An Efficient Simulation Algorithm for Cache of Random Replacement Policy

Shuchang Zhou

Key Laboratory of Computer System and Architecture,
Institute of Computing Technology, Chinese Academy of Sciences,
100190, Beijing, China,
`zhoushuchang@ict.ac.cn`

Abstract. Cache is employed to exploit the phenomena of locality in many modern computer systems. One way of evaluating the impact of cache is to run a simulator on traces collected from realistic work load. However, for an important category of cache, namely those of random replacement policy, each round of the naïve simulation can only give one out of many possible results, therefore requiring many rounds of simulation to capture the cache behavior, like determining the hit probability of a particular cache reference. In this paper, we present an algorithm that efficiently approximates the hit probability in linear time with moderate space in a single round. Our algorithm is applicable to realistic processor cache parameters where the associativity is typically low, and extends to cache of large associativity. Experiments show that in one round, our algorithm collects information that would previously require up to dozens of rounds of simulation.

Keywords: Simulation, Cache memories, Stochastic approximation

1 Introduction

Modern computer systems depend heavily on the efficacy of cache in exploiting locality for improving performance. There are hardware cache built into processors and disk drivers, as well as software cache used in operating system kernels, proxy and file servers. One way of evaluating cache's impact is to run a cache simulator on traces collected from realistic work load. The simulator will mimic the configuration, topology and replacement policy of the simulated cache. However, for an important category of cache, namely those of random replacement policy, as the replacement policy randomly determines one among multiple candidates for eviction, the naïve simulation only constitutes a *Monte Carlo* simulation, therefore can only give one out of many possible results. In particular, in a single round, naïve simulation cannot give the hit probability of each cache reference, which is of particular interest in program analysis[3][4]. If we maintain n copies of possible cache states and simulate access sequences on these states simultaneously, then the time and space requirement of the simulation will be equal to running n copies of naïve simulation in parallel, with no gain in efficiency. In contrast, we present an algorithm that directly approximates the hit probability. Experiments show that in one round, our algorithm collects information that would previously require up to dozens of rounds of simulation.

The rest of paper is organized as follows. Section 2 gives some background for our algorithm. Section 3 describes our algorithm. Section 4 evaluates our algorithm by simulating realistic work load. Section 5 overviews related work.

2 Background

Cache maps addresses to values. In a fully-associative[1] cache under random replacement policy, when the cache receives a request for an address A, the cache will look up all its content to see if the cache line containing A is already in the cache. If the matching succeeds, the request causes a *hit* event, and the corresponding value in cache is returned to the request. Otherwise, the request causes a *miss* event and the cache will fall back to some backup mechanism to satisfy the request. In case of *miss* event, a random place from the cache is picked, regardless of whether it originally contains valid data or not.[2] The original content at the place will be *evicted* to allow storing the address A and its corresponding value just retrieved through backup mechanism. Two nearest references to the same cache line forms a *reuse window*, with references to other cache lines in between. Cache of random replacement policy[3] are found in Translation Look-aside Buffers[6], and processor cache in ARM [13] processor family.

The Monte Carlo simulation of random replacement policy faithfully models the hardware by using (pseudo) random numbers to select the target for eviction. However, the selection is path dependent. For example, assume a fully-associative cache of size two and let $< a, e >$ denote that cache contains a and e as content, with different letters representing different cache lines, and assume an access sequence of d, a. As d is not in cache, it causes a *miss* event, leaving the cache in either $< a, d >$ or $< d, e >$. Now the next access a will exhibit different behavior under the two possible cache states: for $< a, d >$ it will be a *hit* event, and for $< d, e >$ it will be a *miss* event. An observer that simulates sequence d, a multiple times will either see *miss, miss* or *miss, hit*, with equal likelihood. However, such a phenomenon can not be observed in a single round of simulation. To make the matter worse, the number of possible cache states keeps growing after each *miss* event, and in consequence requires more rounds of simulation to

[1] Hereafter we will limit our attention to fully-associative cache. In a set-associative cache, as the mapping from the address to the cache set is fixed, the selection of eviction targets in different cache sets are statistically independent. Hence a set-associative cache of size S and associativity M is equivalent to $\frac{S}{M}$ fully-associative cache operating in parallel, each of size M.

