Software Analysis Techniques to Approximate Data Centric Direct Measurements

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

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

ABSTRACT

Data centric analysis using direct measurements has been established as a successful performance analysis technique. The information gathered with this technique can be used to address data locality problems and other issues. Existing approaches rely on special hardware support which is needed to negate a 'skid' factor. Our approach is viable on hardware where the skid factor is an issue. Prior methods also rely on maintaining runtime information about memory allocation addresses for variables, which may lead to program perturbation. Our approach uses software analysis to eliminate the need for maintaining allocation and free records. We show that by using heuristics our technique can attribute data centric values to program variables and maintain the approximate rank-order found by using traditional techniques.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement Techniques

1. INTRODUCTION

Data centric measurements are a valuable component of performance tuning. Different approaches can be utilized to collect and present data to the user. One approach involves measuring data centric hardware counters such as cache misses and presenting them to the user in a code centric way (line, basic block, function). While this gives high level information about where some of these cache misses are taking place, it does not address interactions of data structures in the cache or TLB that can only be obtained by mapping the misses directly to the variables that triggered them.

An alternate approach looks at the same data centric hardware counters but presents the information to the user in a data centric format. Mainly, the information is presented in terms of the program variables and data structures whose memory access triggered the miss. The latter approach is an established means of gathering data to improve program performance through performance analysis. [3]. These data centric methods usually rely on extensive source and/or binary instrumentation. This instrumentation can increase program perturbation.

The existing approaches also typically rely on hardware support to negate a skid factor. The skid factor is an artifact of instruction sampling, where a triggered event (such as a cache miss) is attributed to an incorrect instruction. One cause of skid can be due to out-of-order execution, however, the skid factor is still an issue for processors with in-order execution [6]. For time based metrics, skid is not as significant a factor as the instruction the event was attributed to was valid in the instruction pool. However, for data centric events the skid factor can mean assigning the event to an instruction that is accessing a different memory region or even assigning the event to an instruction with no memory accesses. Certain architectures have hardware in place to negate the skid factor and will return the precise IP (instruction pointer) for the event and effective address of the miss when running in that mode. However, in cases where an architecture does not have this hardware in place, a user is completely unable to use the existing approaches. As HPC systems move to less complex CPUs (i.e. stream processors & GPUs), the type of hardware support required for data centric measurement may become less available.

Our approach is concerned with two main ideas. First, we want to have a generic approach that works on any architecture, regardless of hardware support for negating skid. Our only hardware requirement is that some data centric hardware counter exists on the system for measuring cache or TLB misses. Second, we want to minimize program perturbation and program instrumentation as much as possible. We do this by using sampling instead of direct measurement. Furthermore, our instrumentation is at the binary level and is limited to inserting two calls at the beginning and ending to main. These calls simply assign which hardware counter and threshold will be used to trigger the samples.

The remainder of this paper describes our approach in more detail. Section 2 covers past data centric approaches. We begin with related work because much of our motivation and design choices are dictated by prior approaches. We follow in Section 3 with our approach to the problem. Section 4 discusses our experimental results. We conclude with Section 5.

2. RELATED WORK

A very common data centric approach is to generate samples using memory based hardware counters and assign counts to the data structures responsible for those misses by using the effective addresses. This method was introduced by Buck [4]. His approach triggers samples based on cache misses and uses the effective address of the data being accessed at the sample point to increment a counter for the responsible variable or dynamically allocated block of memory. This approach has the requirement that the architecture have the proper skid negating hardware available. This approach also has the minor disadvantage of introducing some program perturbation. In order to map the effective addresses, you need to keep a record of all of the allocations and frees in the program for dynamically allocated memory. For allocation routines, this involves instrumentation and performing a stack walk to gather context sensitive information at runtime.

Itanium is one architecture with proper hardware support [8] for this effective address matching technique and is the system Buck's approach was implemented on with his tool Cache Scope. [3] On Itanium, the proper hardware counters are available and the effective data address of a specific cache miss can easily be retrieved.

HPCToolkit [13] has recently added data centric profiling to their tool [11]. Their approach is very similar to the approach used by Cache Scope. Like Cache Scope, they record the allocations and frees in the program. They also rely on hardware support to negate skid and to provide a precise effective address of the memory that triggered the event. Their tool uses the PEBS (Precise Event Based Sampling) feature [9] on select Intel chips and the IBS (Instruction Based Sampling) feature [7] on certain AMD chips.

Our approach closely resembles that of Cache Scope and the data centric capability of HPCToolkit. As noted earlier, our work differs in that we eliminate the need for monitoring all allocations and frees within the program. It will also be usable on architectures without specific hardware that negates skid.

