UNIVERSITY OF CALIFORNIA, SAN DIEGO

The Chameleon Framework: Practical Solutions for Memory Behavior Analysis

A dissertation submitted in partial satisfaction of the requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Jonathan Weinberg

Committee in charge:

Professor Allan Snavely, Chair
Professor Lawrence Carter, Co-Chair
Professor Jeanne Ferrante
Professor Curt Schurgers
Professor Philip Gill

2008

The dissertation of Jonathan Weinberg is approved, and it is acceptable in quality and form for publication on microfilm:

_____

_____

_____

_____
Co-Chair

_____
Chair

University of California, San Diego

2008

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This work is owed in great part to my advisor and friend Allan Snavely. His thoughtful guidance, tireless support, and infectious enthusiasm have made the years of work not only bearable, but so unrelentingly enjoyable that I nearly regret having trounced him so soundly in the Mission Bay Triathlon.

I would like to thank my labmates Mike McCracken, Cynthia Lee, and Jiahua He for their friendship and professional support. Had they not so graciously endured my ceaseless rumination, I may have never managed to inflict it upon future generations.

I also owe a great thanks to Arun Jagatheesan, who provided me the opportunity to begin my graduate studies. His affable nature and kind heart are unconditional examples for us all.

I would like to thank Jeanne Ferrante and Larry Carter for their interest, thoughtful input, and service alongside Philip Gill and Curt Schurgers on my dissertation committee.

Lastly, I would be hopelessly remiss not to thank my parents, Orit and Eitan Weinberg, for having borne, raised, and loved me so completely; all complaints should be directed to them.

VITA

2001	B.A. Colby College
2005	M.S. University of California, San Diego
2008	Ph.D. University of California, San Diego

PUBLICATIONS

"Accurate Memory Signatures and Synthetic Address Traces for HPC Applications." J. Weinberg, A. Snavely. In The 22nd ACM International Conference on Supercomputing (ICS08), Island of Kos, Greece, June 7-12, 2008.

"The Chameleon Framework: Observing, Understanding, and Imitating Memory Behavior." J. Weinberg, A. Snavely. In PARA'08: Workshop on State-of-the-Art in Scientific and Parallel Computing, Trondheim, Norway, May 13-16, 2008.

"User-Guided Symbiotic Space-Sharing of Real Workloads." J. Weinberg, A. Snavely. In The 20th ACM International Conference on Supercomputing (ICS06), Cairns, Australia, June 28-July 1, 2006.

"Symbiotic Space-Sharing on SDSC's DataStar System." J. Weinberg, A. Snavely. In The 12th Workshop on Job Scheduling Strategies for Parallel Processing, Saint-Malo, France, June 27, 2006 (LNCS 4376, pp.192-209, 2007).

"When Jobs Play Nice: The Case For Symbiotic Space-Sharing." J. Weinberg, A. Snavely. In Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing (HPDC 15), Paris, France, June 19-23, 2006.

"Quantifying Locality In The Memory Access Patterns of HPC Applications." J. Weinberg, M. O. McCracken, A. Snavely, E. Strohmaier. In Supercomputing 2005, Seattle, WA, November 12-16, 2005.

"Quantifying Locality In The Memory Access Patterns of HPC Applications." J. Weinberg. Masters Thesis, University of California, San Diego, August, 26, 2005.

"Gridflow Description, Query, and Execution at SCEC using the SDSC Matrix." J. Weinberg, A. Jagatheesan, A. Ding, M. Faerman, Y. Hu. Proceedings of the 13th IEEE International Symposium on High-Performance Distributed Computing (HPDC 13), Honolulu, Hawaii, June 4-6, 2004.

ABSTRACT OF THE DISSERTATION

The Chameleon Framework:
Practical Solutions for Memory Behavior Analysis

by

Jonathan Weinberg

Doctor of Philosophy in Computer Science

University of California, San Diego, 2008

Professor Allan Snavely, Chair

Though the performance of many applications 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. A concise, observable, and machine-independent characterization of memory behavior is needed.

This dissertation presents the Chameleon framework, an integrated solution to three classic problems in the field of memory performance analysis: reference locality modeling, accurate synthetic address trace generation, and the creation of synthetic benchmark proxies for applications. The framework includes software tools to capture a concise, machine-independent memory signature from any application and produce synthetic memory address traces that mimic that signature. It also includes the Chameleon benchmark, a fully tunable synthetic executable

whose memory behavior can be dictated by these signatures.

By simultaneously modeling both spatial and temporal locality, Chameleon produces uniquely accurate, general-purpose synthetic traces. Results 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.

This work focuses on the unique challenges of high-performance computing (HPC) where workload selection, benchmarking, system procurement, performance prediction, and application analysis present important challenges. The Chameleon framework can aid in each scenario by providing a concise representation of the memory requirements of full-scale applications that can be tractably captured and accurately mimicked.

Chapter 1

Introduction

The field of performance analysis seeks to understand why applications perform as they do across various computer systems. The goals are manifold: performance analysis can help manufacturers design faster systems, customers to decide which systems are right for them, programmers to understand and optimize their applications, and system software designers to bridge the gap between software applications and hardware platforms.

For some years, the performance gap between memory and processor has grown steadily. Processors are now able to operate on data at a rate that easily dwarfs that at which the memory subsystem can deliver that data. This phenomenon, known as the von Neumann bottleneck [12], has anointed memory behavior the principal determinant of whole application performance [21].

It has been widely observed that accesses to memory are not randomly distributed, but rather exhibit some form of *reference locality*; memory references are more likely to access addresses near other recently accessed addresses. This observation has been ubiquitously exploited in today's systems to close the gap between processor and memory. Most often, caches or cache hierarchies are employed to keep the values of recently accessed memory addresses in speedy memory banks near the processor. The success of such a system naturally depends on the level of

reference locality exhibited by any given application.

Despite the importance of reference locality to the performance of today's applications, we currently have no practical, quantitative language that suffices to describe that property. Likewise, we have no practical tools to observe it nor effective techniques to reproduce it. The pursuit of a usable model of reference locality has been an open problem since Mattson's seminal work in 1970 [45], making the topic today seem quaintly theoretical, it's solution comfortably distant, and its pursuit mildly quixotic. It is however, among the central problems in the modern field of performance analysis.

As this chapter outlines, the consequences of our historical inability to describe, capture, and recreate memory behavior are numerous and profound. Much of the presented motivation is drawn from the theater of High Performance Computing (HPC), where performance analysis research has burrowed its most applicable niche. The issues and solutions presented in this work, however, can nonetheless be extended to many other computing scenarios.

1.1 Workload Selection

Consider for example, the most fundamental problem of system performance analysis: choosing an evaluation workload. All performance analysis must be relative to a specific workload, but choosing the applications that constitute that workload is difficult without a way to describe their relative memory behaviors.

Ideally, the benchmark set should not be redundant since the inclusion of multiple benchmarks with highly similar memory behavior can increase the time and cost of system evaluation. While this may seem a trivial concern when only a few small benchmarks are involved, consider the case of large-scale, HPC system procurements. Performance analysts may spend months evaluating and projecting the expected performance of the workload across dozens of proposed systems [22]. Beyond the additional time and cost involved, such unguided evaluations may even

obscure the results, giving undue weight to a specific type of memory behavior in disproportion to its representation among the machine's entire expected workload.

In addition to avoiding redundancy, a memory behavior characterization can further ensure that the evaluation workload is complete. It should include applications representing a wide sampling of the most commonly observed memory access patterns on the target machine. This is impossible to accomplish without some way to describe the memory behavior of applications. In the past, designers of benchmark suites such as the HPC Challenge benchmarks [1] have sought to quantify the memory behavior of each benchmark to demonstrate that the suite covers an interesting space with respect to all possible application behaviors [67].

A measurable characterization of memory behavior is extremely helpful for producing benchmark suites or other evaluative workloads by which to measure systems. It is essential for understanding how those benchmarks relate to each other and the rest of a machine's projected workload.

1.2 System Design and Synthetic Traces

Consider now the predicament of system designers who must tune and evaluate many cache configurations to achieve performance across a wide and volatile workload. Designers not only inherit all the workload selection problems described above, but must then also acquire memory address traces of each application in the workload to drive cycle-accurate system simulations. In addition to being generally cumbersome and inflexible, full memory traces are difficult to produce when applications are large or even impossible when the the code is classified or otherwise proprietary. Even when collected, such full traces are prohibitively difficult to store and replay [29]. A 4-processor trace of the simple NAS benchmark BT.A [13] 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. These produce smaller, more manageable traces and enable evaluations to proceed in seconds instead of days [25, 64, 33]. However, the accuracy of a synthetic trace is intrinsically bound to the memory characterization underpinning it. Because such characterizations have not been forthcoming, studies as recently as 2005 have concluded that no existing trace generation technique can adequately characterize memory behavior with sufficient accuracy [30, 56, 54].

1.3 System Procurement and Benchmarking

Let us now consider the problem of large-scale system procurement. With multiple vendors and designs to evaluate, the procurer must similarly begin by choosing a representative workload by which to compare systems. As before, this choice is made in relative darkness. Unlike the designer however, the consumer cannot normally execute his chosen workload on the target system. Customers may not have access to the systems they are considering and it is time-consuming and expensive for vendors to deploy large-scale applications on their behalf.

Worse yet, HPC systems are normally not fully built before the purchasing decisions are made. Instead, the vendor may produce a small prototype from which procurement decision-makers may acquire some benchmark results. Seldom can they execute an application at scale. In some scenarios, such as system procurement by the U.S. Department of Defense [38], applications in the evaluation workload may be classified or otherwise proprietary, making vendor deployment impossible even when the test systems are large enough.

With no way to describe their applications to vendors, consumers have traditionally turned to small and manageable benchmarks. Such benchmarks can be passed along to the vendors, who in turn, execute them on the partial systems and report the results back to their customers. Unfortunately, the underlying issue quickly resurfaces: how do the benchmark relate to the customer's applications of interest? Application benchmarks [5, 13], made from whittling down full applica-

tions, are laborious to produce and easily relatable to only one other application. Synthetic benchmarks [6, 4], which perform some generic access pattern, are simple to produce and easily relatable to no other application.

Frustratingly, consumers can neither describe their applications to vendors nor reliably interpret benchmark results. Instead of quantifying the behavior of applications and then observing their relationships, they are forced to awkwardly infer this information from observed runtimes.

Benchmark suites such as the classic NAS Parallel Benchmarks [13] or the HPC Challenge benchmarks [43] facilitate this type of induction using a scatter-shot approach. Recent years have seen researchers develop sophisticated methodologies to map the performance of benchmarks to that expected of target applications [50, 65]. Such techniques require performance modeling expertise and can be time consuming and expensive.

1.4 Application Analysis

There are many more scenarios in performance modeling, application tuning, and system tool design that require a robust characterization of memory behavior.

Consider the challenge of the software developer or algorithm designer attempting to craft a performance-tuned application. What are the effects if he partitions the data this way or shapes the stencil that way? Currently, he can gauge the effectiveness of these techniques by the resulting run time on his particular system. Could it be that his optimization made little performance improvement on his test system but resulted in large performance gains on several others? Perhaps the optimization improved memory performance markedly, but the performance of the application is not bound by that of the memory subsystem? The developer could identify these scenarios with an intuitive and observable memory characterization.

The challenge is similar to that faced by a compiler developer or user who

wants to observe the true effects of a certain optimization strategy. With no memory characterization, he is forced to infer the effects through cursory runtime or hardware counter information on one or a few systems.

Consider a performance modeler attempting to understand and predict the performance of his parallel application under various processor counts. Perhaps he can observe the application executing at several different counts and wants to produce a general projection of its behavior on many others. What information should he observe from each of the three runs in order to project the application's behavior on subsequent sizes? He requires a memory behavior characterization more descriptive than the erratic stepwise functions exhibited by run time and cache hit rate observations.

1.5 Symbiotic Space-Sharing

Understanding application behavior can also inform operating system design. The problem of job scheduling on a multi-processor system is one such example. Many of today's symmetric multi-processors share various levels of the memory hierarchy among the processors. Consequently, the runtime behavior of one executing application can affect the performance of the others. The challenge of *symbiotic space-sharing* is for a smart scheduler to execute jobs in combinations and configurations that mitigate pressure on shared resources [68, 69].

To be effective, the scheduler must learn to identify jobs that behave similarly and predict that they will have similar space-sharing impact on and sensitivity to other executing jobs. With no description of memory behavior, the scheduler cannot quantify, or even loosely qualify, the level of similarity between applications. It can only identify recurring jobs and predict their interactions with previously co-scheduled jobs. With a combinatorial number of job pairings to try and a relatively limited number of scheduling iterations with which to try them, the scheduler is unlikely to find optimal combinations quickly enough to affect

meaningful performance gains.

If it were able to capture a core description of each job's behavior, the scheduler would eventually learn to identify symbiotic job combinations even among a completely untested job set. In fact, if we could synthetically replicate memory behavior, developers could train their schedulers to identify symbiotic job combinations even before they are ever deployed into production settings.

1.6 Chameleon Overview

As we have discussed, the consequences of our inability to quantify, compare, and recreate memory behavior are ubiquitous in the field of performance analysis. System designers strain to understand customer workloads and design systems optimized for them; procurement decision-makers are even more precariously situated, neither able to describe their workloads to vendors nor reliably interpret benchmarks results. Application and system tool developers observe only nebulous measures of their success and performance analysts must often ground their models on thin or volatile data.

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. Algorithm designers, system tools developers, performance modelers, and benchmark suite authors alike require a concise description of application memory behavior.

The Chameleon framework is a single integrated solution to each of these problems. It addresses three of the classic problems in the field of performance analysis: memory behavior characterization, synthetic address trace generation, and tunable synthetic benchmarks. The framework includes a series of components and tools, each meant to facilitate solutions to the issues described above. The following subsections provide a brief description of each.

1.6.1 The Cache Surface Model

The framework is grounded in an architecture-neutral model of reference locality that is compatible with theory and inclusive of many previous proposals. We call this the *Cache Surface Model* and have designed it to meet the following requirements:

Hybrid - The model simultaneously describes both spatial and temporal locality, thereby capturing the memory address stream's cache performance across disparate cache sizes and designs without prior knowledge of a target architecture.

Observable - It is possible to extract the model's parameters by observing an application's address stream *online*; there is no need to collect and store the entire memory trace. This obviates the burdening space requirements of trace collection and storage.

Concise - The model parameterization is small, typically between 10-100 numbers, depending on the granularity that the user and application require. This allows any application's memory signature to be easily stored and communicated between interested parties.

Reproducible - Given an application's memory signature, it is relatively straightforward to generate synthetic address streams that match it.

Intuitive - The model is intuitive and simple to reason about, with clear relationships to real world caches and expected performance on various systems.

Chapter 2 provides a full description of the Cache Surface Model.

1.6.2 Tracing Tool

The Chameleon Framework includes two memory tracing tools for extracting memory signatures from applications. The first is built on top of the Pin

instrumentation library [42] and is meant for tracing serial applications on x86 architectures. The second uses the PMaCInst [66] binary re-writer for extracting memory signatures from either serial or parallel codes on the Power architecture.

The modeling logic itself is completely encapsulated and exposes a simple interface that allows developers to mount it easily on any instrumentation library of their choosing. Using several sampling-based optimizations, the framework's stock tracers can extract memory signatures even from HPC applications with as little as 5x slowdown.

Chapter 3 provides a full description of the framework's tracing tools.

1.6.3 Synthetic Trace Generation Tool

The Chameleon Framework includes a trace generation tool that can convert any memory signature into a concise *trace seed*. A trace seed is a short synthetic memory trace that can be used to generate a full trace by *replication* onto discrete sections of memory until the desired footprint is reached and *repetition* until the desired length is likewise [71].

