UPPSALA
UNIVERSITET

A Simple Statistical Cache Sharing Model for Multicores

Andreas Sandberg
Department of Information Technology
Uppsala University, Sweden
andreas.sandberg@it.uu.se

David Black-Schaffer
Department of Information Technology
Uppsala University, Sweden
david.black-schaffer@it.uu.se

Erik Hagersten
Department of Information Technology
Uppsala University, Sweden
eh@it.uu.se

ABSTRACT

The introduction of multicores has made analysis of shared resources, such as shared caches and shared DRAM bandwidth, an important topic to study. We present two simple, but accurate, cache sharing models that use high-level data that can easily be measured on existing systems. We evaluate our model using a simulated multicore processor with four cores and a shared L2 cache. Our evaluation shows that we can predict average sharing in groups of four benchmarks with an average error smaller than 0.79% for random caches and 1.34% for LRU caches.

1. INTRODUCTION

A typical modern multicore processor has a big shared last level cache (SLLC) and relatively small private caches. For example, the Intel Nehalem microarchitecture only has 288 kB (L1+L2) private cache per core, the rest of the cache is in the form of a large shared L3 cache. Since most of the cache available to an application executing on modern processors is shared, analyzing the behavior of this resource is crucial to understanding application performance.

Modeling of caches in single-processor systems has been extensively studied [10, 9, 1, 7]. Recent studies focus more on distributed caches in classical multiprocessor systems [2] and shared caches in multicore systems [4, 5, 6, 11]. Many of the cache sharing models previously require a very detailed description of an application's memory access behavior, e.g. a stack distance trace. Measuring a stack distance trace is generally prohibitively expensive, usually even more expensive than a full address trace.

Our goal is to build a practical cache sharing model that predicts the amount of cache allocated to each process running on a system with a shared cache. Using this information we can predict other performance metrics, such as how often the application will miss in the shared cache. We want to be able to apply this model to real systems with little overhead. This requires us to use high level data that can easily be measured with low overhead, such as that produced using Cache Pirating [8]. Cache Pirating allows us to measure any metric that is exposed by the underlying hardware's performance counters as a function of cache size. Eklöv *et al.* have shown that Cache Pirating can be used to produce application profiles with only 5.5% runtime overhead.

We envision several usage areas for our model. For example, imagine a heterogeneous system with a couple of fat cores with private caches and several small cores that share a cache. When starting a mix of applications on such a system, a cache sharing model could be used to predict which thread placements would be most efficient with respect to metrics such as off-chip bandwidth requirements. Of course, deciding which thread placement to use is likely to also depend on other parameters, such as available execution resources.

Our contributions are:

- We have developed a statistical cache sharing model for random caches. To the best of our knowledge, no such model has been previously documented.
- We have extended that model to LRU caches. Our extended model only requires high level data that can be easily measured.

2. MODELING CACHE SHARING

Two threads sharing a cache can intuitively be thought of as two flows of liquid filling an overflowing bucket. The flows filling the bucket correspond to fetches into the cache and the water pouring out of the bucket is the replacement stream from the cache. Assuming that the inflows are constant, the overflowing bucket will eventually reach a steady state where the concentrations of the liquids in the bucket will be proportional to their relative inflow rates. In fact, this very simple model works surprisingly well for random caches.

Applications transition between stable behaviors, also known as phases, in which the inflow of data into the cache is relatively constant. When two applications execute in such stable regions, they will quickly reach a steady state where their cache shares do not change.

An important property at steady state is that whenever new data is fetched into the cache something else has to be replaced. We define *fetch rate* as the number of fetches from memory that allocate data in the SLLC per cycle. Note that an application's fetch rate normally depends on the amount of cache it has access to. We will assume that we are at steady state and that each fetch corresponds to a replacement.

