CAPS: Cache Allocation with Partial Sharing

Yaocheng Xiang Advisor: Yingwei Luo, Xiaolin Wang, Zhenlin Wang
Peking University Michigan Technological University
{yaocheng_x, yw, wxl}@pku.edu.cn zlwang@mtu.edu

Abstract

In a multicore system, effective management of shared last level cache (LLC) has attracted significant research attention. However, almost none of the existing solutions had been implemented on a real system until Intel introduced Cache Allocation Technology (CAT) to its commodity processors recently. CAT itself implements way partitioning and thus can only allocate at a coarse granularity, which does not scale well for a large thread or program count to serve their various performance goals effectively. We overcome these limitations by deliberately and precisely sharing part of the allocations among programs and cores.

1. Introduction

Almost all existing studies of CAT are targeting quality of service (QoS) (3; 4; 7). They primarily provide high priority programs with enough and dedicated cache resources, while leaving the low-priority programs to share the rest. This simple usage of CAT does not require fine-grained control and not exploit much of its potential either.

In this paper, we propose Cache Allocation with Partial Sharing (CAPS), a framework that manages shared cache occupancy at a fine granularity. It is implemented on top of CAT, and runs on a real system. CAPS aims to achieve pre-specified performance goals, such as minimizing misses, maximizing throughput, or ensuring fairness. Our result shows that CAPS is able to support a wide range of performance targets and can scale to a large core count.

Partial-sharing allows multiple cache partitions to overlap with one another. We use an example to demonstrate how partial-sharing can outperform non-sharing and full-sharing. Two SPEC CPU2006 benchmarks, 470.lbm and 471.omnetpp, are executed on 2 separate cores, sharing a 4-way LLC, with 2816KB per way. We experiment on a real machine with all possible CAT allocation schemes and select the best non-overlapping and partially-overlapping schemes that maximize the sum of instructions per cycle (IPC). Figure 1 (left) shows the allocation layout of the three schemes. Figure 1 (right) compares IPC of the three. It is notable that the partial-overlapping scheme outperforms the other two.

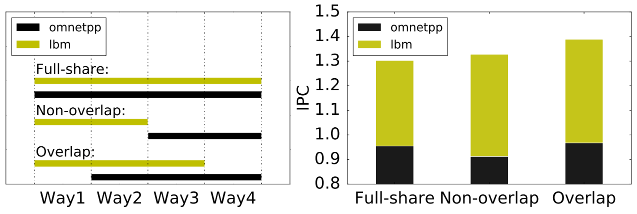

Figure 1. An illustration of full-sharing, non-overlapping and overlapping schemes

The main contributions of CAPS are: (1) a prediction model that estimates miss rates and IPCs of multiprogrammed workloads under any partially-overlapping CAT scheme, and (2) with the prediction as input, a simulated annealing algorithm that outputs a near-optimal solution given a specific performance goal.

2. Prediction Model

Performance modeling and prediction is the first step and the foundation of the optimization process. We derive a new prediction model for CAPS which can accurately estimate the miss rate and instructions per cycle (IPC) of a program when it co-executes with other programs under any CAT scheme. A CAT scheme is a collection of each core's CLOS (allocation). The inputs of the prediction model are miss ratio curve (MRC) and accesses per instruction (API) of each individual co-running program, as well as the CAT scheme that is applied to them. We adopt the average eviction time (AET) model (5) to construct MRC in linear time. Both MRC and API are profiled offline in this work.

Since a program may share part of its cache space with others, we first predict its real cache occupancy in each shared partition. Inspired by a prior study of online cache occupancy prediction for two threads (9), we establish a similar iterative equation for multi-programmed cases as follows.

SOSP'17 October 29–31, 2017, Shanghai, China

ISBN [to be supplied]. . . $15.00

Consider a cache of size C is shared by N programs. Each program currently occupies $C_1, C_2, ..., C_N$ respectively, and generates $M_1, M_2, ..., M_N$ misses during a small period of time. Let M be the sum of all misses, then the new occupancy for program i is: $C'_i = C_i + \frac{C-C_i}{C} \cdot M_i - \frac{C_i}{C} \cdot (M - M_i)$.

The equation is applied iteratively until it converges. In practice, when the aggregate change of occupancies is lower than a threshold, we consider an equilibrium has been reached. Our result shows that it can always converge to a stable condition given an initial occupancy.

During every time slice, with occupancy predicted we can utilize the MRC to predict the actual miss rate. Then Equation1 can be used to calculate misses.

$$Misses = MissRate \times API \times IPC \times TimeSlice \quad (1)$$

$$IPC = \frac{1}{CPI_{base} + API \times MissRate \times MissPenalty} \quad (2)$$

IPC is hard to estimate since so many factors can influence it. Here we use Equation 2 for a simple approximation. *MissPenalty* and CPI_{base} are obtained based on the real machine LLC miss latency and an IPC profile collected from a micro-benchmark that incurs no LLC misses, respectively. Our experiments show that this simple model can provide enough accuracy.

