

# Chameleon: A Framework for Observing, Understanding, and Imitating the Memory Behavior of Applications

Jonathan Weinberg and Allan Snaveley

University of California, San Diego  
San Diego Supercomputer Center  
La Jolla, CA 92093-0404, USA

**Abstract.** In this work, we present an integrated solution to three classic problems in the field of performance analysis: memory modeling, synthetic address trace generation, and the creation of synthetic benchmark proxies for applications. First, we describe an intuitive characterization of memory access locality that can accurately predict an application’s hit rates on arbitrary cache configurations, even when block sizes and cache depths change. We then describe the implementation of a memory tracer that can extract this characterization from applications and a software tool that can generate synthetic address traces to match. Lastly, we describe Chameleon, a fully tunable synthetic benchmark whose memory behavior can be dictated by the traces described above. We show that applications and their Chameleon counterparts display highly similar memory behavior as measured by simulated and observed cache hit rates.

## 1 Introduction

For many years, the chasm between memory and processor has focused performance research on the challenge of memory modeling. Conventional wisdom widely admits that memory performance is a function of an application’s reference locality, yet we have no workable, quantitative language that suffices to describe this property. Pursuit of a rigorous model of reference locality may seem quaintly theoretical, but is in fact among the central problems performance analysis.

System designers and procurers alike must evaluate many system configurations to achieve performance across a wide and volatile workload. Without a description of memory behavior, it is difficult to choose a representative and non-redundant evaluation workload. Even when chosen, it is difficult to anticipate the performance of this workload on various systems, which may be only partially built or not at all. Current options include acquiring prohibitively large memory address traces to drive system simulations at great expense in compute time and storage. Alternatively, synthetic traces can drive fast simulations but existing techniques have been shown to be inaccurate [13, 23, 24].

With no way to describe their applications to vendors, consumers have traditionally turned to small and manageable benchmarks that can execute on partially built systems. Unfortunately, without a description of application behavior, the relationship between benchmark and application performance is unclear.

Ideally, system designers would like to generate synthetic traces that cover a verifiably interesting space with respect to the machine’s likely workload. System procurers would like to describe their applications to vendors or produce benchmarks with clear relationships to their applications.

The Chameleon framework is a single integrated solution to each of these problems. It proposes an architecture-neutral model of reference locality and provides tools to capture the model’s parameters from applications and generate synthetic address traces that match those parameters accurately. It also includes the Chameleon benchmark, a fully-tunable synthetic benchmark whose memory behavior conforms to given model parameters.

## 2 Modeling Locality

In order to understand and imitate memory access patterns, one must begin with a model of reference locality. Locality is the principle that whenever a memory address is accessed, it or its neighbors are likely to be accessed again soon. Since the early 1970’s, many such models have been conceived and evaluated with mixed results [20, 10, 6, 1, 28, 4, 3, 23, 30, 15].

Traditionally, locality has been subdivided into *temporal* and *spatial* varieties, where the former is the tendency of an application to access recently referenced addresses and the latter its tendency to access addresses *near* recently accessed ones. With some exceptions [13, 23, 27, 30], most previous models have focused on a single dimension. To contextualize our proposal, we now describe the tradeoffs these models make and the classic abstractions on which they are based.

### 2.1 Temporal Locality

The classic measure of temporal locality is *reuse distance* or *stack distance* distributions [20, 17]. The reuse distance of some reference to address **A** is equal the number of *unique* memory addresses that have been accessed since the last access to **A**. *Reference distance* is the same measure for non-unique addresses [21]. A cumulative distribution function (CDF) of a trace’s reuse distances, sometimes called the LRU cache hit function, is a useful characterization and has been used frequently to model general locality [20, 31, 1, 5, 32, 11, 19, 8, 30].

To use only reuse distances, the modeler must freeze the spatial dimension and fix the size of the reuse unit. The 8-byte word is a popular choice, as is the width of some target cache’s block length which allows modelers to predict the application’s hit rate on a target cache [4, 16, 19, 33].

However, choosing a fixed word size fails to capture spatial locality. A fixed width, perhaps based on some target machine, makes the characterization machine-dependant and less useful for predicting hit rates on caches with different block lengths. Also, since block lengths often vary across levels of a single machine’s memory hierarchy, the model can seldom capture the application’s overall memory behavior, even on a single system. Lastly, access patterns that trigger optimizations such as multi-line prefetching may not be captured.

