To Appear *International Journal of High Performance Computing Applications, Vol. 20, No. 6*, Fall 2006.

A New Hardware Monitor Design to Measure Data Structure-Specific Cache Eviction Information

Bryan R. Buck[†]
Symantec Corporation
6750 Alexander Bell Drive
Columbia, MD 21046
buck@cs.umd.edu

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

Abstract

In this paper, we propose a hardware performance monitor that provides support not only for measuring cache misses and the addresses associated with them, but also for determining what data is being evicted from the cache when a miss occurs. We describe how to use this hardware support to efficiently determine the cache behavior of application data structures at the source code level. We also present the results of a simulation-based study of this technique, in which we examined the overhead, perturbation of results, and usefulness of collecting this information.

1 Introduction

Because of the relatively slow speed of accessing a system's main memory, the effective utilization of memory caches is important for high performance applications. While many commercial applications efficiently use the cache, HPC applications frequently only achieve single digit percentage of peak performance due to poor cache utilization. As processor speeds continue to increase faster than memory speeds, the significance of this problem continues to grow. This makes information about the cache behavior of an application extremely valuable to a programmer who is trying to tune its performance. In order to provide information that a programmer can easily understand and act upon, this information should be presented in terms of data structures at the source code level.

It is desirable to use hardware performance monitors to gather such information, since they allow this to be done with lower overhead than all-software alternatives such as simulation. However, for this to be possible, the processor must provide the necessary features. Although such features have been limited in the past, the trend is toward including increased support for performance monitoring, including that for measuring cache behavior. 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 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 [1] supports this feature, and reportedly so does the IBM POWER4 [23] (although documentation on its use is not publicly available).

This paper describes a next generation of processor monitor that can also gather information about the addresses of data that are evicted as a result of cache misses. Such a feature would allow a tool to determine how data structures are interacting in the cache. As an example, if two data structures are causing many evictions of each other, then the accesses to them may be conflicting in the cache.

2 Data Structure-Specific Miss and Eviction Information

In this section, we will discuss the hardware features that are necessary to measure data structure-specific information about cache misses and evictions, and how measurement code can efficiently make use of these features.

2.1 Hardware Features

In order to measure the number of cache misses and evictions that relate to specific data structures, it is necessary for instrumentation code to be able to determine the addresses that are being accessed to cause the misses, and of the data that are evicted. Our proposed hardware monitor for doing this is illustrated in Figure 1. In general, the mapping of addresses to data structures 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; the processor can simply save it or pass it to a routine that is triggered by the cache miss (such as an interrupt handler).

