Using Hardware Performance Monitors to Isolate Memory Bottlenecks

Bryan R. Buck Jeffrey K. Hollingsworth
Computer Science Department
University of Maryland
College Park, MD 20742
{buck,hollings}@cs.umd.edu

Abstract

In this paper, we present and evaluate two techniques that use different styles of hardware support to provide data structure specific processor cache information. In one approach, hardware performance counter overflow interrupts are used to sample cache misses. In the other, cache misses within regions of memory are counted to perform an n-way search for the areas in which the most misses are occurring. We present a simulation-based study and comparison of the two techniques. We find that both techniques can provide accurate information, and describe the relative advantages and disadvantages of each.

1 Introduction

As processor speeds have rapidly increased, the gap between these speeds and the access time of main memory has widened. Because of this, it is becoming ever more important for applications to make effective use of memory caches. Information about an application's interaction with the cache is therefore crucial to tuning its performance. To be most useful to a programmer, this information should be presented in terms of program objects (data structures) at the source code level. In the past, this has been difficult to do, due to limited hardware support. For instance, processors that include support for counting cache misses have often not provided a way to determine the addresses being accessed to cause them. An alternative to hardware support is simulation, which can provide the information but is often prohibitively slow.

The situation is now changing as newer processors include more and more support for performance monitoring. Many processors have for some time supported counting cache misses. Some, including the MIPS R10000 [12] and the Compaq Alpha [3], include a way to generate an interrupt after some number of misses have occurred. In addition to such features, the Itanium from Intel [5] provides a means to determine the address associated with a cache miss. It also provides a way to limit cache miss counting to misses associated with a user-determined area of memory. The MIPS R12000 and Alpha 21364 are rumored to have similar functionality.

This paper presents two techniques for using such hardware support to provide source code level feedback to a programmer about data structures with poor cache behavior. The first uses sampling, and the second performs a search through the address space using hardware counters that can selectively count events depending on the address at which a cache miss occurs. We also describe the results of a simulation-based study of the effectiveness of the two techniques.

2 Description of Techniques

The techniques described in this paper utilize two different approaches to hardware support for measuring cache miss information. In order for a tool to relate cache misses to data structures, the hardware must provide knowledge about the addresses that are being accessed when cache misses occur. However, running instrumentation code at every cache miss to check the address would likely cause an unacceptable slowdown in the application being measured. It is desirable to limit the frequency with which instrumentation code must run to read the hardware counters, and the amount of information that must be processed by the tool. The following sections describe two approaches to doing this and how they could be used.

2.1 Cache Miss Address Sampling

One way to process cache miss information less frequently is to sample it. This can be accomplished with the cache miss counters available on some processors. For instance, the MIPS R10000 and R12000, Compaq Alpha, and Intel Itanium can generate an interrupt after a user-defined number of misses. The Itanium also provides a way to determine the address of the last cache miss. On modern processors without specific hardware to report the address of a miss, features like multiple instruction issue and out-of-order execution make it difficult to determine what instruction caused the miss much less the effective address being accessed. In our study, we will assume that the processor provides the address of the last cache miss.

The technique that we will examine is to associate a count with each memory object in an application