2.1 Modeling random caches

In random caches, sharing only depends on two events, fetches into the cache and replacements. At steady state, each fetch will cause a replacement. Upon a replacement, a random cache line is selected for replacement. That means that the likelihood that a cache line belonging to a specific thread is replaced is proportional to its cache share. Intuitively, a thread, n, gets a fraction of the cache ($\frac{\mathrm{c}_n}{\mathrm{C}}$) proportional to its fraction of the total fetch rate ($\frac{\mathrm{f}_n}{\mathrm{F}}$):

$$\frac{\mathrm{c}_n}{\mathrm{C}} = \frac{\mathrm{f}_n}{\mathrm{F}} \tag{1}$$

The thread increases its cache share for every fetch it makes and decreases it whenever one of its lines is replaced by another thread. We call the probability that a cache line is evicted from a specific thread's allocation $\mathrm{P}_n^{\mathrm{repl}}$. Fetches and replacements have to be balanced at steady state, which leads to the following equation:

$$\mathrm{f}_n = \mathrm{F}\,\mathrm{P}_n^{\mathrm{repl}} \tag{2}$$

The probability that a specific cache line will be evicted from a thread's allocation is a property of the replacement policy. In the case of a random cache, the likelihood of replacing something in a thread's allocation is proportional to the size of that thread's cache share. For example, if a thread owns 4 out of 16 lines in the cache, the probability of replacing a cache line belonging to that thread is $\frac{4}{16}$. In general:

$$\mathrm{P}_n^{\mathrm{repl}} = \frac{\mathrm{c}_n}{\mathrm{C}} \tag{3}$$

Inserting Equation 3 into Equation 2 yields Equation 1 above. Together with the requirement that the sum of all cache fractions equal the total shared cache size, we get an equation system that we can solve. In the simple case of two threads the equation system looks like:

$$\begin{cases} \frac{\mathrm{c}_0}{\mathrm{c}_0+\mathrm{c}_1} = \frac{\mathrm{f}_0}{\mathrm{f}_0+\mathrm{f}_1} \\ \mathrm{c}_0 + \mathrm{c}_1 = \mathrm{C} \end{cases} \tag{4}$$

The equation system above assumes that both threads actually compete for the shared cache. If all of the threads in the system have a working set size that is smaller than the total cache size, the system breaks down. We handle this case using a fallback mechanism when the fetch rates are low. In this case, we determine each threads cache allocation by finding the cache size where its fetch rate falls below a small threshold.

2.2 Modeling LRU caches

Unlike in the random replacement policy, the LRU policy depends on access history. A cache line is less likely to be replaced if it has been accessed recently. We consider an element to be a candidate for replacement if it has not been accessed recently. Intuitively, the higher the access rate, i.e. accesses per cycle, to a cache line, the less likely it is to be replaced. If we know the access rate per cache line, $\hat{\mathrm{a}}$, we could model this behavior using:[1]

$$\mathrm{P}_n^{\mathrm{cand}} = 1 - \frac{\hat{\mathrm{a}}_n}{\sum_i \hat{\mathrm{a}}_i} \tag{5}$$

Unfortunately, there is no efficient way to measure the access rate per cache line. Instead we approximate it using the average access rate, a_n, and cache size for a given thread. That is:

$$\hat{\mathrm{a}}_n \approx \frac{\mathrm{a}_n}{\mathrm{c}_n} \tag{6}$$

Using Equation 5 and Equation 6 above we can estimate the expected number of replacement candidates for a thread:

$$\mathrm{E}_n^{\mathrm{cand}} = \mathrm{c}_n - \mathrm{c}_n \frac{\frac{\mathrm{a}_n}{\mathrm{c}_n}}{\sum_i \frac{\mathrm{a}_i}{\mathrm{c}_i}} \tag{7}$$

[1]Assuming that accesses are exponentially distributed, this is the probability that the next cache line to be accessed is *not* from thread n.

Cache line size	64 B
L1 latency	3 cycles
L1 associativity	16
L1 size	64 kB
L2 latency	30 cycles
L2 associativity	16
L2 size	8 MB
Memory latency	200 cycles

Table 1: M5 Simulator parameters

We assume that all of the potential replacement candidates are equally likely for replacement. This is effectively the same as doing random replacement on a subset of the cache. Therefore, if we substitute the cache fractions with the expected number of replacement candidates, we can simply use Equation 3. This leads to the following equation for LRU caches:

$$\mathrm{P}_n^{\mathrm{repl}} = \frac{\mathrm{E}_n^{\mathrm{cand}}}{\sum_i \mathrm{E}_i^{\mathrm{cand}}} \tag{8}$$

To find the cache shares, we solve the same equation system as in random replacement, but use the replacement probability for LRU instead of random.

