation results we showed earlier. An important difference between the simulated system and the Itanium 2 is that on the Itanium 2, we collect information about the latency of each cache miss.

We found that perturbation and overhead were acceptable for the lower sampling frequencies we tested, one in 512K and one in 32K cache events, but that they can become significant when the sampling frequency is increased. The increased perturbation at higher sampling frequencies shows that sampling more frequently is not always more accurate, due to the instrumentation code's own effect on the cache.

We used Cache Scope to analyze two example applications, equake and twolf, and we then optimized them based on the results. We found the latency information to be useful in this analysis. Especially in the case of twolf, we were able to reduce the observed latency per event (L1 data cache miss or floating point load), due to optimizing 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 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. In addition, the optimizations used could not easily have been performed by a compiler, proving the value of a tool that provides this kind of feedback to a programmer.

Chapter 5: Cache Eviction Monitoring

In addition to knowing what data structures are being accessed to cause cache misses, another piece of information that would be valuable in tuning applications is an indication of the reason the misses are occurring. Cache misses are usually categorized into cold misses, conflict misses, and capacity misses. However, it would be difficult for hardware to determine to which of these categories a particular miss belongs. For example, to distinguish a cold miss from the others, the hardware would need to keep a record of whether each cache line-sized block of main memory had ever been loaded into the cache.

An alternative way to obtain information about why cache misses are occurring would be to look at data that is already in the cache, and record the circumstances under which it is evicted. This would require significantly less hardware support than attempting to classify misses as described above. This chapter will describe a novel hardware monitoring feature that would provide the address of the data that is evicted from the cache when a miss occurs. Using this feature, a tool could provide feedback to a user about how the source code level data structures in an application are interacting in the cache. This chapter will discuss an example of such a tool, implemented in simulation. This tool samples both cache miss and eviction information, providing a superset of the information gathered by the Cache Scope tool described in Section 4.3.1.

5.1 Proposed Hardware Feature

In order to provide data centric information about cache eviction behavior, we propose a new hardware monitoring feature. When a cache miss occurs, in addition

to storing the address of the data that missed in the cache, the new feature would store the address of the data that was evicted as a result. The hardware should also provide the cache miss counter and interrupt on counter overflow features that we previously discussed for sampling cache misses, so that the eviction information can be gathered by sampling cache misses. The proposed hardware features are depicted in Figure 10.

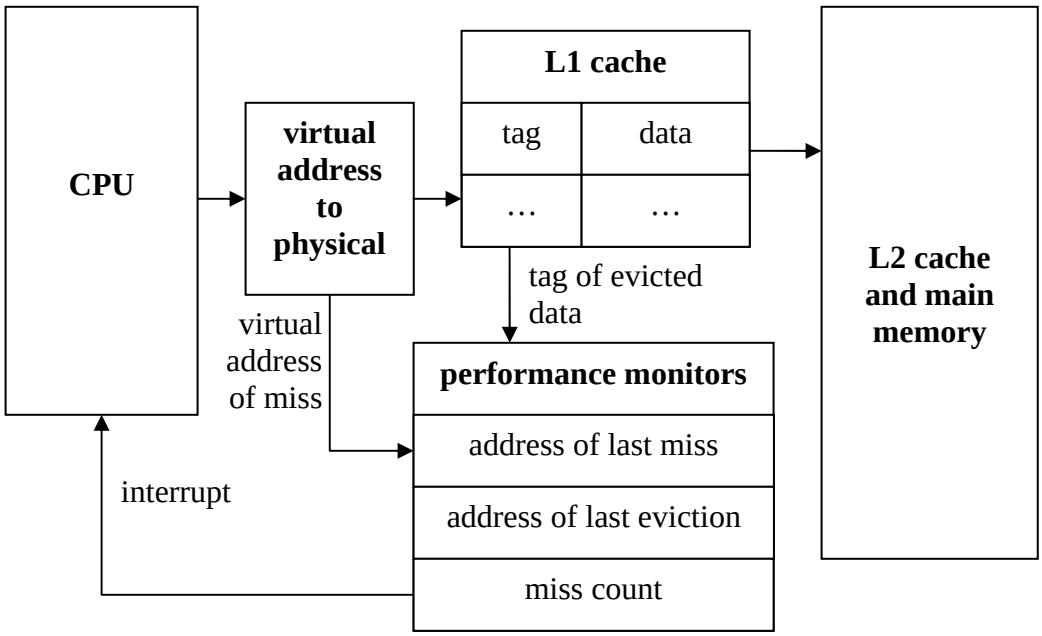

Figure 10: Performance Monitor for Cache Evictions

It should be relatively simple for hardware to provide the eviction address. The cache maintains a tag for each line in the cache, which identifies the area of memory the line is caching. When a cache miss occurs, before replacing a cache line, the hardware could store the tag of the line that is about to be replaced. The saved tag or its associated address could then be made available to software through a special purpose register.

In general, gathering data centric cache information requires using virtual, rather than physical addresses. For cache misses, this is quite simple, since the virtual

address being accessed is available at the time the cache miss occurs, and the processor can simply save it. For cache evictions, it is more difficult to provide a virtual address, since the cache line tags are usually based on the physical address. In order to simplify the hardware requirements, we propose that the cache eviction hardware monitor provide only the physical address, and leave it to software to map this to a virtual address within an application's address space. This requires the inverse of the mapping that must normally be performed, which is to map virtual addresses to physical. Operating system features such as paging to disk and the ability to map the same physical page into multiple virtual locations complicate this mapping. For the experiments described in this dissertation, we have assumed that perfect information is available about the virtual addresses of cache misses and evictions. Since most HPC users size their application data and/or systems so that the working set fits into memory, paging is usually infrequent, so this is not a serious limitation.

5.2 Instrumentation for Sampling Cache Evictions

Since the addresses of evicted data are not provided by any existing processor, we implemented eviction sampling in code that runs under a cache simulator. The cache simulator used is a modified version of the one described in Section 3.2, which was previously used to study sampling cache misses. It consists of a set of instrumentation code that is inserted into an application using the ATOM [81, 84] binary rewriting tool. In addition to providing an emulated register containing the address associated with the most recent cache miss, the modified simulator also provides a register that contains the address of the data evicted as a result of the miss.

The instrumentation code for sampling cache evictions is also inserted into the application using ATOM. As was the case for our experiments in sampling cache misses, this instrumentation runs under the simulator, so that we can measure its overhead and effects on the cache. The cache eviction sampling instrumentation is based on the same code as that described in Section 4.3.1 for sampling cache misses. We use the interrupt on overflow feature of the emulated cache miss counter to set up an interrupt that will occur after some number of cache misses. When a miss occurs, we retrieve the address that was accessed to cause the miss, and also the address of the data that was evicted from the cache as a result of the miss.