0-7803-9802-5/2000/$10.00

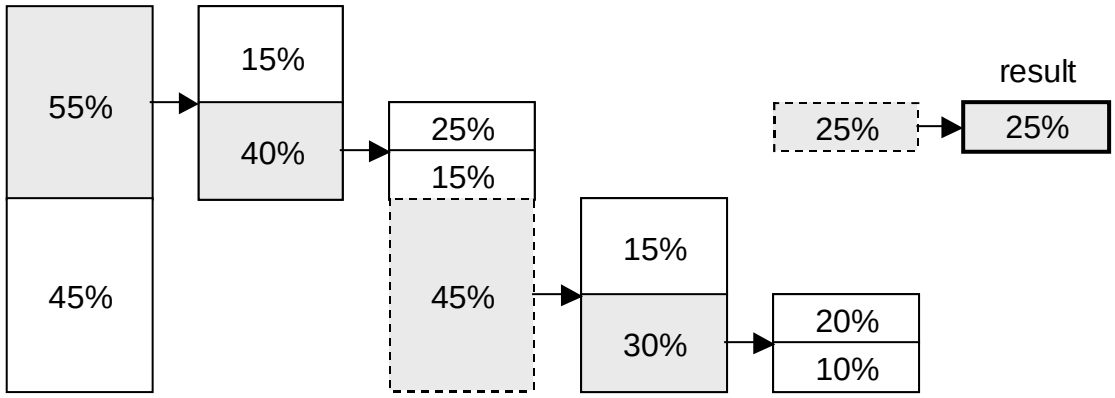

Figure 1: Searching for a Memory Bottleneck

to be measured, meaning each variable and dynamically allocated block of memory. We then set the hardware counters (simulated in this study) to cause an interrupt after some number of misses. When an interrupt occurs, the address of the last cache miss is matched to the program object containing it, and the corresponding count is incremented, after which the process is repeated. After a representative portion of the execution of the application has completed, we can examine the counts and rank the program objects by the number of misses incurred while accessing them. If the number of misses we sampled for each object is proportional to the total number, this will provide a programmer with an accurate idea of which program objects are causing the worst cache behavior.

An advantage of this technique is that it is simple, if there is a method available to map addresses to program objects. For global and static variables, this can be done easily using data from symbol tables and debug information. The location of dynamically allocated memory objects can be tracked by instrumenting memory allocation library functions. Support for variable on the stack is discussed in section 5.

2.2 N-Way Search

A second way to provide data structure specific cache statistics is to have the hardware itself separate the misses into memory regions. Some processors, including the Intel Itanium, allow for counting cache misses conditionally based on the addresses being accessed. The second technique we will describe uses this feature to perform a search through the address space of an application for bottlenecks, the regions of memory in which the most cache misses are occurring.

We will assume that a number of cache miss counters are available, each with its own associated set of base and bounds registers that specify an area of memory in which to count cache misses. Although current processors that provide conditional counting of cache misses typically allow only one region to be specified at a time, multiple counters with separate base/bounds could be simulated by timesharing the single conditional counter between regions of interest.

Our search technique uses an *n*-way search to narrow down on the areas of memory that are causing the most cache misses. Figure 1 shows an example of a two-way search. At the beginning of the search, the address space is divided into *n* areas, each assigned to a miss counter. An additional cache miss counter is used to measure the total number of cache misses in the entire address space. The application is then allowed to run to collect information.

At the expiration of a timer, the instrumentation examines the values in the counters and uses them to compute the percentage of the total cache misses that were caused by each region. It then places the information about the measured regions into a priority queue, which ranks them by this percentage (the queue starts out empty at the beginning of the search).

Next, the instrumentation code takes the *n*/2 regions that caused the largest percentage of total cache misses off the priority queue. It splits each of these *n*/2 regions in half, to form *n* new regions for the next iteration of the search, and the process is repeated. When the top regions in the priority queue reach the size of individual memory objects (and therefore cannot usefully be split further), the search is complete, and the objects are listed, along with an estimate of the percentage of all cache misses caused by each. This last information is collected by taking additional samples with each cache miss counter set to cover exactly the area of one of the found objects. The current version of the search code identifies only global and static variables and dynamically allocated blocks of memory, although the entire address space is included in the search.

One problem for the naïve n-way search algorithm is memory objects that lie only partially within a region. In such cases, the rate of cache misses in the whole object may not be adequately represented by the

data used in the search. In particular, an array causing many cache misses that spans a region boundary may not cause enough cache misses in any single region to attract the search to it. The solution to this problem is to adjust the extents of the regions each time they are split so that objects do not span region boundaries. This is done efficiently in our implementation using information about object extents kept in a sorted array for variables and a red-black tree for heap blocks (since this data will change as allocations and deallocations take place).

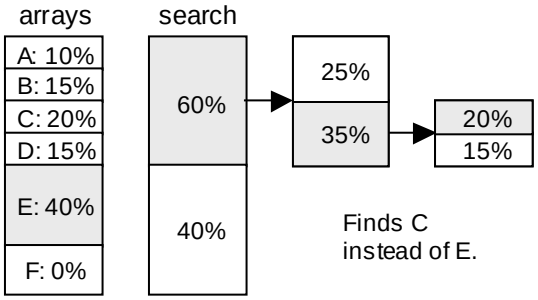

Figure 2: Search without Priority Queue

Another potential problem for the search algorithm, which is solved by the priority queue, is that the region causing the most cache misses does not necessarily contain the single object causing the most misses, unless the region contains only that object. This is illustrated in Figure 2. This figure shows the layout of a group of arrays on the left, along with the percentage of cache misses that is due to each. On the right, the figure shows the areas examined by a search in which the region with the most cache misses in each iteration is chosen for refinement. In the first iteration, the region causing 60% of the cache misses is selected, even though none of the arrays within it cause as many cache misses as array E. This removes the region containing the array E from consideration, causing the search to eventually terminate at array C. The priority queue fixes this problem by providing a mechanism that causes the search to "back up" when necessary. In the second search iteration above, the bottom region from the first iteration would be on the priority queue when the region to refine is chosen. Since it accounted for a greater percentage of cache misses than any other region examined up to that point, it would be chosen for refinement, allowing the search to correctly identify region E as the one causing the most cache misses.