[2] Even if the replacement algorithm takes care to not evict valid data when there are free slots, as the cache is soon filled up with valid data, there will be no difference in practice.

[3] Sometimes referred to as pseudo random replacement policy due to difficulty, if not impossibility, of obtaining true randomness.

capture the cache behavior. As mentioned above, maintaining n copies of possible cache states and simulating access sequences on these states simultaneously will not improve efficiency compared to running n copies of simulation in parallel, or running n rounds of simulation in sequence.

In theory, we can represent the state of a cache under random replacement policy as a probability distribution among all possible states, and each possible address as a transition probability matrix. The simulation of cache is then reduced to multiplying the original probability distribution with a chain of matrices. However, the probability distribution has $\sum_{i=0}^{M} \binom{Z}{i}$ size, and each transition matrix has $O(M \sum_{i=0}^{M} \binom{Z}{i})$ non-zero elements, where Z is the number of distinct elements in trace, rendering this simple approach infeasible in general.[4] Instead, we present an algorithm that implicitly calculates the probability distribution.

3 Algorithm

3.1 The Foundation

Assume the trace of cache lines of an access sequence with indices in logical time to be:

$$a_0, a_1, .., a_N.$$

We represent the *miss* event of time i by an indicator random variable[1] X_i, such that $X_i = 1$ when a *miss* event happens at i and $X_i = 0$ otherwise, forming another sequence:

$$X_0, X_1, .., X_N.$$

The indicator for *hit* event of time i is just $1 - X_i$. The hit probability of a_i is $1 - E(X_i)$, where $E(x)$ is x's expectation. The expected count of *hit* event of a sequence S is just $\sum_{i \in S} (1 - E(X_i))$. Let the cache size be M. We can determine X_i in the following way. At time i, we inspect the subsequence $\{a_j | j < i, a_j = a_i\}$. If the subsequence is empty, then a_i will not be present in cache, hence X_i is 0. Otherwise, there is a *reuse window* and we assume the largest index of the subsequence, i.e. the start of the reuse window, is k. We let the number of misses since k be Z_i.

$$Z_i = \begin{cases} \sum_{l=k+1}^{i-1} X_l, & \text{if there is a reuse window} \\ \infty, & \text{otherwise.} \end{cases} \tag{1}$$

Due to linearity of expectation, we get

$$E(Z_i) = \begin{cases} \sum_{l=k+1}^{i-1} E(X_l), & \text{if there is a reuse window} \\ \infty, & \text{otherwise.} \end{cases} \tag{2}$$

Proposition 1.

$$E(X_i) = 1 - E((1 - \frac{1}{M})^{Z_i}). \tag{3}$$

[4] $\binom{n}{k}$ is the number of k-element subsets of an n-element set

Proof. We observe that a_k is definitely in cache after time k. As every *miss* between k and now will evict a_k with probability $\frac{1}{M}$, the probability of a_k still being present in cache, which is also the probability of observing a *hit* event, is $(1-\frac{1}{M})$ to the power of number of misses since (excluding) k. Then with a given Z_i, we have:

$$E(X_i|Z_i) = 1 - (1 - \frac{1}{M})^{Z_i}. \tag{4}$$

Taking the expectation over Z_i gives equation 3. □

However, as X_i are related to the preceding elements in $\{X_i\}$, hence in general $E(a^{X_i+X_j}) \neq E(a^{X_i})E(a^{X_j})$. Therefore we cannot calculate right hand side of equation 3 easily. However, we note an approximation:

Proposition 2.

$$\textit{When } M \textit{ is large, } E(X_i) \approx 1 - (1 - \frac{1}{M})^{E(Z_i)}. \tag{5}$$

Proof. Let

$$E(X_i) = 1 - E((1 - \frac{1}{M})^{Z_i}) = 1 - (1 - \frac{1}{M})^{E(Z_i)} + \delta. \tag{6}$$