Other data centric tools include StatCache and Memphis. StatCache [2] is a probabilistic model of the cache. It does this by wrapping all loads and stores in the program, which can create a large overhead. The information gathered at runtime is used to simulate the memory hierarchy and to apply the post-mortem model to predict cache performance. This information is used to improve data locality. Memphis [12] is a data centric toolset that is limited to the AMD architecture due to its reliance on IBS. It focuses primarily on finding NUMA-related problems.

3. OUR APPROACH

Our approach first generates information about the program through static analysis. At runtime, we minimize instrumentation and program perturbation to obtain only the instruction pointers for the event-based samples. We also perform a stack walk at each sample point to gather context sensitive information about the sample. Then, by applying heuristics to the information obtained at runtime and the static analysis we can approximate the direct data methods. Since at runtime we are only interested in IPs, and not the effective addresses of the misses, we do not need any specific hardware support to perform our approach.

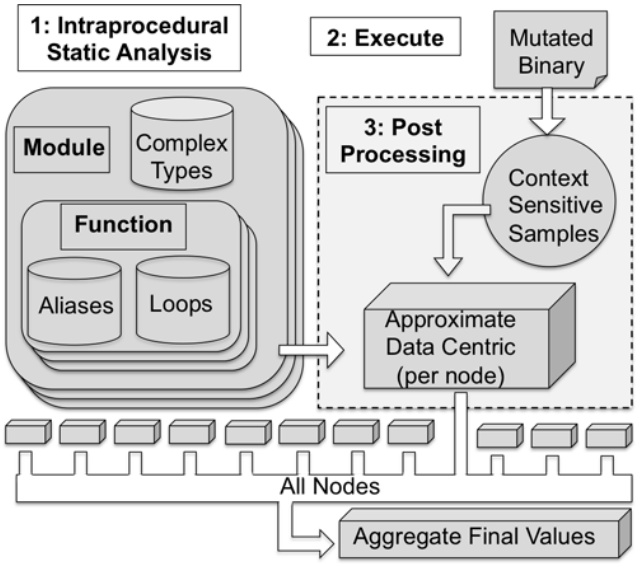

Figure 1: Computing the Approximate Data Centric Values

In this paper, we will use the term "direct data centric" to refer to the approach utilized by Cache Scope and HPCToolkit for their data centric profiling. This is because the performance counter relating to each variable is only incremented during a direct access of that variable. This is verified by the effective address information provided by the hardware. Our approach will be designated as "approximate data centric" since we are using software techniques to approximate which variable triggered the cache miss.

Figure 1 shows the process for determining approximate data centric values. The steps involved are designed to push as much processing as possible to static analysis (pre-run and postmortem) to minimize program perturbation at runtime. They are described in detail in this section. Step 1 can be run on a single node. Step 2 is the running of the program, either in serial or parallel. Step 3 is an embarassingly parallel post processing step, but could be run on a single node.

3.1 Intraprocedural Static Analysis

The majority of the analysis occurs at this step before the program is run. Using the LLVM intermediate format [10], we can analyze different properties at the program, function, and variable level. For the variables, we record the line number information for each read and write performed to that variable. At the function level, we can record the line numbers within every loop within the function. Loop information is utilized in tuning the results and is discussed later in this section. We also use the LLVM information to perform intraprocedural alias analysis. The alias analysis is a very important factor in generating approximate data centric values. Consider the following snippet,

```
1  int * x = (int *) malloc (sizeof(int) * N);
2  int z = x[N-1];
3  int * y = x;
4  z = y[N-2];
```

In a direct data approach, a miss on line 2 or 4 would be attributed to variable x, allocated on line 1, because effective addresses of the read would resolve to the memory range allocated to x. In an approximate approach without alias analysis, a miss on line 2 would be attributed to x and a miss on line 4 would be attributed to y. With alias analysis, x and y would be identified as aliases to one another, and the results would line up with the direct method. Interprocedural alias analysis is also performed after we gather the runtime information and know the full call trace for the samples.

3.2 Execution

The execution step involves running an instrumented program. Instrumentation is accomplished by modifying a binary program to add code at the beginning of program execution to trigger periodic sampling and to record results at the end of the program. The resulting binary is also linked with a shared library containing the handler routine that performs the stack walk to get the full call path for each sample. Since the instrumentation is done on the binary and the new executable is rewritten, no source changes need to be made and the program need not be re-compiled.

The modified binary is run exactly the same way as the original. By pushing most of the analysis to pre-run, and by using sampling, we can minimize program perturbation relative to that of a traditional context sensitive profiling approach. A file is output for each node. This file contains raw addresses of the sampled instruction at each level of the call trace for each sample.