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

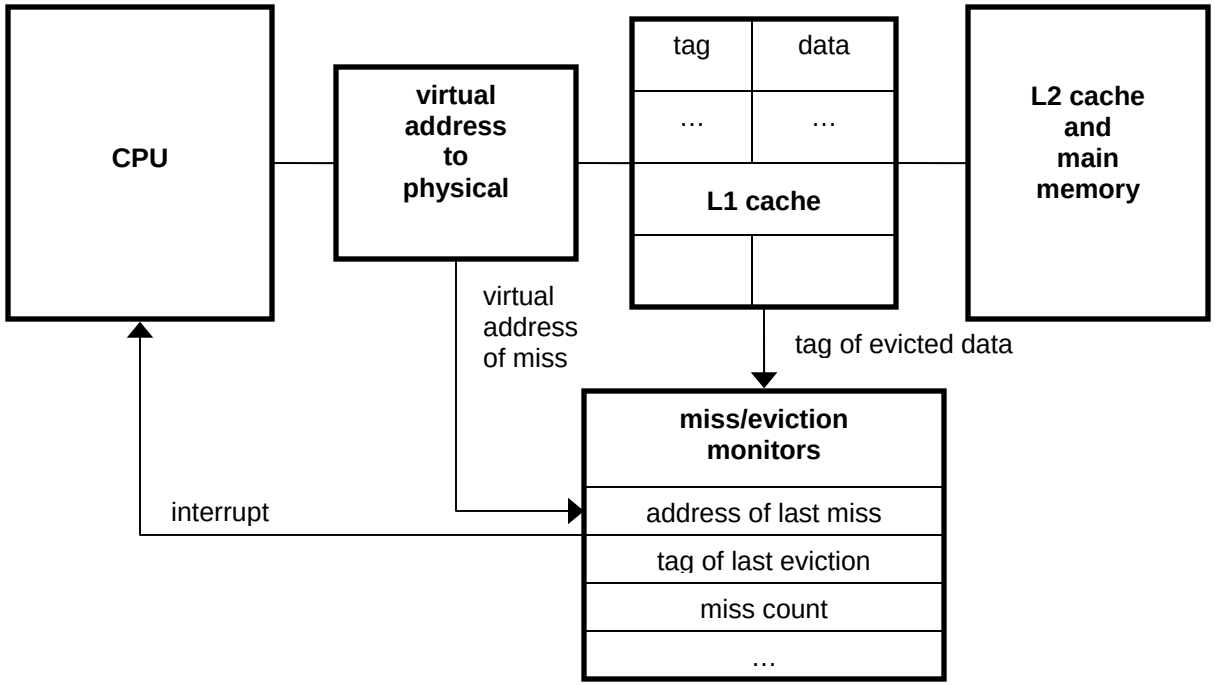

Figure 1: Performance Monitors for Cache Misses and Evictions.

Knowing the address of a block of memory that is evicted from the cache is somewhat more difficult. A cache must of course maintain information about the mapping of the data it contains to main memory. This is done by tagging each line with a number that represents the corresponding block of memory. When a cache replacement occurs, the tag of the evicted data could be written into a special register, and then used to determine the address. In caches that use a tag based on a physical address, it is necessary to map this to a virtual address in the program's address space. This can be done using information about the mapping of physical to virtual pages of memory, which could be supplied by the operating system. One potential problem with this is that operating system features such as page replacement and the ability to map the same physical page into multiple virtual locations can make this mapping difficult. For the experiments described in this paper, we have assumed that perfect information is available about the 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.

As we will discuss in the next section, one more feature that is necessary in order to allow the efficient collection of this data is a way to cause periodic interrupts when cache misses occur. For example, a number of existing processors allow an application to set an initial value in the cache miss counter, and to cause an interrupt to be triggered when the counter overflows. Instrumentation code can use these features together to cause an interrupt to occur at one out of every *n* cache misses.

2.2 Software Support for Measuring Misses and Evictions

Each time the cache miss interrupt is triggered, the monitoring software examines: the address of the data that missed in the cache, the cache tag that was evicted from the cache, and the program counter. The addresses are then mapped to the source code level memory objects (variables or dynamically allocated data) that contain them. This is done using debugging symbols and information maintained by instrumentation code inserted into the memory allocation functions. The program counter is used to map back to source line numbers (using the line number table provided by the compiler's debug flag).

Memory objects are grouped into equivalence classes, which we refer to as *stat buckets*. Each variable in the application makes up a stat bucket. Dynamically allocated memory is assigned to a stat bucket based on the execution path leading to the allocation function that created it; this tends to group heap blocks that are used similarly together.

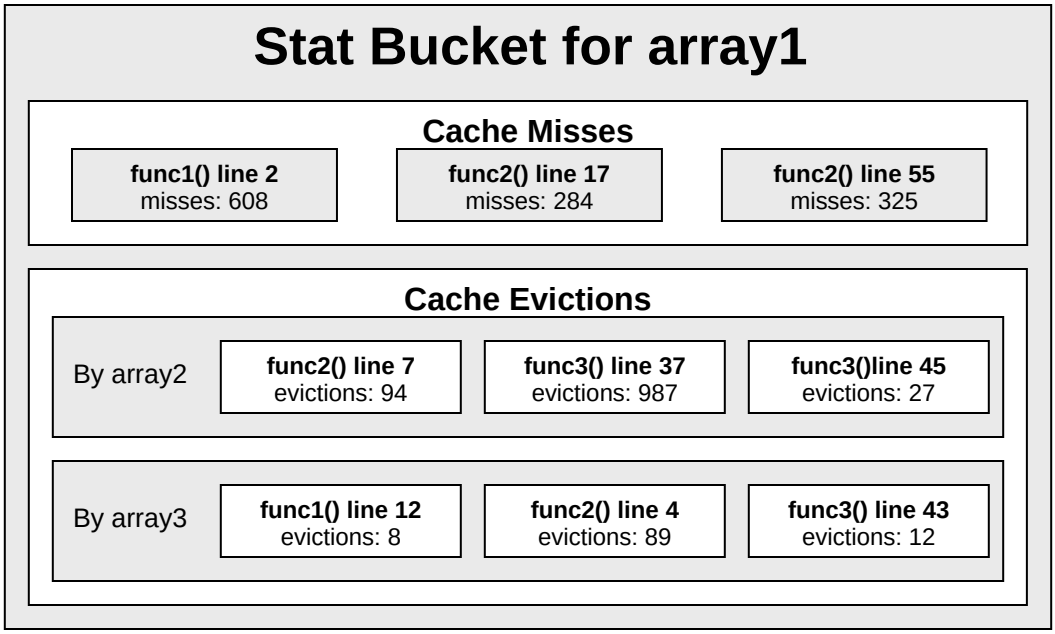

Figure 2: Stat Bucket Data Structure.