When $Z_i = \infty$, $\delta = 0$ as both $(1 - \frac{1}{M})^{E(Z_i)} = 0$ and $E((1 - \frac{1}{M})^{Z_i}) = 0$.
When $Z_i < \infty$

$$\delta = (1 - \tfrac{1}{M})^{E(Z_i)} - E((1 - \tfrac{1}{M})^{Z_i}) \tag{7}$$

$$= e^{E(Z_i)\ln(1-\frac{1}{M})} - E(e^{Z_i \ln(1-\frac{1}{M})}) \tag{8}$$

$$= \textstyle\sum_{n=0}^{\infty} \frac{(E(Z_i)\ln(1-\frac{1}{M}))^n}{n!} - E(\sum_{n=0}^{\infty} \frac{(Z_i \ln(1-\frac{1}{M}))^n}{n!}) \tag{9}$$

$$= -\textstyle\sum_{n=0}^{\infty} (-1)^n \frac{(|\ln(1-\frac{1}{M})|)^n}{n!} (E(Z_i^n) - E(Z_i)^n). \tag{10}$$

We note δ is the sum of an infinite converging alternating series $\sum_{n=0}^{\infty}(-1)^n a_n$ where it can be observed that $a_0 = a_1 = 0$ as $E(Z_i^0) = E(Z_i)^0$ and $E(Z_i^1) = E(Z_i)^1$. As δ is an alternating series with coefficients $\frac{(|\ln(1-\frac{1}{M})|)^n}{n!}$ decreasing rapidly , we may expect $\delta \approx 0$ and then obtain approximation 5. □

Now we can can use equation 2 and 5 to approximate $1 - E(X_i)$, the hit probability of each cache reference.

However, we are unable to give a rigorous bound of δ and have to resort to empirical evaluation in section 4.

3.2 The Plain Algorithm

The key observation is that we can use a map m and an offset b to efficiently calculate $E(Z_i)$ from preceding $E(X_i)$. When we are at i, we want $m[x] + b$ to equal to the number of misses in the *reuse window*, i.e. since last occurrence of x to time i. By definition, $E(Z_i) = m[a_i] + b$. If x is not seen before time i, we let m map x to ∞.

We see m and b can be constructed and maintained in the following way.

- **Initialization**: $\forall x, m[x] \leftarrow \infty$, $b \leftarrow 0$.
- **Maintenance**: At time i, $b \leftarrow b + E(X_i)$, $m[a_i] \leftarrow -b$.

We can use a hash table h to implement m, and let the absence of x from h to indicate $m[x] = \infty$. As maintenance of h can be done in $O(1)$, we have obtained an algorithm that approximates $\{E(X_i)\}$ with time complexity $O(N)$.

3.3 The ϵ-approximation

The main problem with the algorithm presented above is that the hash table size will grow up to the number of distinct elements in trace. We observe from equation 5 that within an ϵ absolute error of the hit probability $1 - E(X_i)$, we can stop the backward summation of $E(X_i)$ at l when

$$(1 - \frac{1}{M})^{\sum_{i \geq l} E(X_i)} \leq \epsilon. \tag{11}$$

That is when

$$\sum_{i \geq l} E(X_i) \geq \frac{\log \epsilon}{\log(1 - \frac{1}{M})}. \tag{12}$$

Let K be $\frac{\log \epsilon}{\log(1-\frac{1}{M})}$. Intuitively, we are ignoring reuses that are under ϵ probability, which means we can prune h of an element x if $h[x] \geq K - b$ under ϵ-approximation of $1 - E(X_i)$.

In light of this observation, we can improve the space complexity using sliding-window technique. We split h into two hash tables h' and h'', and use them in round-robin fashion. Both tables are used for query. Putting an address to one table will remove its entry from another table, if there should be one.

We use a counter c to keep track of when to swap h' and h'', and record the number of swaps with d.

- **Initialization**: $c \leftarrow 0$, $d \leftarrow 0$, h' and h'' points to two empty hash tables.
- **Maintenance**: At time i, $c \leftarrow c + E(X_i)$, $b \leftarrow b + E(X_i)$, $h'[a_i] \leftarrow -b$. If $c > K$ then $c \leftarrow c - K$, clear h'', swap h' and h'', and $d \leftarrow d + 1$; otherwise continue the loop.