3.3 Post Processing (per Node)

The next step is to take the raw context sensitive samples and the stored intraprocedural analysis to determine the approximate data centric values for each of the variables per node. We start by generating a raw approximate assignment to the variables. We then apply further passes to refine those assignments.

3.3.1 Raw Approximate Assignment

After resolving the addresses to functions and line numbers, the raw values attributed to each variable are calculated by taking the number of misses per source line and dividing them equally among the reads for a line. Further passes may modify these values, but the starting point for our approximation uses this method. Only unique reads are counted towards the total. Given the snippet in Figure 2, assume exactly one cache miss occurs at each line in the body of the loop making three total misses. Our approach would initially attribute the following miss numbers, shown in the table below, to the variables within the above code snippet.

```
1  for (a = 0; a < b; a++) {
2        i = x[a] + y[b]
3        j = y[a] + y[b]
4        k = x[b] + x[b]
5  }
```

Figure 2: Sample Code Snippet

	Approx. Misses Raw			
Line	x[]	y[]	a	b
2	0.25	0.25	0.25	0.25
3	0	0.5	0.25	0.25
4	0.5	0	0	0.5

Line 2 has the misses appropriated evenly across the two arrays and two integers that index the arrays. Line 3 has half of the misses appropriated to array y, because y has two reads indexed by two separate integers. In the case that y was indexed twice by the same integer, y would only have one read counted. For the accesses on line 3, it is possible that a equals b, meaning the same memory location would be accessed twice. However, we assume each access to be different unless a constant is used to access the array. Finally, for line 4 the allocation is split between x and b. Four reads take place on this line, but only two unique memory addresses are read.

3.3.2 Loop Compensation

The values assigned to the variables are initially divided equally based on the reads per source line. However, there are common cases with primitive variables used successively as indices, where their cache miss rate should intuitively be much less than an access to a variable or complex data type in the same line of code. Consider the following snippet,

```
1  for (i = 0; i < N; i++)
2   {
3       int x = a[i];
4       x += b[i];
5       x += c[i];
6   }
```

In this example, the variable i is read on every line. At line 5, the likelihood of a miss for i is likely lower than that of the index to the array c. Using the raw approach previously described would result in an inflated miss count for the index variable i, while having lower than expected counts for each of the variables a,b,and c. We try to account for this by assigning weights to the variables within loops based on the frequency that they are accessed.

We define the source line where a miss occurred as S. We define the inner most loop that S is contained in as L. Within S, we define V to be the set of variables where a read occurs. For each variable v in V, equation 1 details the calculation of the raw weights,

$$v_{raw\_weight} = 1.0 - \left(\frac{\text{Number of lines v is read in } L}{\text{Number of lines in } L}\right)^2 \quad (1)$$

The raw weights for all variables will be below 1.0 as they will have at least one read within L. The weights will be redistributed favoring those variables with higher initial weights, with the sum of the final weights for all variables equaling the number of reads for the original line S. The equations for calculating the final weights are given below.

$$weight\_to\_redistribute = \sum(1.0 - v_{raw\_weight}) \quad (2)$$

We set a threshold for the raw weights. All variables above the threshold have their weights distributed according to their raw weight. Set V' is the set of variables with raw weights above the threshold. For all variables, v' within V',

$$threshold\_weights = \sum v'_{raw\_weight} \quad (3)$$

The adjusted weight is then calculated in equation 4.

$$v'_{adj\_weight} = \left((\frac{v'_{raw\_weight}}{threshold\_weights}) * _{weight\_to\_redistribute} \right) \quad (4)$$

Finally, the adjusted weight is added to the raw weight for our final weight. For those variables below the threshold, the final weight is equal to the raw weight.

$$v'_{final\_weight} = v'_{raw\_weight} + v'_{adj\_weight} \quad (5)$$

With the loop adjustment, the values for the weights from the code in Figure 2 are shown below. We assume a threshold of 0.75, which is the point where a read occurs for that variable in half of the lines within the loop.

	Weights		
Variable	**Raw**	**Adjusted**	**Final**
x			
x[a]	.94	.52	1.46
x[b]	.94	.52	1.46
y			
y[a]	.94	.52	1.46
y[b]	.75	.42	1.17
a	.44	.00	.44
b	.00	.00	.00
Sum	4.0	2.0	6.0

The different accesses for x and y are treated independently. We see that the final weight is equal to the number of memory locations (as distinguished through static analysis) that are read within the loop. When these final weights are applied to the original approximate misses (one miss from each source line) from before, we get the following numbers, with "LA" corresponding to the loop adjusted numbers.

