Accurate Memory Signatures and Synthetic Address Traces for HPC Applications

Jonathan Weinberg
University of California, San Diego
9500 Gilman Drive
La Jolla, California 92093-0404
jonw@cs.ucsd.edu

Allan E. Snavely
San Diego Supercomputer Center
9500 Gilman Drive
La Jolla, California 92093-0505
allans@sdsc.edu

ABSTRACT

Though the performance of many scientific codes is dominated by memory behavior, our ability to describe, capture, compare, and recreate that behavior is quite limited. This inability underlies much of the complexity in the field of performance analysis: it is fundamentally difficult to relate benchmarks and applications or use realistic workloads to guide system design and procurement. An observable, reproducible, and machine-independent memory characterization is needed.

The Chameleon framework is a software suite that includes tools to capture a concise memory signature from any application and produce synthetic memory address traces that mimic that signature. By simultaneously modeling spatial and temporal locality, Chameleon produces uniquely accurate, general-purpose synthetic traces. We demonstrate that the cache hit rates generated by each synthetic trace are nearly identical to those of the application it targets on dozens of memory hierarchies representing many of today's commercial offerings.

We apply the framework to high-performance computing (HPC) by leveraging sampling techniques to capture the memory signatures of full-scale, parallel applications with only a 5x slowdown. The overall result is therefore a concise, observable, and machine-independent representation of the memory requirements of full-scale applications that can be tractably captured and accurately mimicked.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Measurement, Performance

Keywords

locality, caches, synthetic memory traces, HPC

ICS'08, June 7–12, 2008, Island of Kos, Aegean Sea, Greece.

1. INTRODUCTION

To evaluate systems, both designers and procurement decision makers must understand the workloads for which those systems are intended. Since one of the most important determinants of scientific application performance is memory behavior [16], a precise understanding and quantification of that behavior is central to the field of performance analysis. Though this problem has been widely recognized and thoroughly studied for almost 40 years, no satisfactory solution has emerged [24, 43, 41].

The consequences of this historical inability to describe, capture, and recreate memory behavior are numerous and profound. Consider, for example, the difficulty of benchmarking when we cannot meaningfully describe and relate executables. Application benchmarks [4, 10], made from whittling down full applications, are laborious to produce and easily relatable to only one other application. Synthetic benchmarks [5, 3], which perform some generic access pattern, are simple to produce and easily relatable to no other application. Instead of quantifying the behavior of applications and then observing their relationships, we are forced to awkwardly infer this information from observed runtimes.

Benchmark suites such as the classic NAS Parallel Benchmarks [10] or the more recent HPC Challenge benchmarks [33] facilitate this type of induction using a scatter-shot approach. The reasoning may still be complex [38, 50] and efforts to verify that a suite's components cover an interesting behavioral space still require quantification [52].

Benchmarking alternatives are likewise problematic. Analysts may spend months creating complex mathematical models of specific codes [35, 28] or antithetically, collect full memory address traces to drive cycle-accurate simulations. In addition to being incomparable to one another, full memory address traces are prohibitively difficult to store and replay [23]. A 4-processor trace of the simple NAS benchmark BT.A for example, requires more than a quarter of a terabyte to store and consequently relies on disk performance to replay; a 100x slowdown is realistic.

These difficulties have spurred long-running research into synthetic address trace generation techniques whose smaller, more manageable traces enable evaluations to proceed in seconds instead of days [20, 49, 27]. However, while the accuracy of a synthetic trace is inescapably bound to the memory signature underpinning it, recent studies have shown that no suitable proposal has yet come forward [24, 43, 41].

Consequently, system designers strain to understand customer workloads and design systems optimized for them. Procurement decision-makers are even more precariously sit-

uated, neither able to describe their workloads to vendors nor reliably interpret benchmarks results.

The Chameleon framework is a single integrated solution to each of these problems. Based on an accurate and observable model of memory reference locality, the framework provides tools to extract the model's parameters from any application and create synthetic address traces, or even standalone benchmarks, that adhere to that signature.

In the following Section, we provide an overview of the Chameleon framework and present results that demonstrate its unique capability to capture and mimic the core locality properties of any memory address stream. In Section 3, we extend this work to HPC by introducing two advanced sampling techniques that enable us to capture the memory signatures of full-scale, parallel applications with only a 5x slowdown. We thereby demonstrate how to extract memory signatures for large-scale applications and produce synthetic address traces that accurately match the cache hit rates of those applications on many disparate real-world cache hierarchies.

2. CHAMELEON FRAMEWORK

Central to the Chameleon framework are a model of reference locality, a memory tracer that extracts the model's parameters from applications, and a synthetic stream generator that uses these memory signatures to create a compliant synthetic address trace. In this section, we provide an overview of these components and present results demonstrating that Chameleon can mimic arbitrary address streams with great accuracy.

2.1 Modeling Locality

To understand the memory behavior of applications, one must necessarily begin with a model of *reference locality*. The principle of locality asserts that whenever a memory address is referenced, it or addresses near it, are likely to be referenced again soon. The principle has traditionally been decomposed into *spatial* and *temporal* varieties [39].