Proposition 3. *The ϵ-approximation is correct and works within linear time and $O(\frac{\log \epsilon}{\log(1-\frac{1}{M})}) \approx O(M \ln(\frac{1}{\epsilon}))$ space.*

Proof. Correctness: We observe that $c = b - dK$. When x is updated, it holds that $h'[x] + b = 0$. As b is monotonically increasing, it holds that $0 \leq h'[x] + b = h'[x] + c + dK$. As $c \leq K$, we have $h'[x] \geq -c - dK \geq -(d+1)K$. h'' contains elements of step $d-1$, therefore $h''[x] \geq -dK$. At the time of clearing h'', $c > K$, all values x in h'' satisfy

$$h''[x] \geq -dK = K - (dK + K) > K - (dK + c) = K - b \tag{13}$$

and can be safely pruned.

Complexity: Let s be the sum of size of h' and h'', we next show that $s \leq 2K$. At time i, if a_i is found in h' or h'', then s will not increase, but b may be incremented; if a_i is not found in either table, both s and b are incremented by 1. Thus increment of s is bounded by the increment of b when c goes from 0 to K. As h' and h'' are cleared every two steps, $s \leq 2K$. In this way we achieve space complexity of $O(K) = O(\frac{\log \epsilon}{\log(1-\frac{1}{M})})$. As the $O(K)$ cost of clearing tables is amortized among at least K cache references, the time complexity remains $O(N)$. $\square$

4 Empirical Evaluation

We implement two algorithms: the plain and the ϵ-approximation variation of our algorithm presented above, and a naïve simulation algorithm that directly simulates current cache state. We perform our experiments on traces collected by HMTT, a platform independent full-system memory trace monitoring system[5]. The traces are collected from running LINPACK[10] and CPU2000[11] benchmark.

Both our algorithm and the average of many rounds of naïve simulation give an approximation of hit probability of each reference. In figures 1 2 3 4 5, we compare the inferred probability from average of 5, 50, and 500 rounds of the naïve simulation, and the plain and ϵ-approximation variation of our algorithm. We use trace fragments of length 10^6 and use the average of 500 rounds of the naïve simulation as reference, which is depicted in the figures as square-dotted lines with absolute value of hit probability as x-value and the distribution as y-value. All the other lines depict the distribution of absolute error of hit probability from the reference. A curve decreasing faster towards right indicates better approximation precision. It can be observed that both the plain and ϵ-approximation variation of our algorithm provide approximation of precision comparable to what is obtained from average of 50 rounds of naïve simulation, and consistently outperforms average of 5 rounds. Our algorithm is applicable to realistic processor cache parameters, for example when $M = 2$, and extends to cases of large associativity. As indicated by the proof of 5, larger M leads to better approximation precision, such that our algorithm outperforms average of 50 rounds when $M \geq 8$. We also observe that choosing ϵ to be 0.01 or even 0.1 only mildly affects the precision of approximation. Experiments with other M values and benchmarks show similar results and are not shown here for lack of space. $M = 1$ case is not considered as in this case eviction targets are picked deterministically.

5 Related Work

Cache simulation has long been an important tool in studying the impact of cache on program execution[7]. Much of literature is devoted to study of cache under *Least Recently Used* policy[8]. However, techniques that are developed for