A continuum of reuse distributions with corresponding block sizes would make a more complete, machine-independent model.

## 2.2 Spatial Locality

A classic measure of spatial locality is *stride*, simply the distance from one memory address to another. Stride distributions have been used by numerous models to characterize locality [28, 30, 7] but necessitate the modeler to freeze the temporal dimension. Thiebaut’s fractal model, for example, measures the stride distance from the single previous access [28]. Other approaches sometimes examine a larger number of previous addresses in hopes of approximating cache sizes of interest [7, 30].

A continuum of stride distributions with corresponding history sizes would make a more complete, machine-independent model.

## 2.3 Unified Models

Several researchers have recognized the need to fuse both varieties of locality into a single measure. The idea of a three dimensional *locality surface* is owed to Grimsrud [13]. For every memory reference in a trace, he calculated the stride and reference distance to every other reference and plotted the totals on a three dimensional histogram. Sorenson later refined the idea by replacing reference distance with reuse distance [23].

Both Grimsrud in 1994 [13] and Sorenson in 2002 [24] and 2005 [23], concluded that no existing trace generation technique adequately captures locality of reference for their hybrid models. Unfortunately, neither author also proposed a methodology for translating their own characterizations into synthetic traces.

## 2.4 The Cache Surface Model

Overwhelmingly, the accuracy of locality models has been measured using cache hit rates. Working backwards from the goal, the most trivially correct model is a list of cache descriptions and corresponding hit rates. We assume only fully-associative caches with a least recently used (LRU) replacement policy.

An application’s *cache surface* is the function  $z = hit(x, y)$  where  $z$  is the hit rate of the application on a cache with  $x$  cache lines (*cache depth*), each containing  $y$ -bytes (*cache width*). The function  $hit(x, C)$  is therefore the application’s *LRU cache hit function* for caches with  $C$ -byte blocks. It is also the application’s *reuse CDF*, that is, the cumulative distribution function of its reference stream’s reuse distances, using  $C$ -byte words.

There are several shortcomings to this approach. One may be the size of the characterization, which could require hundreds of numbers to cover a relevant space. Further, acquiring this characterization may be time-consuming as fully-associative, LRU caches are notoriously expensive to simulate. Lastly, the model does not admit of any obvious techniques for generating synthetic traces.

We mitigate these concerns by observing that the points of the cache surface are not independent. Rather, the function  $hit(x, C)$  is a predictable statistical permutation

of the function  $hit(x, D)$  for all  $D > C$  as defined by the spatial locality parameter  $\alpha$ . We define  $\alpha$  as the probability of reusing some working set of size  $C$  during two consecutive references to its containing contiguous superset of size  $D$ .

To illustrate the relevance of this parameter, consider the effects that halving the cache line length would have on a memory stream. Let  $L_i$  be a reference to the  $i$ th word in cache line  $L$  and consider the following trace using cache lines of 8 words:  $A_0, B_0, C_0, B_1, C_6, A_3$ . The reuse distance of the reference to  $A_3$  is 2, because there are two unique cache line addresses separating it from the previous access to  $A_0$ . If we halve the cache line length, the trace becomes  $A_0, B_0, C_0, B_1, D_2, A_3$ , and the reuse distance of  $A_3$  is now 3.

More generally, each unique element in the interval between  $A_0$  and  $A_3$ , if reused inside the interval, could potentially increment the reuse distance of  $A_3$ . The probability of this happening is controlled by the reuse distribution describing the stream, and the probability that any given reference will reuse the same half cache line as the previous access to that cache line.

Because an  $\alpha$  value can therefore relate any two reuse CDF's, an application's cache surface can be described using one reuse CDF with word size  $D$  and a series of  $\alpha$  values, one describing its relationship to  $hit(x, C)$  for some  $C < D$ , another describing the relationship of  $hit(x, C)$  to  $hit(x, B)$  for some  $B < C$  and so forth.

For the remainder of this work, we bound cache surfaces by depths from  $2^0$  to  $2^{16}$  and widths from  $2^2$  to  $2^9$  bytes, both measured at exponential intervals. The characterization is therefore 24 numbers: 17 points to represent the reuse distance CDF with 512-byte lines and 7  $\alpha$  values to make the iterative projections.