	Approximate Misses							
	x[]		y[]		a		b	
	Raw	LA	Raw	LA	Raw	LA	Raw	LA
Line								
2	.25	.37	.25	.29	.25	.10	.25	.00
3	.00	.00	.50	.60	.25	.10	.25	.00
4	.50	.73	.00	.00	.00	.10	.50	.00
Sum	.75	1.10	.75	.89	.50	.30	.50	.00

3.3.3 Skid Negation

The raw assignment and loop correction pass work best when skid is not a factor. However, when skid is taken into account we need to provide a further pass to help negate the skid. The range and effect of skid manifests differently between architectures. We wanted our approach to be general, so that a minimal number of parameters would have to be specified to have our approach apply to a given architecture. By using an approach similar to that utilized by ProfileMe [6], we can generate a histogram for the distance between the true instruction and where the event based sample landed (the skid factor). Using this information, we can generate a probabilistic model for the effect of the skid. It should be noted that this histogram would not need to be generated per test program, but rather only once when wanting to run code on a new architecture. With this information, our approach need only two parameters: the mean and the variance. Using that information, we assign a weight to the reads within the skid range using a standard Gaussian distribution.

An additional mapping is needed since our approach is source line based, and the skid distribution would be in terms of instructions. There are a few possibilities for this mapping. The first would be to do a one-time linear scan of the instructions in the program and the associated valid source lines. With this information you can calculate the average number of instructions associated to a source line within the program. This could then be used to determine all possible "real" source lines the miss could correspond to. The second approach is more exact, and may be useful for programs with very large skid. This approach would look at the exact instruction given by the event and all of the instructions within the range of the Gaussian distribution for that architecture. The associated source lines would be mapped accordingly.

Formally, for set V which is all variables with reads within the possible distribution of the skid, for each v in V let p be probability assigned to the source line containing the read to v.

$$v\_raw\_skid\_val = p \quad (6)$$

We then take the sum of all the values in the skid distribution.

$$sum\_skid\_vals = \sum v_{raw\_skid\_val} \quad (7)$$

We use that sum to normalize all weights, so the sum of all weights for a particular cache miss will equal one. For all v,

$$v\_final\_skid\_val = (\frac{v\_raw\_skid\_val}{sum\_skid\_vals}) \quad (8)$$

The final skid value is not used as a weight, such as is the case with the loop compensation weights. It is used as a replacement to the raw approximate assignment values. In cases where there is skid, the skid negated value would be the base for passes such as the loop compensation. The following table shows the comparison between raw approximate assignment values versus skid adjusted values. We assume the cache miss was given as being on line 4, the skid distribution has a mean of line 3, with the variance being $\pm$ 1. We assign a probability of .5 to line 3 and .25 to line 2 and 4.

	Approximate Misses							
	x[]		y[]		a		b	
	Raw	SA	Raw	SA	Raw	SA	Raw	SA
Line								
2	0	.07	0	.07	0	.07	0	.07
3	0	.14	0	.14	0	.14	0	.14
4	.5	.07	0	0	0	0	.5	.07
Sum	.5	.28	0	.21	0	.21	.5	.28

For both approximate miss methods, the sum of the attributed cache misses for the variables is 1. The raw methods assumes correct data for the attributed cache miss, which is why values are only present for line 4. The skid negation technique assumes a probabilistic distribution over the source lines, which is why there are values assigned for lines 2, 3, and 4, even when the given line for the cache miss was line 4. Since this miss occurred within a loop, the next step would be to apply the loop compensation weights to the new values, which would see an increase in the attributed cache misses for the arrays, and a decrease for the primitives used to index them.

4. EXPERIMENTAL RESULTS

For our experiments, we ran tests on three programs from the SPEC CPU2000 benchmark suite on an Intel Itanium 2 machine running Linux. The test programs and hardware configurations are the same Buck used for his experiments. As we are comparing our approach against existing direct methods, it was important to have verified direct data centric results to compare against.

For our experimental programs, we gather data from three types of runs. All runs use event driven sampling to measure cache misses. For each type of run, we take the average values from five runs.

1. Sampling with skid free IP and precise effective address gathered using hardware support that negates the skid
2. Sampling with skid free IP without effective address gathered using hardware support that negates skid
3. Sampling with skid using generic hardware counter that does not negate skid

We use the values from the first type of run to perform a traditional direct data centric approach. This serves as a baseline to compare our approach against. For each graph in the results section, the given approximation approach will be compared against the direct approach. We use the second type of run to perform our loop compensation adjustment. The lack of effective address information means we use our approach exclusively to assign the cache misses to the proper variables. The data from the final type of run suffers from skid, so we perform the loop compensation and the skid negation pass to assign miss responsibility to variables.

