Author Retrospective

Analytical Cache Models with Applications to Cache Partitioning

G. Edward Suh
Electrical and Computer Engineering
Cornell University
Ithaca, NY
suh@csl.cornell.edu

George Kurian, Srinivas Devadas, Larry Rudolph*
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA
{gkurian, devadas, rudolph}@csail.mit.edu

ABSTRACT

We summarize the history of the work, revisit primary observations and lessons that we learned from the modeling effort, and also briefly describe follow-up work to show how the research direction evolved over time.

Original Paper: http://dx.doi.org/10.1145/377792.377797

Categories and Subject Descriptors

B.3.2 [**Design Styles**]: Cache Memories; D.4.1 [**Process Management**]: Scheduling; I.6.5 [**Simulation and Modeling**]: Model Development

Keywords

Cache Partitioning; Process Scheduling; Cache Models

1. HISTORY

The cache modeling work in the 2001 ICS paper was carried out within a larger project named Malleable Caches. While cache design and performance had been extensively researched, the Malleable Caches project was motivated by three observations that were novel at the time: (1) the traditional assumptions on the locality of memory accesses were no longer true for emerging applications such as streaming and real-time applications, (2) the emergence of more spacious caches enabled program blocks to reside longer within the cache, even across scheduling time slices, and (3) caches were increasingly shared amongst multiple processors. The project aimed to improve the efficiency of large caches by dynamically allocating cache resources to run-time workload and sharing behaviors. The analytical model introduced in the ICS paper was a departure from cache optimizations, verified via simulation, that largely focused on static allocation for a single program.

*Also affiliated with Two Sigma

ICS 25th Anniversary Volume. 2014
ACM 978-1-4503-2840-1/14/06.
http://dx.doi.org/10.1145/2591635.2591662.

Dynamic partitioning of a shared cache, one thrust of the Malleable Caches project, aimed to more efficiently allocate cache space among multiple tasks. An effective cache partitioning technique requires solving two main challenges: a low-cost, fine-grained mechanism to control cache space for each task, and an algorithm to determine how much cache space that each task needs. For our project, column caching [2] provided a mechanism to partition caches. However, at the time, there was no work that answered the question of how to determine the best allocation.

The cache modeling work was largely motivated by this need to understand the impact of cache sharing and develop an effective cache allocation policy. The goal of the model was to estimate the overall cache miss-rate (or the number of misses) of a shared cache given individual program characteristics, a cache configuration, and a sharing pattern. The model assumed that a cache is accessed by one program at a time in a time-shared fashion, essentially modeling a single-core system. It was a pleasant surprise to discover that the model can also be used to estimate the performance of shared caches in multicore systems.

The main challenge in characterizing the performance impact of multiple programs sharing a cache is estimating the amount of cache space used by each program without simulating every type of workload combination. One key insight is that the individual isolated program miss-rate curves can be combined to determine the shared cache performance. The miss-rate curve represents the probability of a cache miss as a function of cache size, and enables an estimation of the number of unique cache blocks accessed over a given time period. The number of cache blocks that each program accesses over its time quantum can be combined across context switches to determine the cache footprint of a program at the beginning of its time quantum. Once the initial cache footprint is known, the miss-rate curve can be used to obtain the miss-rate for each program over a time quantum.

2. OBSERVATIONS AND LESSONS

Role of Analytical Modeling.

Analytical methods are rarely used in designing practical systems because it is often extremely difficult, if not impossible, to accurately model complex systems mathematically. As a result, most architectural studies rely on simulations or emulations. This cache modeling effort shows, however, that there is a role for analytical studies in system designs. For example, the modeling process forced us to identify first-

order effects and tradeoffs in a shared cache, and provided insight and intuition on the range of interference behaviors. The computational efficiency of the analytical model enabled our study of the high-level tradeoffs in a large design space.

It was both interesting and surprising to discover that the individual miss-rate curves were sufficient to accurately capture the performance impact of sharing a cache. The model showed that there were two program properties in the miss-rate curves that largely determine the shared cache performance. One, the number of cache blocks that are accessed over a period largely determines how much cache space is allocated to each program. Traditional cache policies, such as LRU, prioritize recently accessed data. As a result, programs that access a large amount of data quickly are likely to keep more data in the cache compared to the ones with high locality. Two, the cache performance for each program is heavily influenced by how sensitive the program's miss-rate is to the cache size. Some programs may benefit significantly from additional cache space, yet programs such as streaming applications may be largely insensitive to the cache size.