Many programs have phases in which access patterns may be very different from those observed at other times. This can lead to incorrect results, as the search algorithm may remove from consideration regions that generally cause many cache misses but did not do so in the most recent sample interval. Ordinarily, regions that incur no cache misses during a sample period are discarded immediately, but an exception is made in order to account for phases. Regions that previously were ranked in the top $n/2$ but which show no misses during the current interval are not discarded unless the situation persists for several interactions. In addition, each time a region with zero misses is kept, the duration of future sample intervals is increased. When phases are short, this can have the effect of automatically adapting the length of the sample interval to cover multiple phases, effectively averaging their cache miss rates.

Another important consideration is when to declare the results of the search final. Since the goal of the search is to identify the memory objects causing the most cache misses, an obvious point is when some number of the top regions have been refined until they contain only individual objects or cache lines. We chose to terminate the search when the top n-1 regions reach this point. This value was chosen so that the search algorithm could continue to include regions containing only one object in the search until all final objects have been found. When a region contains only one object, it cannot be further split, but it is kept in the priority queue and may be selected for measurement in each iteration. When such a region is selected, the search algorithm measures the cache misses within it again and averages the results with the results from previous iterations. This allows the objects to be ranked with increasing accuracy. When n-1 of the top regions all contain only a single object, there would not be enough counters available to measure them along with two new regions formed by splitting the nth region, so the search is terminated. The search is also terminated if the percentage of cache misses within unsearched regions drops below a selectable threshold, in order to handle applications in which there are fewer than n-1 significant regions. When the search terminates, the instrumentation code prints the names of the objects within the regions from the last search iteration that were found to be causing the most cache misses. Only regions containing single objects are included in these results (others have not been fully examined).

It would be possible to return more objects if regions containing only a single object were removed from the search after being measured, although this may lead to less accuracy since the results for the found regions would be from single iterations only, instead of averaged over many as is the case in the current algorithm.

3 Experiments

To investigate the effectiveness of the two approaches described in section 2, we performed a simu-

lation-based study. The simulator runs real applications, with load and store instructions instrumented to track memory references and calculate their effects on a simulated cache. The instrumentation is performed using the ATOM [10] binary rewriting tool. Basic blocks are also instrumented to keep 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 simulator does not model details such as pipelining and multiple instruction issue. The cache simulated is a single-level set associative cache (2MB in size for these experiments).

The simulator provides a software-selectable number of cache miss counters, each with a base and bound register designating an area of memory for which to collect information, as well as interrupts that can be requested to occur after some number of cycles or cache misses. These are used by additional instrumentation code that implements the techniques we have discussed. This code runs inside the simulation, so it can be timed using the virtual cycle counter, and it can affect the cache, making it possible to study perturbation of the results.

In the following sections, we will describe the results of running the two measurement techniques under the simulator on a number of applications from the SPEC95 benchmark suite. The applications tested were tomcatv, su2cor, applu, swim, mgrid, compress,

Application	Variable Memory Block	Actual		Sample		Search	
		Rank	%	Rank	%	Rank	%
tomcatv	RY	1	22.5	2	17.6	1	22.5
	RX	2	22.5	1	37.1	2	22.5
	AA	3	15.0	5	10.1	3	15.1
	DD	4	10.0	3	15.0	5	10.1
	X	5	10.0	6	9.8	7	9.9
	Y	6	10.0	7	0.2	6	9.9
	D	7	10.0	4	10.2	4	10.1
swim	CU	1	7.7	3	8.2	3	7.7
	H	2	7.7	4	8.1		
	P	3	7.7	1	8.4		
	V	4	7.7	2	8.3	1	7.7
	U	5	7.7	5	7.8	2	7.7
	CV	6	7.7	13	6.7	4	7.7
	Z	7	7.7	12	6.8	5	7.7
su2cor	U	1	57.1	1	57.5	1	56.8
	R	2	6.9	3	6.8	2	7.2
	S	3	6.6	2	7.2	3	6.8
	W2 - intact	4	3.9	4	4.1	4	3.8
	W2 - sweep	5	3.7	5	3.5		
	B	6	2.3	7	2.0	5	2.3
mgrid	U	1	40.8	1	40.7	1	40.8
	R	2	40.4	2	39.8	2	40.6
	V	3	18.8	3	19.5	3	18.6
applu	a	1	22.9	2	23.0	1	22.7
	b	2	22.9	3	19.9	2	22.6
	c	3	22.6	1	25.8	3	22.4
	d	4	17.4	4	16.7	4	17.4
	rsd	5	6.9	5	7.7	5	7.2
compress	orig_text_buffer	1	63.0	1	67.4	1	63.6
	comp_text_buffer	2	35.6	2	30.2	2	35.9
	htab	3	1.3	3	2.3		
	codetab	4	0.2				
ijpeg	0x141020000	1	84.7	1	95.8	1	85.2
	jpeg_compressed_data	2	12.5	2	4.2	2	12.7
	0x14101e000	3	0.5			3	0.0
	std_chrominance_quant_tbl	4	0.0				