A trace seed is not only more flexible and useful than a full-scale trace, but it can also be easily stored and communicated due to its small size. The trace generator anticipates and pre-adjusts for the errors of seed replication, so synthetic traces of any footprint size remain accurate.

The trace generator is written as a stand-alone Java application to maximize portability and enable vendors or performance modelers to generate synthetic traces from memory signatures they receive from clients.

To assist users with comparing the generated traces to the originals and to each other, the framework includes a graphing tool that can output the points on a trace's *cache surface*, a visualization of the trace's hit rates on LRU caches of various dimensions. The cache surface visualization is convenient for making quick, qualitative or even quantitative comparisons of applications or synthetic traces.

Chapter 4 provides a full description of the framework's synthetic trace generation tools.

1.6.4 The Chameleon Benchmark

The framework's namesake, Chameleon is a fully-tunable, executable benchmark whose memory behavior conforms to given model parameters. Users can utilize the trace generator to produce a Chameleon-specific seed that can be used as input to the benchmark. Chameleon, written in C++, can then be executed on target systems to foreshadow cache hit rates or application performance.

As a performance benchmark, Chameleon also allows users to experiment with the amount of data parallelism it exposes, revealing more about the characteristics and capabilities of applications and target systems.

Chapter 5 provides a full description of the Chameleon benchmark.

1.7 Test Platforms

This work uses the following three systems for evaluation throughout:

Intel Pentium D820 - The Intel Pentium D820 is a dual core x86 machine running at 2.8GHz with 1GB of shared memory. Each of the two processors leverages its own 16KB, 4-way set associative L1 cache with 64-byte blocks and 1MB, 8-way set associative L2 cache with 128-byte blocks. Both caches employ a Least Recently Used (LRU) replacement policy. The two processor connect to main memory over a shared 800MHz front side bus. The PentiumD is run by the Linux operating system using kernel version 2.6.

Intel Core Duo 2500 - The Intel Core Duo 2500 is also a dual processor machine running the same Linux kernel as the Pentium D. Each of it's processors runs at 2.0GHz and leverages its own 32KB, 8-way set associative

L1 cache with 64-byte blocks. Unlike the Pentium D however, the two processors share a combined 2MB, 8-way set associative L2, also with 64-byte blocks. All caches use a LRU replacement policy and connect to 1GB of main memory over a shared 667MHz front side bus.

IBM Power4 - The last system is the San Diego Supercomputer Center's DataStar system, a Power4-based supercomputer composed of 272, p655 nodes. Each node contains 8 processors running at 1.7GHz, paired into four 2-processor chips. Each processor utilizes its own 32KB, 2-way, LRU L1 cache with 128-byte blocks. The two processors on each chip share a 1.5MB, 8-way L2 with 128-byte blocks and the 8 processors on each node share a combined 128MB, 8-way L3 cache with 512-byte blocks.

Chapter 2

Modeling Reference Locality

In order to understand and replicate memory access patterns, one must begin with a model of reference locality. Locality is the principle that whenever a memory addresses 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 [45, 25, 20, 8, 64, 30, 18, 17, 15, 56, 67, 32].

2.1 Background

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.

This dichotomy is somewhat artificial since both types of locality have spatial and temporal dimensions [52]. With some exceptions [31, 55, 59, 67], most previous models have focused only on a single dimension. Though they are thus able to simplify their locality characterizations, these proposals may forfeit accuracy or generalization. To contextualize this proposal, we briefly describe the tradeoffs these models make and the classic abstractions on which they are based.

2.1.1 Temporal Locality

The classic measure of temporal locality is *reuse distance* or *stack distance* distributions [45, 36]. 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 [47].

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 [45, 71, 8, 18, 72, 26, 44, 23, 67].

Reuse distance was first studied by Mattson et. al around 1970 [45], and multiple studies, some as recently as 2007, have leveraged these ideas to create locality models and synthetic trace generators that function by sampling from an application's reuse distance CDF [11, 18, 26, 32, 33]. Many works have also used reuse distance analysis for program diagnosis and compiler optimization [26, 49, 72].

While reuse distance characterization is a powerful tool with practical use, such quantifications cannot dynamically capture spatial locality. They require the modeler to choose a specific block size, thereby freezing spatial locality at a single value. One can just as easily examine the reuse distances of 4-byte integers as 512-byte cache lines. Which block size is most advisable?

The 8-byte word is a popular choice, as is the width of some target cache's block length. Choosing the latter also allows modelers to predict the application's hit rate on a target cache [17, 33, 44, 73].

However, choosing a fixed cache width is undesirable. The same reuse patterns of an 8-byte word can have variable hit rates on caches with larger block sizes. Choosing the length of the target cache's block size is similarly problematic. While useful for predicting hit rates on the target caches, these choices make characterizations machine-dependent. Further, since block sizes often vary across levels of a single machine's memory hierarchy, the model can seldom capture the

application's overall memory behavior across an entire memory hierarchy, even on a single system.

Lastly, access patterns that trigger optimizations such as multi-line prefetching may not be captured. Such optimizations are triggered by spatial access patterns such as a series of short strides or access to consecutive cache lines. A temporal only characterization cannot describe such behavior.

Enabling this type of model to capture general locality would require a continuum of reuse distributions with corresponding block sizes.

2.1.2 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 [63, 67, 22]. The most straightforward approach is the *distance model*, which captures the probability of encountering each stride distance [57]. Thiebaut later refined this idea by observing that stride distributions exhibit a fractal pattern governed by a hyperbolic probability function [63, 62, 64]. In recent years, the PMaC framework has focused on spatial locality by quantifying an application's stride distributions.

Analogously to the freezing of spatial locality by temporal locality characterizations, these spatial characterizations require a particular temporal value. Thiebaut's fractal model, for example, measures the stride distance from the single previous access [62]. The PMaC framework measures the distance from the nearest of the previous 32 addresses [22, 51]. This value is similarly used in other spatial locality quantifications in hopes of approximating cache sizes of interest [67].

The number of previous addresses used to measure stride is generalized as the *lookback window* and can be measured terms of either the number of unique or non-unique memory references. Clearly, the greater the size of the lookback window, the more local the address stream appears.

Enabling this type of model to capture general locality would require a continuum of stride distributions with corresponding lookback window sizes.

2.1.3 Hybrid 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 [31, 30]. For every memory reference in a trace, he calculates 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 [53, 37].

Both Grimsrud in 1994 [30] and Sorenson in 2002 [56] and 2005 [54], concluded that no existing trace generation technique adequately captures locality of reference. Unfortunately, neither author also proposed a methodology for translating their own characterizations into synthetic traces.

2.2 A Unified Model

As we have discussed, to be complete, traditional temporal models would require a series of reuse CDF's, each with a unique block size. Traditional spatial models similarly require a series of distributions, each with its own lookback window size. We observe that both approaches actually converge on the same characerization.

2.2.1 Convergence of Abstract Locality Models

Although they start from seemingly disjoint abstractions, the classic quantifications of spatial and temporal locality actually converge into a single model. This is because reuse distance is essentially analogous to the unique access definition of lookback window and reference distance to the non-unique. Similarly, block

size is analogous to stride.

To illustrate, let us describe the temporal model's reuse distance CDF with a block size equal to B as the function $t(d, B)$, where d is the reuse distance. The temporal-based hybrid model that generalizes all such CDF's is $t(d, b)$.

This surface can also be described using the spatial model's *stride* and *lookback window* parameters. If we assume that strides are measured in absolute distance, then any point $t(d, b)$ is equal to the fraction of memory operations with stride $< b$ *and* lookback window $< d$.

The hybrid spatial characterization can similarly be described using the temporal parameters of reuse distance and block size. Because either set of parameters can be used to describe the other, they are essentially equivalent.

2.2.2 Abstractions to Cache Hit Rates

Now that we have established a unified model of reference locality, we must undertake to understand its relationship to cache hit rates. Overwhelmingly, the accuracy of locality models has been measured using cache hit rates [20, 8, 40, 63, 18, 17, 73, 15, 44, 16, 67, 33, 32]. Instead of starting with abstractions and then correlating those to cache hit rates, let us work backwards from the goal: the most trivially correct characterization of an application's cache-exploitable locality is a series of cache descriptions and the application's hit rate on each.

Many variables describe a cache, but for simplicity, let us assume only fully-associative caches with a least recently used (LRU) replacement policy. In exchange for this simplification, we trade our ability to capture and predict conflict misses. However, research has repeatedly shown conflict misses to be far more rare than capacity misses and that hit rates on such LRU caches are consequently similar to analogously sized set-associative configurations with alternate replacement policies [17, 33, 44, 73]. It is also worthwhile to note that conflict misses, unlike capacity misses, are not dependent on reference locality. We can consequently expect their

effects to apply uniformly across memory traces and the relationship between LRU and set associative cache hit rates to be preserved.

Given these observations, we can 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. We refer to this collection of hit rates as an application's *cache surface*.

Figure 2.1 displays an example of such 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; Figure 2.1 is an example of such a surface, representing various cache configurations from 64 bytes to 33.5MB.

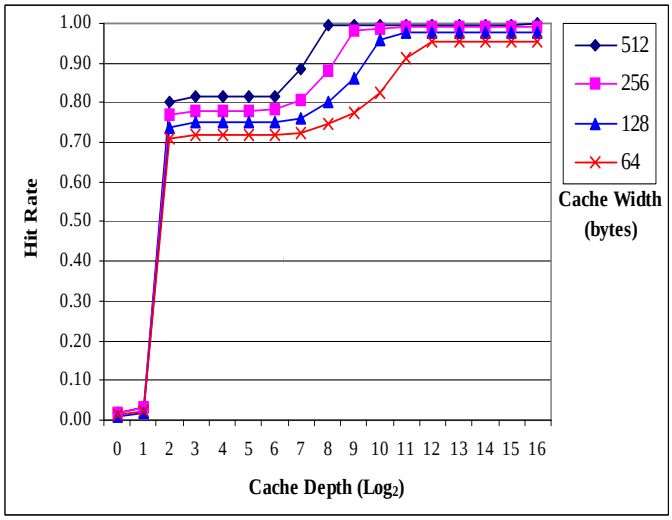

Figure 2.1: Cache surface of CG.A

Note that the cache surface is nearly identical to the unified locality model proposed in the previous section. This is because a memory reference hits in a fully-associative, LRU cache of depth D only if it has a reuse distance of less than $D-1$ [17]. Therefore, the function $hit(d, W)$ is the reuse distance CDF with block

size W, and the surface $hit(d, w)$ is the collection of all such functions.

There is however, one subtle but important difference between the classic locality abstractions and cache-exploitable locality. Caches pre-partition the address space into discrete blocks. This is an implementation detail not mandated by the original locality abstractions. Because they do so, caches are not perfect locality filters. For example, consider accessing the address sequence $\{2, 4, 6\}$. If the block size is 2, then the reuse distance sequence is $\{\infty, 0, 0\}$. However, because a cache would partition these addresses onto separate cache lines, the sequence would have no reuse at all. The relationship between reuse distance and stride is therefore somewhat altered. However, because we are interested in predicting cache hit rates and consequent performance on real machines, we prefer the cache surface as a characterization of cache-exploitable locality.

As shown, the cache surface is a complete description of cache-exploitable reference locality with close ties to the classic abstractions and cache hit rates. We therefore argue that it is at least as relevant and complete as any characterization based on these abstractions [31, 53].

2.2.3 A Definition of 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 obviate these concerns by capturing only the statistical relationships between

them. We observe that the functions $hit(d, C)$ that comprise the surface, as shown in Figure 2.1, are not composed of independent points. Importantly, the function $hit(d, C)$ is a predictable, statistical permutation of the function $hit(d, C+1)$. This permutation is spatial locality.

Intuitively, we understand that caches use larger block sizes in order to exploit spatial locality and not temporal locality. The difference in hit rates between some cache with N-byte blocks and another with $N-1$ 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 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 *reference distance* of that access increases from 1 to ∞ and the *reuse distance* of A_3 consequently increases to 3.

Suppose an address stream T contains a reference at $T[i]$ that is first reused at $T[i+j]$. Any element $T[i+k]$ with $0 < k < j$, whose reference distance has become greater than k will increment the reuse distance of $T[i+j]$. Because the length of the new reference distance is a function of the reuse distribution, the only parameter we need obtain is the probability that a reference distance will change at all when we decrease the cache line length.

To predict the misses generated when we halve the cache line length, for example, we need only determine the probability that two consecutive references to some cache line will reuse the same half of that line. The lower this probability, the lower the hit rate of the shorter cache. Let us define this probability as follows:

Spatial Locality $= \alpha(S, U) =$ *the probability of reusing some contiguous subset of addresses* S *during 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, we can 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 2.1, 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 using an application-wide basis. Because the brevity of the characterization is not important for this study, the results we present were gathered using separate α values for each reuse distance in the CDF, each corresponding to a bin of reuse distances. The total parameterization can therefore be up to 119 numbers per application.

2.2.4 Limitations

As we demonstrate in the remainder of this work, the reference locality model described in this section is a useful characterization of memory access patterns. Because it is a statistical summarization however, it has some limitations for characterizing whole programs with discrete behavioral phases on production cache hierarchies.

From the perspective of the cache abstraction, separate behavioral phases do not pose any difficulty for capturing hit rates using a statistical summarization. If such a summarization can describe the cache surface, then it properly describes the application's overall hit rate on any cache. Additionally, if we view each memory access as a discrete event, requiring a certain amount of time to complete depending on the level of cache it must access, then the distribution can describe performance as well. It is unimportant in what order those operations are executed, but only that they are properly proportioned.

The matter becomes more muddled when cache optimizations are introduced. Take multi-line prefetching for example. This optimization essentially detects if the recent pattern of memory accesses has been highly strided and if so, fetches several consecutive cache lines on the next miss. A trace-cache, which prefetches instructions based on more complex access patterns, is another such example [48].

In these scenarios, the grouping and ordering of memory operations is consequential. However, ordering cannot be described effectively with a statistical distribution. Memory access streams can have the same distribution of reuse distances but different access streams. Here are two examples of sequences with identical signatures but different access ordering:

$Example1 : 1, 0, 0, 1, 0 \Rightarrow 1, 0, 1, 0, 0$

$Example2 : 1, 2, 3, 4, 1, 2, 3, 4 \Rightarrow 3, 4, 1, 2, 3, 4, 1, 2$

This does not mean that the model cannot differentiate between certain stride patterns, which it can, but rather that it does not capture the order in which they occur. This implies that if some program goes through a highly strided phase, followed by a non-strided phase, then the overall characterization would capture the proper hit rates but not necessarily the phased behavior.

If we know in advance that the application exhibits phased behavior and that we would like to trigger prefetching type optimizations, then it would be better to characterize the application piecewise and concatenate the traces later.

2.3 Summary

In this chapter, we have reviewed the classic abstractions with which researchers have traditionally described reference locality, including reuse/reference distance and stride distributions. To be complete, a characterization must incorporate both spatial and temporal locality abstractions, and that doing so from either end, melds the two approaches into a single unified model.

This chapter has introduced the *cache surface* as an intuitive visualization of reference locality and has argued that it embodies the cache-exploitable characteristics of this unified theoretical model.

We have observed an important statistical correlation between the points of the cache surface and used it to define a new, practical definition of spatial locality, α. The new definition theoretically enables us to condense the surface characterization, capture its points with less overhead, and create synthetic address traces that match it.

The following chapters validate this definition of spatial locality by describing how they can be leveraged to create synthetic memory address traces and showing that such traces closely match the cache surfaces they are intended to target.