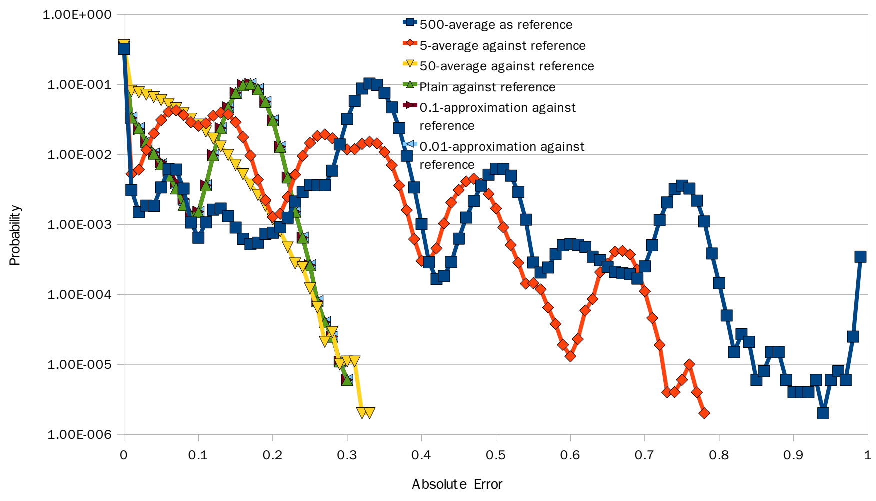

Fig. 1. Absolute error of various methods when M=2 for LINPACK

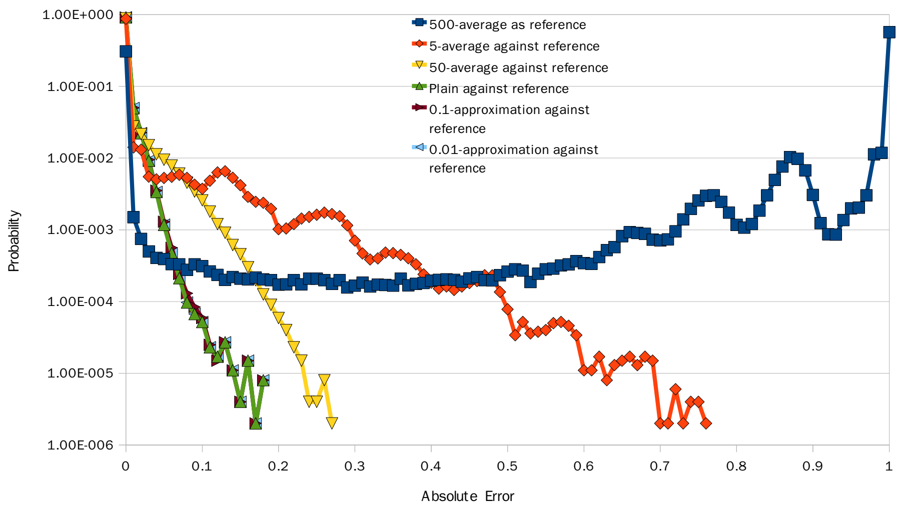

Fig. 2. Absolute error of various methods when M=8 for LINPACK

studying LRU cache, like the *reuse distance* technique, do not apply to cache of random replacement policy in general. In [12], random replacement cache is investigated under a simplifying assumption that miss rate should stay stable over a short period of execution. Under this assumption, [12] uses approximation 5 to calculate the *gross* cache hit probability. In contrast, we exploit approximation 5 without the assumption, and are able to give hit probability of *each* cache