Cache miss and eviction statistics are kept in an extended version of the stat bucket structure described in Section 4.3.1. The extended stat bucket structure is shown Figure 11. As before, the stat buckets contain a vector that is used to store information about cache misses and where in the code they take place. In addition, they contain a vector of information about cache evictions. There is an entry in this vector for each stat bucket with an object that has caused an eviction of an object in the bucket in question. Each entry in the eviction information vector contains another vector that holds the actual eviction counts, broken down by the area of code in which they took place. For instance, in Figure 11, array3 has caused 89 evictions of array1 at line 4 in the function func2. Note that the code areas used are lines of code, not whole functions as was the case with Cache Scope. This is made possible by using functionality of ATOM that can return more detailed information from the line number table in the executable. One other difference between the cache eviction tool and Cache Scope is that the eviction tool names dynamically allocated blocks of memory

differently. Dynamically allocated memory is assigned to a named bucket based on the execution path leading to the allocation function that created it. This includes not only the names of the functions but also the address from which each function was called, so that two blocks of memory allocated from two different places in the same function will be given different names. This did not result in a significant difference in the applications we tested.

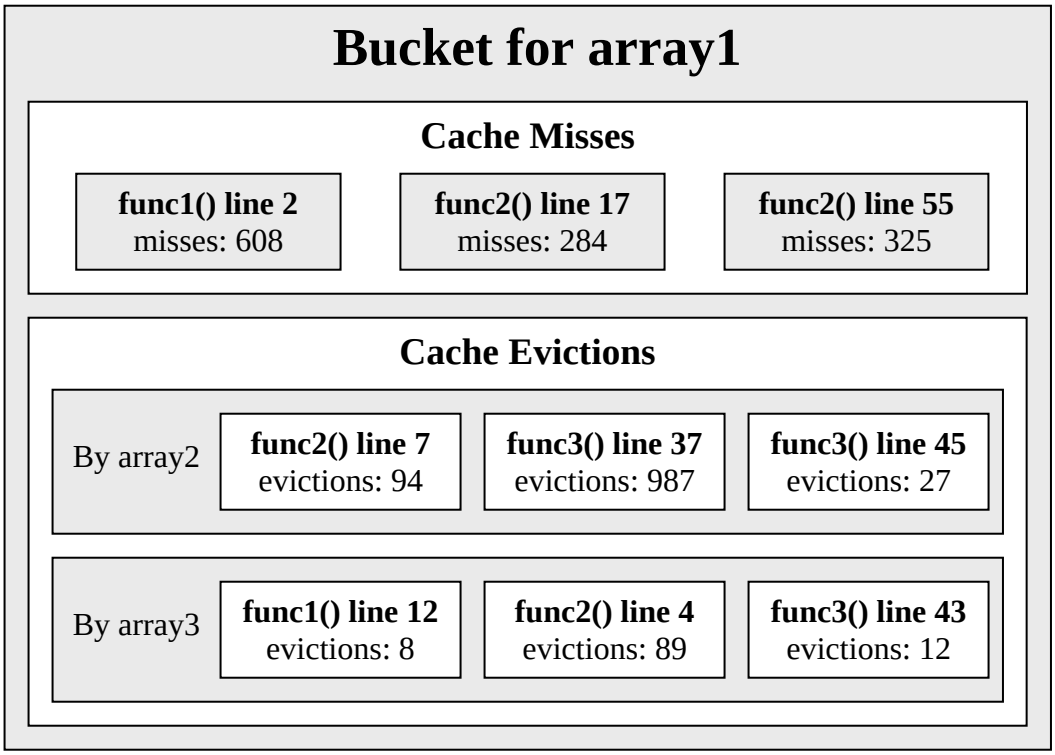

Figure 11: Bucket Data Structure for Cache Evictions

Since misses and evictions occur together, i.e. every miss triggers an eviction, it is not strictly necessary to maintain information about both. Cache misses could be derived from eviction information by adding the number of times an object was the cause of an eviction. A possible disadvantage of this doing this is that the miss information would not be easily available in real-time as the application executes. Our

current implementation does maintain miss information separately from eviction information.

The data gathered by the tool is written out in a compressed format, and can be examined using a separate tool named read_data. The read_data program can display the data about both cache misses and evictions in a number of ways. It can show the stat buckets or functions causing the most cache misses, and the stat buckets causing the most evictions of a particular bucket. It can combine code centric and data centric information, for instance to show the stat buckets causing the most cache misses in a particular function, or the stat buckets causing the most evictions of a given bucket in a given function.

5.3 Experiments

In order to evaluate the accuracy and overhead of sampling cache miss and eviction information, we performed a series of experiments in which we ran the cache miss and eviction sampling instrumentation on a set of applications from the SPEC CPU95 [50] and SPEC CPU2000 [34] benchmark suites. From SPEC CPU95, we used the application su2cor. From SPEC CPU 2000, we used applu, gzip, mgrid, swim, and wupwise. The applications were compiled with the Compaq C compiler V6.3-028 and Compaq Fortran compiler V5.4A-1472.

For these experiments, we simulated a 64KB four-way set associative cache with a line size of 32 bytes. These values were chosen as realistic ones for a RISC processor. Cache misses are assigned an average penalty of 20 cycles. This is based on the assumption that L1 cache misses that are satisfied by the L2 cache incur a penalty of 12 cycles, and that accesses that must go to main memory require 60 cycles;

these values were again chosen to model current processors. Unless otherwise noted, the sampling interval was set to sample an average of one in every 25,000 cache misses, with the actual value pseudo-randomly varied throughout the run in order to obtain a more representative sample.

5.3.1 Accuracy of Results

We will first examine the accuracy of sampling cache miss and eviction information. Our tool records information at a number of levels of granularity. At the coarsest level of data centric information, it records the number of cache misses that took place when accessing the objects in each stat bucket. This can be broken down into the misses for each stat bucket that occurred in each line of code in the application. At this level, the tool collects information similar to that gathered by Cache Scope, described in 4.3.1.

Cache eviction information represents another level of granularity. The tool records the number of times memory associated with each stat bucket is evicted by loads of memory associated with each other stat bucket. This can also be broken down by the line of code at which the evictions took place.

In order to allow a comparison of actual statistics versus those estimated by the instrumentation code, the simulator collects the same types of information gathered by the instrumentation. This is done at a low level in the simulator, and counts every cache miss and eviction. This provides us with exact values to compare the results from the instrumentation code against.

5.3.1.1 Cache Misses

We will first examine the results of sampling cache misses. Although this information is similar to the data collected by the tools already described in this dissertation, it is necessary to revisit it to determine whether its accuracy is affected by the additional overhead and perturbation of collecting detailed cache eviction information, which involves more instrumentation code and larger instrumentation data structures.

Table 11 shows the results of sampling cache misses in the set of applications we tested. It lists the five objects causing the most cache misses in each application, excluding any objects causing less than 1% of the total number for the application. The "stat bucket" column lists the names of the stat buckets, which may represent variables or data structures in dynamically allocated memory. As explained in section 5.2, buckets representing dynamically allocated memory are named by the code path through which they were allocated. This is shown as a series of function names with line numbers. For example, gzip contains a block of memory that was allocated by spec_init at line 88, which was called from main at line 276.

The "rank" columns show the order of the objects when ranked by number of cache misses, and the percent columns show the percentage of all cache misses that were due to the named stat bucket. The actual values were collected at a low level in the simulator, and are therefore precise, whereas the sampled value are as estimated by the instrumentation code. This information was gathered from separate instrumented and uninstrumented runs during the same portion of the applications' executions (this is made possible by the simulator).