3. MODEL EVALUATION

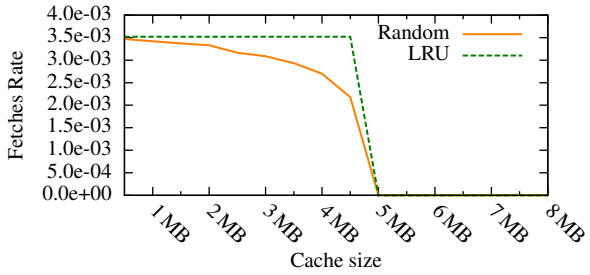
To evaluate the quality of the model for both 2 and 4 threads, we simulated a simple in-order processor with four cores using M5 [3] with the parameters in Table 1. We used the classical M5 memory system, which implements a snooping MOESI protocol, i.e. all L1 caches are connected to a shared bus linking them to the L2 cache, and does not enforce inclusion between cache levels. We measured fetch rates and access rates as a function of cache size by changing the L2 cache size in steps of 512 kB up to 8 MB.[2] We used linear interpolation of the measured values when solving the equation systems in the model. All the benchmarks were allowed to execute for 2 billion instructions.

Since our analysis assumes steady state, we sliced the execution into small windows. The benchmarks executed for 200 million cycles in each window. To avoid having to deal with window alignment, we used alignment information from the reference runs when running the model.

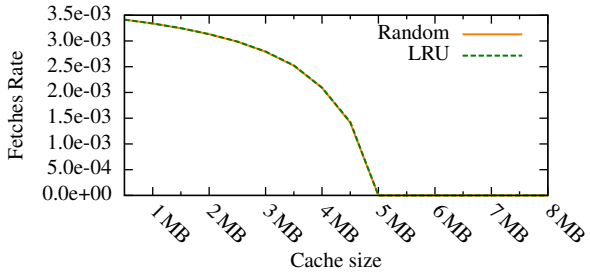
We selected benchmarks from SPEC2006 based on their fetch rate behavior in the SLLC. We generally wanted to analyze applications with a high fetch rate or a fetch rate that changes significantly when the amount of cache allocated in the SLLC is restricted. These are generally the applications that are affected by SLLC contention and are both harder to analyze and more interesting.

In addition to the SPEC benchmarks, we included two classes of microbenchmarks, their fetch rate behavior is shown in Figure 1. The first class, *block*, accesses its data set sequentially over and over again. As seen in Figure 1a, this access behavior causes the fetch rate curve to drop sharply for LRU caches when the cache size is larger than the data set size. In random caches, the fetch rate gradually decreases as the cache size approaches the data set size.

[2]On a real system, the same data would have been provided measured using Cache Pirating



(a) Block, 5 MB data set



(b) Random, 5 MB data set

Figure 1: SLLC behavior of the block_5M and random_5M microbenchmarks. The block microbenchmark accesses a 5 MB array sequentially, while the random microbenchmark accesses the array randomly.

The *random* microbenchmark class accesses its data set randomly. As seen in Figure 1a, this causes the fetch rate to decrease with cache size. An interesting observation is that both replacement policies behave the same in this case. Since the access pattern is completely random, all cache lines are equally likely to be reused next, which effectively makes this microbenchmark behave the same under any replacement policy.

We exhaustively ran all pairs of the following benchmarks from SPEC2006: *hmmer, lbm, leslie3d, libquantum, mcf, soplex, sphinx3, zeusmp*; and the following microbenchmarks: *block (1 MB, 3 MB, 5 MB, 7 MB), random (3 MB, 5 MB, 7 MB)*. Since the simulation time needed to simulate all possible combinations of four applications would be prohibitive, we limited our study to the groups shown in Table 2.

3.1 Random replacement

Figure 2 shows a scatter plot with predicted cache sizes and simulated cache sizes. The better a prediction is, the closer it is to the diagonal. The error when executing pairs of applications was generally very low, the average error as a fraction of cache size was 0.98% for pair runs and 0.79% for groups of four application.