The information maintained about each stat bucket is shown in Figure 2. In addition to the number of misses and evictions that take place within the objects of each bucket, we also maintain information about what objects are causing the evictions, and where in the code the misses and evictions take place. Since misses and evictions generally 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 available in real-time as the application executes. Our current implementation does maintain miss information separately from eviction information.

3 Experiments

In order to evaluate the overhead, correctness, and usefulness of gathering cache miss and eviction information using the approach described above, we implemented a simulator for the necessary hardware. The simulator is implemented using the ATOM [21] binary rewriting tool. All load and store instructions in the application are instrumented with code that tracks memory references and simulates an L1 cache. Each basic block is also instrumented with code that maintains a virtual cycle count for the execution. The cycle counts do not represent any specific processor, but are meant to model RISC processors in general. The simulated L1 cache has configurable parameters such as cache size, associativity, and line size. For our experiments, we used a 64KB four-way set associative cache with a line size of 32 bytes. These values were chosen to represent current RISC processors. 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.

The simulator does not model details such as pipelining or multiple instruction issue. Not modeling out of order execution does not cause a serious problem for our results. Processors such as the Itanium and Power 4 include extensive out of order execution and still provide sampling based monitors that can relate a given miss to the associated program counter, what they lack is the ability to record information about the evicted cache line. In a previous study[4], we validated the accuracy of the simulator used in the paper with hardware counters available on the Itanium 2.

Unless otherwise noted, the sampling interval was set to sample an average of one in every 25,000 cache misses (the actual value was pseudo-randomly varied throughout each run).

The code that implements the miss and eviction sampling is written in C and is linked into the application. As a result, it runs under the simulator, and is included in the virtual cycle count and cache simulation. This allows us to study the overhead and perturbation of our instrumentation software.

In the next sections, we will describe the results we obtained when simulating miss and eviction sampling on a number of applications from the SPEC CPU95 [16] and SPEC CPU2000 [9] benchmark suites. From SPEC CPU95, we used the application su2cor. From SPEC CPU 2000, we used applu, gzip, mgrid, swim, and wupwise. All programs were compiled with either the Compaq C compiler V6.3-028 or the Compaq Fortran compiler V5.4A-1472. The C compiler flags used were –fast -arch host (which includes among other things –O3) and for Fortran we used the flags -arch host -O5 -tune host.

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.

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 simu-

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 1: Cache Misses Sampled With Eviction Information

lator, and counts every cache miss and eviction. This provides us with exact values to compare the results from the instrumentation code against.

3.1.1 Cache Misses

We will first examine the results of sampling cache misses. Although previous works of the authors have addressed gathering cache miss information using sampling [4, 5], here we will examine whether the accuracy of the information obtained 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 1 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 2.2, 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 are the precise values, collected at a low level in the simulator, whereas the sampled values 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).

Stat Bucket	Evicted By	Actual Rank	Actual %	Sampled Rank	Sampled %
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 2: Cache Evictions in Mgrid

3.1.2 Cache Evictions

Table 2 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.

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 2 is for the evictions of U by itself, for which the percentage value estimated by sampling is approximately 0.5 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%.

applu

	c	a	b	d	rsd
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

su2cor

	U	W1-i	W2-i	W2-s	W1-s
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

gzip

	prev	window	spec_init
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

swim

	UNEW	PNEW	VNEW	CU	CV
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

mgrid

	U	R	V
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

wupwise

	U	UD	T	S	P
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 3: Percentage of Evictions of Variables (Columns) Caused by Each Other Variable (Rows)

Table 3 shows 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.

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 4 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.

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 5 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 5: Evictions of T by U in Wupwise

3.2 Perturbation of Results

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

We will now look at how sampling cache evictions affects the cache behavior of an application. Figure 3 shows the percent 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.

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%. Similar to 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.

3.3 Instrumentation Overhead

Figure 4 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.

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%.