Table 1: Results for Sampling and Search

and ijpeg. For the search algorithm, we assumed ten hardware cache miss counters were available, and set the value of *n* for the *n*-way search to ten.

3.1 Quality of Results

We will first examine the quality of the results returned by each approach. Table 1 shows the five objects in each application that caused the most cache misses, as well as the up to five objects found to be causing the most cache misses by sampling or by the *n*-way search algorithm, and excluding objects causing less than 0.01% of the total misses. Object names that consist of a hexadecimal number represent dynamically allocated blocks of memory. The search results were obtained using a ten-way search, which generally identifies up to nine objects, as described in section 2.2.

The "rank" columns show how each technique ranked the given object. The percentages are the percentage of all cache misses in the application that occurred due to references to the object. For the "actual" column, this value was measured by lower levels of the simulator, separate from the sampling and search code. For the sample and search columns, these values are as estimated by each technique. The results were gathered over the same portion of each application's execution for all methods. The sampled values represent the results of sampling one in 50,000 cache misses.

Generally, both techniques returned results that were indicative of the actual number of cache misses occurring due to accessing each object. For almost all applications, both algorithms ranked the objects they found 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 (generally less than 2%).

For sampling, the largest error occurs in tomcatv, where the variable RX, causing the second largest number of cache misses at approximately 22.5%, is estimated to be causing 37.1%. RY, which causes an almost identical number of cache misses, is estimated to be causing only 17.6%. We observed similar results in runs for which we increased the sampling frequency to 1 in 100 cache misses. However, when we changed the sampling frequency slightly by basing it on a nearby prime number, to sample 1 in 50,111 cache misses, sampling achieved significantly more accurate results. In such a run, the largest difference between the percentage of cache misses for a variable estimated by sampling and the actual percentage was approximately 0.3%, versus a difference of approximately 14.6% for the variable RX in the original set of runs. This demonstrates the necessity of ensuring that the sampling interval does not coincide with an application's memory access patterns. This could be achieved by basing the sampling interval on prime numbers as described above, or by varying it pseudo-randomly.

3.2 Perturbation of Results

Figure 3 shows the percentage increase in cache misses for each application when run with instrumentation code. The "search" bars show results for the search, and the "sample(number)" bars show the increase for sampling, with the number indicating the number of cache misses between each sample. Note that the scale of the y-axis, showing the percentage, is logarithmic.