3.2 LRU replacement

As seen in Figure 2, the prediction error for LRU caches was generally low. The average error was 2.92% for pairs and 1.34% for groups of four applications. The error was slightly higher for the LRU model than for the random model. In particular, the block_5M microbenchmark was troublesome (two benchmark pairs involving this microbenchmark are highlighted in Figure 2c). It turns out that some pairs involving this benchmark have multiple possible ways of sharing cache.

Some benchmarks, particularly benchmarks with sharp knees in their fetch rate curves, have a tendency to have multiple stable cache sharing behaviors. For example, consider our block_5M microbenchmark running together with another application. When the microbenchmark has access to less than 5 MB of cache, it misses on every single memory access. This causes it to run significantly slower than it would have if its data set had been in the cache. One could imagine a setup where our microbenchmark gets different amounts of cache depending on if it is allowed to warm its part of the cache before the second application is allowed to start. In such a setup, we would have two steady state solutions.

Looking at Figure 2c we immediately see two outliers, both involving the block_5M microbenchmark. Both of these outliers are in fact correct solutions that the simulator finds if we offset the applications slightly. Figure 3 shows the left and right hand side of the equation describing how block_5M shares cache with LBM and random_7M, three different solutions are clearly visible. Note that the unnaturally sharp step is the miss rate curve is an artifact of the microbenchmark and not likely to occur in real-world applications.

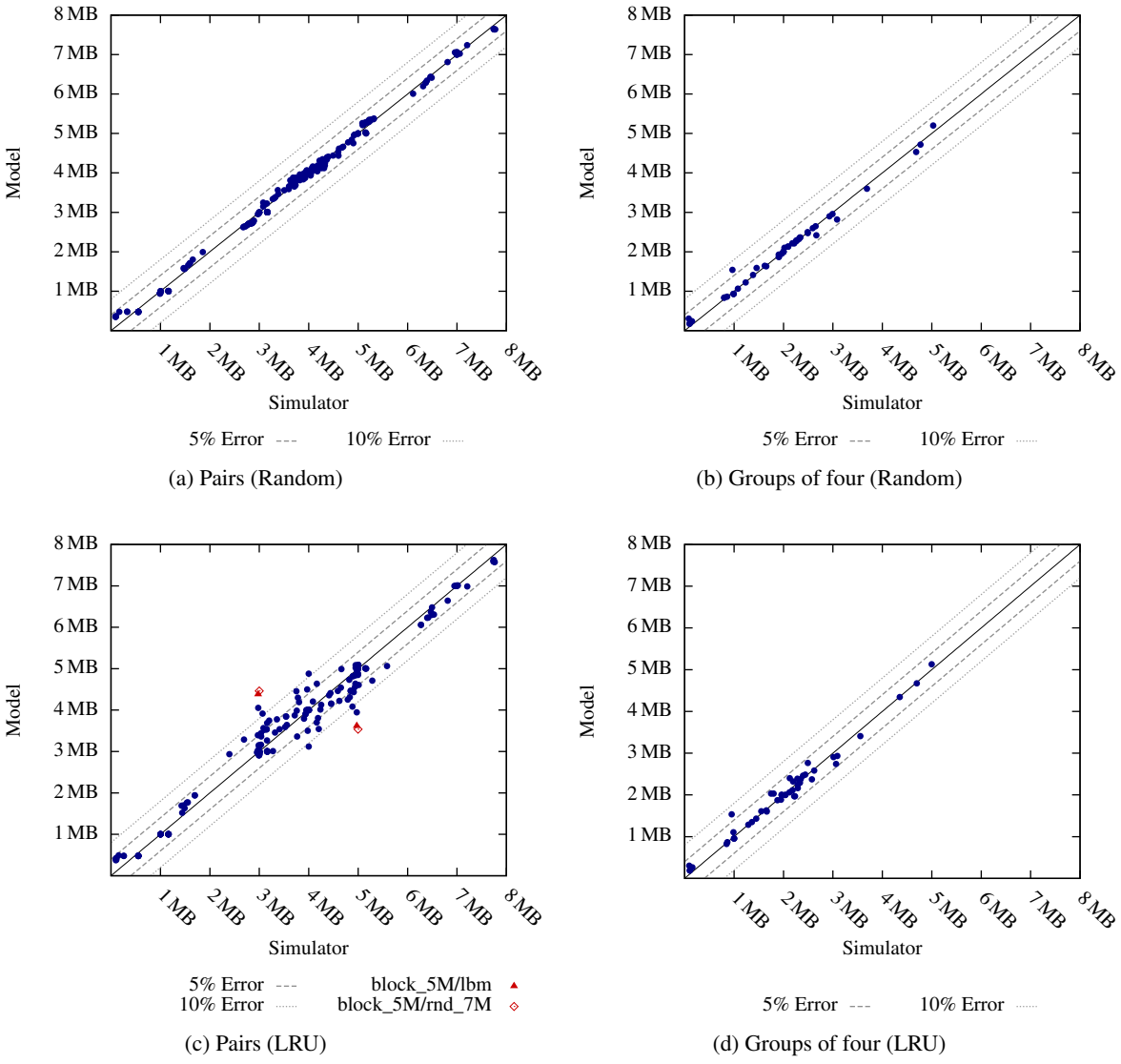
Another potential error source is sparse, interpolated, data. Looking at Figure 1a we see that the fetch rate curve for an LRU cache, has a sharp knee right at the data size. This makes it very sensitive to changes in cache size when the size is in the range of its data set size. Since our measurements of f_n and a_n are relative coarse (steps of 512 kB), interpolation may introduce some undesired behavior.