Though dozens of studies dating back to the early 1970's have focused on quantifying locality in memory behavior [36, 20, 15, 6, 56, 49, 24, 14, 13, 11, 43, 21, 52, 26], very few have focused on capturing both types. Ordinarily, a temporal locality oriented solution might focus on reuse distance analysis with fixed word sizes or cache hit rate distributions with fixed line sizes [6, 56, 14, 57, 21, 34, 18, 52]. Spatial locality oriented approaches might freeze the size of the *look-back window*, that is, the number of previous addresses examined to characterize the current address [48, 52, 17].

These models certainly have important uses, such as predicting approximate cache hit rates on single-level caches of pre-known dimensions [13, 27, 34, 58]. Any model intended to create general purpose address streams however, must address both spatial and temporal locality. Unfortunately, previous hybrid model proposals have not been accompanied by techniques for converting their parameterizations into synthetic traces [25, 40].

2.1.1 Cache Surfaces

Overwhelmingly, previous works have measured the accuracy of locality models using cache hit rates [15, 6, 30, 48, 14, 13, 58, 11, 34, 12, 52, 27, 26]. Instead of creating abstractions and then measuring their relationship to hit rates, we work backwards from the goal: the most trivially correct model is a series of cache descriptions and an application's hit rate on each.

Many variables describe a cache, but for simplicity and analogy to theory, we assume only fully-associative caches with a least recently used (LRU) replacement policy. Research has repeatedly shown that because capacity misses are most common, hit rates on such caches are highly similar to analogously sized set-associative configurations with alternate replacement policies. [13, 27, 34, 58].

We therefore describe a cache simply by its two dimensions: width and depth. We refer to the block size of a cache as its *width* and the number of blocks as its *depth*. We can thus visualize an application's locality signature as a surface $hit(d, w) = z$ with each $\{d, w, z\}$ coordinate representing a cache depth, cache width, and the corresponding hit rate for the application.

Figure 1 displays a cache surface for CG.A, one of the NAS Parallel benchmarks. The granularity at which one samples points on this surface naturally depends on the precision one requires. For the remainder of this study, we use the points shown in Figure 1, representing various cache configurations from 64 bytes to 33.5MB.

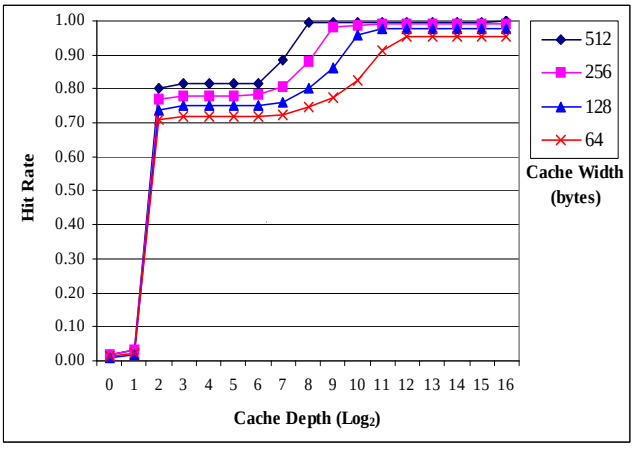

Figure 1: Cache surface of CG.A

2.1.2 Spatial Locality

For all the of the cache surface's simplicity, there may be good reason why researchers have traditionally preferred incomplete abstractions. One objection is the size of the characterization. Even if we sample the surface at log intervals, the area of interest for an application may consist of over a hundred points. Perhaps speed is the problem. Fully associative, LRU caches are notoriously expensive to simulate. Why simulate over a hundred points when each modeler can instead simulate just those he is interested in? Lastly, the model does not readily admit of any obvious techniques for generating synthetic traces.

Fortunately, instead of capturing all the points of the cache surface, we can mitigate these concerns by capturing only the statistical relationships between them. We observe that the functions $hit(d, W)$ that comprise the surface, as shown in Figure 1, are not independent of one another. Importantly, the function $hit(d, W)$ is a predictable, statistical permutation of the function $hit(d, W_1)$ for any constant $W_1 > W$. This permutation is spatial locality.

Intuitively, we understand that caches use larger block sizes to exploit spatial locality and not temporal locality. The difference in hit rates between some cache with W-byte blocks and another with W_1 byte blocks is defined by the spatial locality of the application at that point.

To illustrate, let L_i be a reference to the *ith* word in cache line L and consider the following sequence 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 in the *interval* separating it from the previous access to A_0. If we halve the cache line length, then the index 6 no longer exists and the trace becomes $A_0, B_0, C_0, B_1, D_2, A_3$, where D is some other cache line. Because C_6 becomes D_2, the reuse distance of A_3 consequently increases to 3.