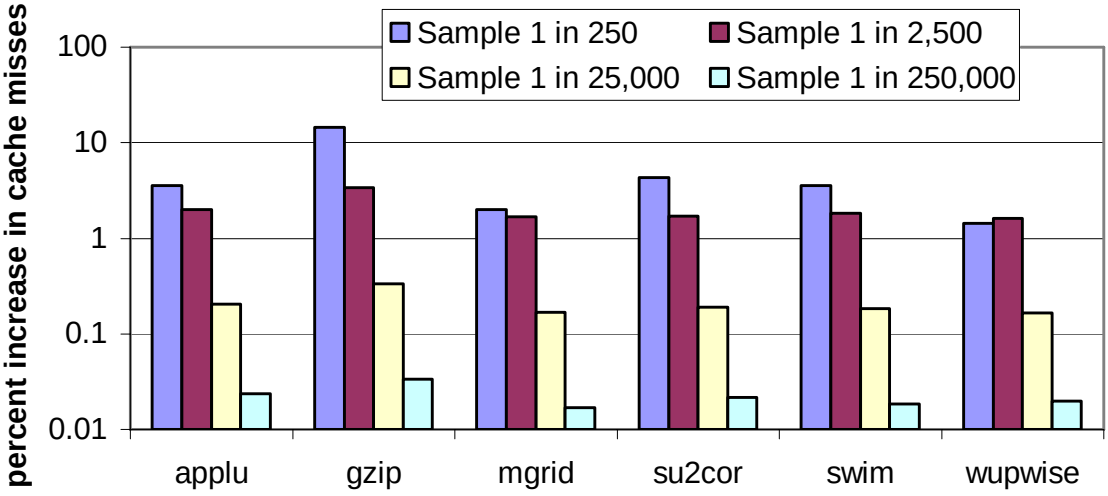

Figure 3: Percent Increase in Cache Misses When Sampling Evictions

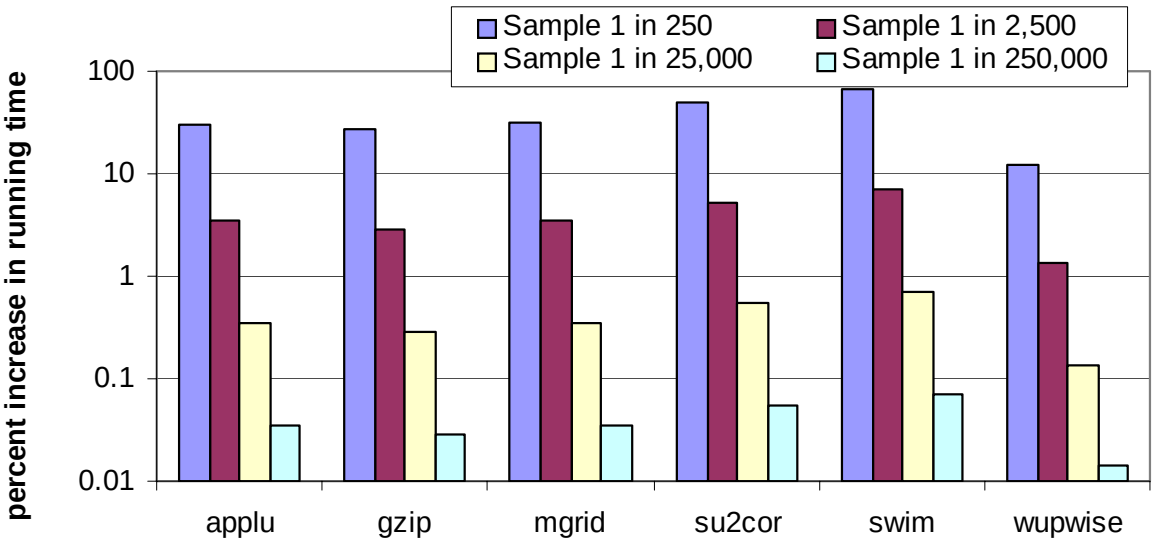

Figure 4: Instrumentation Overhead When Sampling Cache Evictions

4 Tuning Using Eviction Data

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 3, 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 6 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 6: Evictions by Code Region in Mgrid

Three lines together cause almost 36% of all evictions of U by itself, one in the function psinv and the others in the function resid. 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 5: 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 [14, 24] 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 6 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%.

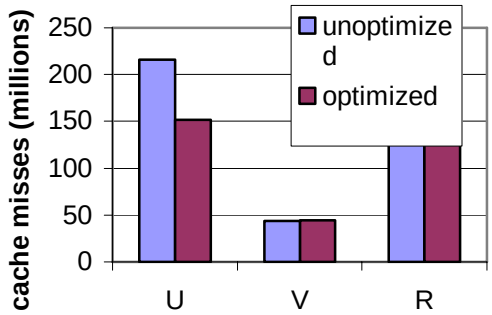

Figure 6: Optimizing Cache Misses in Mgrid.