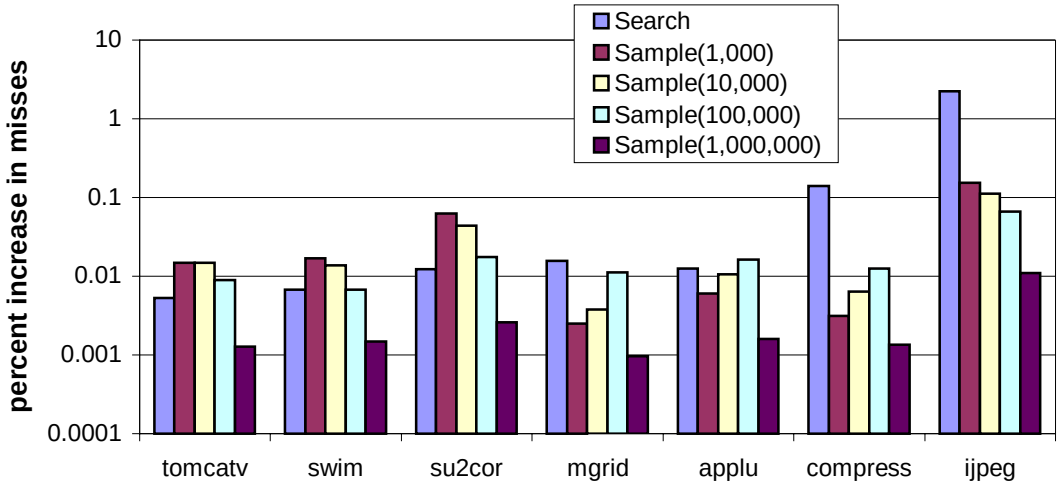

Figure 3: Increase in Cache Misses Due to Instrumentation

These results were derived by running the code for each type of instrumentation under the cache simulator, along with the application code. They were compared to runs with no instrumentation to calculate the percentage increase in cache misses. 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 simulation, so the effects of kernel code for context switches and the delivery of signals was not modeled.

In all cases except for the ijpeg application, the effects of instrumentation code on the cache were almost negligible, with the highest increase in cache misses aside from ijpeg being seen when running compress with a ten-way search, which caused an approximately 0.14% percent increase in misses. For ijpeg, the increase with a ten-way search was still only 2.4%. The larger increase relative to the other applications is due to the fact that ijpeg normally has a much lower cache miss rate, only 144 misses per million cycles (the next lowest is compress, with 361 misses per million cycles, and after that is mgrid with 6,827).

Interestingly, for sampling with mgrid, applu, and compress, the number of additional cache misses goes up as the frequency of sampling goes down, until we reach a sampling rate of one sample per million misses. This is likely due to the data used by instrumentation code being evicted from the cache more often as the sampling frequency is reduced. When sampling frequency becomes low enough, this effect is no longer important. Factors that would affect whether or not this phenomenon will occur with a given application include the cache miss rate, the size of the program object map used by the instrumentation, and the frequency with which misses land in the same set of objects.

3.3 Instrumentation Cost

Figure 4 shows the percent slowdown of each application due to instrumentation code. The "search" bars show results for the search algorithm, and the remaining bars show results for sampling with the given frequency. Again, the scale of the y-axis is logarithmic.

The values shown include the time spent executing search or sampling code (in virtual cycles), plus a cost for receiving each interrupt signal. For this value, we used results obtained experimentally on an SGI Octane workstation with 175Mhz processors. We used the performance 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.

The figure shows that both techniques are extremely efficient, unless sampling is performed too often; sampling one miss in every thousand leads to a slowdown of as much as 16% (in tomcatv). Although the *n*-way search is generally more efficient except at low sampling frequencies, at a frequency of one in 10,000 misses, which was shown in section 3.1 to be sufficient, the worst slowdown for sampling is approximately 1.6% (again for tomcatv).