Shared Cache Interference.

While well-known now, our study based on the cache model showed that the performance impact of interference in shared caches could be significant and that the traditional cache management policies could exacerbate the problem.

The study showed that the cache performance of a memory-intensive program with a large cache footprint could be significantly degraded when a cache was simultaneously used by another memory-intensive program. Streaming applications turned out to be particularly damaging because traditional cache policies blindly allocated space to a program with lots of misses even when the program does not benefit from the additional space. In certain cases, we even found that the overall throughput of a multicore system could be improved by leaving some cores idle to reduce the cache interference.

While not as significant as interference from simultaneous sharing, our study also suggested that context switches could have a noticeable impact on cache performance, especially for large shared caches. In a traditional round-robin schedule, the LRU policy evicts cache blocks for old jobs. These blocks may get evicted just before the job is rescheduled leading to lots of cold misses after a context switch.

These observations suggested that cache performance could be significantly improved with more intelligent allocation or scheduling that minimizes harmful cache interference. As an example, for a cache that is shared amongst multiple cores, partitioning can allocate space based on the utility of additional space to each program and limit the pollution from streaming applications. Under time sharing, cold misses can be reduced by keeping a small amount of critical data for each program across context switches.

Limitations.

While the analytical model provided valuable insight, we found that there was still a gap between the model and practical systems. In particular, an accurate prediction for set-associative caches turned out to be difficult because real-world applications often accessed cache sets in a non-uniform manner. In certain cases, this non-uniform access pattern could artificially reduce cache interference among programs because accesses from programs might use different sets. Our attempt to incorporate the non-uniform accesses complicated the cache model with only a limited improvement in its accuracy. In practice, however, we found that the interference that was captured by the fully-associative cache model was often sufficient to make partitioning or scheduling decisions even for set-associative caches.

As another limitation, our cache model targeted multi-programmed workloads and did not consider accesses to common memory locations by multiple threads. As a result, the model could not handle multithreaded programs.

3. OUR FOLLOW-UP WORK

Cache and Memory Monitors [8].

The cache model showed that miss-rate (or miss) curves contained key program characteristics to understand shared cache performance. To obtain individual program's miss(-rate) curves at run-time for optimizations, we introduced *Recency Hit Counters*. The scheme leverages the LRU stack used for replacement policy decisions. The LRU stack consists of addresses from a sequence of cache or memory accesses such that the distance from the top (i.e., stack distance) represents how recently the address was accessed. A counter, maintained for each LRU stack distance, is incremented when the address in that LRU distance is accessed. Because the stack distance can be used to predict whether an access would be a hit or not given a certain cache/memory size, these counters effectively encode the number of hits as a function of cache size. In essence, the counters represent marginal gains ($g(x)$), which is the number of additional hits for a particular job when the number of allocated memory blocks is increased from $x-1$ to x.

Thread Scheduling [10, 8].

We studied how thread scheduling could be improved taking memory contention into account in a shared-memory multiprocessor system. The goal was to find the job schedule that minimizes the processor idle time due to either page faults or processors with no jobs scheduled. The study first found that job scheduling has a significant impact on shared memory performance due to interference. We then investigated algorithms to find good schedules. For a small number of jobs, the analytical model could be used to estimate the memory performance of all possible job schedules and determine the best one. For a large number of jobs, a brute-force search based on the model was intractable so a new heuristic method was developed. It first identified the memory needs of jobs by allocating available memory capabilities based on marginal gains. Then, the jobs were grouped together to balance the total memory needs in each time slice.

Cache Partitioning [8, 9, 11].

We investigated partitioning of shared last-level caches in the context of both chip multiprocessors (CMP) and simultaneous multithreading (SMT). The recency hit counters were used to obtain marginal gains and cache space was allocated based on the gains. While the high-level approach was simple, a few challenges had to be addressed to make partitioning effective for set-associative caches. First, because the LRU stack was only maintained within a set, the marginal gains were obtained at a coarser granularity, such as one counter for each cache way. Second, partitioning based on cache ways effectively reduced the associativity for each program and increased conflict misses. To address this problem, we introduced a modified LRU policy that replaces a cache block based on the owner and its cache allocation.

Finally, cache miss curves were often non-convex and finding the optimal allocation efficiently was difficult. Our solution relied on a heuristic that uses greedy algorithms with multiple starting points.