Application	Stat Bucket	Actual		Sampled	
		Rank	%	Rank	%
applu	C	1	19.2	3	18.6
	B	2	19.2	2	19.3
	A	3	19.1	1	19.7
	D	4	14.4	5	13.9
	rsd	5	13.9	4	14.1
gzip	spec_init(88)-main(276)	1	99.5	1	100.0
mgrid	U	1	50.5	1	51.3
	R	2	39.0	2	39.0
	V	3	10.2	3	9.6
su2cor	U	1	16.8	1	16.7
	W1-intact	2	9.2	2	9.0
	W2-intact	3	8.1	3	8.2
	W2-sweep	4	6.9	4	7.0
	W1-sweep	5	5.8	5	5.8
swim	UNEW	1	13.3	2	13.4
	PNEW	2	13.3	3	13.2
	VNEW	3	13.3	1	13.6
	CU	4	6.7	9	6.6
	CV	5	6.7	5	6.7
	U	10	6.7	4	6.7
wupwise	U	1	30.3	1	29.0
	UD	2	15.1	2	15.5
	T	3	13.4	3	13.6
	S	4	12.5	4	13.0
	P	5	11.3	5	11.3

Table 11: Cache Misses Sampled With Eviction Information

In general, the sampling technique ranked the variables correctly except when the actual difference between the number of cache misses being caused by two objects was small. The value reported for the percent of cache misses due to each variable was also accurate to within a small range of error. The largest error seen with sampling was for wupwise, in which the percentage for the array U reported by the sampling algorithm was off by 1.3 percentage points. Therefore, we can conclude

that the cache miss information gathered by the tool is sufficiently accurate, even with the extra overhead and perturbation caused by adding the sampling of cache eviction data.

5.3.1.2 Cache Evictions

Table 12 shows information about the evictions taking place in one of our test applications, mgrid. It lists the three objects that caused the most cache misses in the application in the "stat bucket" column. In the "evicted by" column, it lists the objects that caused more than 1% of the total evictions of each variable. The "rank" columns show the order of the objects when ranked by number of evictions of the variable they caused. The "%" columns show the percentage of all of evictions of the object in the "variable" column that were caused by the object in the "evicted by" column. Again, the "actual" columns show precise information as gathered by the simulator in a run with no instrumentation, while the "sampled" columns show the values collected by the instrumentation code.

Stat Bucket	Evicted By	Actual		Sampled	
		Rank	%	Rank	%
U	U	1	60.3	1	60.8
	R	2	20.2	2	20.1
	V	3	19.5	3	19.1
V	U	1	97.0	2	96.8
	V	2	2.9	3	3.2
R	R	1	73.8	1	74.1
	U	2	25.9	2	25.7

Table 12: Cache Evictions in Mgrid

Sampling one in 25,000 cache misses returned accurate information for the applications we tested. The largest difference between the actual and sampled values

in Table 12 is for the evictions of U by itself, for which the value estimated by sampling is approximately 0.5 percentage points higher than the actual one.

To quantify the accuracy of sampling across all the applications we tested, we measured the difference between actual and sampled values for the buckets that were identified as the top ten in terms of cache misses for each application. It is important to note that when measuring the error in the sampled data, we are only concerned with variables that are causing a large number of cache misses (which implies that they are experiencing a large number of cache evictions as well). Any variable identified as causing few cache misses can be disregarded as unimportant to performance tuning. Reflecting this, we discarded any buckets in the top ten that did not cause at least 10% of the total cache misses in an application. Out of the remaining buckets, the largest difference in the estimated percentage of evictions caused by a bucket to the actual value was seen in wupwise, with a difference of 5.1%.

Table 13 through Table 18 show eviction results from all applications tested. The objects shown are the top five in terms of cache misses, in order, excluding objects causing less than 1% of all cache misses. The row labels identify the objects causing evictions, and the column labels show the objects being evicted. The numbers in each box are the percentage of the total evictions of the column object that are caused by the row object. The variable names in su2cor that include the suffixes –i and –s indicate variables of the given names that are defined in the subroutines "intact" and "sweep," respectively, and the variable "spec_init" in gzip represents a block of memory dynamically allocated by the function "spec_init." We can see from the percentages shown that all applications show significant patterns in evictions of

some of the objects listed in the tables. For all six applications, there is at least one object listed that causes 35% or more of the cache evictions of another.

applu

		evicted				
		c	**a**	**b**	**d**	**rsd**
evicted by	**c**	41.5	4.0	21.1	27.9	8.4
	a	11.3	44.5	22.7	17.8	11.3
	b	23.7	21.9	43.7	1.9	7.8
	d	11.9	21.4	2.4	44.4	7.8
	rsd	5.5	4.1	6.7	4.6	52.2
	other	6.1	4.1	3.4	3.4	12.5

Table 13: Cache Eviction Matrix for Applu

gzip

		evicted		
		prev	**window**	**spec_init**
evicted by	**prev**	61.2	82.5	44.5
	window	36.0	15.2	39.9
	spec_init	1.0	0.1	5.2
	other	1.8	2.2	10.4

Table 14: Cache Eviction Matrix for Gzip

mgrid

		evicted		
		U	**R**	**V**
evicted by	**U**	60.8	25.7	96.8
	R	20.1	74.1	0.0
	V	19.1	0.2	3.2
	other	0.0	0.0	0.0

Table 15: Cache Eviction Matrix for Mgrid

su2cor

		evicted				
		U	**W1-i**	**W2-i**	**W2-s**	**W1-s**
evicted by	**U**	20.0	57.9	32.6	25.1	27.0
	W1-i	27.8	0.2	45.2	2.3	4.2
	W2-i	26.7	0.5	2.7	29.2	12.8
	W2-s	7.6	17.9	13.5	3.0	0.4
	W1-s	2.5	22.1	3.1	30.0	4.1
	other	15.4	1.4	2.9	10.4	51.5

Table 16: Cache Eviction Matrix for Su2cor

swim

		evicted				
		UNEW	**PNEW**	**VNEW**	**CU**	**CV**
evicted by	**UNEW**	6.7	51.3	23.5	0.0	32.7
	PNEW	28.8	7.0	30.5	0.0	0.0
	VNEW	31.5	25.1	11.7	0.1	0.0
	CU	0.0	0.0	0.0	35.1	34.9
	CV	0.0	0.0	16.0	0.0	0.0
	other	33.0	16.6	18.3	64.8	32.4

Table 17: Cache Eviction Matrix for Swim

wupwise

		evicted				
		U	**UD**	**T**	**S**	**P**
evicted by	**U**	41.4	35.7	40.8	25.2	27.3
	UD	16.0	50.3	0.0	0.0	8.8
	T	13.2	0.0	47.7	7.4	0.0
	S	11.2	0.0	6.5	67.4	0.8
	P	8.2	7.1	0.0	0.0	63.1
	other	10.0	6.9	5.0	0.0	0.0

Table 18: Cache Eviction Matrix for Wupwise

5.3.1.3 Evictions by Code Area

At the finest level of granularity supported by the eviction sampling instrumentation code, we keep counts for how many times each variable was evicted by each other variable at each line of code in the application. Table 19 shows an example, again from mgrid. For each variable named in the left column, it lists the five lines of code at which the most evictions of U caused by the named variable occur (excluding lines at which less than 1% of the total evictions of U occur). The lines are ranked by the number of evictions, and the percentages shown are the percent of all evictions of U caused by the given variable and line, both actually and as estimated by sampling. Even at this level of granularity, the results returned are close to the actual values, with the largest difference being the number of evictions of U

caused by accessing V at line 204 in the function resid; the estimated value is 1.1 percentage points lower than the actual one.