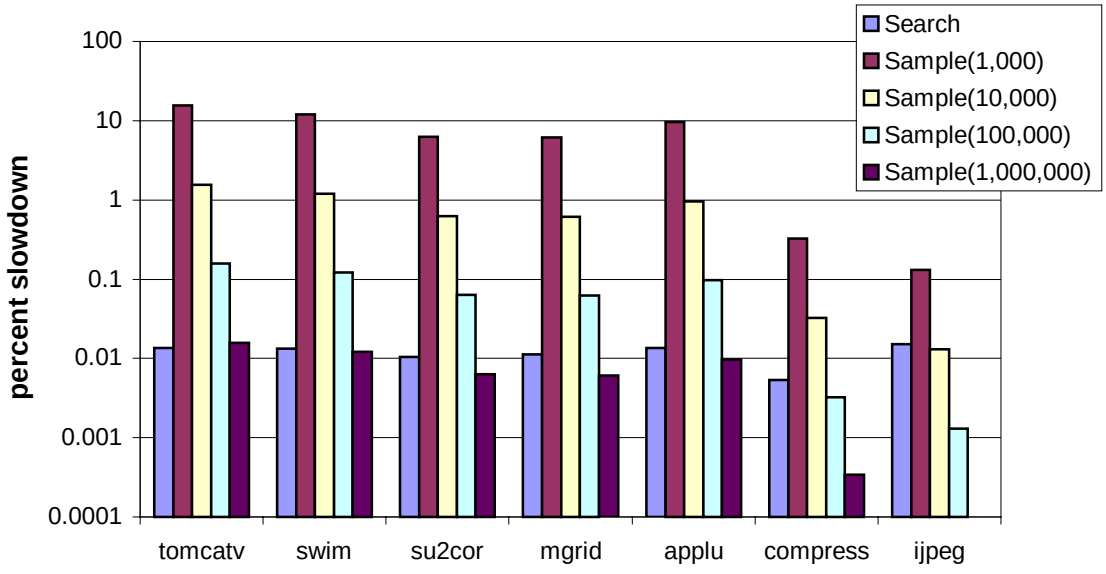

Figure 4: Instrumentation Cost

Application	Variable Memory Block	Actual		2-Way Search		10-Way Search	
		Rank	%	Rank	%	Rank	%
tomcatv	RY	1	22.5	2	22.4	1	22.5
	RX	2	22.5	1	22.4	2	22.5
	AA	3	15.0			3	15.1
	DD	4	10.0			5	10.1
	X	5	10.0			7	9.9
	Y	6	10.0			6	9.9
	D	7	10.0			4	10.1
swim	CU	1	7.7	1	7.8	3	7.7
	H	2	7.7				
	P	3	7.7				
	V	4	7.7			1	7.7
	U	5	7.7			2	7.7
	CV	6	7.7			4	7.7
	Z	7	7.7			5	7.7
	VOLD	8	7.7	2	7.6	6	7.7
su2cor	U	1	57.1			1	56.8
	R	2	6.9	1	0.0	2	7.2
	S	3	6.6			3	6.8
	W2 - intact	4	3.9			4	3.8
	W2 - sweep	5	3.7				
	B	6	2.3			5	2.3
mgrid	U	1	40.8	1	40.6	1	40.8
	R	2	40.4	2	40.3	2	40.6
	V	3	18.8			3	18.6
applu	a	1	22.9			1	22.7
	b	2	22.9	1	22.7	2	22.6
	c	3	22.6	2	22.4	3	22.4
	d	4	17.4			4	17.4
	rsd	5	6.9			5	7.2
compress	orig_text_buffer	1	63.0	1	63.6	1	63.6
	comp_text_buffer	2	35.6	2	36.0	2	35.9
	htab	3	1.3				
	codetab	4	0.2				
ijpeg	0x141020000	1	84.7	1	84.9	1	85.2
	jpeg_compressed_data	2	12.5	2	12.6	2	12.7
	0x14101e000	3	0.5			3	0.0
	std_chrominance_quant_tbl	4	0.0				

Table 2: Results of Two-Way Versus Ten-Way Search

Note that the search code is much more expensive per interrupt; 26,000 to 64,000 cycles across the applications including the time for the OS to deliver the interrupt signal, versus approximately 9,000 cycles per interrupt for sampling. The search algorithm achieves its efficiency by requiring very few interrupts, from 1.6 to 4.1 per billion cycles with the tested applications, versus 13 to 1,727 per billion cycles for sampling 1 in 10,000 misses. This results in a lower overall overhead for the search code versus sampling even one in every 100,000 misses, except for the ijpeg and compress applications (which have lower cache miss rates than the others).

3.4 Number of Regions

The results for the search algorithm that have been presented up to this point have been generated using a ten-way search. This section compares these with the results of a two-way search, shown in Table 2. The question of choosing the value for *n* in the *n*-way search is an important issue, since an *n*-way search requires *n* cache miss counters with associated base and bounds (plus one global counter, for use in calculating the percentage of total cache misses caused by each region). An alternative is to timeshare fewer

registers to measure *n* regions, but this may lead to increased inaccuracy.

Since a processor is likely to provide only limited support for performance measurement, reducing the number of required registers is desirable. However, an important consideration is that as discussed in section 2.2, in general an *n*-way search will return *n*-1 objects as results (it may return one more result if the *n*th ranked region contains only one object). Therefore, a two-way search could be expected to identify only the top one or two objects causing the most cache misses. An early version of the search algorithm, without the priority queue for previously examined regions, failed to find the top object because the coarser granularity made the two-way search more likely to discard important regions. A region may cause fewer cache misses than another and yet still contain an object causing more misses than any individual object in the other region. Table 2 shows that with the current version of the algorithm, using a priority queue (which allows backtracking to previously measured regions), even a two-way search is able to accurately identify the top one or two regions causing the most cache misses for almost all of the applications we tested. The only exception is su2cor. In this application, changing memory access patterns caused it assign a low rank a region that would later cause the most cache misses, so that it was never selected for refinement. The fact that it estimated the percentage of cache misses in the array that was found to be zero is also due to changing access patterns. The estimation is based on measurements taken after the search has concluded; in this case the access patterns had changed since the region was found. Possible solutions to the problem of changing access patterns will be discussed in the next section.