For the distribution of skid values, we use a probabilistic distribution similar to the one we used in our prior example. The Itanium used in these experiments utilizes in-order execution of instructions, meaning the skid is an artifact of the pipeline and is less severe compared to most out-of-order

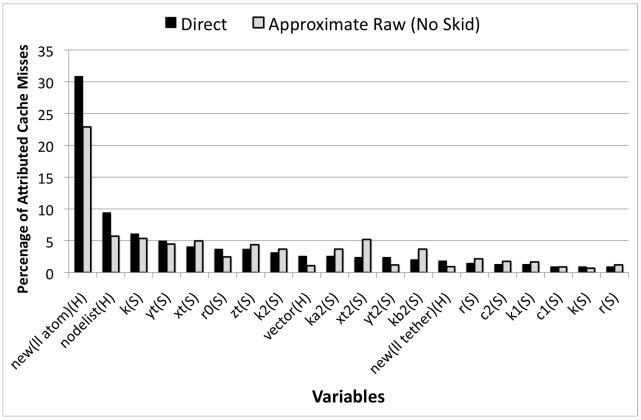

Figure 3: ammp cache misses for direct versus approximate raw with no skid

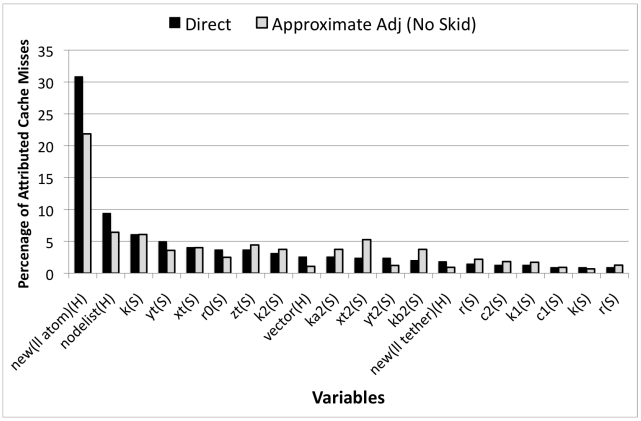

Figure 4: ammp cache misses for direct versus approximate adjusted with no skid

execution architectures. We set the skid distribution to be 3 source lines, with the mean being one source line prior to the one given. We use the same probabilistic distribution given in the example, .5 for the mean instruction and .25 for the other two (one of which is the source line given by the event driven sample).

4.1 188.ammp

The ammp benchmark is a C program that runs molecular dynamics on a protein-inhibitor complex that is embedded in water [14].

4.1.1 No Skid

We begin by applying our raw approximate assignment to the no-skid run. The results of this run are shown in Figure 3. The top 20 variables are shown, sorted in rank by the direct method values. The variable names are listed followed by the allocation method for the variable in parentheses. 'S' is for stack allocated variables, 'G' is for global, and 'H' is for heap. The source lines given for this run are guaranteed to be correct, so we compare our software only approach to the direct method, which was accomplished by comparing the effective address of the miss to a record of all the allocations and frees within the program. Our method compares

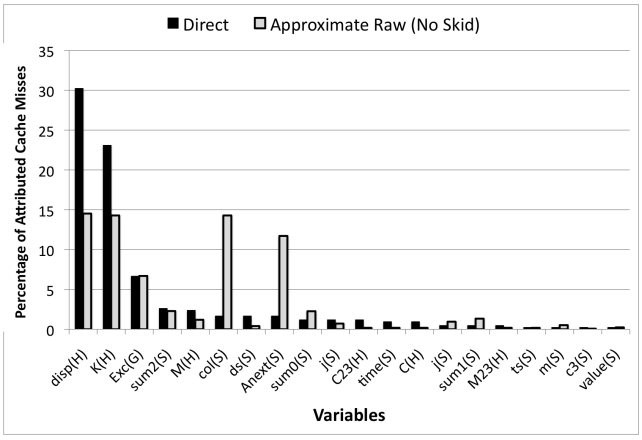

Figure 5: equake cache misses for direct versus approximate raw with no skid

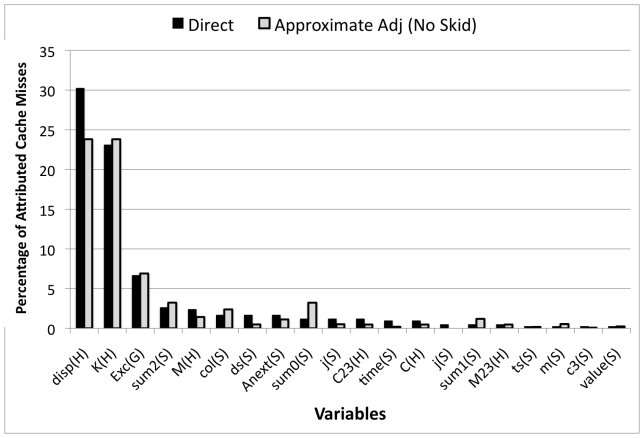

Figure 6: equake cache misses for direct versus approximate adjusted with no skid