When a cache width is halved, the reuse distance of each reference can remain unchanged, or at most, double. To predict the new reuse distance for each element, and therefore the new distribution, one needs three pieces of information. First is the number of unique elements in a given interval. This is equal to the element's reuse distance. Second is the number of reuses inside the interval. This is a function of the first parameter and the reuse distance distribution describing the elements in the interval. Third is the probability of each reuse in the interval reusing the same half cache line as its previous instance. Because the first two pieces of information are already described by the stream's reuse distribution, the only parameter we need obtain is the third. Let us define this probability as follows:

Spatial Locality $= \alpha(S, U) =$ *the probability of reusing some contiguous subset of addresses* S *among consecutive references to a contiguous superset* U

Because a single α value theoretically suffices to relate any two reuse distance CDF's, we can parameterize this model as a single reuse distance CDF and a series of α values that iteratively project that CDF to ones with shorter word sizes. Technically, since $hit(d, F) = 1$ for any application with footprint less than or equal to F, it is possible to characterize an application using only α values. We choose to add a reuse CDF however, both to limit the number of α parameters necessary and to mitigate error in the trace generation tools as described later.

For this work, we use $hit(d, 512)$ as the top end CDF, meaning we can model caches with block sizes up to 512-bytes. As in Figure 1, we constitute this CDF from points sampled at log intervals up to a maximum cache depth of 2^{16}. We then use six α values to capture the temporal behavior of caches with identical depths but widths of 256, 128, 64, 32, 16, and 8 bytes; the resulting characterization is therefore 23 numbers altogether.

In practice, we can increase accuracy by pooling memory references by their reuse distances and characterizing each pool's α values independently instead of doing so on an application-wide basis. Because the characterization's terseness is not important for this study, the results we present here were gathered using separate α values for each reuse distance in the CDF, meaning 119 numbers per application.

2.2 Observing Memory Behavior

To obtain the model's parameters, we instrument an application's binary with simulation logic that is invoked upon each memory reference. The simulation logic is essentially independent of the instrumentation library, making it simple to port onto various architectures. For this work, we have deployed the tracer using the Pin instrumentation library [32] for tracing on x86 architectures, and using PMaCInst [51] for tracing on the Power architecture.

2.2.1 Obtaining the Reuse CDF

Our approach first requires that we obtain hit rates for the 17 caches with maximum width (512-bytes in this case). Simulating multiple, large, fully-associative, LRU caches requires specialized software as conventional simulators are untenably slow [8, 21, 29, 46].

Our simulator uses an approach similar to that described by Kim et. al. [29] with some modifications. We maintain an LRU ordering among all cache lines using a single, doubly-linked list. To avoid a linear search through the list 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 [21, 29, 57].

2.2.2 Obtaining Alpha Values

We earlier defined α as the probability that a certain sized working set will be reused between consecutive references to a superset. In this case, the superset is initially a cache block and the subset its halves. What is the probability that two consecutive accesses to some block will hit the same half? What is the probability that two accesses to a half will hit the same quarter, etc? We stop after reaching a non-divisible working set: 4 bytes in our case, corresponding to a single 32-bit integer.

To calculate these probabilities, each simulated block maintains its own access history as a binary tree. Each leaf represents a 4-byte working set. The parent of two siblings represents the 8-byte superset and so forth until the root, which represents the entire cache block.

On an access to some 4-byte word, a function traverses the accessed block's tree from root to the corresponding leaf. At each node, it marks the edge along which the traversal proceeded. Before doing so however, it observes whether or not the edge is the same as the one chosen during the previous visit to this specific node. If so, it increments the global *yes* counter corresponding that that tree level and if not, the global *no* counter. If the node had never been visited, no counter is incremented.

The end result is a list of reuse and non-reuse counts for each tree level across all cache lines. $\alpha(256, 512)$ for example, is equal to the number of reuses reported by root nodes divided by the number of non-reuses such nodes reported.

We can thus determine each of the α values we seek, revealing how frequently two consecutive accesses to some working set reused the same half.

2.3 Generating Synthetic Traces

The Chameleon framework includes a tool to convert model descriptions into synthetic traces. It accepts as input, the unmodified output file from the tracer and creates a small file containing the new synthetic trace's *seed*. A trace seed is a minimally sized trace that can be used to generate larger traces of arbitrary length and footprint through replication and repetition [56]. These concise seeds are preferable to the full traces both because of their flexibility and their ease of handling.

To create a synthetic trace, we begin by creating a trace conforming to the CDF by sampling reuse distances from it. We create an initial linked list of all possible cache blocks, each with a unique identifier and representing a unique block. Using inverse transform sampling, we sample reuse distances from the CDF and record the identifier of the element at that index. The element then moves to the head of the list as the most recently used element.

We compensate for cold cache misses by iterative adjustment. At regular intervals, the stream generator checks the average and maximum discrepancy between the points of the input CDF and those of the sample it has generated. The trace continues to grow until it is either within some given error bounds or has exceeded a maximum length. If the latter occurs before the former, the generator scales each point of the original CDF by T/R where T is the original target and R is the achieved hit rate. The generator uses the resulting CDF as its new input and repeats the process until finding a satisfactory trace. Though multiple iterations are seldom needed for producing general-purpose synthetic traces, this mechanism becomes important when the trace must meet various constraints as with other Chameleon tools [54].