Stat Bucket	Function	Line	Actual		Sampled	
			Rank	%	Rank	%
U	resid	218	1	20.6	1	21.0
	psinv	162	2	10.8	2	9.9
	resid	216	3	4.6	3	4.2
	interp	287	4	3.1	6	3.1
	interp	308	5	2.9	4	3.5
	interp	296	7	2.8	5	3.1
V	resid	204	1	16.4	1	16.7
R	resid	204	1	19.6	1	20.1
	Psinv	176	2	0.2	2	0.2

Table 19: Percent of Total Evictions of U by Stat Bucket and Code Line

The accuracy seen with mgrid was typical of the applications we tested. To verify this, we again looked at the 10 buckets causing the most cache misses in each application, excluding any buckets causing less than 10% of the total cache misses. The largest error in the reported percentage of cache evictions of a given bucket caused by a particular combination of another bucket and a line of code was approximately 3.9 percentage points, seen in wupwise, for evictions of the variable T caused by cache misses in U. The error in the estimation accounts for only 0.7% of the total evictions of the variable. Table 20 shows the evictions of T by U caused by each line of code, excluding information about lines that cause less than 1% of the evictions of T. Although the error causes the two lines of code shown to be ranked incorrectly, the estimates made by sampling are sufficiently close to be useful.

Function	Line	Actual		Sampled	
		Rank	%	Rank	%
zgemm	263	1	16.8	2	12.8
zgemm	250	2	15.8	1	14.1

Table 20: Evictions of T by U in Wupwise

5.3.2 Perturbation of Results

As we did for sampling cache misses in Section 3.3.2, we will now look at how sampling cache evictions affects the cache behavior of an application. Figure 12 shows the percentage increase in cache misses due to instrumentation code when running each of the applications and sampling at several sampling frequencies. This information was obtained by comparing the number of cache misses in a run without instrumentation (cache misses are still measured by the simulator) with the number of misses in a set of runs in which we sampled one in 250, one in 2,500, one in 25,000, and one in 250,000 cache misses. Note that the scale of the y axis is logarithmic.

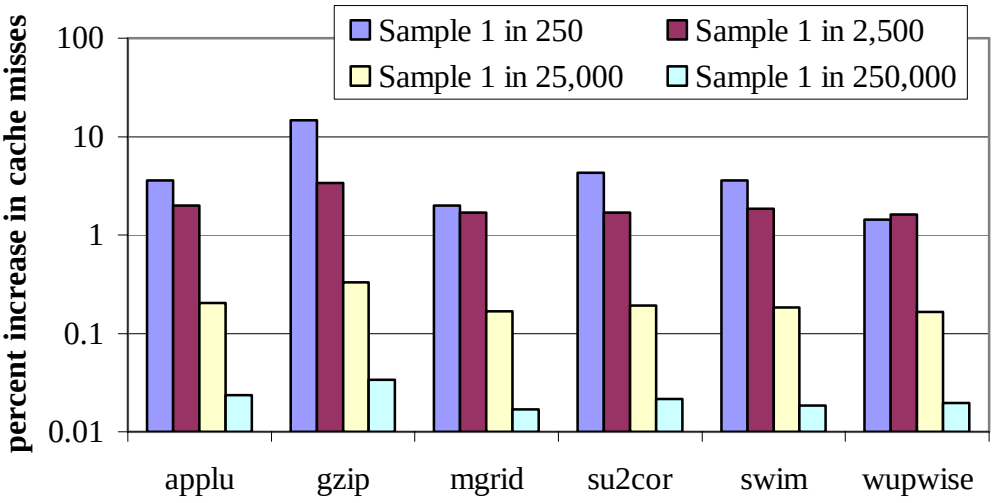

Figure 12: Percent Increase in Cache Misses When Sampling Evictions

At our default sampling frequency, one in 25,000 misses, the increase in cache misses was extremely low for all applications. We see the largest increase with gzip, which experienced approximately 0.3% more cache misses with instrumentation than

without. At higher sampling frequencies, the instrumentation code begins to significantly perturb the results; for gzip, sampling one in 250 misses results in a 15% increase in cache misses. The average increase across all applications at this sampling frequency was approximately 5%. As we saw when sampling only cache misses, this shows that sampling more frequently does not always lead to higher accuracy, due to the instrumentation code's effect on the cache.

5.3.3 Instrumentation Overhead

Figure 13 shows the overhead that is added to the execution time of each application by the instrumentation code when sampling cache evictions at several frequencies. This includes the virtual cycle count of the instructions executed in the instrumentation code, as well as a per-interrupt cost for handling an interrupt and delivering it to instrumentation code.

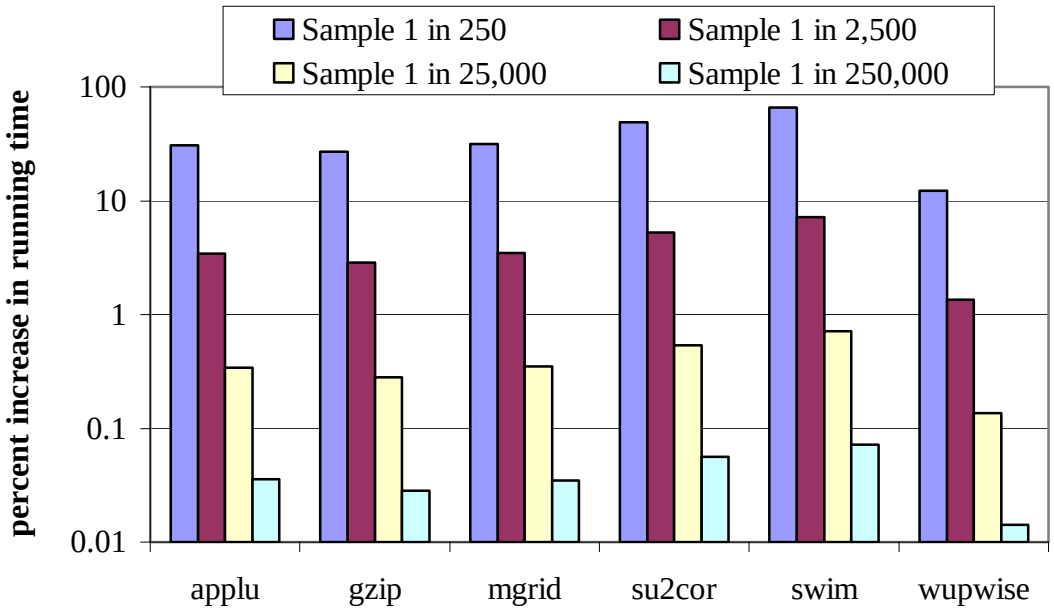

Figure 13: Instrumentation Overhead When Sampling Cache Evictions

At the default sampling frequency of one in 25,000 misses, the highest overhead was seen in swim, which had an increase in execution time of slightly less than 1%. The overhead becomes more significant at higher samples frequencies, with the overhead for swim rising to 66% when sampling one in 250 cache misses. The average overhead over all applications when sampling one in 250 cache misses was approximately 36%.