Chapter 3

Collecting Memory Signatures

Now that we have a model of reference locality, we must build tools that are able to collect the model's parameters from any given application. To do this, we can use binary instrumentation, a process by which a call to some modeling logic is inserted at each point in an application's binary where a load or store instruction exists. The modeling logic can thereby observe the application's dynamic memory stream and extract the model parameters from it.

Because the modeling logic can be encapsulated, it is essentially independent of the instrumentation library and is simple to port onto various architectures. The Chameleon framework currently includes one tracer built using the Pin instrumentation library [42] for tracing on x86 architectures and another using PMaCInst [66], a Power-based instrumentation library for tracing on that architecture.

These tools allow us to characterize applications of interest in the most common contexts. The Pin tool can be deployed to investigate serial desktop applications running on the most common x86 personal computer architectures. The deployment on PMaCInst allows us to model scientific applications executing on supercomputers or other large-scale, parallel resources.

For the remainder of this work, all traces using the Pin tool have been collected using the dual-processor Pentium D820. All PMaCInst traces were collected

using one node on DataStar.

Further ports using libraries such as ATOM [58] or Dyninst [35] would be straight forward and require little, if any, modification to the modeling logic.

This chapter describes the modeling logic and how it extracts the previously described model parameters from a given stream.

3.1 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 [10, 26, 39, 61].

Our simulator uses an approach similar to that described by Kim et. al. [39] 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 increased locality and the overall performance is comparable to the most efficient published solutions [26, 39, 72].

3.2 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 use the same half? What is the probability that two accesses to a half will use 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. We are interested in the values:

$$\alpha(256, 512), \alpha(128, 256), \alpha(64, 128), \alpha(32, 64), \alpha(16, 32), \alpha(8, 16), \alpha(4, 8)$$

To calculate these probabilities, we first set up two counters for each α value we expect to derive. The counters represent the number of times each subset was reused and the number of times it was not reused during the run.

Every modeled 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 512-byte 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.

3.3 Performance

It is well known that memory instrumentation often causes prohibitively high slowdown [28]. The initial performance measurements of the framework's tracing tools bear out this conventional wisdom. Table 3.1 displays the slowdown manifested by two of the NAS Parallel Benchmarks, CG and SP, both at size A.

We observe slowdown near 1000x over uninstrumented runtimes for Pin executables and approximately 300x for PMaCInst. 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. Additionally, PMaCInst shaves some runtime by assuming that all memory references load a single word instead of checking the exact size of the load. It also enjoys the added benefit of instrumenting only the most important sections of an application due to a preprocessing step described in Section 3.3.2.

Table 3.1: Slowdown caused by memory tracing

Application	Tracing Slowdown
CG.A(Pin)	1160
SP.A(Pin)	915
CG.A(PMI)	276
SP.A(PMI)	348

The slowdown numbers are clearly prohibitive even for small codes and outright impractical for larger scientific applications whose uninstrumented runtimes alone can be several hours long. At 350x slowdown, it would require several months of execution to extract the model parameters from some HPC applications. We therefore investigate sampling techniques aimed at mitigating this slowdown.

3.3.1 Interval Sampling

The first optimization we evaluate is interval sampling. Rather than send every memory access through the modeling 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 [28]. We follow this guideline and modify the tracer to iteratively use a consecutive stream of 1M addresses and then ignore the subsequent 9M.

Performance

As expected, the performance increase is around one order of magnitude. Table 3.2 lists the slowdown numbers for CG.A and SP.A when traced with the Pin and PMaCInst (PMI) instrumentation libraries.

Table 3.2: Slowdown caused by memory tracing

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

Accuracy

We must also examine the error introduced into the characterizations by applying the interval sampling technique. Because we are using a cache-type 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 two NAS benchmarks. 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.

The interval sampling technique 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.

3.3.2 Basic-Block Sampling

While the tracing slowdown with interval sampling is tractable for smaller serial applications, 30-40x slowdown is high for the larger, parallel codes of interest to the HPC community; such applications normally display base runtimes of several hours. We must therefore search out yet more powerful sampling techniques to characterize HPC applications.

One such technique is *basic block sampling*. This technique, successfully employed by mature performance prediction methodologies [50], 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 sent through the modeling logic. The modeling logic, 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 pre-processing 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 [66]. We then modify the framework tracer to isolate the locality model results by basic blocks. To incorporate each memory reference, the modeler invocation now requires a block id to accompany the address. Internally, it executes identical logic as before, but then places the outcome of each reference into the bin specified by its block id. Note that this is not the same as modeling each block separately; the internal state of the modeler is shared among all blocks but only the *results* are separated. Consequently, this separation increases neither the modeler'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.

Performance

To evaluate the performance gains of this technique, we replace the two NAS benchmarks with larger-scale, parallel applications. AMR [70] is an adaptive mesh refinement code that we deploy across 96 processors of DataStar. S3D [41] 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 both the interval and block sampling techniques concurrently 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 3.3 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 modeling 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 modeling logic rather than other tracing overheads, we replace the locality logic with a simulation of 15 arbitrary caches and retrace. The slowdown is nearly identical. Finally, we execute a trace with both the locality modeling *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 modeling logic itself accounts for only a small fraction of the tracing overhead.

Table 3.3: Application slowdown due to memory tracing

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

With such low overhead for the modeling logic, it is feasible to simply insert it into a performance analysis group's existing memory tracing activities. For example, a modeler trying to evaluate an application's performance on these 15 caches can concurrently collect its 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. Sending the synthetic trace through a cache simulator requires only a few seconds and is almost surely faster than even executing the original application without instrumentation

Accuracy

We address the accuracy of memory signatures acquired using this technique in Chapter 4.

3.4 Summary

This section describes the implementation of a memory address tracer that can collect Chameleon's memory signatures from any given application with minimal slowdown. The framework deploys this tracing logic using both the Pin and

PMaCInst instrumentation libraries for tracing of serial codes on x86 and parallel codes on Power architectures respectively.

To achieve its runtime goals, the Pin implementation uses a novel LRU cache simulation technique and 10% *interval sampling*. The PMaCInst version adds *basic block* sampling to reduce slowdown to approximately 5x, only marginally higher than empty instrumentation. We have verified that the inaccuracy introduced by these sampling techniques is marginal. As a result, the Chameleon modeling logic can be inserted into application tracing activities to extract accurate memory signatures without significant performance penalty.

The following chapter describes a technique for generating synthetic address traces from the collected signatures and the evaluation of those traces' effectiveness in mimicking the locality properties of the original applications.

Chapter 4

Generating Synthetic Traces

To prove that the compressed cache surface representation can effectively capture locality of reference and is therefore a useful memory signature, one must go beyond previous hybrid models and convert these characterizations into synthetic traces; similarity between cache hit rates of the synthetic and original traces would indicate a sound model.

Chameleon's trace generation tool can be used 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. These concise seeds are preferable to the full traces both because of their flexibility and their ease of handling.

4.1 Generation Technique

Recall that the model consists of temporal and spatial components. We have a reuse distance CDF with block sizes of 512 bytes and a series of α values that iteratively project that CDF to shorter block sizes.

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 list element is given a unique identifier and represents a unique block. The list is marginally longer than the largest cache is deep, ensuring we can perform random accesses. Using inverse transform sampling, we sample reuse distances from the input 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.

Error stems from two sources: chance and cold cache misses. To eliminate the former, we need only generate a sufficiently sized sample. The latter is somewhat trickier. Cold cache misses occur because the cache is initially empty and a reference to the third element in the list for example, might therefore correspond to a random access rather than a reuse distance of 2. While this effect too can be mitigated with large enough sample lengths, we do not want to grow trace seeds too large.

Instead of the brute force approach, the generator can simply adjust its aim. 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. These error bounds are tunable and both are set to half of one percent for the experiments described in this work. The trace continues to grow until it is either within the error bounds or has exceeded a maximum length. If the latter occurs before the former, then 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 entire process until finding a satisfactory trace. The generator additionally keeps track of the incremental improvement that each iteration has made. If an iteration does not improve the trace accuracy above a tunable improvement threshold, then the generator stops.

Two or three iterations are typical and, depending on the data set, each requires between a few seconds to one minute on the Pentium D820 we used. For these tests, we cap the number of iterations at five, the minimum length of each seed at 200,000 and the maximum length at 1,000,000.

At this stage, the trace generator has created a trace of *block* addresses. In order to add spatial locality, it must then convert these to 4-byte word addresses. The generator iterates over the trace and for each block, it chooses an offset as dictated by the series of α values in the input. Each block maintains its own history using a tree structure identical to that used by the tracer.

The final product is a series of indices to words. The size of each word, most often 4 or 8 bytes, can be set as a compile-time parameter in the tracing logic. The generator writes this trace to an output file no larger than a few megabytes and prefaces it with some metadata as follows:

Length - The number of elements in the trace

Signature - The target memory signature passed as input into the trace generator

Size of Word - The trace output is a series of word addresses. This value reports the size of each word and is necessary for replaying the trace.

Working Set Size - The total memory footprint touched by this trace. The reported working set size helps determine the number of times this seed should be replicated in order to achieve a certain footprint.

Minimum Replications - The minimum replications figure is the fewest number of replications needed to ensure full accuracy. Why does the number of replications affect accuracy? Recall that when the generator calibrates the trace, it includes cold cache misses. However, if we repeat the trace, the cache warms and these misses may become hits of unpredictable reuse distances. To avoid this, we must ensure that no element in some replication's working set is resident in cache when we begin that replication's simulation. During seed creation, the generator counts the number of unique block addresses used in the trace. In order to flush a cache of depth d, a seed using u unique block addresses must be replicated at least $\frac{d}{u} + 1$ times.

4.2 Accuracy on Cache Surfaces

This section compares the cache surfaces of the synthetic traces with those of some target benchmarks. We use serial versions of the NAS benchmarks CG.A, SP.A, and IS.B running on the Pentium D820. We collect the memory signatures of each benchmark using 10% interval sampling and create a synthetic trace for each. We then feed the synthetic traces through the framework's cache surface tool, effectively determining their cache hit rates on 68 LRU caches of various dimensions. We do the same with the original traces.

Figures 4.1, 4.2, and 4.3 plot the discrepancy in hit rate between each synthetic trace and the benchmark it is targeting on the 68 LRU caches. The X-axis represents cache depths on a log_2 scale. Each line, represents a different cache width as labeled in the legend. The figure plotted is calculated as $T - A$ where T is the trace's hit rate and A is the application's.

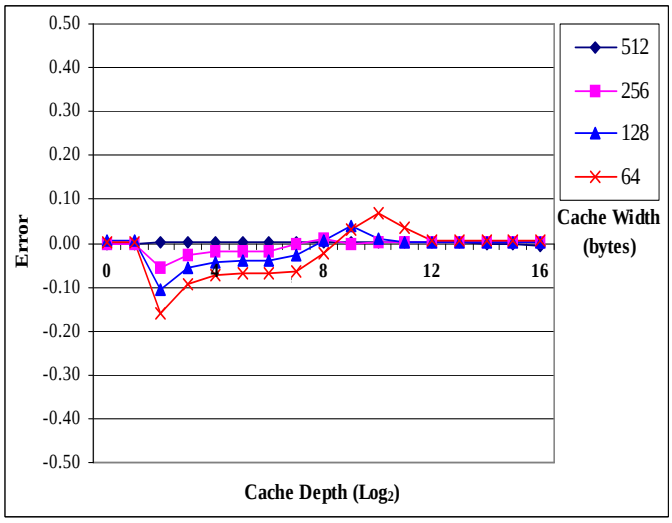

Figure 4.1: Synthetic trace hit rates vs CG.A

Overall, the traces are quite accurate. The absolute average error for CG, SP, and IS is only 1.9%, 1.6%, and 8.1% respectively. Notice that there is no error for caches specified by the model's temporal parameters (512-byte width).

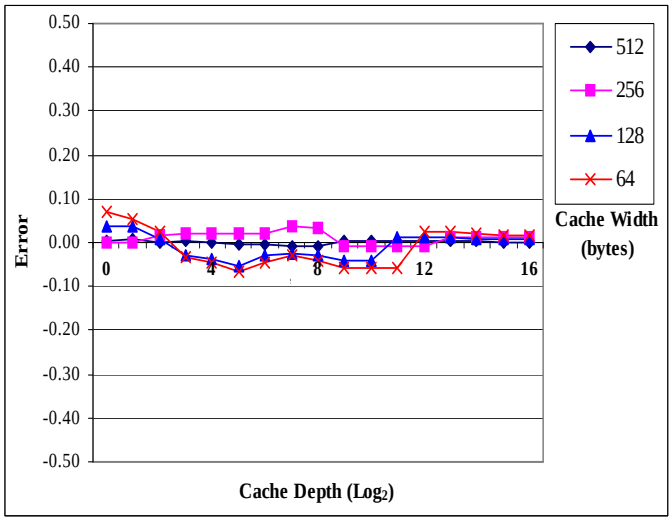

Figure 4.2: Synthetic trace hit rates vs SP.A

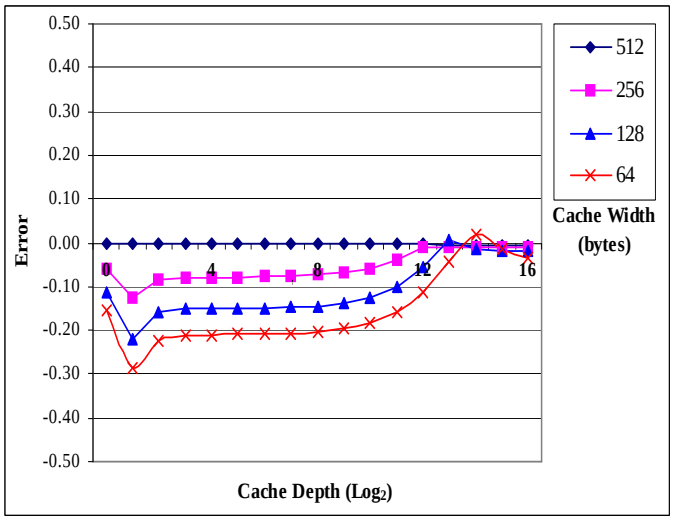

Figure 4.3: Synthetic trace hit rates vs IS.B

Chameleon's trace generation technique is therefore the most accurate possible for any temporal locality based solution. Moreover, because the memory signatures capture spatial locality using the α parameters, the resulting synthetic traces *also* match the hit rates of caches with shorter widths. In this capacity, Chameleon is distinct from all previous memory modeling and synthetic trace generation proposals.

Next, observe that the magnitude of error increases as cache widths decrease. This is not due to some inherent difficulty with emulating shorter widths, but rather to the statistical mechanism by which we recursively project each function from the previous. Starting from the perfectly accurate 512-byte function, we project to 256 and then from there to 128, etc. Subsequent projections thus accumulate error. This may not be critical since most memory hierarchies do not present more than two cache widths and a single projection may suffice.

Even though some points on the IS plot err by as much as 20%, and one point by almost 30%, the condensed cache surface model is nevertheless accurate and generally useful, particularly when more is known about the target cache configurations. If, however, we desire more accuracy under ambiguity, we can barter some of the model's brevity for it.

4.3 Multiple Alpha Values

Examine Figure 4.3 again. Notice that the α values we use are not particularly good for projecting even from very shallow caches. Because the shallow caches are subsets of the deeper caches, any error they incur is propagated. However, we notice that as the caches deepen, the error eventually gravitates back towards zero. This indicates that while the α we used was too low for references with short reuse distances, it was actually too high for those with longer ones.