We now have a trace of *block* addresses. We must then convert these to 8-byte or 4-byte word addresses as desired. The generator iterates over the trace and for each block, chooses an offset as dictated by the series of given α probabilities. Each block maintains its own history using a tree structure identical to that used by the tracer.

The generator writes the resulting trace to a small output file and prefaces it with some metadata. That metadata includes the original input, the size of the trace's working set, the size of each word, and a "minimum number of replications" needed to flush the cache and ensure that multiple replications of the trace will always begin with the cold cache on which its accuracy is predicated.

2.4 Trace Accuracy

The resulting synthetic traces mimic those of the original applications quite accurately. Figures 2, 3, and 4 illustrate the disparity in hit rates between serial versions of three NAS benchmarks and their corresponding synthetic traces on 68 LRU cache configurations. The traces emulating CG.A, SP.A, and IS.A require only a few seconds to simulate and err on absolute average by only 1.0%, 1.5%, and 0.1% respectively.

Notice no error exists for caches specified by the model's temporal parameters (512-byte width). Chameleon's trace generation technique is therefore the most accurate possible for any temporal locality based solution. Moreover, because the *alpha* parameters capture spatial locality, the resulting synthetic traces *also* match the hit rates of caches with shorter widths. In this capacity, Chameleon is distinct from all previous modeling and trace generation proposals.

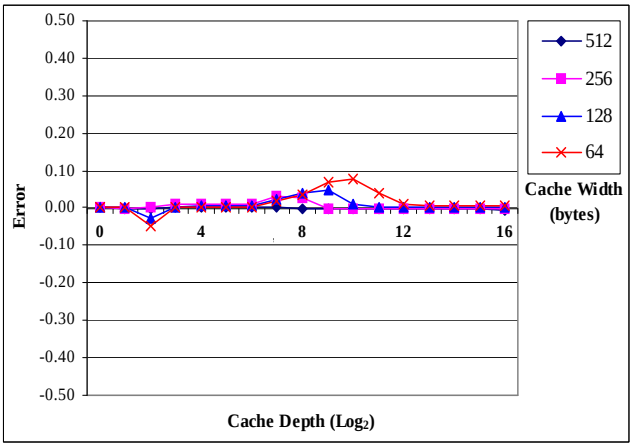

Figure 2: Trace vs CG.A

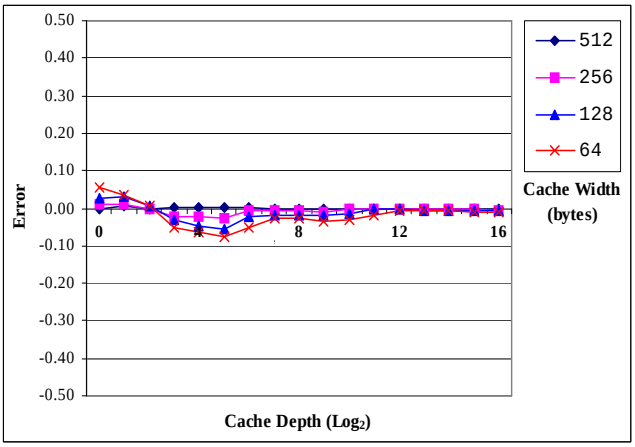

Figure 3: Trace vs SP.A

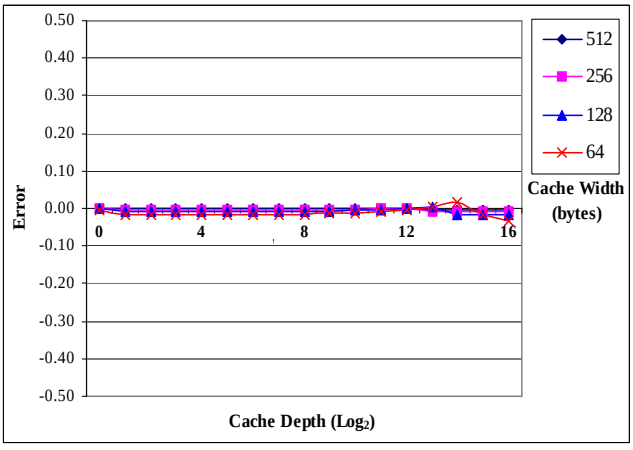

Figure 4: Trace vs IS.B

Application	Avg. L1 Error	Avg. L2 Error
IS.B	.05	.05
CG.A	.01	.02
SP.A	.03	.09

Table 1: Avg. absolute error of benchmarks on 48 simulated cache hierarchies

Application	Full Trace	Sampling (10%)
CG.A(Pin)	1160	163
SP.A(Pin)	915	112
CG.A(PMI)	276	27
SP.A(PMI)	348	36

Table 2: Slowdown caused by memory tracing

Beyond LRU caches, the traces also competently capture the behavior of applications on real-world, set-associative caches. To test this, we compare the hit rates of the actual and synthetic traces for the three NAS benchmarks on all 48 real-world and theoretical cache *hierarchies* recently evaluated by the U.S. Department of Defense's HPCMO program to support equipment procurement decisions in 2008 [1].