5.4 Performance Tuning Using Data Centric Eviction Information

This section will present an example of using the data provided by the cache eviction tool to optimize an application. We will examine mgrid from the SPEC CPU2000 benchmark suite. For this application, our tool indicates that two arrays, U and R, cause approximately 90% of all cache misses. Looking at the eviction information for mgrid in Table 12, we find that each of these is most often being evicted by accesses to itself.

To better understand this problem, we looked at the next finer level of granularity in the data to determine what parts of the code are causing this to happen. Table 21 shows the lines of code at which the most evictions of U by U and R by R are occurring.

Bucket/Evicted By	Function	Line	% Evictions
U evicted by U	resid	218	21.0
	psinv	162	9.9
	resid	216	4.2
R evicted by R	psinv	168	14.9
	psinv	174	14.6
	psinv	176	13.8

Table 21: Evictions by Code Region in Mgrid

Three lines together cause almost 32% of all evictions of U by itself, one in the function resid and the others in the function psinv. For evictions of R by itself, the table shows that a small set of lines from psinv cause approximately 43% of all such evictions.

Looking at the function "resid," we find the loop shown in Figure 14. The array U that is used in this loop is declared elsewhere as a large single-dimensional array, parts of which are passed into resid and other functions in such a way that they are interpreted as one- or three-dimensional arrays of various sizes; in the case of resid, part of U is passed in as an N by N by N array. The array R is used similarly. The fact that these arrays are declared and used in this way may prevent the compiler from performing optimizations that would involve changing their layout in memory, since the layout depends on values computed at runtime.

```
      DO 600 I3=2,N-1
      DO 600 I2=2,N-1
      DO 600 I1=2,N-1
600   R(I1,I2,I3)=V(I1,I2,I3)
     >      -A(0)*( U(I1,  I2,  I3  ) )
     >      -A(1)*( U(I1-1,I2,  I3  ) + U(I1+1,I2,  I3  )
     >                  +  U(I1,  I2-1,I3  ) + U(I1,  I2+1,I3  )
     >                  +  U(I1,  I2,  I3-1) + U(I1,  I2,  I3+1) )
     >      -A(2)*( U(I1-1,I2-1,I3  ) + U(I1+1,I2-1,I3  )
     >                  +  U(I1-1,I2+1,I3  ) + U(I1+1,I2+1,I3  )
     >                  +  U(I1,  I2-1,I3-1) + U(I1,  I2+1,I3-1)
     >                  +  U(I1,  I2-1,I3+1) + U(I1,  I2+1,I3+1)
     >                  +  U(I1-1,I2,  I3-1) + U(I1-1,I2,  I3+1)
     >                  +  U(I1+1,I2,  I3-1) + U(I1+1,I2,  I3+1) )
     >      -A(3)*( U(I1-1,I2-1,I3-1) + U(I1+1,I2-1,I3-1)
     >                  +  U(I1-1,I2+1,I3-1) + U(I1+1,I2+1,I3-1)
     >                  +  U(I1-1,I2-1,I3+1) + U(I1+1,I2-1,I3+1)
     >                  +  U(I1-1,I2+1,I3+1) + U(I1+1,I2+1,I3+1) )
```

Figure 14: Loop from Function Resid

With the reference data set from the SPEC2000 benchmarks, resid is called with varying values for N, up to 130. Each element of U is eight bytes, so the array U

can be over 16MB in size. Because of the large size of the array, the references to U with subscripts I2-1 to I2+1, and I3-1 to I3+1 will likely be evicted from the cache before being reused in other iterations, suggesting that tiling [45, 85] would be effective at increasing reuse. We tiled the loop with a tile size of 8 by 8 by 8, which allowed an entire tile for each of the three arrays accessed to fit into the L1 cache. We also padded the first dimension of the array to make its size a multiple of the cache line size and in such a way as to help eliminate conflicts within tiles. We then padded the beginning of the arrays so that they would start on cache line boundaries as used in resid. Note that as mentioned above, the arrays are not used as first declared in the program, which must be taken into account when padding. For instance, the main program passes part of U, offset from the beginning, into resid as resid's argument U, so the main program's U must be padded such that the offset begins on a cache line boundary. Finally, since the code inside the loop is short, we unrolled the innermost loop over a tile, in order to eliminate some of the extra overhead of the new loops introduced for tiling. The function "psinv" has a loop similar to the one in "resid," to which the same optimizations were applied.

While a compiler could potentially apply the code transformations mentioned automatically, for the reasons discussed above it would be difficult for it to combine them with changing the layout of the arrays, making it advantageous to perform the transformations manually.

Figure 15 shows the number of cache misses in U, V, and R before and after the optimizations. Although slightly more cache misses take place in V (2%), there are 29% and 20% fewer misses in U and R, respectively. Overall, cache misses were

reduced by 22%. Looking only at cache misses in resid, our simulator shows that there are 48% fewer misses in U, but approximately 2% more misses in V and R, for an overall improvement of 29%. The "psinv" function shows a similar pattern; R causes 48% fewer cache misses, while U causes 2% more. These provide a speedup in these functions of 11% for resid and 7% for psinv, and an overall speedup of 8%.