With a small modification to the tracer and without palpable runtime penalty, we can break down the α values by reuse distance. The results dis-

played in Table 4.1 show this breakdown for $\alpha(256, 512)$, the 512-byte to 256-byte projection.

Table 4.1: $\alpha(256, 512)$ by reuse distance in IS.B

Reuse Distance	α
2^0	1.0
2^1	.98
$\geq 2^2$	.50
Weighted Avg:	.84

As hypothesized, the α value of .84 is indeed too low for short reuse distances and too high for long distances. The same pattern holds in IS for every working set size, down to 8-bytes. This tells us much about the access patterns of IS and enables us to emulate it much more accurately.

We modify the memory tracing logic to collect and report α values according to exponentially sized reuse distance bins as in Table 4.1. An example memory signature is shown in Appendix A. In addition to the original α values, now reported as "Avg", the new format breaks down each value into its contributing parts by reuse distance range. The Appendix explains the format in more detail.

We modify the trace generator to accept this new format and conform its traces to the detailed α value breakdowns. To do this, the generator must record the reuse distance that generated each block address. To convert each block address to a word address, the generator looks up the block address's reuse distance and then retrieves the corresponding α value set. Lastly, it uses that set of α values to choose an appropriate index as before.

Repeating the experiments in the previous section produces the results depicted in Figures 4.4, 4.5, and 4.6. The traces are clearly superior to those derived using only a single spatial parameter. Traces emulating CG.A, SP.A, and IS.B err on absolute average by only 1.0%, 1.5%, and 0.1% respectively. Given these

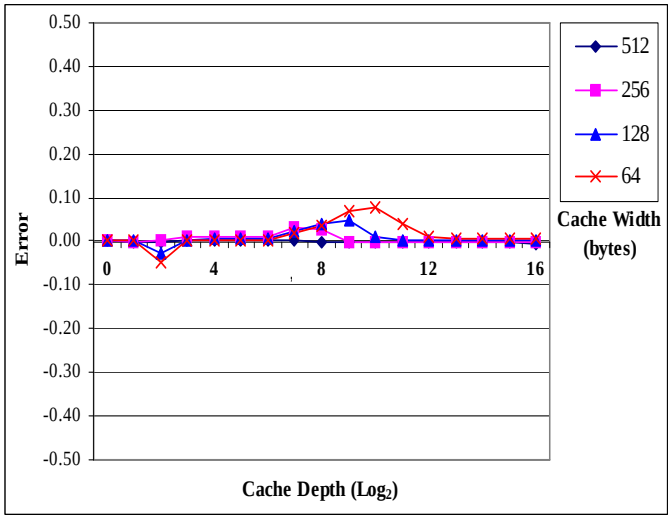

Figure 4.4: Trace vs CG.A (multiple alphas)

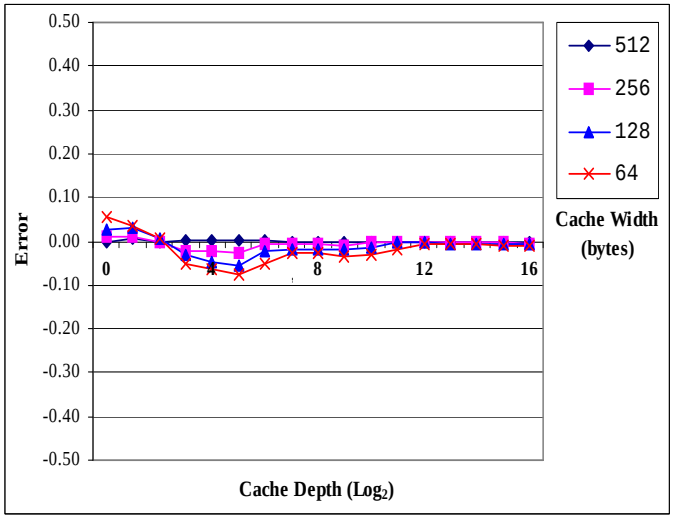

Figure 4.5: Trace vs SP.A (multiple alphas)

results, we can conclude with some confidence that the α parameter does indeed quantify the spatial locality of applications usefully and accurately.

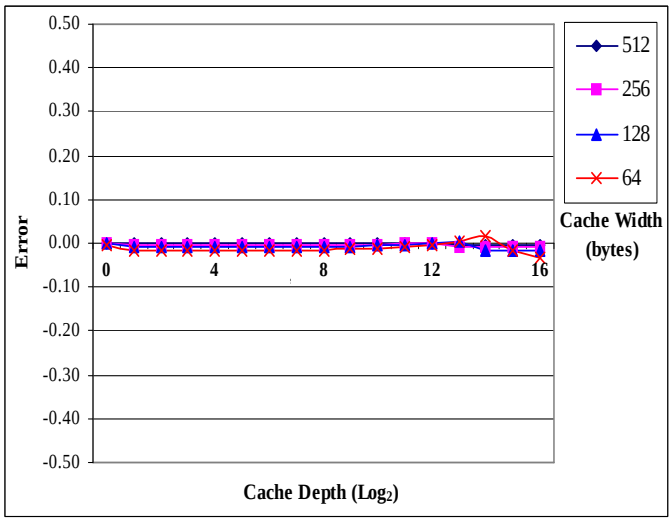

Figure 4.6: Trace vs IS.B (multiple alphas)

4.4 Accuracy on Cache Hierarchies

To this point, we have shown that Chameleon's synthetic traces produce cache surfaces that are highly similar to those of their target applications. As described in Chapter 2, this implies that the synthetic traces and target applications have nearly identical theoretical locality properties.

The last step is to determine the extent to which these locality properties can dictate hit rates on real-world, set-associative, caches. To test this, we compare the hit rates of the actual and synthetic traces for the three test benchmarks on 46 real-world and theoretical cache *hierarchies*. We choose these as representative of many modern commercial offerings; the same set was evaluated by the U.S. Department of Defense's HPCMO program to support equipment procurement decisions in 2008 [38]. These cache hierarchies are listed in Table 4.2.

Table 4.3 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 highly comparable, even when multi-level caches of disparate widths and depths inhabit a single hierarchy.

Table 4.2: Cache Hierarchies used to evaluate synthetic trace accuracy

ID	size/width/assoc		ID	size/width/assoc	
	L1	L2		L1	L2
1	32768/128/2	786432/128/4	2	32768/128/2	786432/128/8
3	65536/64/2	1048576/64/16	4	262144/128/8	6291456/128/12
5	16384/64/8	1048576/64/8	6	32768/128/4	983040/128/10
7	32768/64/8	2097152/64/16	8	262144/128/8	4194304/128/4
9	16384/32/2	2097152/32/2	10	262144/128/8	7340032/128/12
11	262144/128/8	9437184/128/12	12	32768/128/2	499712/128/4
13	32768/128/2	366592/128/8	14	204800/128/8	5372928/128/12
15	32768/64/8	1597440/64/16	16	32768/128/4	458752/128/10
17	204800/128/8	8518656/128/12	18	243712/64/8	1048576/64/8
19	65536/64/2	524288/64/16	20	65536/64/2	2097152/64/16
21	65536/64/2	4194304/64/16	22	16384/32/2	524288/32/4
23	204800/128/8	3275776/128/4	24	65536/64/2	983040/64/16
25	32768/128/4	1966080/128/10	26	32768/128/4	1433600/128/10
27	32768/64/8	3145728/64/24	28	32768/64/8	1253376/64/24
29	32768/64/8	983040/64/16	30	32768/64/8	565248/64/16
31	32768/64/8	1253376/64/16	32	262144/128/8	6291456/128/4
33	204800/128/8	5505024/128/4	34	65536/64/2	811008/64/16
35	32768/32/64	2097152/128/8	36	32768/32/64	2099200/128/8
37	262144/128/8	9437184/128/4	38	204800/128/8	7372800/128/4
39	32768/128/8	4194304/128/8	40	32768/128/8	2099200/128/8
41	32768/128/8	3637248/128/8	42	32768/32/4	262144/64/2
43	32768/32/4	131072/64/2	44	65536/64/2	524288/64/16
45	65536/64/2	524288/64/16	46	65536/64/2	524288/64/16

Table 4.3: Avg. absolute error on 46 simulated cache hierarchies

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

There are several possible sources of error. First, is the linear interpolation used by the trace generator. The memory signature given as input is only sufficient to define 119 points on the cache surface. The trace generator uses linear interpolation between these points, possibly causing some error when the traces execute on arbitrary cache sizes.

An additional source of error may be the model's omittance of conflict misses. The similarity of the synthetic and actual trace's cache surfaces portends similar cache hit rates on fully associative caches, but does not explicitly address conflict misses on set-associative configurations. We assume the level of conflict misses should be similar, perhaps even a uniform penalty across all traces. However, we cannot guarantee these rates to be identical.

Lastly, because these are cache hierarchies, errors in the L1 hit rates may propagate to L2. This may help to explain the relatively larger error margins we observe in L2.

The results of this section demonstrate that the memory signatures collected from serial benchmarks using the interval sampling technique can be used to generate accurate synthetic memory traces for set-associative, multi-level cache hierarchies, even when multiple cache widths exist.

4.5 Accuracy for Parallel Applications

In this section, we address whether or not we can collect useful signatures from parallel applications using the block-sampling technique. To do so, we again

use the parallel applications from Section 3.3.2, AMR and S3D at 96 and 8 processors respectively.

We wish to compare the cache hit rates of these parallel applications with those produced by their synthetic counterparts 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 15 real-world, set-associative caches chosen arbitrarily from Table 4.2. 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% [22]. 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 4.4 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.

These results demonstrate that the interval and block sampling techniques can help us accurately characterize whole, parallel-applications. In addition to whole program characterization, it is often useful in performance modeling to produce per-block characterizations [22].

We confirm that Chameleon can accurately model the behavior of each block of the applications should a stream at that level of detail be needed. It is important to recall here that the characterizations are not of each block per se, but rather of each block's *behavior*. This is a subtle difference. Because of cache warming and cooling effects, a block that executes in isolation may exhibit different hit rates than it would had it executed among the others. The characterization produced by

Table 4.4: Average absolute error in cache hit rates between applications and synthetic traces over 15 cache configurations by processor

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	–	–

the Chameleon Framework implicitly captures cache state. The characterization is therefore not a representation of the block in isolation, but rather of its behavior in the context of the application.

To model application behavior at the block level, we decompose the locality signature of processor 0 into its separate blocks and produce a trace for each. We use each synthetic trace to drive a simulation of the same 15 caches and compare the results to the actual hit rates achieved by each of these blocks.

We examine the 10 most heavily executed basic blocks of each application, which constitute approximately 55% and 40% of AMR's and S3D's dynamic memory references on the processor respectively. In all cases, the hit rates produced by the actual basic blocks are within 1% of those achieved by the synthetic traces. Such a capability is important for basic-block oriented performance analysis tools such as the PMaC prediction framework, which we have earlier discussed.

The results in this section demonstrate that the proposed characterization is able to describe memory access patterns effectively at the granularity of either whole applications or basic blocks for parallel applications, even when both interval

and basic block sampling is employed. We can thus extract usefully accurate memory signatures from parallel applications while imposing only a 5x tracing slowdown.

4.6 Summary

This chapter introduced a software tool and technique for generating accurate synthetic memory traces from an application's Chameleon memory signature. We have verified that the resulting traces are characterized by cache surfaces that are nearly identical to those of the original applications.

We have observed that α values may not be uniform across all reuse distances discussed how Chameleon's tracing tools can therefore parse and report those values by reuse distance. Trace generation tool accepts these delineated α values as input and produce conforming traces.

By observing that the synthetic traces produce cache surfaces that are highly similar to those of the original applications, we have verified that α values do indeed define the relationships between the points of the cache surface, and consequently, are a useful characterization of locality.

Lastly, we have verified that cache surfaces correlate strongly to cache hit rates. We have done so by demonstrating that the framework's synthetic traces produce highly similar cache hit rates to those of their target applications across 46 modern memory *hierarchies*, even when multiple block sizes exist within a single hierarchy. We have verified the accuracy of the parallel application tracer by demonstrating that it produces hit rates that are nearly identical to those produced by existing simulation tools on 15 memory hierarchies.

The following chapter describes the implementation of a tunable synthetic memory benchmark based on this framework.

Chapter 5

The Chameleon Benchmark

The previous chapters have described an observable characterization of reference locality and a technique for generating accurate synthetic traces based on that characterization. While these are powerful tools for system designers and performance modelers, they may have limited utility for less sophisticated users or for those outside of our framework. For more accessibility, users may favor an executable benchmark over a trace.

5.1 Motivation

Benchmarking is currently the technique of choice for end-user performance evaluation of systems. Users can simply attain benchmark runtimes on target systems and use those to extrapolate a notion of overall performance. In HPC for example, the Top500, a sorting of the world's fastest 500 computers [7], orders systems according to the speed at which each can execute the Linpack benchmark [27]. The problem is that the significance of benchmark results to application runtimes is not always clear [21]. Teasing out such relationships can be a significant endeavor, which is nonetheless undertaken regularly for HPC procurement cycles [50, 65].

Further complicating the undertaking is the sometimes volatile nature of benchmark behavior across platforms. It is not always clear that a certain benchmark is actually doing the same thing across platforms. If not, then the results may be a measure of some compiler optimization or failing instead of hardware performance. Even simple synthetic benchmarks intended to perform some uninvolved access pattern are vulnerable to this volatility, perhaps even more so.

This chapter proposes a fully tunable memory benchmark based on the Chameleon framework's memory signatures. If we can dictate benchmark behavior with a memory signature, we can eliminate the uncertainty surrounding the benchmark's relationship to a given application. The relationship would be clearly described by their respective memory characterizations, which could be set as equivalent to produce benchmark proxies for any application. Users could calibrate the benchmark to imitate any application and deploy it outside the Chameleon framework as a self-contained executable, independent of a cache simulator or trace generator.

A further requirement is that the behavior not be volatile. It is often difficult to know what a particular benchmark is doing across systems or compiler optimizations. We would like to eliminate this complexity by creating a memory benchmark that behaves predictably.

5.2 Background

Few tunable memory benchmarks exist today and none, of which we are aware, is based on an observable characterization of memory. *MultiMAPS* [3] for example, is a two-dimensional extension of the traditional MAPS benchmark that has gained acceptance in recent years as part of the HPC Challenge suite [1]. MultiMAPS performs regularly strided access through memory; the memory footprint and length of the strides is tunable. The abstractions used to parameterize MultiMAPS however, are inadequate for describing arbitrary memory be-

havior. Because memory access patterns are more complex than regular strides, MultiMAPS's coverage of memory behavior space quite sparse.

Researchers have recently attempted to deploy MultiMAPS as a performance benchmark by using a parameter sweep to correlate cache hit rates and corresponding memory bandwidths on a target machine. The data is then used to extrapolate the expected memory bandwidth of some other application, given its cache hit rates [65]. Unfortunately, the extrapolation is complex and the assumption that a single cache hit rate on a particular machine can predict the achieved memory throughput is tenuous.

A similarly cache-based level of abstraction has been used to relate the tuning parameters of the Apex-Map benchmark [59, 60] to other applications [67]. Unlike MultiMAPS, the tuning parameters of Apex-Map are based on principles of spatial and temporal locality, but like MultiMAPS, these parameters do not exhibit a one-to-one correspondence with applications.

Apex-MAP performs L consecutive stride 1 accesses through memory at starting points chosen from an exponential distribution. The nature of the exponential distribution, the memory footprint, and L are tunable. Again, these parameters are not extractable from an arbitrary stream and only sparsely cover the space of possible memory behavior [67].