Table 1 lists the average absolute difference between the hit rates produced by the actual traces and those produced by the synthetics as measured by cache simulation. The results demonstrate that the two are comparable, even when multi-level caches of disparate widths and depths inhabit a single hierarchy. The disparity in error between levels stems from L1 errors propagating to L2.

3. HPC APPLICATIONS

The results in the previous section were collected on an x86 machine by applying our Pin-based tracer to small scientific benchmarks. While the results are promising, there are several additional challenges one must overcome to extend this framework to HPC applications. Most importantly, the memory tracing is prohibitively slow, sometimes several orders of magnitude over uninstrumented runtimes. This slowdown in untenable for full-scale applications that execute for several hours at a time. Additionally, this framework must be extended to and evaluated on HPC resources executing parallel applications.

In this section, we address these issues by investigating the accuracy and performance tradeoffs made by two sampling-based performance optimizations to the memory tracer. We also extend our results to a full-scale parallel application running on SDSC's 2000+ processor Power4 system.

3.1 Interval Sampling

The first optimization we evaluate is interval sampling. Rather than send every memory access through the simulation logic, the tracer will send some consecutive number through and then ignore a subsequent set. Previous research has shown that memory tracing of scientific applications can maintain good accuracy even when as few as 10% of addresses are sampled [22]. We follow this guideline and modify the tracer to iteratively simulate 1M addresses and then ignore the subsequent 9M.

3.1.1 Performance

As expected, the performance increase is around one order of magnitude. Table 3 lists the slowdown numbers for CG.A and SP.A when traced with the Pin and PMaCInst (PMI) instrumentation libraries.

The Pin traces were performed on a Pentium D820 while the PMaCInst traces were collected on SDSC's Power4-based DataStar machine [2]. The base performance discrepancy between the instrumentation libraries likely arises from their underlying mechanisms: the Pin library is a dynamic instrumentation tool that performs instrumentation at runtime while PMaCInst is a static binary re-writer that produces an instrumented binary in advance of execution. PMaCInst also shaves some runtime by assuming that all memory references load a single word instead of checking the exact size of the load.

3.1.2 Accuracy

Because we are using a locality model, the penalty we pay for this sampling is simply the cold cache misses at the start of each sampling interval. For the temporal locality portion of the model, this is not a significant penalty. Recall that this portion of the model is simply the reuse distance CDF with word sizes of 512 bytes, corresponding to the cache hit rates of 17 increasingly deep LRU caches. The maximum number of additional cold-cache misses introduced by the start of each sampling interval is at most equal to the depth of each cache. Because this number is so small with respect to the size of the sampling interval, there is no palpable inaccuracy introduced to the model's temporal elements.

The spatial parameters, represented by the α values, are somewhat more susceptible to perturbation by interval sampling. Intuitively, this is because each cache line carries with it more spatial history than temporal; while the temporal history is simply the line's reuse distance, the spatial history is an entire tree representing every recursive subset in the line. This history is lost at the end of each interval.

To determine the error that interval sampling introduces into the spatial characterization, we use the same NAS benchmarks from the previous section. We first perform five full traces of each benchmark and record the average value of each α parameter. We assume this average to be the *actual* value. We then calculate the average deviation from the actual value between the five runs. These values constitute the *natural* average deviations of each α value.

Next, we perform five traces of each benchmark using interval sampling and record the average deviation of each α value from the previously calculated actual. The error introduced by the sampling technique is then equal to this average deviation minus the natural deviation.

The maximum error among the α values characterizing each test application is only 1.2%, meaning that interval sampling does not cause a significant error in the spatial characterization of these two benchmarks. Interval sampling therefore reduces slowdown by an order of magnitude without a significant accuracy penalty. The resulting 30-40x figure is tractable for common desktop applications and we use this sampling mode as the default memory tracing technique for serial applications in the remainder of this work.

3.2 Basic Block Sampling

The interval sampling technique reduces slowdown by an

order of magnitude without a significant accuracy penalty. However, while the resulting 30-40x figure is tractable, it is still high for working with codes whose base runtimes are several hours long.

Another powerful sampling technique we may leverage is *basic block sampling*. This technique, successfully employed by the PMaC performance prediction framework [38], works as follows: A preprocessing step decomposes the binary into basic-blocks and instruments it to count the number of times each is executed during runtime. This information is then used by the instrumentor to identify the most important basic-blocks to trace (perhaps those collectively constituting 95% of all memory operations). A cap (e.g. 50K visits) is placed on the number of times that each basic block can be simulated. The simulator, for its part, must output a separate characterization for each basic block. In a post-processing step, these per-block characterizations may be combined with the execution counts collected in the preprocessing step to derive the full application's characterization.

To deploy this strategy, we leverage preexisting PMaCInst tools to decompose the binary into basic blocks and perform the preprocessing runs [51]. We then modify our simulator to isolate the locality model results by basic blocks. To simulate each memory reference, the simulator invocation now requires a block id to accompany the address. Internally, it performs an identical simulation as before, but then places the outcome of each reference's simulation into the bin specified by its block id. Note that this is not the same as performing a separate simulation for each block; the internal state of the simulator is shared among all blocks but only the *results* are separated. Consequently, this separation increases neither the simulator's runtime nor space complexity by any meaningful measure.