favorably against the direct method. The heap allocated variable *new*, which is a node from a linked list, and the heap allocated variable *nodelist* suffer a decrease in the attributed cache misses, but the relative rank-order remains consistent. Since our approach is a software approximation, our goal is not an exact match to the 'true' data, but rather a light-weight mechanism to find the same hotspots that a direct approach (if possible on the hardware) would provide.

We also apply our loop compensation pass to the data. This is shown in Figure 4. This program has few loop nests. The loops within this program are very large with little use of loop iterators serving as indices into the arrays. Because of this, there are few artificially inflated outliers that the loop compensation pass would need to take care of, and the results are similar to that found in Figure 3.

4.2 173.equake

The equake benchmark is a C program that simulates the propagation of waves in large, heterogeneous valleys [1].

4.2.1 No Skid

We begin with our raw approximate assignment comparison, shown in Figure 5. For this benchmark, there are outliers based on misappropriated misses to primitives. We apply our loop compensation pass and get the values shown in Figure 6. The main inaccuracy, as compared to the direct method, is the lower than expected values for variable *disp*. The reason for this is that *disp* and k are present together on the same lines for most of their reads and have the same number of unique reads on each one of these lines. In terms of the lines where the misses actually occur, their signatures are virtually identical in regards to our analysis. This leads to a close to expected value for k, but a lower value for *disp*.

4.2.2 Skid

The prior runs benefited from having the correct source line associated with the cache miss. The raw assignments with the skid active give us the data in Figure 7. Due to the skid factor, the outliers are more apparent and the variable assignments are much different than the baseline direct data method. By applying our software skid negation, we achieve