The Apex-Map benchmark also has the issue of using an *index array* to dictate memory behavior. In this approach, an initialization phase creates separate data and index arrays. At each iteration of the work loop, the benchmark reads the next value from the index array and then accesses the element of the data array at that index. The problem is that since the index array is touched between every reference to the data array, at least 50% of the benchmark's memory accesses do not conform to the target distribution. Worse yet, the spacing of these overhead accesses *between* every intended access, further warps the locality properties of the intended stream. This sort of problem is common among performance benchmarks. One may argue that superfluous L1 hits in such circumstances do not impose

a significant runtime impact. Nevertheless, this inaccuracy makes relating the benchmark behavior to applications difficult to impossible, even through cache hit rates.

5.3 Benchmark Concept

The goal of the Chameleon benchmark is to build a *fully* tunable memory benchmark based on the framework's memory signatures. One approach is to use the synthetic address streams produced by the trace generator to define the memory access pattern of the benchmark. Because the synthetic address trace consists of a series of word addresses, the benchmark simply needs to read the seed file, initialize a data array of the properly sized types (e.g. 4-byte integers, 8-byte longs, etc), and then access the data array in the pattern specified by the file. In much the same way as creating a synthetic trace, the benchmark can expand its footprint by replicating the seed pattern onto discrete portions of the data array and control its runtime by repeating the full access pattern as desired.

The only question is how the benchmark will keep the access pattern itself in memory. One approach is to hold it in index array, but as discussed in the previous section, this would cause a significant perturbation of the intended pattern.

Rather than using separate index and data arrays, Chameleon uses a single array to double as both. Each data array element contains the value of the next index to touch. In this way, the read from the data array actuates a reference to the desired memory location while simultaneously providing the benchmark with the information it needs for the next iteration.

The work loop is essentially:

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

The benchmark ensures that the pointer chasing forms a closed loop by pointing the last index touched by each replication to the first one touched by the next replication. The final replication points back to the first memory access of the original replication. The value of `numMemAccesses` can therefore be any value without additional modification to the data array. To ensure that every memory address is performed, the function returns the final value of `nextAddress`.

There is, of course, the caveat that no two elements of the array can contain the same value. The loop can only touch a particular array element once per iteration. This requirement, that trace seeds never repeat an index, obligates the generator to make extra considerations in its work.

5.4 Modified Seed Generation

In addition to the name of the trace file to use, the seed generator accepts a boolean parameter to indicate if indices can be used only once. If the seed is intended for the Chameleon benchmark, this flag must be set to true.

Significant challenges arise when the generator cannot repeat indices. Recall that seeds are generated in two phases: first the generator creates a stream of block addresses using the model's temporal parameters and then converts those to 4-byte word addresses using its spatial parameters. Both phases are affected by the new requirement. The blocks chosen by the spatial process may have already been used. Further, a block may only be touched a certain number of times, affecting the temporal distribution.

The following two sections discuss the necessary adjustments to each phase of the seed generation.

5.4.1 Adjusting Spatial Locality

During the spatial phase, the generator takes a series of cache block identifiers and translates them to word addresses according to the given spatial parameters. Because now it cannot reuse any word, the generator needs to track those it has already touched.

Recall that each block is already represented by a tree structure with the leaves representing individual words. Once a word is used, the generator must now delete the node representing it. Any node in the tree without children must be deleted as well.

There are two implications of this policy. First, missing nodes may prevent the generator from traversing the tree along its desired path. For instance, suppose some node has only one child. Even if the next memory access needs to traverse down a different path, it cannot. The generator makes reuse decisions independently for each level of the tree, so even when a subtree is unavailable, it will complete an identical traversal pattern on the symmetric subtree.

The net result is that the generator is unable to comply accurately with the requested α values. As done for cold cache misses in the temporal phase, this error can be corrected by iteratively adjusting the generator's aim. During the spatial phase, the generator tracks the α values it *actually* achieves and compares them to those requested. If sufficient error exists, the generator scales each value by its ratio to the original target and repeats the index generation phase. Most often, the error is sufficiently mitigated within a few iterations. Depending on the requested values, it may not always be possible to eliminate the error completely using this technique. Section 5.4.3 discusses this in more detail.

The second problem is that some blocks fill to capacity before the last access to them is to occur. Since each 512-byte block contains 64 4-byte words, the entire block disappears after that number of accesses. To deal with this, the generator *virtualizes* blocks. Each time it encounters a new identifier in the block trace, the

generator allocates a new physical block. When that block fills to capacity, the generator reassigns the identifier to refer to a new physical block.

This technique allows the generator to accommodate any number of references to a single block. In doing so, however, it perturbs the block trace's temporal distribution. Luckily, the generator can anticipate this effect and make preemptive corrections in the temporal locality phase.

5.4.2 Adjusting Temporal Locality

Recall that the temporal locality phase produces a trace of block addresses by sampling reuse distances from the application's distribution and compensating for cold cache misses. Block virtualization compounds this error since the generator incurs a cold cache miss for every *physical* block touched. Remember that this happens whenever a M-word block is touched N times and N%M=0.

To anticipate when unwanted misses occur, the generator counts the number of accesses to each virtual block and determines when a new physical block would become necessary. With this information, it determines the actual reuse CDF it achieves and iteratively compensates its aim as described in Section 4.1.

5.4.3 Limitations

The requirement that the Chameleon benchmark can never touch the same memory address twice, places certain limitations on the level of locality it can exhibit. Chameleon's miss rate on an N-word cache must be at least $1/N$. For a L1 cache with 64-byte blocks, this implies that a Chameleon instance using 4-byte integers cannot achieve a hit rate higher than $15/16 \approx .94$. For 128-byte blocks, it can be no higher than $31/32 \approx .97$.

This may be somewhat problematic for imitating the memory behavior of highly local applications. The most local behavior that Chameleon can produce is a regular, one word stride through memory. This may not be local enough for

some benchmarks and Chameleon's hit rates on first level caches may consequently be too low.

This does not necessarily mean that Chameleon cannot achieve higher hit rates. Indeed, its regularly strided access pattern is exploitable by prefetching optimizations, particularly at higher levels of cache.

5.5 Benchmark Initialization

Chameleon accepts the following input parameters:

Trace File - The name of the file containing the trace seed description

Footprint - The size of memory for the data array in megabytes

Number of Operations - The number of memory operations the benchmark should execute (x2 billion)

Processor - For multiprocessor systems, the processor to bind onto

Print Counters - On x86 systems with an installation of the Performance API (PAPI) [19], setting this boolean flag prints the value of all available hardware counters after the execution

Delay - Delay is the number of seconds to wait after initialization before beginning the actual run. This is useful for performing co-scheduling tests.

Using the working set size and minimum number of replications reported in the seed file, Chameleon calculates the minimum footprint for this run. If the memory footprint requested by the user is smaller, then Chameleon uses the calculated minimum instead. Otherwise, Chameleon uses the smallest multiple of the working set size that is greater than the requested footprint.

The benchmark then allocates a data array and finds the index of the first array element that begins a new cache block. The benchmark detects this by

checking that the address of the element is evenly divisible by 512. This element would be mapped onto the beginning of a block for any cache with exponentially sized block widths less than or equal to 512.

Chameleon reads the input into an `indices` array and for each replication, the initializer uses the seed and offset to populate the data array as follows:

```
dataArray[offset+indices[i]] = offset+indices[i+1]
```

When each replication is finished, the initialization function adds the working set size to the offset and points the last element of the previous replication to the first element of the new one. Once the initialization has created enough replications, it points the final element to the first element of the first replication and the data array is complete. At runtime, Chameleon simply follows the indices as shown in Section 5.3, making the number of hops dictated by the input parameter.

The processor binding parameter is a zero-indexed integer specifying the processor on which to execute Chameleon, on multiprocessor, unix-based systems. This is important to ensure uniformity across runs by preventing process migration. This, along with the *delay* parameter can be used for studying symbiotic space-sharing [68, 69] by measuring destructive interference between benchmarks running concurrently on different processors of an SMP. Chapter 6 describes this in more detail.

5.6 Accuracy on Cache Surfaces

Given that the Chameleon benchmark is based on the same synthetic trace methodology evaluated in Chapter 4, we can anticipate that it would accurately mimic target applications on most real world caches. However, this would only hold if the benchmark's additional requirement, that the synthetic traces never reuse the same address, can be shown to perturb the trace accuracy only minimally.

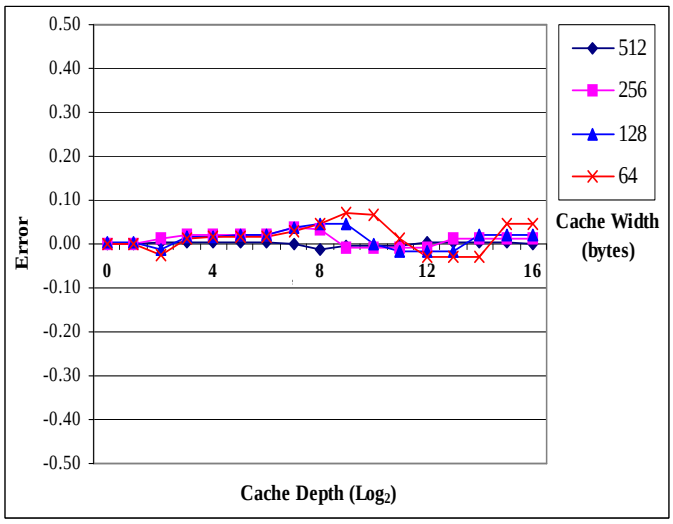

Figure 5.1: Chameleon hit rates vs CG.A

To test the accuracy of the Chameleon benchmark's input traces, we create seeds for each of the three NAS benchmarks evaluated in Section 4.2, using the benchmark's one address, one access requirement. We then feed the synthetic traces through the framework's cache surface tool, which effectively captures their hit rates on 68 LRU caches.

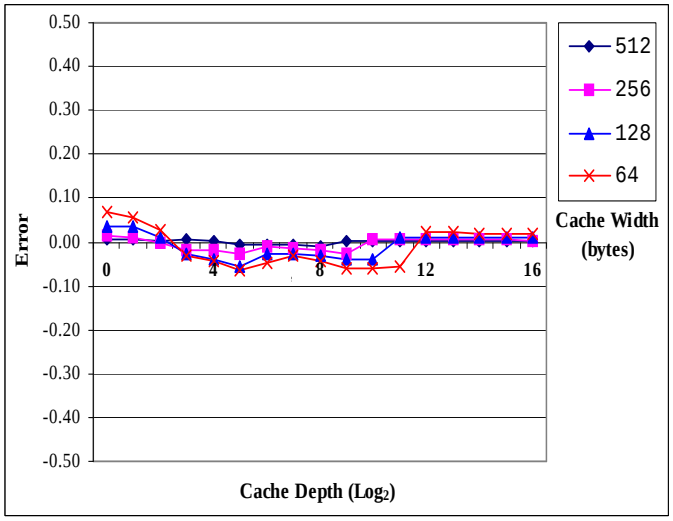

Figure 5.2: Chameleon hit rates vs SP.A

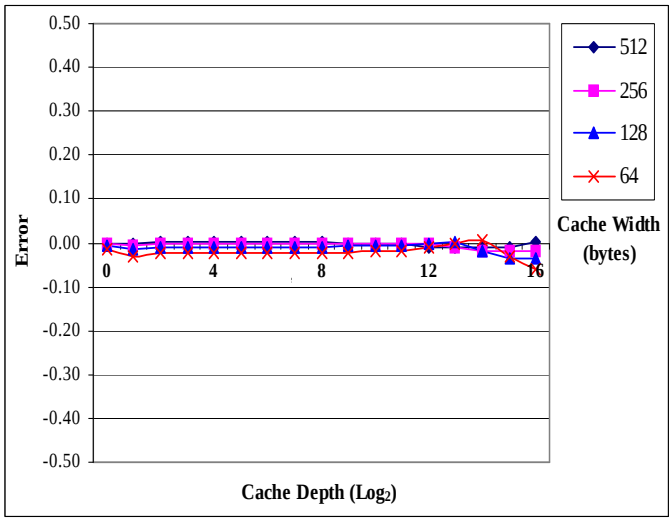

Figure 5.3: Chameleon hit rates vs IS.B

Figures 5.1, 5.2, and 5.3 plot the difference between Chameleon's hit rates and those of the original three applications. The results demonstrate that the benchmark requirement of reference uniqueness does not introduce significant error beyond that present in the unconstrained traces. The average absolute difference in hit rates between the benchmark and CG.A, SP.A, and IS.B is only 1.7%, 2.0%, and 1.1% respectively. These values are only marginally higher than the uninhibited Chameleon trace's respective errors of 1.0%, 1.5%, and 0.1%.

5.7 Accuracy on Cache Hierarchies

Section 4.4 demonstrated that the Chameleon framework's synthetic traces produce hit rates that are highly similar to those of target applications on dozens of real-world cache *hierarchies*, even when disparate block sizes exist within the hierarchy. Given that the Chameleon benchmark's restricted traces produce cache surfaces that are highly similar to those of the unrestricted traces, we would anticipate that the executable benchmark can similarly mimic the target applications on real-world cache hierarchies.

We can verify this by using the Pentium D820 and the PAPI performance counter library [19] to compare the cache hit rates achieved by the NAS benchmarks and those achieved by their Chameleon counterparts. Recall that the D820 uses different block sizes in its first and second level caches, making the system challenging for synthetic address traces.

Table 5.1 displays cache hit rates produced by each NAS benchmark versus those produced by its Chameleon counterpart. Because the D820's Linux kernel uses a time-sharing scheduler, hardware counter statistics are prone to perturbation by system and other background processes. To mitigate this effect, we repeat each test 10 times and chose only the highest hit rate combination observed in those trials.

Table 5.1: Cache hit rates of NPB and Chameleon on Pentium D820

Application	L1 — L2	Chameleon	Error
BT.A	.96 — .98	.93 — .98	.03 — .00
CG.A	.66 — .99	.67 — .99	.02 — .00
FT.A	.86 — .97	.88 — .99	.02 — .01
IS.B	.67 — .85	.58 — .83	.09 — .02
LU.A	.94 — .95	.94 — .97	.00 — .02
SP.A	.93 — .94	.94 — .97	.01 — .03
UA.A	.91 — .91	.94 — .97	.03 — .06

These measurements demonstrate that the hit rates of Chameleon and the actual applications are quite similar. The errors are almost uniformly under 5%, with an average of only 2.4% and maximum of 9%. The magnitude of the error rates is similar to that presented for unrestricted traces in Section 4.4.

Some of the error in emulating the L1 rates stems from the choice of benchmarks and architecture. Recall that the Pentium D's L1 cache uses a 64-byte block length, enough to hold 16 32-bit integers. As we discussed earlier, this makes hit rates above 94% highly unlikely. Three of the benchmarks in Table 5.1 (BT, LU, and SP) exhibit L1 hit rates above that mark, causing unavoidable error for the

benchmark on this system.

5.8 Instruction Level Parallelism

The results in this chapter have thus far shown that the Chameleon benchmark is capable of imitating the memory behavior of arbitrary applications as measured by cache hit rates. The benchmark can therefore serve as an effective application proxy for projecting memory hierarchy hit rates for target applications.

Previous work has shown that such information can be sufficient for generating detailed performance models and predictions for applications [65]. However, one would prefer to dispense with the modeling phase and simply infer the anticipated performance of a given application based on the observed performance of its Chameleon counterpart. Unfortunately, the similarity of an application's hit rates to those of its Chameleon counterpart is not necessarily indicative of highly similar performance.

Table 5.2 compares a series of NAS benchmarks and their Chameleon counterparts, listing the cache hit rates and corresponding performance for each on the Pentium D820. Observe that despite its highly similar hit rates, Chameleon consistently under-performs its target benchmark by 2-4x.

