LLNL-TR-604112

Cache Performance Analysis and Optimization

K. Mohror`, B. Rountree

November 26, 2012

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Cache Performance Analysis and Optimization

Kathryn Mohror and Barry Rountree

1 Overview

There is a critical lack of tools for understanding application cache performance. While programmers understand the general benefits of temporal and spatial locality, no tool exists for measuring how close any given piece of code approaches optimal performance, nor do we have a tool that can tell a programmer what optimal performance might be. NUMA architectures, multi-level caches and out-of-order execution all conspire to prevent any trivial solutions from being effective. As we move to ever-larger clusters, the penalty for suboptimal performance in terms of wasted resources continues to increase.

Despite demonstrated potential for application performance improvement, cache usage analysis and optimization have not advanced appreciably past rules of thumb and rough heuristics, especially for complex scientific codes. Generally, any nontrivial optimizations are done "by hand" requiring significant time and expertise. The granularity exposed by currently available tools prevents giving an accurate picture of how cache usage relates to performance, much less to suggest data layout changes for improved performance. As a result, application developers must rely on hard-won expertise and time-consuming hand-tuning of performance critical loops in order to approach optimal performance. While their tuning efforts may result in significant performance gains on a single platform, the improvements from the transformations are not generally portable across platforms or even compilers.

With the advent of enhanced Precise Event-Based Sampling (PEBS) counters on recent Intel processor architectures and equivalent technology on AMD processors, we are now in a position to remedy this problem. Prior to the introduction of PEBS counters, cache behavior could only be measured reliably in the aggregate across tens or hundreds of thousands of instructions. With the newest iteration of PEBS technology, cache events can be tied to a tuple of instruction pointer, target address (for both loads and stores), memory heirarchy and observed latency. With this information we can now begin asking questions regarding the efficiency of not only regions of code, but how these regions interact with particular data structures and how these interactions evolve over time. In the short term, this information will be vital for performance analysts understanding and optimizing the behavior of their codes for the memory hierarchy. In the future, we can begin to ask how data layouts might be changed to improve performance and, for a particular application, what the theoretical optimal performance might be.

2 Approach

In this feasibility study, we began to explore the capabilities of the novel hardware performance counters on the Nehalem, Westmere and Sandy Bridge architectures. These new counters provide aspects of cache and memory performance such as memory latency and hit locations can now be measured with far higher degrees of precision. We need to evaluate the accuracy of these measurements. As part of this work, we have created a runtime library and API that allows these measurements to be made by users. Figure 1 shows an overview of our initial exploratory tool. We use the Pin [1] tool to extract instructions, addresses of memory references, and line numbers from the application binary. Then, we link the application against our runtime library and execute it. Using the output from the Pin tool and our runtime library, we can extract memory reference latencies for specific instructions and addresses for an application run.

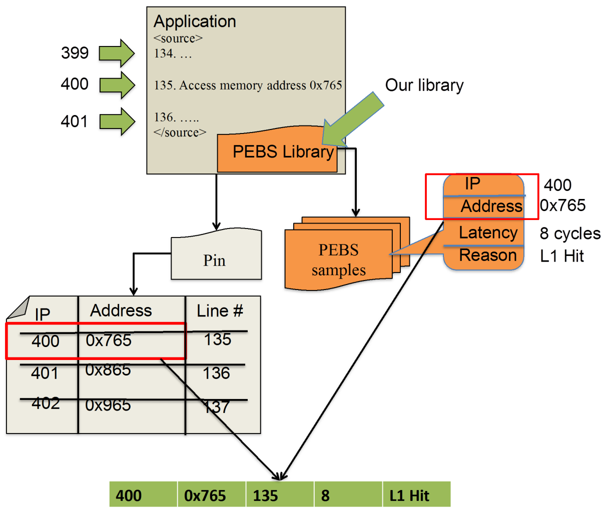
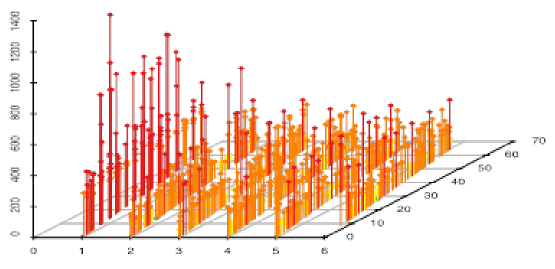


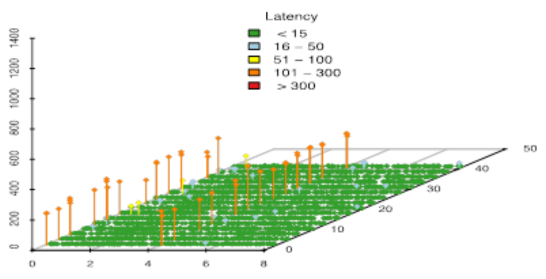
Figure 1: **Overview of our exploratory implementation.** We use the Pin tool to extract instructions, memory reference addresses, and line numbers from binaries. Following this, we run the application linked against our PEBS-enabled library to generate latency information for instructions and memory addresses.

3 Results

Our tool shows that it is feasible to collect and display memory reference latencies for particular application data stuctures. Here we use a simple synthetic benchmark to evaluate our implementation. In Figure 2, we show the latencies for memory references over time for a two versions of a simple benchmark that simply traverses an array called `stomp_buf`. In the figure, green and blue colors indicate low latency values, while red and orange indicate high values. The x-axes show the row numbers of the `stomp_buf` array; the y values are the column numbers of the array; the z-axes give the latency values of accessing each array location. The first version, "stomp," traverses the array in column-major order, which is suboptimal for the C programming language. The second version labeled "swapped" traverses the array in row-major order, which gives better performance in terms of cache utilization. For the "stomp" version, of particular interest is not only the occasional high-latency load, but the high degree of variation in measured latencies of loads. We see that there are generally high memory reference latencies throughout the execution of the program, which are especially bad for the first iteration of the loop, reaching as high as over 1200 clock cycles for one reference. On the other hand, the graph of latencies for the "swapped" benchmark shows lower overall latencies, with maximum times less than $\approx$ 300 clock cycles and much less variation than for "stomp." Existing practice relies on average cache latency performance. We feel that this is insufficient for the most effective optimzation of data structure layout in cache.



