Dynamic Cache Partitioning Based on Software Hints

Grant Jinilang

ABSTRACT

Shared last-level cache has became the bottleneck of scalability in chip-multi-processors (CMPs) as one application's accesses to shared cache suffer from interferences caused by other cores' accesses. Cache partitioning has been proposed to partition shared cache among cores or applications to alleviate interferences in shared cache. Existing cache partitioning techniques assumes no prior knowledge is known about the applications before they starts running and thus may start with a terrible initial partitioning. However, application developers, compilers or operating systems may be able to provide software hints about the application's cache usage beforehand. In our work, we studied what kinds of software hints might be useful and how they might be used assuming those hints could be generated. We designed and implemented a system that uses software hints for cache paritioning. Evaluations showed that software-hinted cache paritioning indeed improves shared cache performance in terms of total number of misses. Our work provides the first step for future works on software-hinted cache paritioning.

1. INTRODUCTION

In traditional chip-multi-processors (CMPs), the last-level cache is shared among all cores and LRU is usually used as the replacement policy. Shared last-level cache is long konwn to suffer from interference among cores. The LRU policy implicitly partitions the shared cache among cores based on their requests for cache resources. The resulted cache partitioning among cores is unlikly to be optimal in terms of the total number of cache misses. This is because the marginal benefit gained from increasing cache space differ across cores. Limited cache resources should allocated to applications that may get more marginal benefits from them.

Application developers might be able to predict their applications' cache usage and thus request an optimal amoumt of shared cache for his application. For example, the application developer might know that his application is a streaming application. In that case, although the application demands a large amount of cache, the marginal benefits the application gains from the additional cache space is small. Thus he requests a small amount of cache according the application's working set size. So application programmers may provide hints for requesting shared cache space.

Compiler might be able to analyze the program at compile time and predict the application's working set size and thus make request accordingly. The operating sytem might be able to analyze the application's live trace and utilize more sophisticated algorithms to predict its working set size that what can be done in hardware. Thus the operating system might as well provide hints for requesting shared cache space.

We understand that generating those hints might be a hard problem itself. However, as a logical first step, we'd like to know whether those hints would be helpful assuming idea hints can be generated. Therefore, in this project, we performed a limit study and designed a cache partitioning mechanism based on software hints to investigate how software hints may be utilized in cache partitioning to minimize cache misses. We believe our work can serve as the foundation of future works that intend to investigate cache partitioning based on software hints.

2. BACKGROUND AND RELATED WORKS

Serveral cache partitioning mechanisms have been proposed by previous works to minimize cache misses in shared cache. [7] proposed a utility-based cache partitioning strategy. In their approach, each core is associated with a utility monitor (UMON). UMON has a tag directory which caches a tag per set per way. In order to reduce hardware overhead, UMON may cache only one tag for the same cache way in all sets. By using UMON, a cache miss line can be obtained, which depicts the relationship between number of cache misss and the number of ways assigned to this core. By greedily assigning each cache way to the core where the cache utility can be maximized (reducing most cache misses), this approach partitions the cache to maximize utility (minimize cache misses). This is called look ahead algorithm.

[6] pointed out that cache misses are not equivalently expensive. Parallel misses are much cheaper than isolated misses because they can be served in parallel. Instead of minimizing the number of cache misses, a better cache partitioning strategy might be minimizing MLP-based cache cost. This strategy was explored by [5] which demonstrated MLP-aware cache partitioning indeed achieves better performance. Recently, [1] studied partitioning fully-associated cache to data and it reduced the cost of look-ahead algorithm by peek-ahead algorithm.

All previous works predict future cache usage based on previous cache miss curve. This works if the cache usage pattern remains similar across different phases of the program.

When the memory usage pattern of the program changes across phases, the prediction might be inaccurate. In our project, we'd like to further improve cache partitioning based on program hints for future memory usage given by users, OS, or compiler.