Table 5.2: Performance of NPB vs Chameleon

Application	L1/L2	L1/L2 [C]	MemOps/s (x10^8)	MemOps/s (x10^8) [C]
BT.A	.96/.98	.93/.98	12.90	4.91
CG.A	.66/.99	.67/.99	9.69	1.84
FT.A	.86/.97	.88/.99	7.39	3.19
IS.B	.67/.85	.58/.83	2.07	.638
LU.A	.94/.95	.95/.97	8.25	4.84
SP.A	.93/.94	.94/.97	9.06	4.46
UA.A	.91/.91	.94/.97	9.19	4.65

This may be somewhat surprising, considering that, unlike the benchmarks, Chameleon performs no floating point work at all. Perhaps the most important factor explaining this discrepancy is the level of instruction level parallelism (ILP) that is exposed by the respective applications. Many modern machines utilize *out-of- order* processors that allow multiple load/store instructions to be outstanding at any given time [34]. While one instruction is waiting for its operands to be retrieved from memory, the system may execute other, non-dependent instructions. Systems are thus able to overlap memory penalties and speed up execution.

Recall Chameleon's work loop from Section 5.3. Because the execution of each iteration is completely dependant upon the results of the previous one, Chameleon's work loop has no instruction parallelism to exploit. Contrast this, for example, with the work loop of CG.A, which uses an inverse power method to find the largest eigenvalue of a random sparse matrix. The pseudocode for its innermost work loop is:

```
p[j] = r[j] + beta*p[j]
```

As we can see, CG has no dependence between work loop iterations; we would consequently expect it to perform significantly faster than Chameleon. It does. A speedup figure in the observed 2-4x range is certainly within the bounds of what can be feasibly explained by ILP.

5.8.1 Adding ILP to Chameleon

In order to get Chameleon's performance closer to that of its target applications, more parallelism must be exposed in its work loop. The problem is that Chameleon's pointer-chasing algorithm causes each instruction to be dependent on the preceding instruction.

While the loop cannot be "unrolled" by traditional compiler optimization,

coordinated changes to both the work loop and data array can expose more parallelism. Instead of a single index pointer traversing the data array, we will need a variable number. Consider the following modification to Chameleon's work loop:

```
1            for(int i = 0; i < numMemAccesses; i+=2)
2            {
3              nextAddress1=dataArray[nextAddress1];
4              nextAddress2=dataArray[nextAddress2];
5            }
```

There is no dependence between lines 3 and 4 of the loop. Ignoring any unrolling the compiler might perform, this modification allows the system to execute two memory operations concurrently in each loop iteration. The increment value for `i` should correspond to the number of memory operations performed in each loop iteration.

In order for this new loop to perform the same memory access pattern as the original, we must interleave the data array. To illustrate, consider the following data array example:

3	5	1	6	2	7	4	0

Data Array A

In this instance, the serial version of Chameleon would start by setting `nextAddress=0` and execute as follows:

```
Iter  1:      nextAddress=dataArray[0];
Iter  2:      nextAddress=dataArray[3];
Iter  3:      nextAddress=dataArray[6];
Iter  4:      nextAddress=dataArray[4];
```

```
Iter  5:      nextAddress=dataArray[2];
Iter  6:      nextAddress=dataArray[1];
Iter  7:      nextAddress=dataArray[5];
Iter  8:      nextAddress=dataArray[7];
--------------
Iter  9:      nextAddress=dataArray[0];
Iter 10:      nextAddress=dataArray[3];
...
```

In order to maintain the same access pattern while exposing parallelism, we must interleave the data array so that each cell does not contain the value of the next index to touch, but rather, of the nth, where n is the level of parallelism.

In the 2-way parallel example, `nextAddress1` would initially be 0 and `nextAddress2` would be set to 3. We then must rearrange the data array as follows:

6	7	5	4	1	0	2	3

Data Array B

Even though the two indices now traverse the array independently, their combined, interleaved instruction stream is actually the same as the original version. An *in-order* machine executing with Array B would issue the following instruction sequence:

```
Iter 1:       nextAddress1=dataArray[0];
              nextAddress2=dataArray[3];
Iter 2:       nextAddress1=dataArray[6];
              nextAddress2=dataArray[4];
```

```
Iter 3:      nextAddress1=dataArray[2];
             nextAddress2=dataArray[1];
Iter 4:      nextAddress1=dataArray[5];
             nextAddress2=dataArray[7];
--------------
Iter 5:      nextAddress1=dataArray[0];
             nextAddress2=dataArray[3];
...
```

This is the same pattern as the original version described by Array A. However, since the instructions of each loop iteration are now independent, an out-of-order processor could overlap their execution and possibly attain significant speedup. We can generalize this technique to any level of parallelism and have done so through a compile-time ILP constant. The constant can currently be set at values from 1-20.

5.8.2 ILP Effects on Hit Rates

While out-of-order execution increases performance, it also perturbs the reference stream. To understand the nature and magnitude of the parallel memory stream's deviation from its serial counterpart, it is important to first introduce some general components and concepts of out-of-order processors. Such processors employ several techniques for exploiting ILP.

Dynamic Execution Overview

Figure 5.4 is a high-level illustration of a dynamic instruction pipeline. Although details may vary across specific implementations, the basic process is as follows: An instruction fetch stage brings instructions into the *instruction queue* and decodes them in order. Following this *in-order issue*, each instruction must

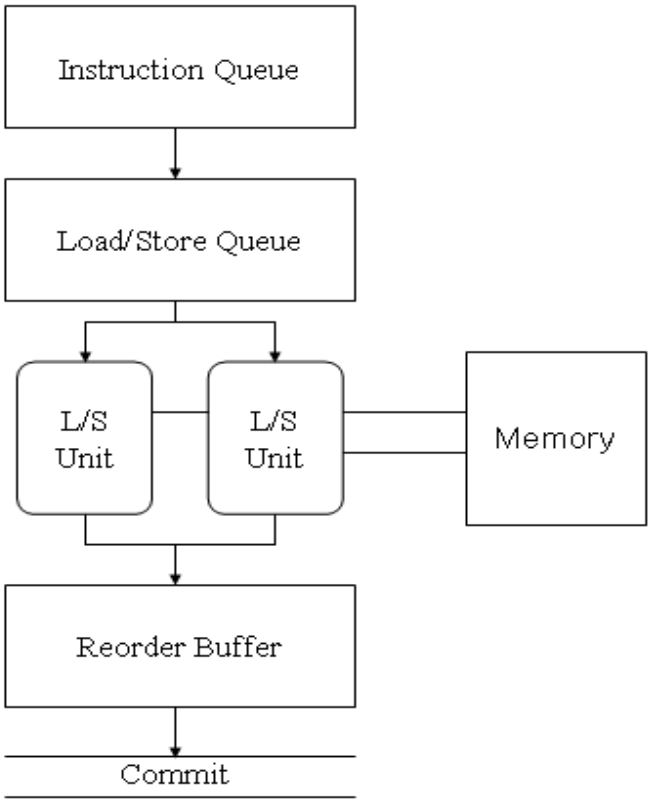

Figure 5.4: An out-of-order execution pipeline

remain in the queue until its operands become available. Because of data dependencies, some instructions may stall and allow other instructions whose operands are ready, to begin execution despite having been issued later. This out-of-order execution may freely cross loop boundaries.

When an instruction is ready for execution, it is sent to an appropriate *functional unit*, such as a floating point, integer, or memory unit. In Chameleon's case, this is always the load unit. To exploit ILP, systems may have multiple functional units of each type, pipelined functional units, or both. The D820 for example, pipelines its load and store units into two stages. Figure 5.4 does not distinguish between these two options and simply represents both as effectively being multiple load/store units.

Depending on how far up the cache hierarchy its requested information

resides, load/store instructions may require several cycles to complete. While a load/store unit waits, it may issue a concurrent request to memory on behalf of another instruction. The number of outstanding load/store instructions that a system can tolerate is variable from system to system.

The ILP capabilities of this execution phase (including replicated functional units, pipelined functional units, and multiple outstanding memory requests) allow a number of independent memory operations to proceed in parallel. They also place a cap on the level of parallelism that can be exploited by the machine.

Lastly, once a functional unit completes the execution of an instruction, it moves the instruction to a *reorder buffer*. To execute instructions out of order while maintaining correct semantics, the processor buffers completed instructions in the reorder buffer and commits them *in order* as the instructions become available. If the reorder buffer fills, then the structural hazard halts the issue of subsequent out-of-order instructions from the instruction queue.

Implications for Chameleon

Based on the description in the previous section, a N-way parallel Chameleon configuration will have exactly N memory operations in the process of execution at any given time. This is because each `nextAddressN` variable inside Chameleon's work loop is independent of the others while simultaneously dependent on its own value in the previous iteration. The operands of any such instruction are available if and only if its previous iteration has completed. The instruction queue can therefore release no more than N instructions simultaneously.

Out-of-order execution does perturb the originally intended memory stream. For example, suppose some memory reference (A) evicts the cache block needed by a later operation (B). If their execution order is inverted, B would hit in cache. This situation is known as a *hit under miss*. More abstractly, out-of-order execution can shorten the reuse distance of some memory operations.

The magnitude of this change is controlled by the reuse distribution and is capped by the size of the reorder buffer. For example, suppose a certain reorder buffer contains 256 entries. Including the loop variables, it is reasonable to assume a 2-way parallel Chameleon benchmark would require 4 instructions per loop iteration, meaning that neither `nextAddressN` index can run ahead of the other by more than 64 loop iterations. Thus, the size of the reorder buffer caps the maximum error.

This error value is quite small with respect to the number of lines in a typical cache, and as parallelism increases, this value decreases even further. It is also highly unlikely that any one index would run far ahead of the others, given that they are all derived from the same statistical distribution and operate on the same set of memory.

The other possible perturbation is a *miss under miss*. Suppose operation A accesses main memory to bring a block into cache that operation B uses. If the two operations do not complete in order, B will miss cache as well, altering the behavior expected by the Chameleon trace generator. The benchmark's hit rates in lower level caches would therefore deteriorate palpably. Despite the adverse effects on hit rates however, miss under miss scenarios do not actually *decrease* performance because the miss penalty for access B is overlapped with that of A.

Given these two sources of error, we would anticipate that the greatest effect of ILP on the Chameleon benchmark's cache behavior would be a decrease in L1 hit rates that is unaccompanied by performance degradation. To demonstrate this effect, we execute Chameleon at four different levels of ILP, targeting an arbitrary benchmark (IS.B). Table 5.3 documents the steady degradation of L1 hit rates exhibited by Chameleon as various levels of ILP are exposed.

To demonstrate that this degradation is due to miss under miss scenarios, we modify the IS.B signature to preclude any memory references with reuse distances equal to 0, instead, assigning them a value of 1. We leave the rest of the distribution unchanged.

Table 5.3: Cache hit rates of Chameleon versions targeting IS.B

Max Reuse	Parallelism	L1	L2
0	1	.58	.83
	2	.49	.84
	3	.44	.85
	4	.40	.86
1	1	.58	.83
	2	.58	.81
	3	.41	.86
	4	.43	.86

Memory operations with a reuse distance of 1, cannot cause miss under miss situations for a 2-way parallel Chameleon implementation. This is because any such reference is dependent on the value of its previous iteration. For example, consider the following access stream where "`A->1`" implies that index A is touching address 1:

```
A->1
B->3
A->5
B->1
...
```

All references with an odd-valued reuse distance refer to an iteration on which they are dependant; such reference can therefore not be executed out of order and cause a miss under miss. Because we eliminate reuse distances of 0, the smallest reuse distance that can cause this is therefore 2. However, IS.B performs almost no memory references with reuse distances between the values of 1 and 128. As more operations are executed, the probability of some earlier operation not having been completed *and* another operation accessing the same address drops

precipitously. Consequently, a miss under miss by a 2-way Chameleon instance for this trace seed is highly improbable.

Table 5.3 lists the observed cache hit rates for variously parallel Chameleon runs using the new trace seed. The L1 cache hit rate for the 2-way Chameleon is identical to that of the serial version, implying that the entire L1 degradation observed with the original seed is due to miss under miss situations.

These observations support the conclusion that the proposed ILP modifications to Chameleon can be deployed without adversely impacting the benchmark's performance and behavioral accuracy for reasonably sized caches.

5.8.3 ILP Effects on Performance

To demonstrate the performance effects of exposing additional ILP in Chameleon, this section presents runtime measurements from a series of performance tests on the Pentium D820. These results are representative of identical tests we carried out on the other two systems.

As anticipated, the addition of ILP into the Chameleon benchmark increases its performance markedly. Table 5.4 displays the memory bandwidth achieved by the Chameleon benchmark while various degrees of parallelism are exposed. The runtimes were collected using the Pentium D820 machine and each test was executed 10 times, though observed performance remained highly stable.

Depending on the application it targets, Chameleon is able to exhibit a speedup of 1.5-2.5x by exploiting parallelism. If we compare the performance results to those of the original applications presented in Table 5.2, we observe that the performance gaps have been shortened to within 10-50%.

One possible causes for the remaining performance gap is that the NAS benchmarks have an even greater degree of ILP than these versions of Chameleon. The most parallel version of Chameleon reported in Table 5.4 is 4, meaning that at any given time, at most 4 memory operations are outstanding. A benchmark

Table 5.4: Performance of Chameleon with parallelism

Target	Parallelism	MemOps/s (x10^8)	Speedup
BT.A	1	4.91	1.00
	2	6.39	1.30
	3	6.99	1.42
	4	7.51	1.53
CG.A	1	1.84	1.00
	2	2.90	1.58
	3	3.88	2.11
	4	4.54	2.47
FT.A	1	3.19	1.00
	2	4.43	1.39
	3	5.28	1.65
	4	6.02	1.89
IS.B	1	.64	1.00
	2	.90	1.40
	3	1.04	1.62
	4	1.36	2.13
LU.A	1	4.84	1.00
	2	6.31	1.31
	3	6.96	1.44
	4	7.47	1.54
SP.A	1	4.46	1.00
	2	5.97	1.34
	3	6.68	1.50
	4	7.24	1.62
UA.A	1	4.65	1.00
	2	6.16	1.33
	3	6.82	1.47
	4	7.35	1.58

such as CG.A for example, exposes the maximum level of ILP, perhaps explaining why it exhibits the greatest performance discrepancy from Chameleon of all the measured benchmarks. LU on the other hand, designed to allow somewhat less parallelism than the other benchmarks in the suite, performs at a rate within 10% of Chameleon.

Unfortunately, the Pentium D820 cannot execute Chameleon instances with more parallelism than 4 because of insufficient register space. Recall that Chameleon depends on the general purpose registers to store all the `nextAddress` indices. If too many of these indices exist to store in registers, the system is forced to use L1 cache to store the overflow, and the memory access pattern is affected. Even though the extra indices occupy a very small part of cache, their repeated access in each iteration slows the benchmark and complicates the hit rate figures reported by hardware counters or tracing.

Still, a figure of four outstanding memory references on average is a sufficient number for many real-world applications, which may have significantly less ILP than the NAS Benchmarks. Ideally, one would like to extract some ILP measure from an application during the tracing phase and set Chameleon's level to the same value. We leave exploration of this possibility to future work.

Even with the register-imposed limit on producible ILP and our uncertainty about the level of parallelism in target applications, Chameleon's tunable parallelism scheme nonetheless allows for interesting studies and insight into anticipated performance. For example, suppose we determine through tracing, or infer through observation on other machines, that the NAS Benchmarks in this section average 6 outstanding memory operations during execution. For each benchmark, we can execute Chameleon at the first four levels of ILP and extrapolate the performance anticipated when we expose 6 instructions. Using a simple least squares linear regression on the data in Table 5.4, we can extrapolate the anticipated performance as displayed in Table 5.5. This rough performance prediction is relatively accurate with respect to modern performance prediction techniques [44, 50].