The output format is as before, but instead of containing a single memory signature, the output file lists one signature per basic block, preceded by that block's unique identifier. To produce a synthetic address trace, we pass this file, along with one detailing the basic block execution counts, to the stream generator. The generator combines the block signatures into a single signature according to the given weights and uses the resulting model to generate a synthetic trace.

3.2.1 Performance

To evaluate the performance gains of this technique, we replace the two NAS benchmarks with larger-scale, parallel applications. AMR [55] is an adaptive mesh refinement code that we deploy across 96 processors of DataStar. S3D [31] is a sectional 3-dimensional, high-fidelity turbulent reacting flow solver developed at Sandia National Labs. We deploy that code across 8 processors of the same machine.

We employ the interval and block sampling techniques to observe the memory address streams of the applications on each of their processors. The output is a separate locality signature for each processor involved in each run.

Table 3 lists the results of several performance experiments. The original, uninstrumented, AMR and S3D codes execute for 140 and 23 minutes respectively. The preprocessing that collects the basic block counts slows the applications by less than 2x while the simulation logic that collects our memory signatures causes a total slowdown of approximately 5x.

To determine how much of this 5x slowdown is due to the simulation logic rather than other tracing overheads, we replace the locality logic with a simulation of 15 caches from Section 2.4 and retrace. The slowdown is nearly identical. Finally, we execute a trace with both the locality *and* cache simulation logic present, observing a total slowdown of only 5.16x and 5.73x for the two applications. We can therefore conclude that the locality simulation logic itself accounts for only a small fraction of the tracing overhead.

With such low overhead for the simulation logic, it is feasible to simply insert it onto existing memory tracing activities. For example, a modeler trying to evaluate an application's performance on these 15 caches can concurrently collect it's locality signature at little extra cost. If he later wishes to determine the application's expected performance on a different cache, he could use the Chameleon framework to generate a synthetic trace and make a hit rate approximation without incurring the significant time and cost of retracing. In the next section, we investigate just how close that approximation is likely to be.

Configuration	Runtime (min)		Slowdown	
	AMR	S3D	AMR	S3D
Uninstrumented	140	22	1.00	1.00
Basic Blocks	257	26	1.84	1.18
Memory Signature	721	111	5.09	5.05
Caches	710	120	5.09	5.45
Signature+Caches	710	126	5.16	5.73

Table 3: App. slowdown due to memory tracing

3.2.2 Accuracy

To evaluate the accuracy of these techniques, we wish to compare the cache hit rates of our applications with those produced by their synthetic traces across several systems. First, we employ the interval and block sampling techniques to observe the memory address streams of each application's processors. We use each stream to drive a simulation of the 15 real-world, set-associative caches from Section 3.2.1. Mature performance modeling results from recent HPC system procurement efforts have shown that cache hit rates gathered in this way are accurate enough to predict the performance of full scientific applications within 10% [17]. They are therefore useful approximations of the actual hit rates of each processor on the 15 systems.

During this simulation, we concurrently collect each processor's locality signature and later use it to create a unique synthetic address trace. Lastly, we use the synthetic traces to drive a full simulation of the 15 caches with no sampling.

Table 4 lists the average absolute difference between the hit rates produced by the processors of each application and those produced by the corresponding synthetic traces. The results sample an evenly distributed set of processors across each application and demonstrate that the hit rates are virtually identical across the 15 caches.

Lastly, we confirm that we can accurately characterize the individual behavior of each block in the two applications. To do this, we decompose the locality signature of processor 0 into its separate blocks and produce a trace for each of the 10 most important basic blocks. These blocks constitute approximately 55% and 40% of AMR's and S3D's dynamic memory references on the processor respectively.

AMR		S3D	
Proc	Abs. Error	Proc	Abs. Error
0	.00	0	.01
10	.04	1	.00
20	.00	2	.01
30	.00	3	.01
40	.00	4	.01
50	.00	5	.02
60	.00	6	.01
70	.00	7	.01
80	.00	–	–
90	.00	–	–

Table 4: Average absolute error in cache hit rates between applications and synthetic traces over 15 cache configurations by processor

We use each synthetic trace to drive a simulation of the same 15 caches and compare the hit rates to those achieved when the simulation is driven by the actual memory stream of those blocks. The error rate for each block in the applications is less than 1%. The observed errors are likely so low because the memory access patterns of each basic block are short, stable, and less complex than the application's aggregate memory behavior.

These tests confirm that the framework can model each block accurately across multiple cache configurations. Such a capability is important for basic-block oriented performance analysis tools such as the PMaC prediction framework.

4. RELATED WORK

Over the past 40 years, a humbling breadth of work has addressed locality modeling and the creation of synthetic address traces. While a complete survey would merit a survey publication, we mention some important contributions here.

One of the earliest reference models, the *independent reference model*, was introduced in 1971 by Denning [7, 20]. It is noteworthy because unlike most subsequent models, it is not based on locality per se, but rather, on the independent probability of referencing each address.