With the occupancy predicted, we can estimate the actual miss rate and IPC. Our evaluation with a total of 750 experiments on 4-program to 15-program workloads show that our model has an average accuracy of 90.5% in reporting miss rate and 80.1% for IPC. The time consumption of a prediction process is about 0.01 seconds, on average.

3. Optimization Algorithm

The cache optimization problem can be summarized as follows: Given a performance target, find the optimal allocation scheme. Optimization under CAT has a significantly larger search space than previous techniques, since CAT requires that each allocation is a set of contiguous cache ways and overlapped allocations are also considered. Finding the optimal solution is an NP-hard problem. Here we choose three metrics to cover different aspects of an optimization goal:

- **Average MPKI.** This metric indicates the average misses per 1000 instructions (MPKI) among the co-executing programs and is a lower-is-better metric. Let MR_i be the miss rate of program i and $APKI_i$ the number of accesses per 1000 instructions.

$$AverageMPKI = \sum(MR_i \times APKI_i)/\#program \quad (3)$$

- **Throughput.** Throughput is defined by the sum of all programs' IPCs. It is a higher-is-better metric.

$$Throughput = \sum IPC_i \quad (4)$$

- **Fair slowdown.** Fair Slowdown balances both fairness and performance. The idea is borrowed from previous research (2; 8). In our work, it is defined as the harmonic mean of per-program slowdown. Fair Slowdown is a lower-is-better metric.

$$FairSlowdown = \#program/\sum \frac{IPC_i}{SingleIPC_i} \quad (5)$$

With a performance target specified, to approximate a global optimal solution in such a large search space, we adopt the wisdom from the classic simulated annealing (SA) algorithm. SA is a probabilistic technique to approximate global optimization in a large search space (1; 6). The process of the SA algorithm can be modeled as a random walk in the search space of CAT schemes. We start from a random CAT scheme. At each step, the algorithm decides whether moving to a neighbor scheme or staying at current scheme. We call two schemes neighbors if the difference between the two is exactly one cache way. Different from a hill climbing algorithm that always goes to a better state at each step, SA can move to a worse state. The possible neighbor scheme is generated by randomly choosing a program, a side (left or right) and an operation (add or remove). After we generate a new neighbor scheme, it is fed into our prediction model to estimate the miss rate and IPC of each program and calculate the metric. If the metric improves, the algorithm will accept the new scheme. If it is worse, there is still a possibility to accept it to avoid getting stuck in a local optimum.

4. Results and Analysis

We implement three polices in CAPS for evaluation. For each metric, CAPS will output an optimized scheme. We run this partially-overlapping allocation scheme, as well as the full-sharing and the best non-overlapping scheme on the real machine and compare the corresponding metrics measured by hardware performance counters. We experiment on a total of 75 multiprogrammed workloads: 10 workloads of 4, 6, 8, 12 and 15 programs, respectively, and 25 workloads of 10 programs. For $AverageMPKI$, CAPS can reduce it by 16.96% compared to full-sharing on average, and up to 23.1%. $Throughput$ increases by 11.11% on average, and by 31.3% in the best case. $FairSlowdown$ is reduced by 8.17% on average, and up to 13.2%. Typically, CAPS yield more improvement over full-sharing with higher core count and more sensitive programs. The average time for CAPS to generate a solution is about 20 seconds.

5. Conclusion

In this paper, we propose Cache Allocation with Partial Sharing (CAPS), a software framework that can (1) *predict shared cache occupancy at a fine granularity*, (2) *support a wide range of performance goals*, (3) *scale to a large core count*, and (4) *work on a real system.*

References

[1] Aarts, E., and Korst, J. Simulated annealing and boltzmann machines.

[2] Chang, J., and Sohi, G. S. Cooperative cache partitioning for chip multiprocessors. In *ACM International Conference on Supercomputing 25th Anniversary Volume* (2014), ACM, pp. 402–412.

[3] Funaro, L., Ben-Yehuda, O. A., and Schuster, A. Ginseng: Market-driven llc allocation. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, 2016), USENIX Association, pp. 295–308.

[4] Herdrich, A., Verplanke, E., Autee, P., Illikkal, R., Gianos, C., Singhal, R., and Iyer, R. Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2016), IEEE, pp. 657–668.

[5] Hu, X., Wang, X., Zhou, L., Luo, Y., Ding, C., and Wang, Z. Kinetic modeling of data eviction in cache. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, 2016), USENIX Association, pp. 351–364.

[6] Hwang, C.-R. Simulated annealing: theory and applications. *Acta Applicandae Mathematicae 12*, 1 (1988), 108–111.

[7] Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., and Kozyrakis, C. Heracles: improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News* (2015), vol. 43, ACM, pp. 450–462.

[8] Luo, K., Gummaraju, J., and Franklin, M. Balancing thoughput and fairness in smt processors. In *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on* (2001), IEEE, pp. 164–171.

[9] West, R., Zaroo, P., Waldspurger, C. A., and Zhang, X. Online cache modeling for commodity multicore processors. *ACM SIGOPS Operating Systems Review 44*, 4 (2010), 19–29.