(a) Latencies for stomp benchmark



(b) Latencies for swapped benchmark

Figure 2: **Latencies for array locations over time for two toy benchmarks.** The "stomp" benchmark traverses the array `stomp_buf` suboptimally in row major order, leading to high memory reference latencies. The "swapped" benchmark, on the other hand, traverses the array in column major order, which means that the prefetcher can bring in the array elements into the L1 cache for lower memory latencies.

We also investigated using different sampling rates with our tool. There is an inherent trade-off between the number of measurements taken and accuracy and overhead. We want to perturb the application as little as possible by taking as few measurements needed to ensure accurate results. In our experiment, we used 4 different rates, where the rate indicates how many memory reference instructions occur before a PEBS record is written. We used values of 10, 50, 100, and 150 and measured the latencies observed at different levels of the cache hierarchy. In Figure 3, we show the latencies for events occuring in each level of the cache at the different sampling rates for both versions of the benchmark. For the "stomp" version, we again see that there are higher overall latencies, and that there are a larger number of memory accesses that are served from the L3 cache. However for "swapped," most memory references are served from L1 and have much lower latencies.

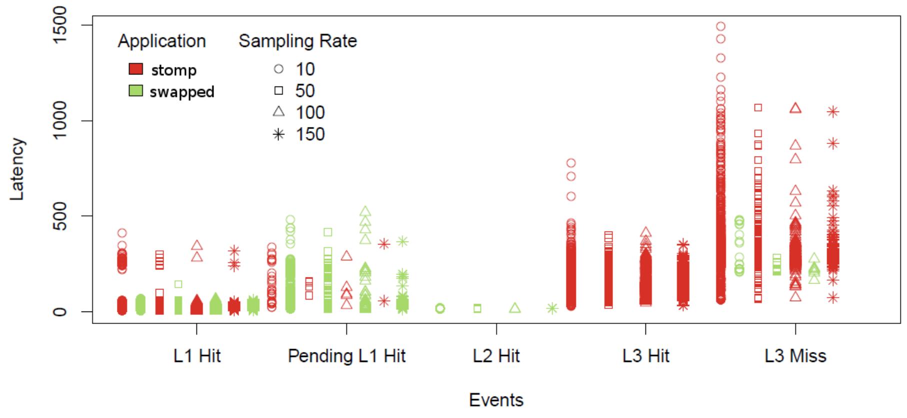


Figure 3: **Latencies for memory references at different levels of the cache at different sampling rates.** The "swapped" benchmark has more memory references serviced by the L1 cache with lower overall latencies than the "stomp" version.

Another observation we can make from Figure 3 is that the conclusions we draw are similar for the different sampling rates. This is further supported by the data in Table 1 which shows the percentage of memory references that were serviced by different levels of the cache hierarchy for the different sampling rates. Here we see that as the number of memory references sampled decreases, the reported percentage of references serviced at each level of cache stays relatively stable. For example, the percentages of L1 hits for the "swapped" version of the benchmark are very close to the value reported at the highest sampling rate of 10 (99.7%) as the sampling rate decreases up to 150 (99.%). The same is true for the "stomp" version, where the percentages for L1 hits are very close to 91.3% even at the lowest sampling rate of 150 which reports 91.8%. These results indicate that it is possible to capture the memory performance of an application at lower sampling rates, which means less overhead and perturbation to the application.

Table 1: **Percentage of memory references at different levels of cache for varying sampling rates.**

Sampling Rate	**10**		**50**		**100**		**150**	
Benchmark	swapped	stomp	swapped	stomp	swapped	stomp	swapped	stomp
L1 Hit	99.7	91.3	99.7	91.0	99.8	91.4	99.8	91.8
Pending L1 Hit	0.22	0.08	0.23	0.05	0.14	0.02	0.01	0.07
L2 Hit	0.03	0.0	0.0	0.0	0.04	0.0	0.02	0.0
L3 Hit	0.0	4.1	0.0	5.9	0.0	5.9	0.0	5.5
L3 Miss	0.01	4.5	0.04	2.9	0.01	2.6	0.05	2.6

4 Conclusions and Future Work

In this work we have succeed in demonstrating that advanced PEBS counters can be gathered reliably independent of specialized kernel modules or third-party tools that depend on them. Because we have shown that these counters can be accessed using the low-level kernel MSR interface, we can now begin planning how to scale up the use of these counters on TLCC2 machines where this interface in enabled. We are well-positioned to generate the first results at scale using this technique.

Looking forward, we are also now well-positioned to leverage the scale available at LLNL by quickly mapping tens of thousands of small code perturbations to not only wall-clock time and cache misses but latency measurements as well. This brute-force examination of the code space will allow us to validate future models and runtime systems for optimizing cache performance.

Finally, we have laid the groundwork for exposing more advanced processor features to user space. Within the past six months we have created a user-level interface to not only gather latency measurements but also power measurements and capping. To the best of our knowledge, this low-level interface work at scale is unique to LLNL. While cache optimization is interesting in its own right, our work becomes even more compelling in combination with other data and control that is only possible at the processor level.

5 Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-TR-604112.

References

[1] Intel Corporation, *Pin - A Dynamic Binary Instrumentation Tool*, http://www.pintool.org/.