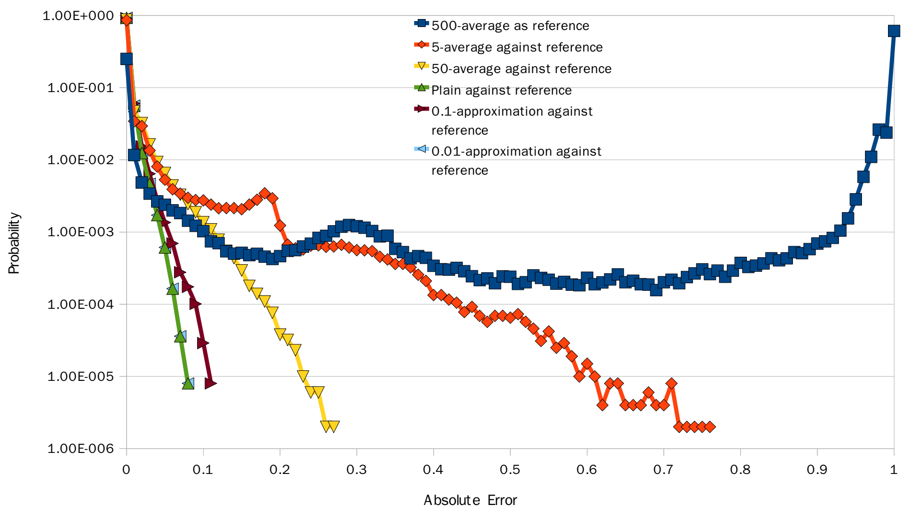

Fig. 3. Absolute error of various methods when M=64 for LINPACK

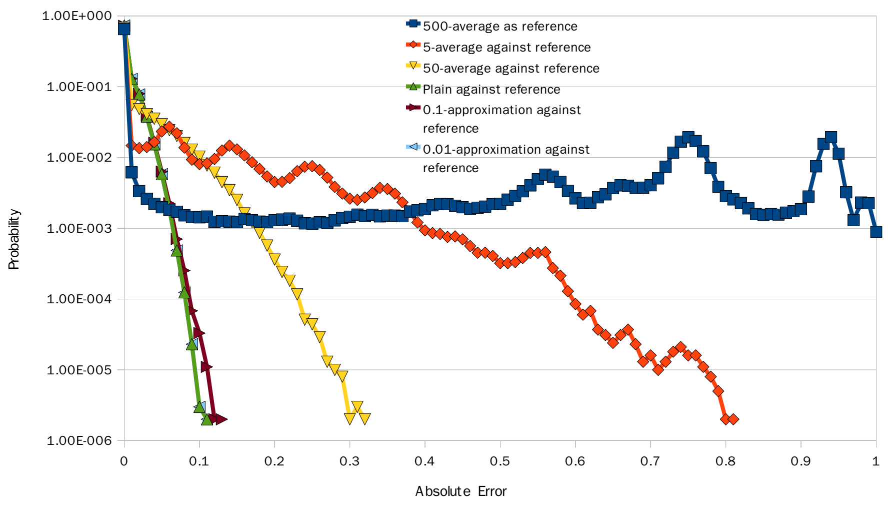

Fig. 4. Absolute error of various methods when M=4 for SWIM

reference. [15] and [16] also make other simplifying assumptions to approximate the *gross* hit probability.

Most of previous studies on cache of random replacement policy use average of a few rounds of naïve simulation as an approximation of the hit ratio of each cache reference. For example, in [9], which studies the impact of replacement policy on instruction cache, average of three rounds of simulation is used.

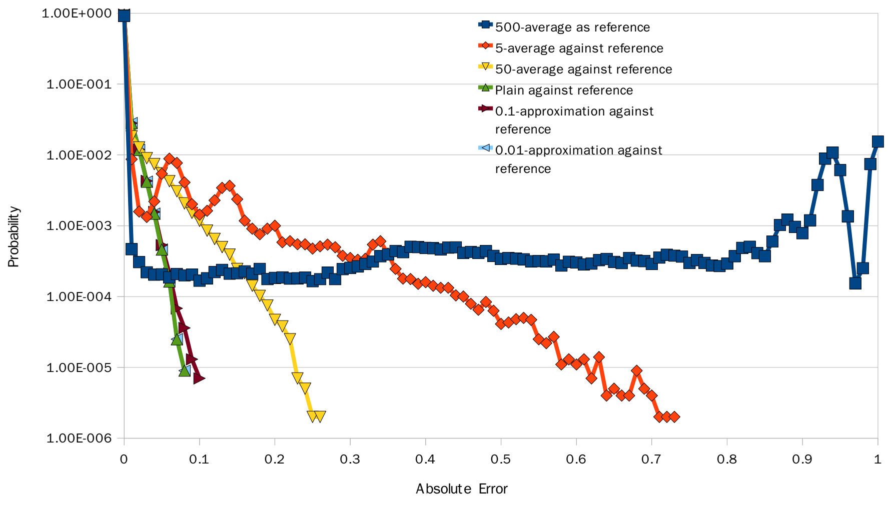

Fig. 5. Absolute error of various methods when M=16 for WUPWISE

6 Conclusion

For an important category of cache, namely those of random replacement policy, it would be necessary to perform many rounds of naïve *Monte Carlo* simulation to approximate the hit probability of each cache reference. To improve the efficiency, we devise a simulation algorithm that can approximate the probability in a single round. We start with a plain algorithm and then reduce its space complexity through ϵ-approximation. The ϵ-approximation variation of the algorithm works in linear time complexity, and has space complexity of $O(\frac{\log \epsilon}{\log(1-\frac{1}{M})}) \approx O(M \ln(\frac{1}{\epsilon}))$, where M is the size of cache. Experiments show that in one round, our algorithm collects information that would previously require up to dozens of rounds of simulation.