4. RELATED WORK

Several researchers have tried to predict how applications share cache. Chandra *et al.* [4] proposed a statistical cache sharing model that is based on stack distance traces (with some additional information) of the target applications. Their model predicts number of unique memory accesses performed by a thread in an interval of a given length. This information is used to prolong local stack distances with the accesses performed by the remote threads. Chandra's model assumes that all applications execute with the same CPI independent of cache size. This was later improved by Chen *et al.* [5], who first run the applications with an initial CPI guess and then rerun the model with the new CPI guess to yield the final cache size. Unfortunately, both of these models depend on stack distances, which make them hard to apply to practical problems. One could argue that the stack distances could be estimated by StatStack [7], however, both models require additional information[3] that StatStack can not predict.

Eklöv *et al.* [6] later proposed StatCC, which uses a combination of StatStack and a first order CPI model to merge reuse distance histograms and predict the behavior of the application mix. The input data to their model is a sparse memory access sample, identical to the data used by StatStack. Using a reuse distance sample makes their model practical since the reuse distance sample can be acquired with relatively low overhead (the authors claim 40% runtime overhead) and is mostly platform independent. However, their method requires a CPI model of the target system. Our approach does not rely on such a model, instead we measure how cache size affects CPI.

[3]Chandra's model uses different stack distance distributions for each reuse distance, while StatStack simply assumes every reuse distance to correspond to exactly one stack distance.



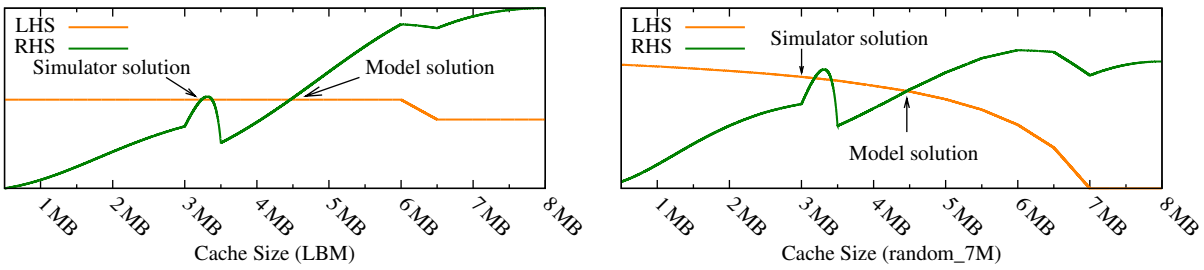
(a) Pairs (Random)

(b) Groups of four (Random)

(c) Pairs (LRU)

(d) Groups of four (LRU)

Figure 2: Scatter plots showing predicted and simulated cache size for benchmarks when running in groups of two (a, c) and groups of four (b, d) with random replacement (a, b) and LRU (c, d). The two outliers when modeling pairs of benchmarks with LRU replacement are a result of the simulator choosing between multiple stable behaviors and settling for a different cache sharing than the model.