### 3 Observing Memory Behavior

We obtain the model parameters by tracing the applications on an x86 architecture using the Pin instrumentation library [18]. However, the modeling logic is contained in an encapsulated library and not specific to the instrumentation library.

First, our approach requires that we obtain hit rates for the 17 LRU caches with 512-byte blocks. We maintain an LRU ordering among all cache lines using a single, doubly-linked list. To avoid a linear search on every memory access, we maintain a hashtable for each simulated cache that holds pointers to the list elements representing blocks resident in that cache. Each hashtable structure also maintains a pointer to its least recently used element.

On each access, we find the smallest hashtable that contains the touched block, recording a hit for it and larger caches and a miss for all smaller caches. The hashtables that missed then evict their least recently used element, add the new, most recently used element, and update their LRU pointer. Lastly, we update the doubly-linked list to maintain ordering.

Our approach simulates all 17 caches concurrently with a worst-case asymptotic running time of  $O(N*M)$  where  $N$  is the number of memory addresses simulated and  $M$  the number of caches. The average case runtime improves with locality and the overall performance is comparable to the most efficient published solutions [11].

To calculate the  $\alpha$  values, each cache block maintains its own access history as a binary tree, with the root representing the whole block, its two children the block's halves, and on until the leaves, which each represent the smallest addressable word.

On an access to some word, a function traverses the tree from root to the corresponding leaf, marking the edges along which the traversal proceeds. As it does, it increments a global counter to indicate if the edge is the same as the one chosen during the previous visit. By examining the global counters corresponding to each level of the tree, we determine how frequently two consecutive accesses to some working set reused the same half.

For finer accuracy, one may wish to describe each projection using multiple  $\alpha$  values at each width. For this purpose, the tracer's output report includes a breakdown of each  $\alpha$  value by reuse distance. There is no palpable overhead for doing this.

The tracing slowdown we have observed for extracting memory signatures using the PIN instrumentation library is up to 1000x. Approaches for reducing this figure include a faster instrumentation library such as ATOM [26] or PMaCInst [29] and trace sampling techniques [12].

However, because this study is intended to measure the potential of the proposed approach, we avoid such optimizations so to minimize the number of experiment variables. Sampling optimizations will be evaluated in future studies.

## 4 A Synthetic Trace Generator

The Chameleon framework includes a tool to convert model descriptions into synthetic traces. The trace generator actually creates a small trace *seed*, which can be converted to a full trace of arbitrary footprint and length through replication onto discrete pieces of memory and repetition.

The generator first creates a trace of block addresses by sampling reuse distances from the input CDF. For full accuracy, it continually monitors the CDF describing the trace output and compares it to the input. The generator eliminates cold cache error by growing the trace until the output is within a given maximum error. If the seed grows beyond a given maximum length, the generator adjusts the input CDF upward by the error margin and begins again, effectively preempting cold cache misses.

The generator then converts the block addresses to word addresses by choosing an index into each block using the probabilities defined by the  $\alpha$  values. The trace is prefaced with some metadata, including the original input, the size of its working set, and a "minimum number of replications" needed to flush the cache between repetitions.

### 4.1 Trace Accuracy

We create synthetic traces to mimic each of the three NAS benchmarks CG.A, SP.A, and IS.B. We use these traces to drive cache simulations of the 68 LRU caches defined in Section 3 and compare them to the hit rates derived by using the benchmarks' actual traces. On absolute average, these hit rates are within 1.9%, 1.6%, and 8.1% of the actuals for CG.A, SP.A, and IS.B respectively.