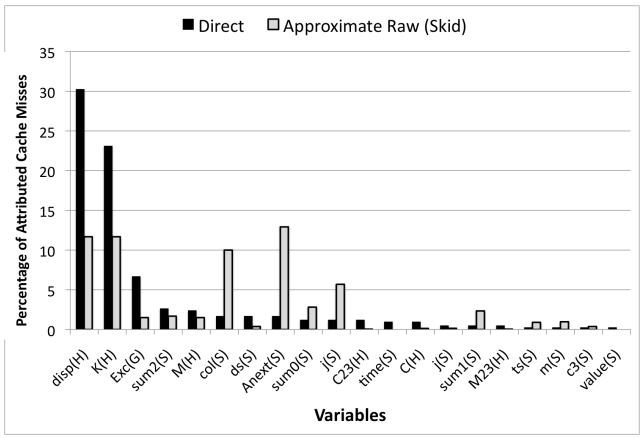

Figure 7: equake cache misses for direct versus approximate raw with skid

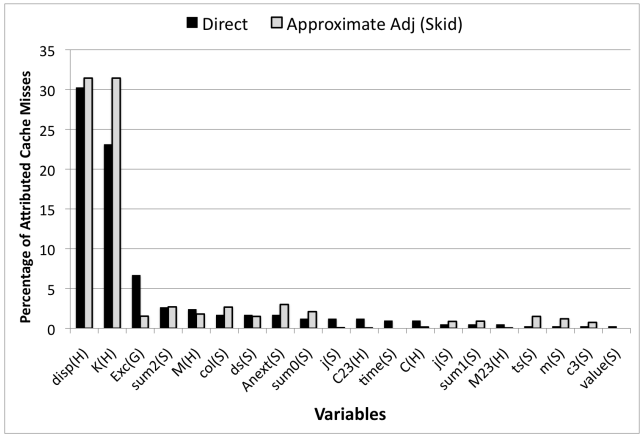

Figure 8: equake cache misses for direct versus approximate adjusted with skid

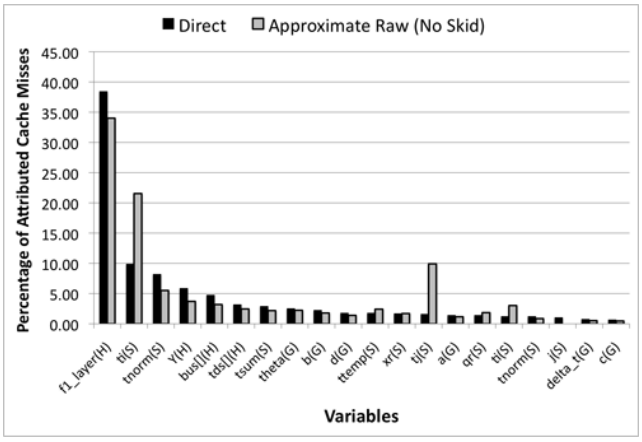

Figure 9: ART cache misses for direct versus approximate raw with no skid

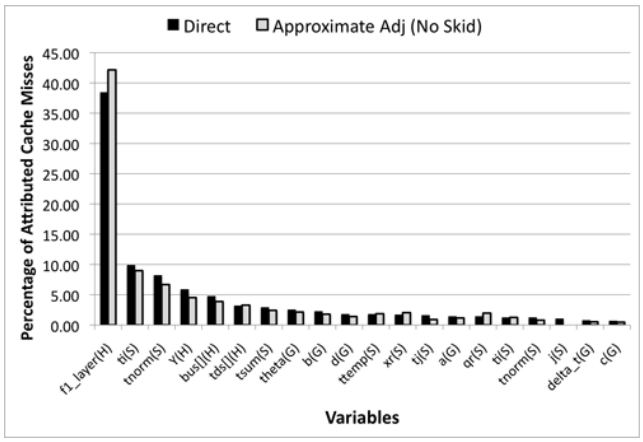

Figure 10: ART cache misses for direct versus approximate adjusted with no skid

more reasonable data. The results of the skid adjustment (with combined loop adjustment) are shown in Figure 8.

The two main variables, *disp* and *k*, are once again linked together and at the top of the list of the appropriated misses. The skid adjustment helped to improve their performance because the source line wrongly attributed for many cache misses was one line ahead of a line that performed multiple reads on the two variables. The attributed misses for variable *Exc* was low before the skid adjustment and the value remained low after the skid adjustment. This was due to the misattributed source line being directly after a call in which a read to *Exc* was the comparison operation before the return. At this time, our analysis does not move backwards through calls, so we were unable to redistribute values to that variable.

4.3 179.art

The ART 2 benchmark is a C program that uses neural network models to recognize objects in a thermal image. [5]

4.3.1 No Skid

We first examine our raw approach versus the direct data method, shown in Figure 9. Our initial method performs comparably against the direct method with the exception of the outliers for stack allocated variables *ti*, *tj*, and *ti*. These are all loop iterator variables, and are the motivation for the loop adjustment pass. The loop adjustment pass, shown in Figure 10, eliminates the outliers. For this program, our method lines up almost perfectly with the direct method.

4.3.2 Skid

The raw approximate values for the skid run are shown in Figure 11. The effect of the skid results in diminished values for most of the top 10 variables found by the direct method. The results for the skid adjustment pass are shown in Figure 12. Our skid adjusted approach gives us a rank-order approximately the same as the direct method. The following table shows the rank-order for the top ten variables (as given by the direct data approach).