Of course, we are not bound to apply a uniform 6-way estimation onto every benchmark application, but these results demonstrate that we can gain some insight into anticipated performance, even with only broad strokes.

Table 5.5: Projected Chameleon Performance with ILP of 6

	MemOps/s(x10^8)		
Benchmark	Original	Chameleon	Error
BT.A	12.1	9.38	0.23
CG.A	9.69	6.46	0.33
FT.A	7.39	8.00	0.08
IS.B	2.07	1.80	0.13
LU.A	8.25	9.38	0.14
SP.A	9.06	9.25	0.02
UA.A	9.19	9.31	0.01

5.9 Summary

This Chapter has described the Chameleon benchmark, a fully-tunable synthetic memory benchmark based on the framework's traces. The benchmark is able to replicate the memory access patterns of target applications without an index array by effectively pointer chasing through a data array in a pattern dictated by the framework's synthetic traces.

The benchmark concept requires the pointer chasing to progress in one large loop and therefore demands that Chameleon's trace generator produce synthetic traces that never reuse a given word. To enable this, the generator must choose the word nearest to that dictated by the target statistical distribution and compensate for errors by extending it's existing cold-cache miss compensation mechanisms.

Verification tests show that the cache surfaces of three NAS benchmarks are within 1.7%, 2.0%, and 1.1% of those produced by their respective Chameleon counterparts. Further, runs on a real-world system with hardware performance counters show that the Chameleon executable benchmark normally produces cache hierarchy within 5% on a set-associative cache hierarchy with nonuniform block sizes. The presented results confirm this on seven of the NAS Benchmarks.

Lastly, this chapter describes modifications to the benchmark that allow users to expose various amounts of instruction-level parallelism using a compile-time flag. The final section argues that exposing parallelism causes moderate but tolerable perturbation to the benchmark's memory access patterns and enables interesting performance studies and prediction techniques.

Chapter 6

Use Cases

The previous chapters have outlined the ideas and implementation of a practical framework for observing, understanding, and imitating the memory behavior of applications. This chapter describes a series of hypothetical use case scenarios in which the Chameleon tools could be leveraged.

6.1 Workload Selection

As described in Section 6.1, workload selection is an important problem for system benchmarking. When performance analysts look to evaluate systems or benchmarkers look to compose effective benchmark suites, both aim to choose a minimal but complete set of executables that spans the behavioral space of some target system's expected workload without being redundant. As importantly, they would like to know where in this behavioral space the chosen benchmarks reside.

The discussion in Chapter 2 of a uniform model of reference locality enables them to describe the full behavioral space of applications using cache surfaces. The Chameleon tools also enable users to capture and compare this characterization more quickly and easily than would otherwise be possible

Suppose for example, that a performance analyst is working on a large-

scale system procurement effort. He would like to evaluate a slate of several dozen possible systems and chooses the NAS Parallel Benchmark set, including BT, CG, FT, IS, LU, MG, SP, and UA. He is told however, that the staff does not have enough time to evaluate all of the systems across all eight benchmarks and that he must eliminate at least two. Without intimate knowledge of each benchmark's implementation, the analyst can use the cache surface model and Chameleon tools to narrow the evaluation workload, qualify the tradeoffs, and justify his decisions.

He begins by using Chameleon's tracing tools to extract the memory signature from each benchmark. With interval sampling and modest benchmark parameters, the extraction requires only a few minutes. A cursory review of the signature data reveals that some of the benchmarks do in fact appear similar. To confirm, the analyst would like to compare their respective cache surfaces.

He decides that a cache surface of 17 exponentially deeper and 5 exponentially wider caches should cover his systems of interest. However, it would take many hours to perform full memory traces and simulations of all 85 LRU caches for every benchmark. Instead, he uses Chameleon's trace generator to create a synthetic address trace for each memory signature and then the framework's cache surface tool to extract a cache surface from each synthetic trace. The trace generation and surface extraction phases each require only a few minutes and seconds respectively. The process creates the surfaces pictured in Figures 6.1-6.8.

The analyst can expect these synthetic surfaces to approximate those of the actual benchmarks reasonably. With this visualization, he can quickly confirm that up to half of the proposed benchmarks are really quite similar. While the first four benchmarks (CG, FT, IS, MG) each exhibit a unique cache surface, the second four (BT, LU, SP, UA) appear very much alike.

More quantitatively, the analyst may even use some similarity metric to measure the difference between the surfaces. Table 6.1 presents a simple example, reporting the average absolute difference between the analogous points of each benchmark's surface. Only the four identified benchmarks are within 5%.

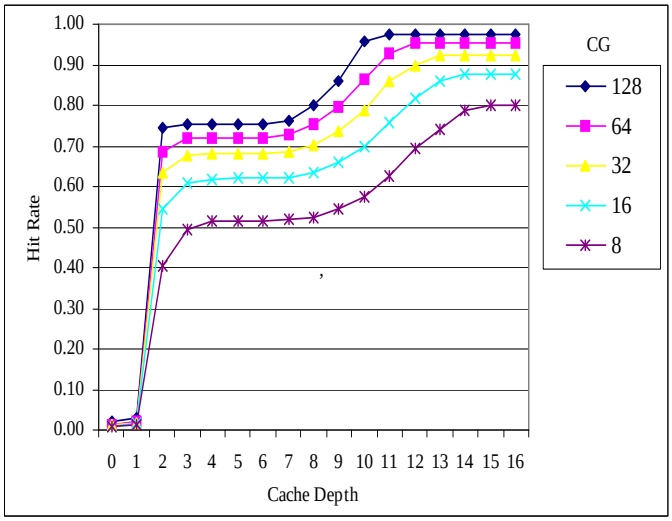

Figure 6.1: Synthetic cache surface for CG

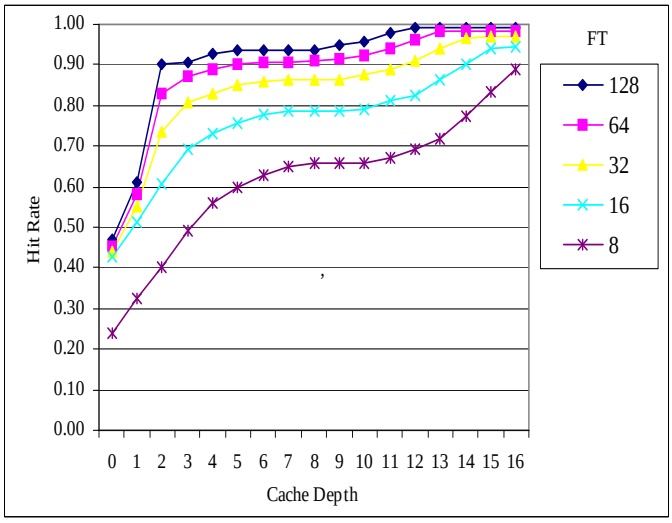

Figure 6.2: Synthetic cache surface for FT

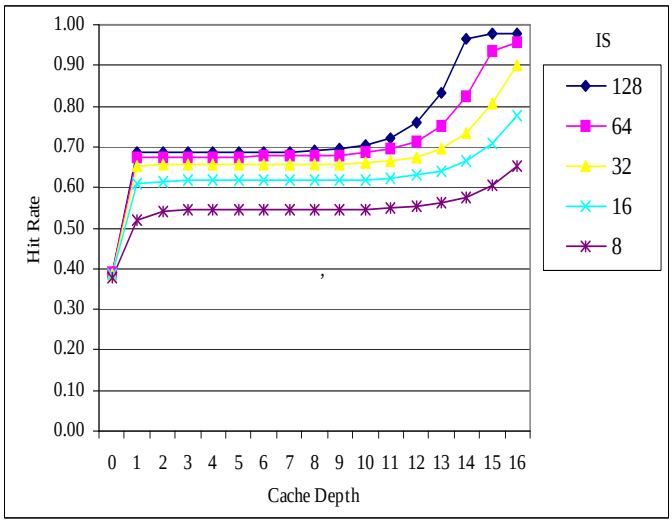

Figure 6.3: Synthetic cache surface for IS

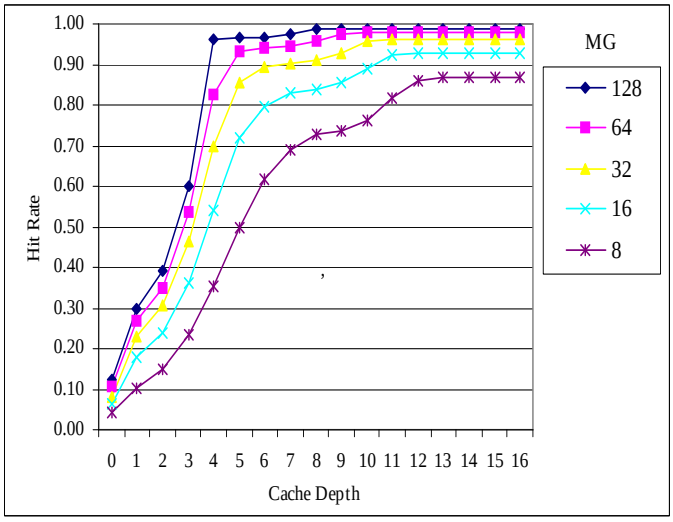

Figure 6.4: Synthetic cache surface for MG

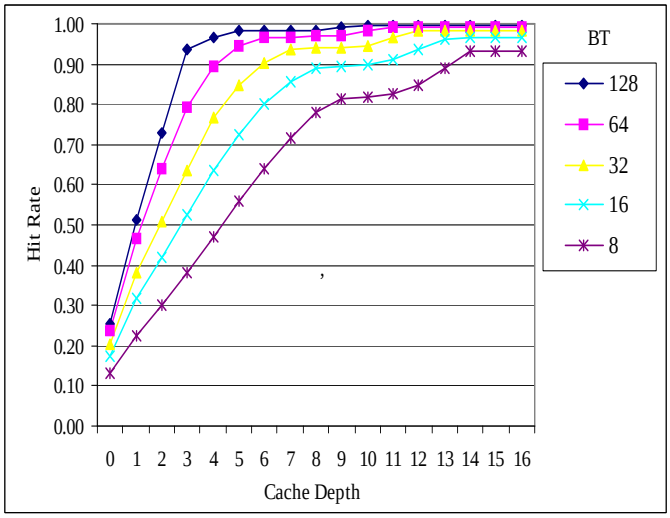

Figure 6.5: Synthetic cache surface for BT

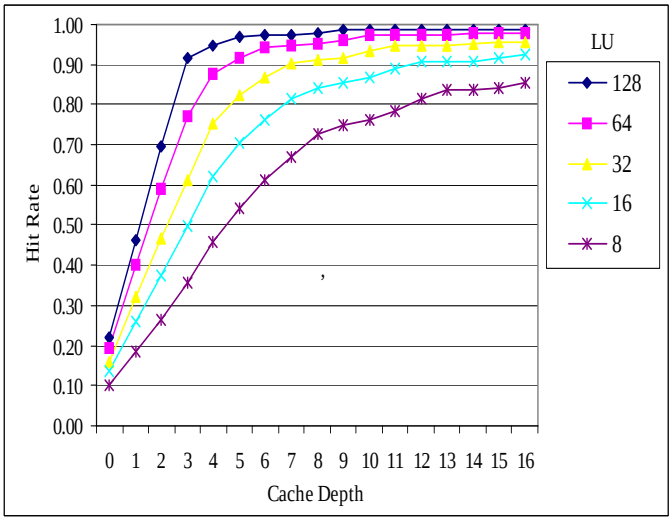

Figure 6.6: Synthetic cache surface for LU

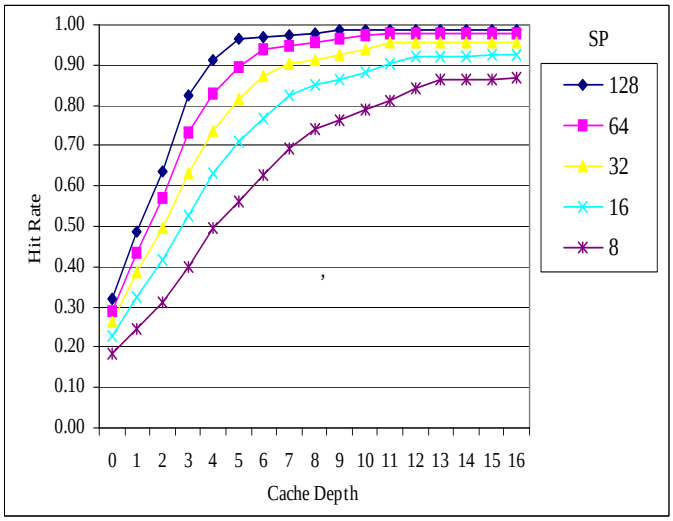

Figure 6.7: Synthetic cache surface for SP

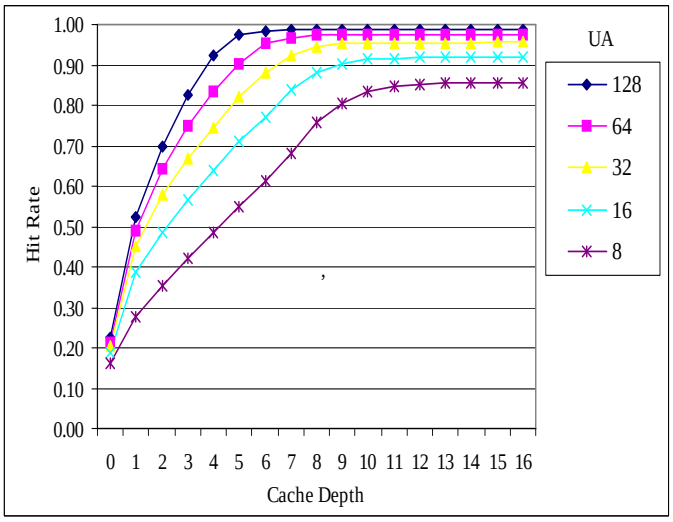

Figure 6.8: Synthetic cache surface for UA

Table 6.1: Similarity between synthetic cache surfaces

	CG	FT	IS	MG	BT	LU	SP	UA
CG	–	0.13	0.14	0.14	0.15	0.12	0.13	0.13
FT	0.13	–	0.16	0.12	0.08	0.08	0.07	0.07
IS	0.14	0.16	–	0.24	0.22	0.20	0.19	0.20
MG	0.14	0.12	0.24	–	0.06	0.05	0.05	0.06
BT	0.15	0.08	0.22	0.06	–	**0.03**	**0.03**	**0.03**
LU	0.12	0.08	0.20	0.05	**0.03**	–	**0.02**	**0.03**
SP	0.13	0.07	0.19	0.05	**0.03**	**0.02**	–	**0.02**
UA	0.13	0.07	0.20	0.06	**0.03**	**0.03**	**0.02**	–

The verified similarity between BT, LU, SP, and UA does not necessarily imply that they are all superfluous, but only that their LRU cache hit rates are highly comparable. The benchmarks may yet have differences, such as various levels of ILP, propensities for prefetching, or communication patterns when parallelized.

Fortunately, it is relatively simple to check for variation in ILP or prefetching levels among the benchmarks: the analyst need only execute them once on an arbitrary machine capable of exploiting such optimizations. Benchmarks for which these optimizations can be disproportionately leveraged will likely outperform the others. For example, the runtime observations in Table 5.2 reveal that BT is just such an outlier among the four similar benchmarks.