[2] describes a novel memory controller design which would use adaptive scheduling based on machine learning. Using reinforcement learning, their scheduler would optimize scheduling on the fly. Controller-state action pairs are assigned reward values, and when commands are issued, the controller tries to choose the command with the greatest long term value. A learning controller brings about some great benefits to program performance. Primary amongst these is that the controller optimize for bus bandwidth, and does so on the fly. Many scheduling algorithms attempt provide the best bandwidth in the general case, though there often weak spots in their approaches which reduce memory throughput. By learning and being adaptable, the authorsâĂŹ scheme fights this weakness. In addition, the rewards system takes the core where the memory request originated from, allowing the scheme to fight against starvation as well. However the capabilities of the scheduling system are limited. While in theory, the machine learning algorithm they chose should be able to take into account infinitely many states and inputs, hardware and computation time hampers the scheduling optimization possibilities. While the authors optimized their algorithm for the resources they had available, there are various scenarios which they were not able to account for due to hardware limitations, creating weak spots in their system.

[3] proposes a thread scheduling scheme to minimize LLC con-tention based on architectural observations made by the OS at run- time. While the paper does not discuss the modification of architectural features, it demonstrates how the OS can interface with the architecture to find optimize system behavior. In this particular instance, by leveraging features of the chosen architecture, the authors were able to track cache hit/miss ratios as well as absolute counts of hits and misses on caches in the system. Using these metrics, threads running in the system were assigned weights which corresponded to their cache usage. Threads were then scheduled on the cores in such a way that these weights were spread as evenly as possible across shared caches. The benefits to this approach are easily tangible. Scheduling threads in this way ensures that memory intensive processes are given an appropriate amount of resources rather than being forced to share an unnecessary and counter productive portion of cache. The feature we are proposing could possibly extend this scheme by allowing the OS to interface with shared caches to get better partitioning.

Previous approaches assume nothing is known about the application in prior and thus can only partition the shared fairly among cores. Since those approaches rely on analyzing runtime statistics and dynamically adjusting cache partition, it may take too long for them to converge to an optimal partition. Also, since they rely on analyzing runtime application behavior, those approaches demand higher hardware cost and introduce runtime overhead. In contrast, our approach introduce lower hardware complexity and lower runtime overhead.

3. PROPOSED TECHNIQUE

An application may have multiple phases and it may have different needs for cache in different phases. Our framework allows the application to issue cache requests anytime during the course of the program. The request specifies the working set size of the application in the next phase. The shared cache takes into account the requests from all cores and partition the shared cache propotionally according to their working set size. We refer to this as *working-set-size-based partitioning.* Our technique is visualized in Figure **??**

Besides the basic partitiong approoach above, we also propose a more sophisticated partitioning strategy. Instead of providing a single number of working set size, the software hint may provide a cache miss curve at each phase. Knowing the cache miss curve allows the shared cache to analyze the marginal benefits of allocating shared cache space to each core and thus get the optimal partition that miminizes cache miss rate. We refer to this as *cache-miss-curve-based partitioning.*

Programs consist of functions. In our approach, application phases are marked by function boundaries. Thus reqeusts for shared cache are issued every time the application enters a function.

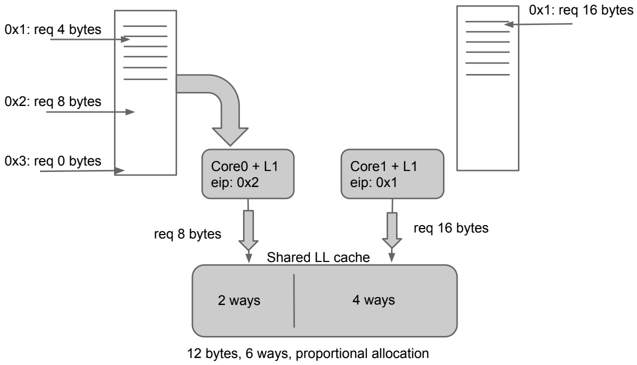
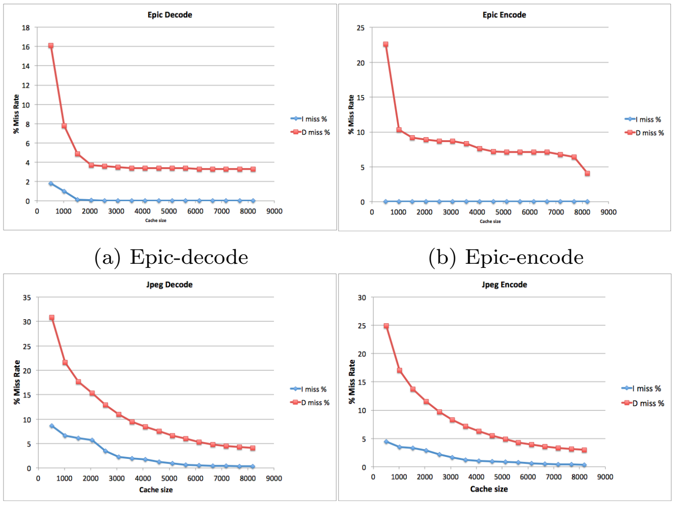