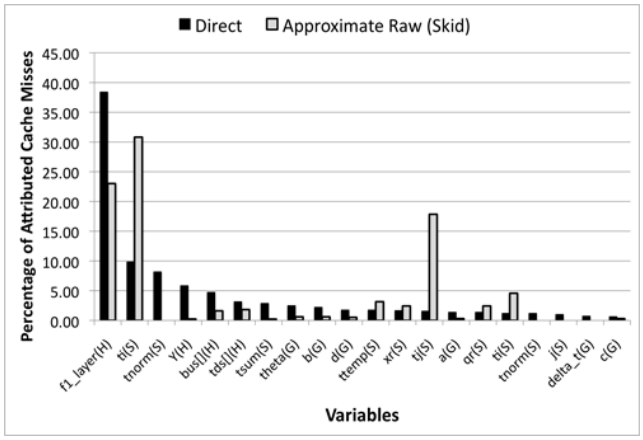

Figure 11: ART cache misses for direct versus approximate raw with skid

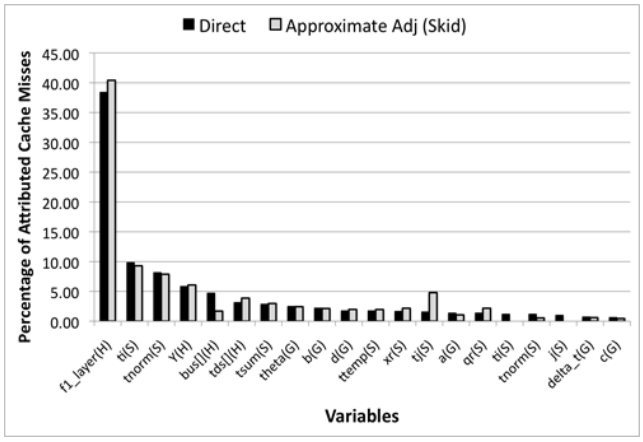

Figure 12: ART cache misses for direct versus approximate adjusted with skid

	Rank Order		
Variables	**Direct**	**Skid Raw**	**Skid Adjusted**
f1_layer(H)	1	2	1
ti (S)	2	1	2
tnorm(S)	3	19	3
Y(H)	4	15	4
bus[](H)	5	9	14
tds[](H)	6	8	6
tsum(S)	7	16	7
theta(G)	8	10	8
b(G)	9	12	11
d(G)	10	11	12

The low number of misses assigned to *bus* is due to the reads for that variable occurring in the middle of a set of conditionals, with the misappropriated instruction occurring immediately after the block for the final 'else.' A solution to this problem would be assigning probabilistic weights to different regions in the conditional block based on generic profiling numbers, but that is not present in our current analysis.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have discussed a technique for approximating cache miss totals for variables using only static analysis and runtime data gathered from event driven sampling utilizing generic hardware counters. Existing techniques resolve these misses by matching the effective address to a maintained list of allocated memory by monitoring allocations and frees. Our approach allows this type of analysis to run on systems without dedicated hardware support that provides exact effective address and IP information that is unaffected by skid. It also removes the need to maintain the allocation list. Our approach is meant to be a supplementary method to existing techniques, in cases where the hardware does not allow the existing types of analysis to be run or monitoring the allocations adds too much overhead.

The main area of future work is to apply these techniques to other architectures that have skid-negating hardware, specifically those platforms with out of order execution. An increased distribution of skid values creates backtracking problems in terms of control flow. We have already encountered issues with conditionals and function calls. The greater the skid factor, the more considerations about control flow become a factor. Expanding our approach to other platforms would allow us to further test the validity of our technique across different degrees of skid and cache replacement policies, both of which might change how our approach would perform.

Another area of future work would involve doing the mappings at the instruction level to compare the accuracy. This would have the advantage of not having to divide the sample over all of the reads on the mapped source line as is the case in our current approach. However, this would involve having reliable alias analysis and other static analysis at the machine code level for any platform we wished to support.

6. ACKNOWLEDGMENTS

The work supported in part by DOE grants DE-FC02-06ER25763 and ER25925 and NSF grant EIA-0080206.

7. REFERENCES

[1] H. Bao, J. Bielak, O. Ghattas, D. R. O'Hallaron, L. F. Kallivokas, J. R. Shewchuk, and J. Xu. Earthquake Ground Motion Modeling on Parallel Computers. In *Supercomputing '96*, Pittsburgh, Pennsylvania, Nov. 1996.

[2] E. Berg and E. Hagersten. Statcache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *In Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2004.

[3] B. R. Buck. Data Centric Cache Measurement on the Intel Itanium 2 Processor. In *Proceedings of SuperComputing*, 2004.

[4] B. R. Buck. *Data Centric Cache Measurement Using Hardware and Software Instrumentation*. PhD thesis, University of Maryland, 2004.

[5] G. A. Carpenter and S. Grossberg. Art 2: Self-Organization of Stable Category Recognition Codes for Analog Input Patterns. *Appl. Opt.*, 26(23):4919–4930, Dec 1987.

[6] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, pages 292–302, Washington, DC, USA, 1997. IEEE Computer Society.

[7] P. J. Drongowski. "Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors". Technical report, Advanced Micro Devices, Inc., 2007.

[8] Intel. *Intel Itanium 2 Processor Reference Manual*. Intel, May 2004.

[9] Intel. *Intel 64 and IA-32 Architectures Software DeveloperÕs Manual Volume 3B: System Programming Guide, Part 2*. Intel, June 2010.

[10] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGOÕ04)*, 2004.

[11] X. Liu and J. Mellor-Crummey. Pinpointing Data Locality Problems Using Data-centric Analysis. In *International Symposium on Code Generation and Optimization (CGO11)*, 2011.

[12] C. Mccurdy and J. Vetter. Memphis: Finding and Fixing Numa-Related Performance Problems on Multi-Core Platforms. In *In Proceedings of ISPASS*, 2010.

[13] J. M. Mellor-Crummey, R. J. Fowler, and D. B. Whalley. Tools for Application-Oriented Performance Tuning. In *International Conference on Supercomputing*, pages 154–165, 2001.

[14] I. T. Weber and R. W. Harrison. Molecular Mechanics Analysis of Drug-Resistant Mutants of HIV Protease. *Protein Engineering*, 12(6):469–474, 1999.