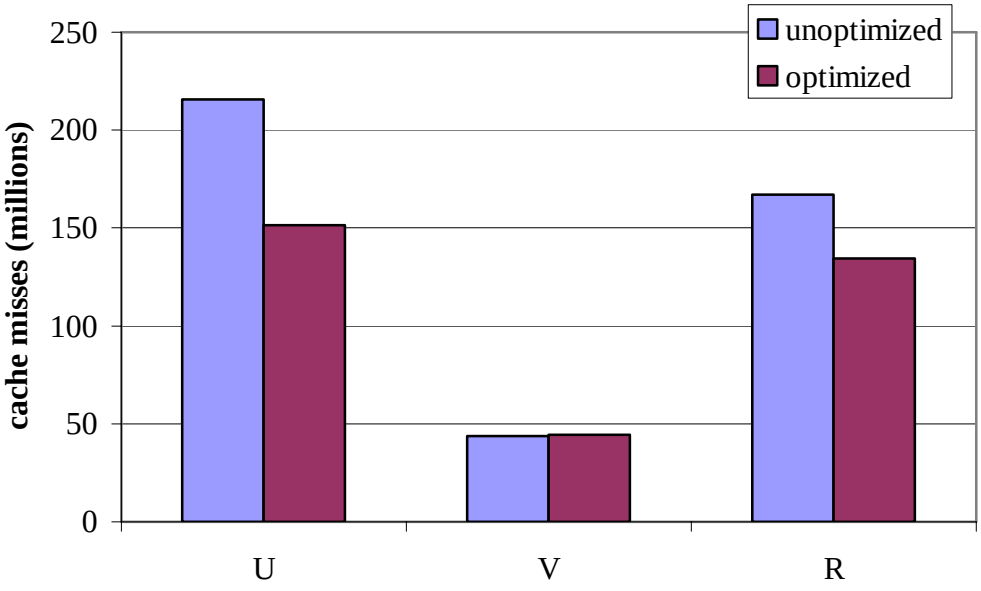

Figure 15: Cache Misses in Mgrid Before and After Optimization

5.5 Conclusions

In this chapter, we have discussed a proposed hardware feature that would capture information about the data that is evicted from the cache when a miss occurs. We presented a technique that uses this feature to sample cache eviction addresses, in order to provide feedback to a user about how data structures are interacting in the cache. We implemented this technique in a simulator, and ran a series of experiments in which we measured the cache evictions in a set of benchmark applications. These experiments showed that accuracy and overhead were acceptable, even though the instrumentation code must use larger data structures that hold more fine-grained in-

formation than was the case when sampling only cache misses. When looking specifically at the finer-grained information, such as the cache evictions taking place at a particular place in the code, accuracy is reduced for objects that do not cause many cache misses. However, this is not a serious limitation, since for performance tuning we are most interested in the objects that cause the most cache misses.

To examine the value of the information gathered by sampling evictions, we used the tool to analyze one application, mgrid, in detail. Using information from the tool, we were able to perform optimizations by hand that reduced the running time of the application by 8%, showing the usefulness of this information.

Chapter 6: Conclusions

In this dissertation, we have examined the problem of providing useful feedback to a programmer about the cache behavior of application data structures at the source code level. We refer to this as data centric cache measurement. We have shown that the information needed to do this can be gathered using hardware performance monitors that can be practically incorporated into processor designs. These monitors are used by software instrumentation that collects and aggregates the data.

We first described a technique in which software instrumentation uses hardware performance monitors that provide information about cache miss addresses to sample those addresses and relate them to data structures. The results of our study of this technique in simulation showed that sampling allows us to gather accurate information with low overhead. We also examined the overhead of the cache simulator, which was high; for the applications we tested, the slowdown when running under the simulator ranged from 37 to 109 times the normal execution time of the applications. This is a strong argument for why hardware support is needed in order to perform data centric cache measurement.

We next described a tool named Cache Scope, which is an implementation of data centric cache measurement on actual hardware, the Intel Itanium 2 processor. Using Cache Scope, we demonstrated the practicality of the sampling technique in a real system. We found that the instrumentation's perturbation of cache behavior and overhead can become significant if samples are taken too often, but that for Cache Scope running on the Itanium 2 sampling approximately one in every 32K cache events provided acceptable accuracy and overhead for the applications we tested.

One difference between the Itanium 2 and the simulator is that the Itanium 2 provides information about the latency associated with each cache miss. We found this information 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. This is especially true because the L1 cache does not store floating point values. Cache Scope therefore ranks data structures by cumulative latency, not by number of cache misses.

We used Cache Scope to analyze two applications, and tuned them based on the results. We were able to achieve a 24% reduction in running time on one application and an almost 19% reduction in the other. This was done almost entirely by changing the data structures in the applications, with few changes to the code. Furthermore, the tuning was performed in a short time, one day in the case of one application, by a programmer who was not previously familiar with the applications' source code. From this, we can conclude that the tool was useful in providing feedback about cache behavior for performance tuning.

This dissertation also discussed a proposed hardware feature that would provide information about the data that is evicted from the cache when a miss occurs. We described a technique for using this feature to sample cache eviction information and to provide feedback to a user about how data structures are interacting in the cache. We implemented this technique in simulation and performed a study in which we used it to measure cache evictions in a number of applications. We found that even though it requires processing of much finer-grained information than was the case when sampling only cache misses, we were still able to collect the information

with sufficient accuracy and low overhead. We did find that the accuracy of the eviction information degrades when looking at objects causing few cache misses. Though this must be taken into account, it does not significantly affect the usefulness of the tool, since for performance tuning we are concerned only with objects that are causing many cache misses.

We found that in many of the applications we tested, there were objects in memory for which some other particular object caused most of the cache evictions, making the data significant for helping to understand how data structures are interacting in the cache. We examined one application, mgrid, in more detail, and using information gained from the tool were able to improve its performance by 8%. From this, we conclude that sampling cache eviction information is useful in providing the user with feedback about cache behavior for performance tuning.

Throughout the studies we have described, we found that hardware support for obtaining cache miss and eviction addresses enables the creation of tools that provide feedback to the user about the cache behavior of data structures at the source code level. The creation of these tools would not otherwise be possible, except through techniques such as simulation, which have severe limitations such as high overhead. We think that this is a strong argument for including such features in future processors, and for making them available to software developers. In some cases, such as the Intel Itanium 2, this has already started to happen, and some of the techniques described in this dissertation can be used on these today. In many cases, however, performance monitoring features continue to be limited, or even if they are present, may be partly or entirely undocumented. An example of this is the IBM POWER4 proces-

sor, which possesses sophisticated performance monitoring features that are unfortunately mostly undocumented (although the PMAPI [1] kernel interface provides access to a subset of these features). We hope that demonstrating the usefulness of these features will lead to more vendors including and documenting them.

6.1 Summary of Contributions

This dissertation has made a number of contributions in answering the question of how to provide feedback to a user about the cache behavior of data structures at the source code level. One such contribution is to show how hardware performance monitors that can provide the addresses related to cache misses, along with the ability to generate periodic interrupts when cache misses occur, can be used to measure the cache behavior of data structures. Using simulation and an implementation on the Intel Itanium 2 processor, we showed that this technique can gather the desired information accurately and with low overhead. This had not previously been done, since prior tools had either used simulation for the tool itself (as opposed to as a method of validating a hardware approach), or were unable to determine the addresses associated with specific cache misses.

This dissertation also demonstrated the usefulness of this data centric information, by describing how it was used to improve the performance of two applications from the SPEC CPU2000 benchmark suite.

Furthermore, this dissertation introduced the idea of a new hardware feature that would provide information about the data that is evicted from the cache when a miss occurs. It described in detail how software instrumentation could use the information from such a feature to provide feedback about how data structures are inter-

acting in the cache. Using simulation, we showed that this technique is able to gather accurate information for the most important objects in an application, while maintaining a low overhead. We showed that this information is useful in performance tuning by using it to improve the performance of a sample application.

6.2 Future Research

In the future, more processors may become available that provide cache miss addresses. It would useful to port the Cache Scope cache miss sampling tool to such processors, and to study the ways in which each architecture's unique features affect cache performance.

It would also be interesting to examine ways to automatically control the overhead and perturbation of the instrumentation code by dynamically changing the sampling frequency. Although we have shown that it is possible to choose a sampling frequency that is appropriate for a wide range of applications, this would further improve the robustness of a sampling tool.