3.5 Changing Access Patterns

One aspect of program behavior that can adversely affect the reliability of the simple search algorithm is phases. Distinct phases of a program's execution may display very different access patterns. It may be the case that no single array causes the most cache misses across all phases; instead, there may be a separate such array for each phase. This would not be expected to affect sampling unless the phases are synchronized with the sample frequency, or short enough to most often fall in between samples.

Figure 5 shows cache misses over time for an application that exhibits phases, applu. The names in the key identify arrays in the application, with "A, B, C" indicating the graph for three arrays, A, B, and C, which have almost exactly the same access pattern.

We can see in this graph that the number of cache misses in the arrays that cause the most misses overall, A, B, and C, periodically dip below the number of misses in other arrays; in fact, A, B, and C periodically cause no cache misses during a sample interval.

The simple heuristic described in section 2.2 for handling changing access patterns sufficiently handles this case. When no cache misses occur in a region that was previously among the areas with the most cache misses, the region is not immediately discarded from the search, and the time between search iterations is extended. In the case of applu, this automatically adjusts the length of an iteration to cover multiple phases. As can be seen in Table 1, this allows the search to find the five arrays causing the most cache misses, demonstrating the effectiveness of this simple mechanism in handling short phases. We plan to extend this technique to cover cases in which the number

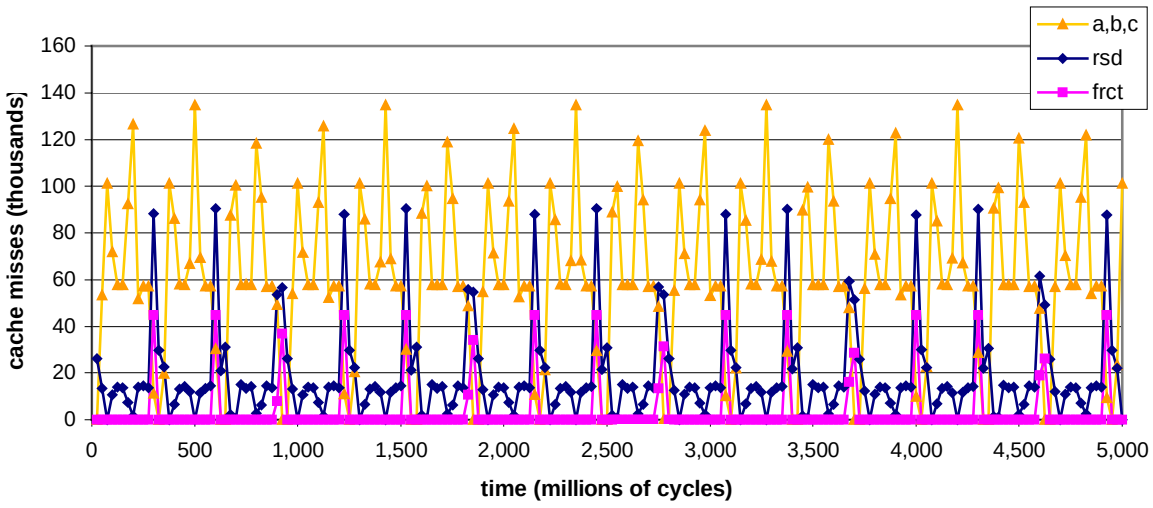

Figure 5: Cache Misses over Time for Applu

of cache misses in a region drops off, but does not reach or approach zero. The technique does not apply to longer phases, which would require more sophisticated handling.

4 Related Work

Most modern processors include some kind of performance monitoring counters on-chip. These typically provide low-level information about resource utilization such as cache hit and miss information, stalls, and integer and floating point instructions executed. Examples include the MIPS R10000 [12], the Compaq Alpha line [3], the UltraSPARC [11], and the Intel Itanium [5]. All of these can provide cache miss information.

Other systems have used flexibility provided by the hardware to add data centric cache instrumentation. ATUM [1] uses the ability to change the microcode in some processors to collect memory reference information. The FlashPoint [8] system uses the fact that the Stanford FLASH multiprocessor [6] implements its coherence protocols in software, allowing instrumentation to be added at this level.

Mtool [4] 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. A system with more similarity to the techniques in this paper is MemSpy [7]. MemSpy provides data-oriented information as well as code-oriented, but uses simulation to collect its data.