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 [36]. 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 [9, 14, 21, 26, 27]. Many works have also used reuse distance analysis for program diagnosis and compiler optimization [21, 37, 57]. 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 [12, 11]. His model is a histogram of reference distances with a fixed cache width. The Chameleon framework also predicts hit rates on fully associative caches of a particular width very effectively. In fact, it does so with no error for any case we have tested so far. In addition though, Chameleon also captures hit rates when cache widths change. The ability to create synthetic traces and benchmarks also distinguishes Chameleon from Berg's 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 [44]. Thiebaut later refined this idea by observing that stride distributions exhibit a fractal pattern governed by a hyperbolic probability function [48, 47, 49]. More recently, the PMaC performance prediction framework had focused on spatial locality but added a temporal element by including a lookback window [38].

An interesting hybrid approach that fuses spatial and temporal locality into *locality surfaces* was introduced by Grimsrud [25, 24]. Sorenson later studied a refinement of this idea extensively [41, 42, 43]. Neither Grimsrud nor Sorenson however, proposed techniques for converting their characterizations into synthetic traces.

Contrastingly, Strohmaier and Shan developed the tunable memory benchmark Apex-MAP [45] which uses spatial and temporal parameters to generate a synthetic address stream. However, Apex-MAP's locality parameters are not observable: their value cannot be determined for a given application or address stream.

Conte and Hwu also described separate locality measures using *inter-reference temporal and spatial density functions* [19]. More recently, spatial and temporal locality "scores" have been proposed for describing the propensity of applications to benefit from temporal and spatial cache optimizations [52].

5. CONCLUSIONS

In this work, we have shown that the Chameleon framework can capture accurate memory signatures from parallel HPC applications with only a 5x slowdown. To achieve this, we used the PMaCInst binary instrumentation library to show that the interval and basic block sampling techniques can reduce memory tracing overheads by as much as two orders of magnitude without significant loss of accuracy. We have further demonstrated that the captured signatures can be used to generate synthetic address traces whose cache hit rates are highly similar to those of target applications across dozens of real-world caches.

Chameleon can thus be used to describe, compare, and mimic the memory access patterns of arbitrary HPC applications. The solution is unique in this space due to its combination of high accuracy, ability to model spatial locality, and tractable tracing time for large-scale, parallel codes.

Chameleon can be leveraged in application analysis, architecture evaluation, performance prediction, and benchmark development. It enables users to understand their workloads and describe them to vendors. It enables vendors to understand customer requirements and evaluate system designs more quickly, accurately, cheaply, and completely by using synthetic memory traces with transparent relationships to realistic workloads.

We intend to incorporate Chameleon's simulation logic into the memory tracing activities of the HPCMO's yearly performance evaluation and system procurement effort. Doing this will allow us to characterize important parallel applications on an ongoing basis while guiding system procurement in a richer fashion than has been possible in the past.

We also intend to expand work on the Chameleon bench-

mark, a tunable memory benchmark that can be calibrated using memory signatures to behave as a memory proxy for any application [54]. Doing so will help us to study the complex relationships between memory behavior, performance, instruction level parallelism, and contention among concurrently scheduled applications [53].

6. 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, Michael McCracken, and Cynthia Lee for their valuable input and Nick Wright for his assistance with S3D.

7. REFERENCES

[1] High Performance Computing Modernization Program: http://www.hpcmo.hpc.mil/.

[2] http://www.npaci.edu/DataStar/guide/home.html.

[3] RandomAccess benchmark: http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/.

[4] Spec benchmarks: http://www.spec.org/.

[5] Stream benchmark: http://www.cs.virginia.edu/stream/.

[6] A. Agarwal, J. Hennessy, and M. Horowitz. An analytical cache model. *ACM Trans. Comput. Syst.*, 7(2):184–215, 1989.

[7] A. Aho, P. Denning, and J. Ullman. Principles of optimal page replacement. *Journal of the ACM*, pages 80–93, January 1971.

[8] G. Almási, C. Caşcaval, and D. A. Padua. Calculating stack distances efficiently. In *MSP '02: Proceedings of the 2002 workshop on Memory system performance*, pages 37–43, New York, NY, USA, 2002. ACM Press.

[9] 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.

[10] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[11] 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.

[12] E. Berg and E. Hagersten. Fast data-locality profiling of native execution. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 169–180, New York, NY, USA, 2005. ACM Press.

[13] K. Beyls and E. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of PDCS'01*, pages 617–662, August 2001.

[14] M. Brehob and R. Enbody. An analytical model of locality and caching. Technical Report MSU-CSE-99-31, Michigan State University, September 1999.

[15] R. Bunt and J. Murphy. Measurement of Locality and the Behaviour of Programs. *The Computer Journal*, 27(3):238–245, 1984.

[16] L. Carrington, M. Laurenzano, A.Snavely, R. Campbell, and L. Davis. How well can simple metrics represent the performance of HPC applications? In *Supercomputing*, November 2005.

[17] L. Carrington, N. Wolter, A. Snavely, 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.

[18] R. Cheng and C. Ding. Measuring temporal locality variation across program inputs. Technical Report TR 875, University of Rochester. Computer Science Department., 2005.