(a) block_5M & LBM

(b) block_5M & random_7M

Figure 3: Right-hand side and left-hand side of the LRU model equation describing sharing between two different benchmark pairs. Three solutions are clearly visible, two around 3 MB and one at 4.5 MB. The solver in our model implementation found the solution at 4.5 MB, while the simulator found the solution at 3 MB.

Mix	App 0	App 1	App 2	App 3
quad0	block_1M	block_1M	block_1M	random_7M
quad1	bzip2	bwaves	astar	hmmer
quad2	bzip2	leslie3d	milc	mcf
quad3	gamess	bwaves	mcf	libquantum
quad4	lbm	libquantum	leslie3d	zeusmp
quad5	lbm	soplex	zeusmp	leslie3d
quad6	libquantum	soplex	zeusmp	leslie3d
quad7	random_3M	random_3M	random_3M	random_3M
quad8	random_5M	random_5M	random_3M	random_3M
quad9	sphinx3	mcf	bzip2	astar
quad10	sphinx3	soplex	mcf	gamess
quad11	zeusmp	bwaves	mcf	gamess

Table 2: Mixes of four applications

5. FUTURE WORK

We are currently working on adapting the system for use on real hardware using Cache Pirating data. We plan to extend these studies to evaluate how bandwidth usage is affected by cache sharing and how cache sharing is affected by limited bandwidth.

As shown in this paper, applications sharing a cache may have multiple stable working points. We did not analyze the stability of those points, we just showed their existence. It is not unreasonable to assume that all stable solutions are as likely to appear, in fact, some are probably more likely than others. Such a quality measure will most likely be important when moving to real hardware since random noise, such as interrupts and preemptions, is likely to push the system away from the less stable solutions.

Another exciting future direction is to analyze the time dependent behavior of cache sharing. Initial time dependent analysis of our simulation data suggests that an application's cache share changes significantly over time. We will likely need phase information, or similar, to enable more efficient study of different interleavings.

6. ACKNOWLEDGMENTS

The simulations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX). This work was financed by the CoDeR-MP project and UPMARC research center.

7. REFERENCES

[1] E. Berg and E. Hagersten. StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on - ISPASS*, pages 20–27, 2004.

[2] E. Berg, H. Zeffer, and E. Hagersten. A Statistical Multiprocessor Cache Model. In *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*, pages 89–99, 2006.

[3] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems,. *IEEE Micro*, 26(4):52–60, Aug. 2006.

[4] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 340–351, 2005.

[5] X. E. Chen and T. M. Aamodt. A first-order fine-grained multithreaded throughput model. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 329–340, Raleigh, NC, USA, 2009.

[6] D. Eklöv, D. Black-Schaffer, and E. Hagersten. Fast Modeling of Cache Contention in Multicore Systems. In *In Proceedings of the the 6th International Conference on High Performance and Embedded Architecture and Compilation (HiPEAC)*, January 2011.

[7] D. Eklöv and E. Hagersten. StatStack: Efficient Modeling of LRU Caches. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2010)*, New York, New York, USA, Mar. 2010.

[8] D. Eklöv, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Cache Pirating: Measuring the Curse of the Shared Cache. In *Parallel Processing (ICPP), 2011 40th International Conference on*, Sept. 2011.

[9] J. P. Singh, H. S. Stone, and D. F. Thiebaut. A model of workloads and its use in Miss-Rate prediction for fully associative caches. *IEEE Transactions on Computers*, 41(7):811–825, July 1992.

[10] D. Thiebaut and H. S. Stone. Footprints in the cache. *ACM Transactions on Compututer Systsems*, 5(4):305–329, Oct. 1987.

[11] X. Xiang, B. Bao, T. Bai, C. Ding, and T. Chilimbi. All-window profiling and composable models of cache sharing. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 91–102, New York, NY, USA, 2011. ACM.