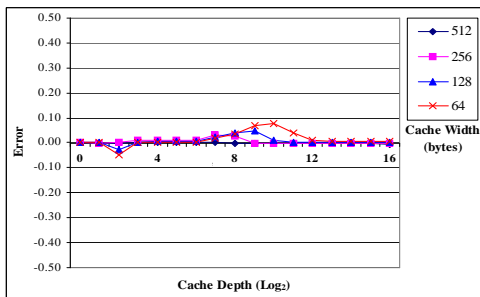


Fig. 1. Trace vs CG.A

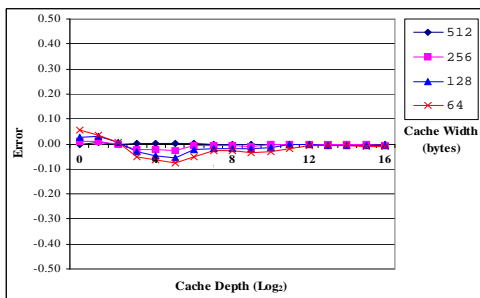


Fig. 2. Trace vs SP.A

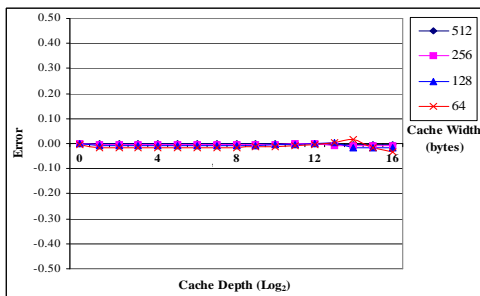


Fig. 3. Trace vs IS.B

There is no error for caches with the maximum width (512), meaning that the approach is the most accurate possible for temporal-locality based solutions. However, the error is increased as cache widths narrow.

To increase accuracy, we modify the trace generator, enabling it to leverage the reuse distance specific  $\alpha$  values reported by the tracer. This reduces the hit rate errors between the three benchmarks and their corresponding synthetic traces to only 1.0%, 1.5%, and 0.1% respectively. Figures 1, 2, and 3 graph the error rates for all 68 caches.

## 5 A Tunable Synthetic Benchmark

The Chameleon benchmark is a tunable memory benchmark based on the proposed locality model. Instead of using an index array to dictate memory behavior, the Chameleon benchmark uses a trace seed to initialize a data array and pointer chase through it:

```
for(int i = 0; i < numMemAccesses; i++)
    nextAddress=dataArray[nextAddress];
```

To do this however, the seed must never reuse any element. We modify the trace generator to accept this option as a boolean parameter. Recall that the index generation phase represents each cache line as a tree structure. To ensure that indices are never reused, used leaves and nodes without children are pruned during this phase. To compensate for the spatial error, we use an iterative aim readjustment approach similar to that used by the block generation phase to compensate for cold-cache misses.

When a tree disappears altogether, the cache block id is remapped onto a new physical cache line with a new spatial history tree. To offset the perturbation of reuse distances introduced by block id virtualization, the block generation phase uses its existing adjustment mechanism to preemptively compensate for additional cold cache misses.

To test Chameleon, we create seeds for each of the three NAS benchmarks and use the cache simulator described in Section 3 to obtain hit rates for each one. The average absolute difference in hit rates between CG.A, SP.A, and IS.B and their Chameleon trace counterparts is only 1.7%, 2.0%, and 1.1% respectively.

Lastly, we use the Pentium D820 and PAPI performance counter library to execute Chameleon and compare its hit rates to those of its target applications. The observed L1 hit rates are within 2%, 1%, and 9% of CG.A, SP.A, and IS.B respectively. The L2 rates are within 0%, 3%, and 2%. Testing on the other NAS benchmarks, including BT.A, FT.A, LU.A, and UA.A all yield similar results.

## 6 Related Work

As we described in Section 2, an incredible breadth of work has addressed locality modeling over the past 40 years. While a complete survey would merit its own publication, we touch on some important contributions here.

Temporal locality, and reuse distance in particular, has been a very popular basis for quantifying locality. Reuse distance was first studied by Mattson et. al around 1970 [20]. Multiple studies, as recently as 2007, have leveraged these ideas to create locality models and synthetic trace generators based on sampling from an application's reuse distance CDF [2, 5, 11, 15, 16]. Many works have also used reuse distance analysis for program diagnosis and compiler optimization [11, 22, 32]. The Chameleon framework distinguishes itself by eliminating error when block widths are known and modeling spatial locality to capture application behavior under various block widths; all previous approaches used fixed block widths.

In 2004, Berg proposed StatCache, a probabilistic technique for predicting miss rates on fully associative caches [3]. His model is a histogram of reference distances

with a fixed cache width. The Chameleon framework also very effectively predicts hit rates on caches of a particular width, but also does so when widths change. The ability to create synthetic traces and benchmarks also distinguishes Chameleon from this work.