StormWatch [2] is another system that allows a user to study memory system interaction. It is used for visualizing memory system protocols under Tempest [9], 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. Because of this, the information provided is also different. This information includes what protocol events are taking place, what code is causing them, and how they are related.

5 Future Work

We plan to extend the techniques we have discussed to gather information about variables on the stack, and to better handle dynamically allocated memory. For sampling, this could be done by aggregating data for all instances of the same local variable, and for related blocks of dynamically allocated memory (for instance, the nodes of a tree). For the search technique, we would need to move related blocks of memory into contiguous regions in order to allow them to be considered as a unit. This could be done by replacing the standard memory allocation functions with specialized ones that arrange memory for measurement. A possible disadvantage to this approach is that the changed placement of the blocks may affect cache behavior. It would be difficult for the search technique to handle individual objects on the stack, so such support would be limited.

Currently, the algorithms depend on certain arbitrarily chosen parameters, such as sampling frequency or the length of a search iteration. We plan to investigate how these values could be adjusted automatically by the algorithms in order to achieve greater accuracy and efficiency.

Much useful information could be gained by running the two algorithms on actual hardware, such as the Intel Itanium, which supports the required features. It would also be useful to examine the results achieved by the algorithms on a variety of other applications, especially applications that make extensive use of dynamically allocated memory. We plan to expand the tested applications to include at least a set taken from the SPEC2000 benchmark suite.

6 Conclusions

In conclusion, the sampling and *n*-way search techniques were able to adequately determine the set of memory objects causing the greatest number of cache misses in our experiments. This kind of feedback is becoming increasingly important to tuning an application's performance, due to the growing disparity between the access time for data in the cache versus data in main memory.

Our simulation based study reveals several differences between the two approaches. The *n*-way search code is more complex, leading to potentially larger overhead and perturbation of results. However, because it runs much less frequently than the sampling code, this was rarely the case in the applications we tested, and in general the *n*-way search had a lower overhead. On the other hand, the sampling technique is able to rank all objects in terms of observed cache misses, while the search is limited in how many bottleneck objects it can identify by the number of region cache miss counters available. This may be correctable by returning to search previously discarded areas after the ones causing the most cache misses have been examined fully.

Most modern processors are now being designed with hardware counters, but often they do not include such features as the determination of cache miss addresses. The latest generation of processors is starting to provide these features; the Intel Itanium is one example. The results of this study show that the techniques we have presented can use them to provide source-code level feedback to a programmer about what data structures to concentrate on when optimizing for the cache.

7 References

1. A. Agrawal, R. L. Sites, and M. Horowitz, "ATUM: A New Technique for Capturing Address Traces Using Microcode," *13th Annual International Symposium on Computer Architecture*. June 1986, pp. 119-127.
2. T. M. Chilimbi, T. Ball, S. G. Eick, and J. R. Larus, "StormWatch: A Tool for Visualizing Memory System Protocols," *Supercomputing '95*. December 1995, San Diego, CA.
3. Compaq Computer Corporation, *Alpha Architecture Handbook (Version 4)*. 1998.
4. A. J. Goldberg and J. L. Hennessy, "MTOOL: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications," *IEEE Transactions on Parallel and Distributed Systems*, 1993, pp. 28-40.
5. Intel, *Intel IA-64 Architecture Software Developer's Manual*. 1.0 ed. Vol. 4. 2000.
6. J. Kuskin, *et al.*, "The Stanford FLASH Multiprocessor," *21st International Symposium on Computer Architecture*. April 1994, Chicago, IL, pp. 302-313.
7. M. Martonosi, A. Gupta, and T. Anderson, "MemSpy: Analyzing Memory System Bottlenecks in Programs," *1992 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. June 1-5, 1992, Newport, Rhode Island, pp. 1-12.
8. M. Martonosi, D. Ofelt, and M. Heinrich, "Integrating Performance Monitoring and Communication in Parallel Computers," *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. May 1996, Philadelphia, PA.
9. S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Typhoon and Tempest: User-Level Shared Memory," *ACM/IEEE International Symposium on Computer Architecture*. April 1994.
10. A. Srivastava and A. Eustace, "ATOM: A system for Building Customized Program Analysis Tools," *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. May 1994, Orlando, FL, pp. 196-205.
11. Sun Microsystems, *UltraSPARC User's Manual*. 1997.
12. M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, "Performance Analysis Using the MIPS R10000 Performance Counters," *Proceedings Supercomputing '96*. November 1996, Pittsburgh, PA.