4. FOLLOW-UP RESEARCH DIRECTIONS

This section provides a brief overview of the research directions that followed our work.

Thread Scheduling.

There has been a large body of work on improving scheduling decisions considering shared cache/memory contention, especially in the context of multi-core processors. For example, Chen et al. [1] presented a scheduling algorithm that leverages the constructive cache sharing behavior of multi-threaded programs. Zhuravlev et al. [15] proposed to improve scheduling using classification schemes which determine how programs affect each other when competing for shared resources such as caches, memory bus, etc.

Partitioning for Efficiency.

There have been efforts to improve both effectiveness and costs of cache partitioning. Qureshi et al. [6] proposed utility-based cache partitioning using dynamic set sampling. The sampling enables obtaining a complete isolated miss curve of each application at run-time without maintaining full recency counters for each application. The number of sampled sets is chosen to obtain good accuracy with negligible hardware overhead. The work also uses a lookahead algorithm to avoid local minima when searching for the best allocation.

To avoid losing associativity, partitioning of cache sets instead of ways has also been investigated [13]. Such schemes, however, require significant redesign of the cache arrays and must do scrubbing, i.e, flush data when resizing partitions. Sanchez et al. proposed Vantage [7] that allows fine-grained partitioning while maintaining high associativity and strong isolation. Vantage requires cache designs with high associativity and good hash functions (e.g., zcaches).

Partitioning for Commodity Processors.

Researchers have investigated partitioning on commodity processors without custom hardware support. One set of techniques obtained the miss curve (or marginal gains) using existing hardware functions such as performance counters. Tam et al. [12] proposed to collect the cache references using the Sampled Data Address Register (SDAR) of IBM Power5 processors and build the LRU stack model at regular intervals to estimate the miss curve of an application. West et al. [14] proposed an analytical model that measures the cache occupancy of an application and combined this with miss rate information from performance counters to estimate the miss curve. Multiple occupancy (/miss-curve) points are obtained by co-scheduling the application with different co-runners or by dynamically throttling its execution rate.

Another set of techniques explored the cache partition space using hill climbing [5], albeit with non-convex problems. To enforce partitioning policies, these approaches use virtual memory and page coloring to constrain the pages of a process to specific cache sets. Repartitioning requires potentially costly recoloring, involving page mapping updates and copying of physical pages.

Partitioning for Quality of Service.

Researchers have shown that cache partitioning can also be used to improve Quality of Service (QoS). For example, Iyer [4] designed a QoS framework for shared caches on CMP platforms. The hardware partitions the cache into multiple regions and directs memory references to separate regions to enforce priority. Cook et al. [3] proposed a partitioning technique that allowed latency-sensitive foreground jobs to run simultaneously with throughput-bound background jobs. The proposal ensures that enough shared cache space is allocated to the foreground job.

5. REFERENCES

[1] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling Threads for Constructive Cache Sharing on CMPs. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures*, 2007.

[2] D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-specific Memory Management for Embedded Systems Using Software-controlled Caches. In *Proceedings of the 37th Annual Design Automation Conference*, 2000.

[3] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic. A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-efficiency While Preserving Responsiveness. In *Proc. of the 40th Annual International Symposium on Computer Architecture*, 2013.

[4] R. Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *Proceedings of the 18th Annual International Conference on Supercomputing*, 2004.

[5] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In *Proc. of the International Symposium on High Performance Computer Architecture*, 2008.

[6] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[7] D. Sanchez and C. Kozyrakis. Vantage: Scalable and Efficient Fine-grain Cache Partitioning. In *Proc. of the 38th International Symposium on Computer Architecture*, 2011.

[8] G. E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *Proc. of the 8th International Symposium on High-Performance Computer Architecture*, 2002.

[9] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Cache Partitioning for Simultaneous Multithreading Systems. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, 2001.

[10] G. E. Suh, L. Rudolph, and S. Devadas. Effects of Memory Performance on Parallel Job Scheduling. In *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001.

[11] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *Journal of Supercomputing*, 28(1), 2004.

[12] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[13] K. Varadarajan, S. K. Nandy, V. Sharda, A. Bharadwaj, R. Iyer, S. Makineni, and D. Newell. Molecular Caches: A Caching Structure for Dynamic Creation of Application-specific Heterogeneous Cache Regions. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[14] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang. Online Cache Modeling for Commodity Multicore Processors. *SIGOPS Oper. Syst. Rev.*, 44(4), 2010.

[15] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.