5 Related Work

Most current 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 [25], Compaq Alpha [7], UltraSPARC [15, 22], and the Intel Itanium [1, 11, 20] families of processors. All of these can provide cache miss information.

Other systems have used flexibility provided by the hardware to add data centric cache instrumentation. ATUM [2] uses the ability to change the microcode in some processors to collect memory reference information. The FlashPoint [18] system uses the fact that the Stanford FLASH multiprocessor [13] implements its coherence protocols in software, allowing instrumentation to be added at this level.

Mtool [8] provides information about the amount of performance lost due to the memory hierarchy, but only relates this information back to program source lines, not data structures. MemSpy [17] provides data-oriented information as well as code-oriented, but uses simulation to collect its data.

StormWatch [6] is another system that allows a user to study memory system interaction. It is used for visualizing memory system protocols under Tempest [19], a library that provides software shared memory and message passing. However, the goal of StormWatch is to study how to adapt a memory system protocol to suit the application, rather than how to change the application to match the memory system.

Bershad et al. [3] 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, which 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.

Another hardware feature that has been proposed as a means of both measuring memory behavior and adapting to it is informing memory operations [10]. An informing memory operation allows an application to detect whether a particular access hits in the cache. 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.

In an earlier paper [5], the authors of this paper compare the effectiveness of sampling with that of using conditional counters with a base and bound for isolating misses to specific data structures. A later paper examines sampling cache misses using features of an existing processor, the Itanium 2 [4]. However, neither of these discusses measuring cache evictions.

Itzkowitz et al. [12] 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. Since the UltraSPARC-III processors used by this tool do not provide information about the instruction and data addresses associated with an event, the values reported by this tool are inferred and may be imprecise.

6 Conclusion

We have shown that information about cache evictions can lead to a better understanding of how application data structures are interacting in the cache. In the mgrid example, the fact that accesses to an array was often causing an eviction of data from the same array allowed us to eliminate many of the evictions and subsequent misses.

We also found that if simple support for measuring information about evictions is provided by the hardware, then we can use sampling to collect accurate eviction information with low overhead. The required hardware support is not substantially different from the information that some processors already provide about cache miss addresses. We believe that the ability to sample cache eviction addresses is a feature that would complement the existing performance monitoring features in current processors, and that it would be practical to implement.

7 Acknowledgements

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

References

1. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*. Intel, Intel Order Number 251110-002, 2003.
2. 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.
3. 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.
4. Buck, B.R. and Hollingsworth, J.K., Data Centric Cache Measurement on the Itanium 2 Processor. In *Proceedings of SC2004*, (Pittsburgh, PA, 2004).
5. Buck, B.R. and Hollingsworth, J.K., Using Hardware Performance Monitors to Isolate Memory Bottlenecks. In *Proceedings of SC2000*, (Dallas, TX, 2000).
6. 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).
7. Compaq Computer Corporation *Alpha Architecture Handbook (Version 4)*, 1998.
8. 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.
9. Henning, J.L. SPEC CPU2000: Measuring CPU Performane in the New Millenium. *Computer, 33* (7). 28-35.
10. 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).
11. Huck, J., Morris, D., Ross, J., Knies, A., Mulder, H. and Zahir, R. Introducing the IA-64 Architecture. *IEEE Micro, 20* (5). 12-23.
12. Itzkowitz, M., Wylie, B.J.N., Aoki, C. and Kosche, N., Memory Profiling using Hardware Counters. In *Proceedings of SC2003*, (Phoenix, AZ, 2003).
13. 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.
14. Lam, M.S., Rothberg, E.E. and Wolf, M.E., The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, (San Jose, California, 1991), 63-74.
15. Lauterbach, G. and Horel, T. UltraSPARC-III: Designing Third Generation 64-Bit Performance. *IEEE Micro, 19* (3). 73-85.
16. Lebeck, A.R. and Wood, D.A. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer, 27* (9). 15-26.
17. 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.
18. 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).
19. 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).
20. Sharangpani, H. and Arora, K. Itanium Processor Microarchitecture. *IEEE Micro, 20* (5). 24-43.
21. 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.
22. Sun Microsystems *UltraSPARC User's Manual*, 1997.
23. 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.
24. 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.

25. 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).