Another interesting area of study would be how to use data centric cache information to provide feedback to a compiler. Based on this information, the compiler could automatically change the layout of data structures or alter the code that accesses them in order to improve use of the cache. This would be especially useful for problems that would be difficult for a compiler to analyze statically, such as difficult to analyze uses of pointers in C.

The idea of automatically using feedback could also be extended to memory allocation. The results from cache miss sampling could be used by the memory allocator to decide where newly allocated blocks of memory should be placed. Possibly

the feedback could be used in the same run of the application in which it was gathered – cache miss addresses could be continuously sampled, and the memory allocator could adapt based on the latest results. If eviction information is available, it could be particularly useful, since it may provide information about cache conflicts between data structures that are being used concurrently; future allocations could attempt to avoid such conflicts.

No existing processor includes a way to sample cache eviction addresses. If this could be implemented in hardware, it would make it possible to use this technique on a much wider set of applications, many of which cannot be run under a simulator due to memory or performance constraints.

There are other uses of eviction addresses that could be examined as well. As one example, by looking at the difference between misses and evictions for a certain data structure, we should be able to estimate how much of the data structure is being kept in the cache at any given time. It would be useful to investigate whether sampling provides sufficient accuracy to make an estimate of this value useful.

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. Agrawal, A., Sites, R.L. and Horowitz, M., ATUM: A New Technique for Capturing Address Traces Using Microcode. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, (1986), 119-127.

6. 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).

7. 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.

8. Berrendorf, R., Ziegler, H. and Mohr, B. The Performance Counter Library (PCL) web site, Research Centre Juelich GmbH, 2003. http://www.fz-juelich.de/zam/PCL/

9. Bershad, B.N., Lee, D., Romer, T.H. and Chen, J.B., Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. In *Proceedings of the 6th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, (1994), 158-170.

10. Bishop, M. Profiling Under UNIX by Patching. *Software Practice and Experience*, *17* (10). 729-739.

11. Bodin, F., Beckman, P., Gannon, D., Gotwals, J., Narayana, S., Srinivas, S. and Winnicka, B., Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. In *Proceedings of the Second Annual Object-Oriented Numerics Conference (OON-SKI)*, (Sunriver, OR, 1994), 122-138.

12. Bodin, F., Beckman, P., Gannon, D., Narayana, S. and Yang, S.X. Distributed pC++: Basic Ideas for an Object Parallel Language. *Scientific Programming, 2* (3).

13. Brantley, W.C., McAuliffe, K.P. and Ngo, T.A. RP3 Performance Monitoring Hardware. in Simmons, M., Koskela, R. and Bucker, I. eds. *Instrumentation for Future Parallel Computer Systems*, Addison-Wesley, 1989, 35-47.

14. 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.

15. Callahan, D., Carr, S. and Kennedy, K., Improving Register Allocation for Subscripted Variables. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (White Plains, NY, 1990), 53-65.

16. Callahan, D., Kennedy, K. and Porterfield, A., Software Prefetching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, (Santa Clara, CA, 1991), 40-52.

17. Chame, J. and Moon, S., A Tile Selection Algorithm for Data Locality and Cache Interference. In *Proceedings of the 1999 International Conference on Supercomputing*, (Rhodes, Greece, 1999), 492-499.

18. Chatterjee, S., Jain, V.V., Lebeck, A.R., Mundhra, S. and Thottethodi, M., Nonlinear Array Layouts for Hierarchical Memory Systems. In *Proceedings of the 1999 International Conference on Supercomputing*, (Rhodes, Greece, 1999), 444-453.

19. 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).

20. Chilimbi, T.M., Davidson, B. and Larus, J.R., Cache-Conscious Structure Definition. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (Atlanta, GA, 1999), 13-24.

21. 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.

22. Chilimbi, T.M. and Hirzel, M., Dynamic Hot Data Stream Prefetching for General-Purpose Programs. In *Proceedings of the ACL SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (Berlin, Germany, 2002).

23. Cmelik, R.F. and Keppel, D., Shade: A Fast Instruction-Set Emulator for Execution Profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, (1994), 128-137.

24. Coleman, S. and McKinley, K.S., Tile Size Selection Using Cache Organization and Data Layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (La Jolla, California, 1995), 279-290.

25. Compaq Computer Corporation *Alpha Architecture Handbook (Version 4)*, 1998.

26. Cox, A.L. and Fowler, R.J., Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, (1993).

27. De Rose, L., Ekanadham, K. and Hollingsworth, J.K., SIGMA: A Simulator Infrastructure to Guide Memory Analysis. In *Proceedings of SC2002*, (Baltimore, MD, 2002).

28. Ding, C. and Kennedy, K., Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (Atlanta, GA, 1999), 229-241.

29. Eggers, S.J., Keppel, D.R. and Koldinger, E.J., Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, (1990), 37-47.

30. Fursin, G., O'Boyle, M.F.P., Temam, O. and Watts, G. A Fast and Accurate Method for Determining a Lower Bound on Execution Time. *Concurrency and Computation: Practice and Experience*, *16* (2-3). 271-292.

31. Ghosh, S., Martonosi, M. and Malik, S., Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, (San Jose, California, 1998), 228-239.

32. Glass, G. and Cao, P., Adaptive Page Replacement Based on Memory Reference Behavior. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, (Seattle, WA, 1997), 115-126.

33. 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.

34. Henning, J.L. SPEC CPU2000: Measuring CPU Performane in the New Millenium. *Computer*, *33* (7). 28-35.

35. Horowitz, M., Martonosi, M., Mowry, T.C. and Smith, M.D., Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, (Philadelphia, PA, 1996).

36. Huck, J., Morris, D., Ross, J., Knies, A., Mulder, H. and Zahir, R. Introducing the IA-64 Architecture. *IEEE Micro*, *20* (5). 12-23.

37. Hundt, R. HP Caliper: A Framework for Performance Analysis Tools. *IEEE Concurrency*, *8* (4). 64-71.

38. Itzkowitz, M., Wylie, B.J.N., Aoki, C. and Kosche, N., Memory Profiling using Hardware Counters. In *Proceedings of SC2003*, (Phoenix, AZ, 2003).

39. 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).

40. Johnson, S.C., Postloading for Fun and Profit. In *Proceedings of the USENIX Winter Conference*, (1990), 325-330.

41. Kandemir, M., Bannerjee, P., Choudhary, A., Ramanujam, J. and Ayguade, E., An Integer Linear Programming Approach for Optimizing Cache Locality. In *Proceedings of the 1999 International Conference on Supercomputing*, (Rhodes, Greece, 1999), 500-509.

42. Kodukula, I., Ahmed, N. and Pingali, K., Data-Centric Multi-Level Blocking. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (Las Vegas, NV, 1997), 346-357.

43. Kodukula, I., Pingali, K., Cox, R. and Maydan, D., An Experimental Evaluation of Tiling and Shackling for Memory Hierarchy Management. In *Proceedings of the 1999 International Conference on Supercomputing*, (Rhodes, Greece, 1999), 482-491.

44. Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., Gupta, A., Rosenblum, M. and Hennessy, J., The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, (Chicago, IL, 1994), 302-313.

45. Lam, M.S., Rothberg, E.E. and Wolf, M.E., The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the International Con-*

ference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV), (San Jose, California, 1991), 63-74.