Figure 1: Visual illustration of shared cache partitioning based on software requests of working set sizes. Firstly, application 1 issues its request of 4 bytes and application 2 issues its request of 16 bytes. Core0 and Core1 forwards the requests to the shared cache. The shared cache then partitions ways propotional to the working set sizes of each application. Thus 2 ways are allocated for core0 and 4 ways are allocated for core1. When application 1 issues a new request for shared cache, the shared cache repartitions ways accordingly.

4. IMPLEMENTATION AND EVALUATION METHOLOGY

The propsed technique requires our framework to take in cache request any time and adjust cache partitioning accoridngly. However, this was too complicated to implement. Due to limitation of time, we resorted to a simplified implementation. Instead of letting the application to issue cache requests at phase changes, we only let applications to issue cache requests when the program starts. The shared cache is partitioned according to the requests and the partitioning is used for the entire course of the program.



(a) Epic-decode

(b) Epic-encode

(c) Jpeg-decode

(d) Jpeg-encode

Figure 2: Example applications' miss curves

Our technique is implemented on a simulator Multi2Sim [**?**], which is used for evaluation. We used Mediabench as our benchmarks [4].

In order to evaluate our proposed partitioning strategy, we need to predict the working set size and the cache miss curve of each function. To learn the cache miss curve of each function, we used a tool called Cachegrind [8] which simulates a cache of user-specified configuration and reports statistics about the program running with the give cache setting. It may also report statistics of each function of the program. Thus, by varying the size of shared cache, we obtained the cache miss curve of the entire program as well as that of each function.

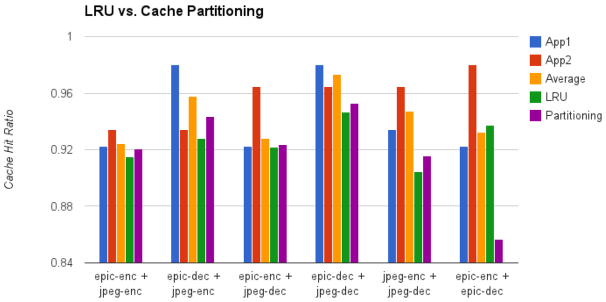
In order to evaluate the *working-set-size-based partitioning*, we need to know the working set size of each applications. We obtained that from each application's cache miss curve. We show a few applications' miss curves in Figure 2. We consider the working set size to be the point where the cache miss rate stops dropping rapidly as the cache size increases. In other words, it's where the slope of the cache miss curve decreases significantly. For some applications, such as those shown in Figure 2a and Figure 2b, this point is obvious. For some other applications, such as those shown in Figure 2c and Figure 2d, it's not obvious where the marginal benefit of increasing cache size drops. In such cases, we might make a bad decision in choosng working set sizes. To evaluate the effects of choosing suboptimal working set size, we conducted sensitivity experiments whose results are shown in Section 5.

5. RESULTS AND ANALYSIS

6. REFERENCES

[1] Nathan Beckmann and Daniel Sanchez. Jigsaw: Scalable software-defined caches. In *PACT*, pages 213–224. IEEE, 2013.

[2] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 39–50, Washington, DC, USA, 2008. IEEE Computer Society.



[3] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using os observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, 2008.

[4] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. of the 30th Int'l Symposium on Microarchitecture*, Dec. 1997.

[5] Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, and Mateo Valero. Mlp-aware dynamic cache partitioning. In *PACT*, page 418. IEEE Computer Society, 2007.

[6] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A case for mlp-aware cache replacement. *SIGARCH Comput. Archit. News*, 34(2):167–178, May 2006.

[7] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechtanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[8] Julian Seward and et. al. Cachegrind: a cache and branch-prediction profiler. `http://valgrind.org/docs/manual/cg-manual.html`. Accessed: 2013-12.

[9] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing . In *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2012.