Spatial locality has traditionally been quantified using strides. The most straightforward approach is the *distance model*, which captures the probability of encountering each stride distances [25]. Thiebaut later refined this idea by observing that stride distributions exhibit a fractal pattern governed by a hyperbolic probability function [28]. In recent years, the PMaC framework has focused on spatial locality but added a temporal element by including a lookback window [7].

An interesting hybrid approach that fuses spatial and temporal locality into *locality surfaces* was introduced by Grimsrud [14, 13]. Sorenson later studied a refinement of this idea extensively [23]. Neither Grimsrud nor Sorenson however, proposed techniques for converting their characterizations into synthetic traces.

Conte and Hwu described the *inter-reference temporal and spatial density functions* to quantify spatial and temporal locality separately [9]. More recently, Weinberg et al have proposed spatial and temporal locality “scores” for describing the propensity of applications to benefit from temporal and spatial cache optimizations [30].

Work on tunable synthetic benchmarks has been significantly more scarce. Wong and Morris argued mathematically that benchmarks can be synthesized to match the LRU cache hit function when block widths are known [31]. They hypothesized that multiple benchmarks could be manually stitched together through replication and repetition to match arbitrary reuse distributions. Chameleon represents the manifestation of these ideas into an automated framework.

More recently, Strohmaier and Shan developed the tunable memory benchmark Apex-Map [27]. The benchmark accepts one spatial and one temporal parameter, allowing users to compare architectures via large parameter sweeps. However, Apex-Map’s locality parameters are not observable and the use of an index array limits the benchmark’s range. Chameleon extends this idea by building on an observable model and eliminating the index array.

## 7 Conclusions

In this work, we have presented the Chameleon framework, an integrated solution for memory locality analysis. We have proposed an observable model of reference locality by which applications can be described and compared. We have implemented a memory tracer to capture the model’s parameters from applications and a trace generator that can aid system designers by quickly producing synthetic memory traces based on model parameterizations. Lastly, we described Chameleon, a fully tunable synthetic benchmark that can imitate the memory behavior of any application.

Using the NAS benchmarks and 68 cache configurations, we have shown that, on average, Chameleon’s LRU cache hit rates are within 2% of the original applications’. Using an arbitrary x86 machine with a non-trivial memory hierarchy, we also confirmed that Chameleon’s hit rates compare well with those of its target applications on a real-world, set-associative memory hierarchy with multiple block widths.

## 8 Acknowledgements

This work was supported by NSF NGS Award #0406312 entitled Performance Measurement & Modeling of Deep Hierarchy Systems. We would like to thank Jiahua He and Michael McCracken for their valuable input.

## References

1. A. Agarwal, J. Hennessy, and M. Horowitz. An analytical cache model. *ACM Trans. Comput. Syst.*, 7(2):184–215, 1989.
2. J. Archibald and J. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.
3. E. Berg and E. Hagersten. Statcache: A probabilistic approach to efficient and accurate data locality analysis. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2004)*, Austin, Texas, USA, March 2004.
4. K. Beyls and E. D’Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of PDCS’01*, pages 617–662, August 2001.
5. M. Brehob and R. Enbody. An analytical model of locality and caching. Technical Report MSU-CSE-99-31, Michigan State University, September 1999.
6. R. Bunt and J. Murphy. Measurement of Locality and the Behaviour of Programs. *The Computer Journal*, 27(3):238–245, 1984.
7. L. Carrington, N. Wolter, A. Snively, and C. B. Lee. Applying an Automated Framework to Produce Accurate Blind Performance Predictions of Full-Scale HPC Applications. In *Proceedings of the 2004 Department of Defense Users Group Conference*. IEEE Computer Society Press, 2004.
8. R. Cheng and C. Ding. Measuring temporal locality variation across program inputs. Technical Report TR 875, University of Rochester. Computer Science Department., 2005.
9. Conte and Hwu. Benchmark characterization for experimental system evaluation. In *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences*, volume 1, pages 6–18, January 1990.
10. P. J. Denning and S. C. Schwartz. Properties of the working-set model. *Commun. ACM*, 15(3):191–198, 1972.
11. C. Ding and Y. Zhong. Predicting Wholeprogram Locality Through Reuse Distance Analysis. In *PLDI 03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 245–157. ACM Press, 2003.
12. X. Gao, M. Laurenzano, B. Simon, and A. Snively. Reducing overheads for acquiring dynamic traces. In *International Symposium on Workload Characterization*, 2005.
13. K. Grimsrud, J. Archibald, R. Frost, and B. Nelson. On the accuracy of memory reference models. In *Proceedings of the 7th international conference on Computer performance evaluation : modelling techniques and tools*, pages 369–388, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.
14. K. Grimsrud, J. Archibald, R. Frost, and B. Nelson. Locality as a visualization tool. *IEEE Transactions on Computers*, 45(11):1319–1326, 1996.
15. R. Hassan, A. Harris, N. Topham, and A. Efthymiou. A hybrid markov model for accurate memory reference generation. In *Proceedings of the IAENG International Conference on Computer Science*. IAENG, 2007.
16. R. Hassan, A. Harris, N. Topham, and A. Efthymiou. Synthetic trace-driven simulation of cache memory. In *AINAW ’07: Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops*, pages 764–771, Washington, DC, USA, 2007. IEEE Computer Society.