46. Larus, J.R. Abstract Execution: A Technique for Efficiently Tracing Programs. *Software Practice and Experience*, *20* (12). 1241-1258.

47. Larus, J.R. and Ball, T. Rewriting Executable Files to Measure Program Behavior. *Software -- Practice and Experience*, *24* (2). 197-218.

48. Larus, J.R. and Schnarr, E., EEL: Machine-Independent Executable Editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (La Jolla, CA, 1995), ACM, 291-300.

49. Lauterbach, G. and Horel, T. UltraSPARC-III: Designing Third Generation 64-Bit Performance. *IEEE Micro*, *19* (3). 73-85.

50. Lebeck, A.R. and Wood, D.A. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer*, *27* (9). 15-26.

51. Lee, H.B. and Zorn, B.G., BIT: A Tool for Instrumenting Java Bytescodes. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, (Monterey, CA, 1997), 73-82.

52. Lipasti, M.H., Schmidt, W.J., Kunkel, S.R. and Roediger, R.R., SPAID: Software Prefetching in Pointer- and Call-Intensive Environments. In *Proceedings of the International Symposium on Microarchitecture*, (Ann Arbor, MI, 1995), 231-236.

53. Lu, J., Chen, H., Fu, R., Hsu, W.-C., Othmer, B., Yew, P.-C. and Chen, D.-Y., The Performance of Runtime Data Cache Prefeteching in a Dynamic Optimization System. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, (San Diego, CA, 2003), 180-190.

54. Luk, C.-K. and Mowry, T.C., Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, (Cambridge, 1996), 222-233.

55. 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.

56. Malony, A.D. and Reed, D.A., A Hardware-Based Performance Monitor for the Intel iPSC/2 Hypercube. In *Proceedings of the 1990 International Conference on Supercomputing*, (Amsterdam, 1990), 213-226.

57. Martonosi, M., Gupta, A. and Anderson, T., Effectiveness of Trace Sampling for Performance Debugging Tools. In *Proceedings of the 1993 ACM SIG-*

METRICS Conference on Measurement and Modeling of Computer Systems, (1993).

58. Martonosi, M., Gupta, A. and Anderson, T., MemSpy: Analyzing Memory System Bottlenecks in Programs. In *Proceedings of the 1992 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, (Newport, Rhode Island, 1992), 1-12.

59. Martonosi, M., Ofelt, D. and Heinrich, M., Integrating Performance Monitoring and Communication in Parallel Computers. In *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, (Philadelphia, PA, 1996).

60. McKinley, K.S., Carr, S. and Tseng, C.-W. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems, 18* (4). 424-453.

61. Mellor-Crummey, J., Whalley, D. and Kennedy, K., Improving Memory Hierarchy Performance for Irregular Applications. In *Proceedings of the 1999 International Conference on Supercomputing*, (Rhodes, Greece, 1999), 425-432.

62. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K. and Newhall, T. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer, 28* (11). 37-46.

63. Mink, A. *Operating Principles of Multikron II Performance Instrumentation for MIMD Computers*. National Institute of Standards and Technology, NISTIR 5571, Gaithersburg, MD, 1994.

64. Mink, A., Carpenter, R., Nacht, G. and Roberts, J. Multiprocessor Performance Measurement Instrumentation. *IEEE Computer, 23* (9). 63-75.

65. Mowry, T.C., Lam, M.S. and Gupta, A., Design and Implementation of a Compiler Algorithm for Prefetching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, (Boston, MA, 1992), 62-73.

66. 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).

67. Nahshon, I. and Bernstein, D., FDPR - A Post-pass Object-code Optimization Tool. In *Proceedings of International Conference on Compiler Construction*, (Linkoping, Sweden, 1996), Springer-Verlag, 355.

68. Noe, R.J. and Aydt, R.A. *Pablo Instrumenation Environment User's Guide*. University of Illinois, 1996.

69. Panda, P.R., Nakamura, H., Dutt, N.D. and Nicolau, A. Augmenting Loop Tiling with Data Alignment for Improved Cache Performance. *IEEE Transactions on Computers, 48* (2). 142-149.

70. Pingali, V.K., McKee, S.A., Hseih, W.C. and Carter, J.B., Computation Regrouping: Restructuring Programs for Temporal Data Cache Locality. In *Proceedings of the 16th International Conference on Supercomputing*, (New York, NY, 2002), 252-261.

71. Qiao, X., Gan, Q., Liu, Z., Guo, X. and Li, X., Cache Optimization in Scientific Computations. In *Proceedings of the ACM Symposium on Applied Computing*, (February 1999, 1999), 548-552.

72. Quinlan, D., Rose: A Preprocessor Generation Tool for Leveraging the Semantics of Parallel Object-Oriented Frameworks to Drive Optimizations via Source Code Transformations. In *Proceedings of the Eighth International Workshop on Compilers for Parallel Computers (CPC '00)*, (Aussois, France, 2000).

73. Reed, D.A., Aydt, R.A., Noe, R.J., Roth, P.C., Shields, K.A., Schwartz, B.W. and Tavera, L.F. Scalable Performance Analysis: The Pablo Performance Analysis Environment. in Skjellum, A. ed. *Scalable Parallel Libraries Conference*, IEEE Computer Society, 1993, 104-113.

74. 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).

75. Ries, B., Anderson, R., Auld, W., Breazeal, D., Callaghan, K., Richards, E. and Smith, W., The Paragon Performance Monitoring Environment. In *Proceedings of Supercomputing '93*, (Portland, OR, 1993), 850-859.

76. Rivera, G. and Tseng, C.-W., Data Transformations for Eliminating Conflict Misses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (Montreal, Canada, 1998), 38-49.

77. Rivera, G. and Tseng, C.-W., Tiling Optimizations for 3D Scientific Computations. In *Proceedings of SC2000*, (Dallax, Texas, 2000).

78. Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., H. Levy, H. and Bershad, B., Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, (Seattle, WA, USA, 1997), 1-7.

79. Sechen, C. and Sangiovanni-Vincentelli, A. The TimberWolf Placement and Routing Package. *IEEE Journal of Solid-State Circuits, 20* (2). 432-439.

80. Sharangpani, H. and Arora, K. Itanium Processor Microarchitecture. *IEEE Micro, 20* (5). 24-43.

81. Srivastava, A. and Eustace, A., ATOM: A system for Building Customized Program Analysis Tools. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (Orlando, FL, 1994), 196-205.

82. Temam, O., Fricker, C. and Jalby, W., Cache Interference Phenomena. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, (Nashville, Tennessee, 1994), 261-271.

83. 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.

84. Wilson, L.S., Neth, C.A. and Rickenbaugh, M.J. Delivering Binary Object Modification Tools for Program Analysis and Optimization. *Digital Technical Journal, 8* (1). 18-31.

85. Wolf, M.E. and Lam, M.S., A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (Toronto, Ontaro, Canada, 1991), 30-44.

86. Wybranietz, D. and Haban, D., Monitoring and Performance Measuring Distributed Systems during Operation. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, (Santa Fe, New Mexico, 1988), 197-206.

87. 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).