[19] 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.

[20] P. J. Denning and S. C. Schwartz. Properties of the working-set model. *Commun. ACM*, 15(3):191–198, 1972.

[21] C. Ding and Y. Zhong. Predicting Wholeprogram Locality Through Reuse Distance Analysis. In *PLDI Š03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 245–157. ACM Press, 2003.

[22] X. Gao, M. Laurenzano, B. Simon, and A. Snavely. Reducing overheads for acquiring dynamic traces. In *International Symposium on Workload Characterization*, 2005.

[23] X. Gao, A. Snavely, and L. Carter. Path grammar guided trace compression and trace approximation. In *HPDC'06: Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing*, Paris, France, June 2006.

[24] 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.

[25] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson. Locality as a visualization tool. *IEEE Transactions on Computers*, 45(11):1319–1326, 1996.

[26] 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.

[27] 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.

[28] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance

and scalability modeling of a large-scale application. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 37–37, New York, NY, USA, 2001. ACM.

[29] Y. H. Kim, M. D. Hill, and D. A. Wood. Implementing stack simulation for highly-associative memories. In *SIGMETRICS '91: Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 212–213, New York, NY, USA, 1991. ACM Press.

[30] S. Laha. *Accurate low-cost methods for performance evaluation of cache memory systems*. PhD thesis, Urbana, IL, USA, 1988.

[31] S. Liu and J. Chen. The effect of product gas enrichment on the chemical response of premixed diluted methane/air flames. In *Proceedings of the Third Joint Meeting of the U.S. Sections of the Combustion Institute*, Chicago, Illinois, March 16-19 2003.

[32] 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.

[33] P. Luszczek, J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Baily, and D. Takahashi. Introduction to the HPC Challenge Benchmark Suite, April 2005. Paper LBNL-57493.

[34] 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.

[35] M. Mathis and D. J. Kerbyson. Performance modeling of mcnp on large-scale systems. In *Proceedings of the LACSI Symposium*, Los Alamos, NM, 2002. Los Alamos Computer Institute.

[36] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[37] 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 Las Alamos Computer Science Institute*, Sante Fe, New Mexico, November 2003.

[38] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A Framework for Application Performance Modeling and Prediction. In *Supercomputing Š02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–17, Los Alamitos, CA, 2002. IEEE Computer Society Press.

[39] M. Snir and J. Yu. On the theory of spatial and temporal locality. Technical Report UIUCDCS-R-2005-2611, July 2005.

[40] E. S. Sorenson. Using locality to predict cache performance. Master's thesis, Brigham Young University, 2001.

[41] E. S. Sorenson. *Cache Characterization and Performance Studies Using Locality Surfaces*. PhD thesis, Brigham Young University, 2005.

[42] E. S. Sorenson and J. K. Flanagan. Cache characterization surfaces and prediction workload miss rates. In *Proceedings of the Fourth IEEE Annual Workshop on Workload Characterization*, pages 129–139, December 2001.

[43] 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.

[44] J. R. Spirn. *Program Behavior: Models and Measurements*. Elsevier Science Inc., New York, NY, USA, 1977.

[45] E. Strohmaier and H. Shan. Apex-map: A global data access benchmark to analyze hpc systems and parallel programming paradigms. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 49, Washington, DC, USA, 2005. IEEE Computer Society.

[46] R. A. Sugumar. *Multi-configuration simulation algorithms for the evaluation of computer architecture designs*. PhD thesis, Ann Arbor, MI, USA, 1993.

[47] D. Thiebaut. From the fractal dimension of the intermiss gaps to the cache-miss ratio. *IBM J. Res. Dev.*, 32(6):796–803, 1988.

[48] D. Thiebaut. On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio. *IEEE Trans. Comput.*, 38(7):1012–1026, 1989.

[49] 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.

[50] M. Tikir, L. Carrington, E. Strohmaier, and A. Snavely. A genetic algorithms approach to modeling the performance of memory-bound computations. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 82–94, Reno, Nevada, November 10-13 2007.

[51] M. Tikir, M. Laurenzano, L. Carrington, and A. Snavely. The PMaC binary instrumentation library for PowerPC. In *Workshop on Binary Instrumentation and Applications*, 2006.

[52] J. Weinberg, M. McCracken, A. Snavely, 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.

[53] J. Weinberg and A. Snavely. Symbiotic space-sharing on sdsc's datastar system. In *The 12th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP '06)*, St. Malo, France, June 2006.

[54] J. Weinberg and A. Snavely. Chameleon: A framework for observing, understanding, and imitating the memory behavior of applications. In *PARA08: Workshop on State-of-the-Art in Scientific and Parallel Computing*, Trondheim, Norway, May, 2008.

[55] T. Wen, J. Su, P. Colella, K. Yelick, and N. Keen. An adaptive mesh refinement benchmark for modern parallel programming languages. In *Supercomputing 2007*, November 2007.

[56] W. S. Wong and R. J. T. Morris. Benchmark synthesis using the lru cache hit function. *IEEE Trans. Comput.*, 37(6):637–645, 1988.

[57] 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.

[58] 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.