17. B. L. Jacob, P. M. Chen, S. R. Silverman, and T. N. Mudge. An analytical model for designing memory hierarchies. volume 45, pages 1180–1194, Washington, DC, USA, 1996. IEEE Computer Society.
18. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.
19. G. Marin and J. Mellor-Crummey. Crossarchitecture Performance Predictions for Scientific Applications Using Parameterized Models. In *SIGMETRICS 2004 /PERFORMANCE 2004: Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 2–13, New York, NY, 2004. ACM Press.
20. R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM System Journal*, 9(2):78–117, 1970.
21. C. Pyo, K.-W. Lee, H.-K. Han, and G. Lee. Reference distance as a metric for data locality. In *HPC-ASIA '97: Proceedings of the High-Performance Computing on the Information Superhighway, HPC-Asia '97*, page 151, Washington, DC, USA, 1997. IEEE Computer Society.
22. X. Shen, Y. Zhong, and C. Ding. Regression-based multi-model prediction of data reuse signature. In *Proceedings of the 4th Annual Symposium of the Los Alamos Computer Science Institute*, Sante Fe, New Mexico, November 2003.
23. E. S. Sorenson. *Cache Characterization and Performance Studies Using Locality Surfaces*. PhD thesis, Brigham Young University, 2005.
24. E. S. Sorenson and J. K. Flanagan. Evaluating synthetic trace models using locality surfaces. In *Proceedings of the Fifth IEEE Annual Workshop on Workload Characterization*, pages 23–33, November 2002.
25. J. R. Spirn. *Program Behavior: Models and Measurements*. Elsevier Science Inc., New York, NY, USA, 1977.
26. A. Srivastava and A. Eustace. ATOM: A Flexible Interface for Building High Performance Program Analysis Tools. In *Proceedings of the Winter 1995 USENIX Conference*, January 1995.
27. E. Strohmaier and H. Shan. Architecture independent performance characterization and benchmarking for scientific applications. In *MASCOTS '04: Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)*, pages 467–474, Washington, DC, USA, 2004. IEEE Computer Society.
28. D. Thiebaut, J. L. Wolf, and H. S. Stone. Synthetic traces for trace-driven simulation of cache memories. *IEEE Trans. Comput.*, 41(4):388–410, 1992.
29. M. Tikir, M. Laurenzano, L. Carrington, and A. Snaveley. The PMaC binary instrumentation library for PowerPC. In *Workshop on Binary Instrumentation and Applications*, 2006.
30. J. Weinberg, M. McCracken, A. Snaveley, and E. Strohmeir. Quantifying locality in the memory access patterns of hpc applications. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Seattle, WA, November 2005.
31. W. S. Wong and R. J. T. Morris. Benchmark synthesis using the lru cache hit function. *IEEE Transactions on Computers*, 37(6):637–645, 1988.
32. Y. Zhong, C. Ding, and K. Kennedy. Reuse distance analysis for scientific programs. In *Proceedings of Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Washington DC, March 2002.
33. Y. Zhong, S. G. Dropsho, and C. Ding. Miss rate prediction across all program inputs. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 79, Washington, DC, USA, 2003. IEEE Computer Society.