Acknowledgments

We thank Ma Chunhui for interesting discussions that lead to this paper. We thank Bao Yungang, the author of HMTT[5], for providing the trace files and insightful comments. We would also like to thank the anonymous reviewers for their precious remarks which help improve this paper.

References

1. Introduction to algorithms, MIT Press, Cambridge, MA, 2001

2. Fang, C., Carr, S., Önder, S., and Wang, Z.: Reuse-distance-based miss-rate prediction on a per instruction basis. In Proceedings of the 2004 Workshop on Memory System Performance (Washington, D.C., June 08 - 08, 2004). MSP '04. ACM, New York, NY, 60-68 (2004)
3. Ding, C. and Zhong, Y.: Predicting whole-program locality through reuse distance analysis. In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (San Diego, California, USA, June 09 - 11, 2003). PLDI '03. ACM, New York, NY, 245-257 (2003)
4. Beyls, K. and D'Hollander, E. H.: Reuse Distance-Based Cache Hint Selection. In Proceedings of the 8th international Euro-Par Conference on Parallel Processing (August 27 - 30, 2002). B. Monien and R. Feldmann, Eds. Lecture Notes In Computer Science, vol. 2400. Springer-Verlag, London, 265-274 (2002)
5. Bao, Y., Chen, M., Ruan, Y., Liu, L., Fan, J., Yuan, Q., Song, B., and Xu, J.: HMTT: a platform independent full-system memory trace monitoring system. In Proceedings of the 2008 ACM SIGMETRICS international Conference on Measurement and Modeling of Computer Systems (Annapolis, MD, USA, June 02 - 06, 2008). SIGMETRICS '08. ACM, New York, NY, 229-240. (2008)
6. Sweetman, D.: See MIPS Run, 2nd edition. Morgan Kaufmann Publishers. ISBN 0-12088-421-6 (2006)
7. Sugumar, R. A., Abraham, S. G.: Multi-configuration simulation algorithms for the evaluation of computer architecture designs. Technical report, University of Michigan (1993)
8. Mattson, R. L., Gecsei, J., Slutz, D., Traiger, I. L.: Evaluation techniques for storage hierarchies. IBM System Journal, 9(2):78-117 (1970)
9. Smith, J. E. and Goodman, J. R.: A study of instruction cache organizations and replacement policies. SIGARCH Comput. Archit. News 11, 3 (Jun. 1983), 132-137 (1983)
10. http://www.netlib.org/linpack/
11. http://www.spec.org
12. Berg, E. and Hagersten, E.: Fast data-locality profiling of native execution. SIGMETRICS Perform. Eval. Rev. 33, 1 (Jun. 2005), 169-180 (2005)
13. ARM Cortex-R4 processor manual, http://www.arm.com
14. Guo, F. and Solihin, Y.: An analytical model for cache replacement policy performance. In Proceedings of the Joint international Conference on Measurement and Modeling of Computer Systems (Saint Malo, France, June 26 - 30, 2006). SIGMETRICS '06/Performance '06. ACM, New York, NY, 228-239 (2006)
15. Chandra, D., Guo, F., Kim, S., and Solihin, Y.: Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In Proceedings of the 11th international Symposium on High-Performance Computer Architecture (February 12 - 16, 2005). HPCA. IEEE Computer Society, Washington, DC, 340-351 (2005)
16. Suh, G. E., Devadas, S., and Rudolph, L.: Analytical cache models with applications to cache partitioning. In Proceedings of the 15th international Conference on Supercomputing (Sorrento, Italy). ICS '01. ACM, New York, NY, 1-12 (2001)
17. Agarwal, A., Hennessy, J., and Horowitz, M.: An analytical cache model. ACM Trans. Comput. Syst. 7, 2 (May. 1989), 184-215 (1989)