Having observed this on his local system, the analyst decides to keep BT and discard two of LU, SP, and UA. According to the similarity measurements in Table 6.1, SP is more similar to the other two benchmark than they are to each other. He keeps SP.

To provide some evaluation of his decision, Table 6.2 presents multi-platform performance data for the entire benchmark set. The performance measurements are in memory operations per second $x10^8$.

These observations confirm that only the three benchmarks identified by

the analyst have uniform performance characteristics on every system. The run time and hit rate measurements also confirm that SP is indeed more similar to LU and UA than those benchmarks are to one another. The cache surface comparison in Table 6.1 identifies FT as the next most similar benchmark; the measurements in Table 6.2 concur.

Table 6.2: Performance of NPB on various systems

App	Pentium D820		Intel Centrino		IBM Power4	
	L1/L2	Performance	L1/L2	Performance	L2/L3	Performance
CG.A	.66/.99	9.69	NA	5.22	.96/1.00	3.18
FT.A	.86/.97	7.39	NA	4.75	.98/.32	9.63
MG.A	.93/.98	13.9	NA	8.37	.94/.45	5.52
IS.B	.67/.85	2.07	NA	1.68	.78/.89	.11
BT.A	.96/.98	12.1	NA	7.59	.94/.96	13.0
LU.A	.94/.95	8.25	NA	5.05	.87/.94	6.64
SP.A	.93/.94	9.06	NA	5.59	.89/.93	7.94
UA.A	.91/.91	9.19	NA	5.86	.87/.93	8.13

Before breaking for lunch, the analyst has successfully trimmed redundancy from the evaluation workload and potentially saved the procurement team significant effort over the coming weeks.

6.2 System Simulation

Having selected a streamlined but complete workload, the performance team benchmarks each of the target systems and eventually settles on an IBM Power4 machine similar to DataStar. The memory hierarchy specifications are presented in Table 6.3.

After the benchmarking activities have been completed however, the vendor offers the company a new "performance option" that will upgrade the size of the

Table 6.3: IBM Power4 memory hierarchy

Level	Size	Block	Associativity	Replacement Policy
L1	32768	128	2	LRU
L2	366592	128	8	LRU
L3	8749056	512	8	LRU

system's L2 caches by 36%. Instead of a 366592-byte, 8-way L2 cache, the system can be built with a 499712-byte, 4-way associative configuration [1]. The new configuration is not available for benchmarking, but the procurement executives ask the performance analyst for guidance nonetheless. What performance returns can they expect for this investment?

Because the system is not available for benchmarking, the analyst must rely on simulation. He has a cache simulator tool on hand but needs memory traces from the evaluation workload to drive it. Collecting, storing, and replaying these traces would require terabytes of storage and days if not weeks of compute time. The executives would like his recommendation more quickly.

Instead of tracing the six evaluation benchmarks, the analyst retrieves the synthetic trace seeds he had produced weeks before during the workload selection phase. Because of the relatively small size of these files, he was able to store them on his local machine for only a few megabytes. He runs the seeds through the cache simulator and within seconds, determines that the new cache configuration would result in only a very modest performance improvement.

Table 6.4 details these results for a few benchmarks in the evaluation workload. The analyst finds no more than a 6% increase in L2 hit rates, barely within Chameleon's usual error margins. The analyst concludes that he cannot project a significant benefit from the upgrade and the executives opt for the original design.

[1] actual commercially available options for IBM Power4

Table 6.4: Simulated cache hit rates of NPB on two potential systems

Benchmark	Hit Rates (L1/L2/L3)	
	Option 1	Option 2
CG	.79 / .84 / .83	.79 / .86 / .81
IS	.79 / .16 / .96	.79 / .22 / .96
SP	.95 / .59 / .79	.95 / .65 / .75

To verify that the analyst has made the proper recommendation, we compare his numbers to those derived using the test workload's actual address streams. Table 6.5 catalogs the cache hit rates yielded when the synthetic address streams used by the analyst are replaced with each benchmark's actual trace. These results confirm the analyst's conclusion, that the 36% increase in cache size would not yield a comparable increase in performance. The L2 predictions he derived were within 0-3% of the actual values.

Table 6.5: Actual cache hit rates of NPB on two potential systems

Benchmark	Hit Rates (L1/L2/L3)	
	Option 1	Option 2
CG	.79 / .84 / .83	.79 / .86 / .81
IS	.84 / .16 / .99	.84 / .22 / .99
SP	.96 / .62 / .83	.96 / .63 / .82

6.3 Synthetic Memory Benchmarking

The company has purchased and deployed its new system when the analyst receives an email from a colleague in the performance community. The colleague inquires about the performance of this new machine with respect to that of other systems around the country. He would like some standard metrics such as the total

throughput for strided and random memory access patterns on this machine.

These two memory access patterns are standards for comparing cross-system performance. In fact, they are represented by the Stream [6] and RandomAccess [4] benchmarks in the community's HPC Challenge benchmark suite [1]. The analyst could find these benchmarks online and build, tune, verify, and execute each. However, this is not always as simple as it should be. Platform-specific compiler and hardware optimizations may alter the benchmark's intended behavior. The analyst would have to performance tune each benchmark, tinker with compiler flags, and likely even hand edit the code to ensure full performance. This effort is repeated anew for each benchmark.

Even after tuning, the benchmark's behavior may still be unclear. For example, performance counters report that the HPCC's RandomAccess benchmark, running on the Pentium D820, hits in L1 96% of the time and in L2, 10%. Is the benchmark actually performing random access? The benchmark, its parameters, the hardware, the performance counters, and the compiler are all suspects. Perhaps none are guilty and the reported throughput is correct after all.

Instead of building and tuning multiple benchmarks, the analyst decides to use the Chameleon benchmark since he is already familiar with it. He writes two memory signatures by hand to describe the strided and random memory access patterns. The random access pattern can be described by setting all of the model's parameters to 0, and the stride pattern by setting them all to 1. Chameleon also allows him to control the size of the working set and the data types to be used.

Because Chameleon's access pattern can be prescribed cleanly without index arrays or vulnerability to compiler optimizations, the resulting benchmarks are more likely to exhibit the intended behavior and report the correct results.

Table 6.6 compares the observed cache hit rates and performances of Stream, RandomAccess, and Chameleon at various levels of ILP. To mitigate error, only the work loops of each application are instrumented. These results were derived using the PAPI performance counter library on the Pentium D820.

Table 6.6: Synthetic benchmark performance on Pentium D820

Benchmark	Pattern	L1	L2	Performance (x10^8)
Stream	stride	0.55	0.81	6.42
Chameleon1	stride	0.93	0.99	5.39
Chameleon4	stride	0.65	1.00	7.56
Chameleon20	stride	0.91	0.99	9.17
RandomAccess	random	0.96	0.15	.05
Chameleon1	random	0.00	0.00	.11
Chameleon4	random	0.00	0.00	.33
Chameleon20	random	0.63	0.00	.45

We observe that the cache hit rates of the two benchmarks do not match expectation. Perhaps this is due to index arrays or other timing overheads, but we cannot know for sure without more in-depth analysis of the benchmarks.

Chameleon's hit rates match expectations more closely. For instance, the stream pattern with Chameleon1 hits within 1% of the theoretical maximum of 93.75% on the 64-byte wide L1. As 4-way parallelism is added, we see the expected L1 drop, accompanied by increased performance. We can safely increase the parallelism to 20 without fear of hit under miss augmentations because the stride pattern has no temporal reuse. We do see an increase in L1 hit rates due to the index overflow issue, but we disregard all of these memory references when calculating the total performance. The same analysis applies equally to the random access pattern. The RandomAccess benchmark makes many superfluous L1 and some L2 references while Chameleon does not.

The unexpected hit rates would not necessarily be concerning, except that the performance figures reported by the Stream and RandomAccess benchmarks are incorrect. As the Chameleon runs demonstrate, it is possible to access the same number of memory addresses, over the same workspace, using the same pattern, in less time. The array sizes and data types are identical for each pattern's tests. The slowdown of RandomAccess may be somewhat explained by it performing

only store operations, while Chameleon performs only loads. Stream only performs loads as well.

6.4 Symbiotic Space-Sharing

Thus far, our discussion has illustrated how memory modeling can inform system evaluation for benchmarking, procurement, and design efforts. It may also go beyond mere observation and actually *improve* system performance as well. Take for example, *symbiotic space-sharing*, a job scheduling technique that attempts to execute parallel applications in combinations and configurations that alleviate pressure on shared resources [68]. Memory behavior analysis can help inform this technique and improve overall system throughput.

6.4.1 Background

Symmetric multiprocessor systems (SMP), such as the three testing platforms used in this study, share memory resources among their processors to various extents. As detailed in Section 1.7, the two processor on the Pentium D820 share a single front side bus to memory. The two processors on the Centrino Duo share that bus, as well as a single L2 cache. Memory resource sharing among the processor of the Power4's 8-way nodes is even more complex: each processor receives a dedicated L1 cache, but two processors must share a L2 cache, and all eight must share L3 and bandwidth to main memory.

As one might expect, memory resource sharing leads to performance degradation. The more heavily coexisting processes make use of a shared resource, the more likely it is that the performance of that resource will suffer. Heavy use of a shared cache might lead to lower hit rates, and consequently, lower per-processor throughput.

Figure 6.9 illustrates how performance degrades on the Power4 as the pro-

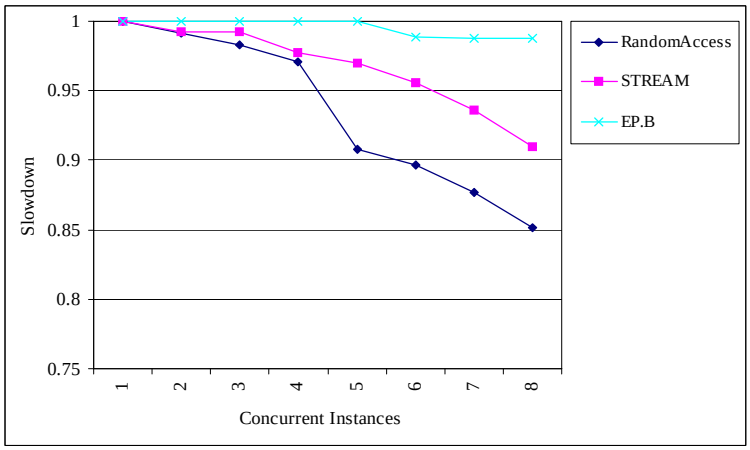

Figure 6.9: Performance degradation from memory sharing on the Power4

cessors on each node fill up. EP.B is a non-memory intensive member of the NAS benchmark set. Stream and RandomAccess are synthetic memory benchmarks from the HPCC as described in the previous section. As the 8-way node fills, the performance of memory-intensive applications degrades precipitously.

Because the consequences of resource sharing are often ill-understood, scheduling policies on production space-shared systems avoid inter-job sharing wherever possible. The scheduling policy for SDSC's DataStar system, for instance, provides jobs with exclusive use of the nodes on which they run [2]. This is not an ideal policy. Resource utilization and throughput suffers when small jobs occupy an entire node while making use of only a few processors. The policy also encourages users to squeeze large parallel jobs onto the fewest number of nodes possible since doing otherwise is both costly and detrimental to system utilization. Such configurations are not always optimal; the processes of parallel jobs often perform similar computations, consequently stressing the same shared resources and exacerbating the slowdown due to resource contention.

In such situations, a more flexible and intelligent scheduler could increase the system's throughput by more tightly space-sharing symbiotic combinations of jobs that interfere with each other minimally. Such a scheduler would need to

recognize relevant job characteristics, understand job interactions, and identify opportunities for non-destructive space-sharing.

Previous studies have demonstrated that users who declare shared-resource bottlenecks in submission scripts can improve job performance by 15-20% [69]. What is needed however, is a more precise and accessible approach.

6.4.2 Symbiotic Space-Sharing Using Memory Signatures

The challenge of symbiotic space-sharing is therefore the same as that of many performance analysis problems, requiring a practical and consequential description of memory behavior. In HPC scenarios, this description could ideally be extracted from applications by non-expert users and submitted to the scheduler along with the job. Over time, the scheduler can learn how different memory signatures interact with one another on its particular system and anticipate that similar signatures would interact similarly.

As shown in Section 6.1, Chameleon's memory signatures are comparable and can convey similarity between between the memory behavior of applications. In this section, we demonstrate that they can also describe and predict runtime interactions on shared memory resources.

Recall from Section 6.1 that four of the NAS benchmarks (BT, LU, SP, UA) have highly similar memory signatures, though only the latter three have comparable ILP. We would therefore predict that LU, SP, and UA should have comparable reactions to being coscheduled with other applications on SMP systems.

An application's space-sharing effects can be measured along two dimensions: *interference* and *sensitivity*. The former is the performance detriment it imposes on other applications and the latter is the reciprocal. We choose eight of the NAS benchmarks and execute every combination on our three target systems, measuring the sensitivity of each benchmark to every other in terms of observed slowdown.

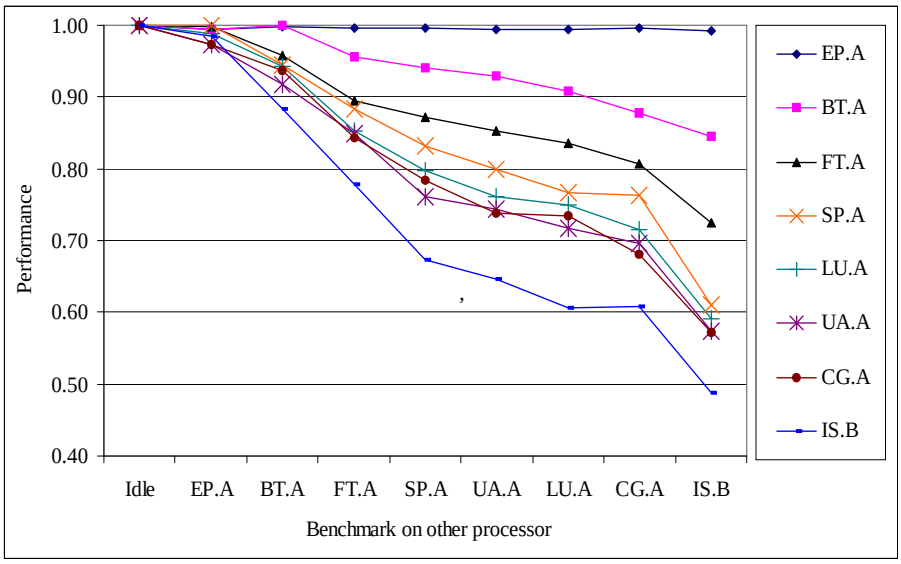

Figure 6.10: NPB performance while space-sharing on Pentium D820

Figures 6.10, 6.11, and 6.12 report these measurements for each system. The benchmarks are ordered along the X-axis by overall interference and in the legend by overall sensitivity. We plot these figures using line graphs to help accentuate emergent trends, not to imply interpolation.

While the interference and sensitivity orderings shift across machines, LU, SP, and UA are almost always adjacent in both categories. The only exception is that the sensitivities of UA on the Centrino, are slightly less aligned with those of LU and SP than are those of FT.

As we anticipate, the performance interactions grow more complex with more involved memory resource sharing. The Pentium D820, which shares only a bus to memory among its processors, produces a clean interference ordering. Presumably, all applications perform better with a cosecheduled process that makes lighter use of the shared bus. The other two systems share multiple resources, and consequently, less consensus exists.

These results imply that Chameleon's memory signature, when coupled with some knowledge of ILP, can be used to inform symbiotic space-sharing de-

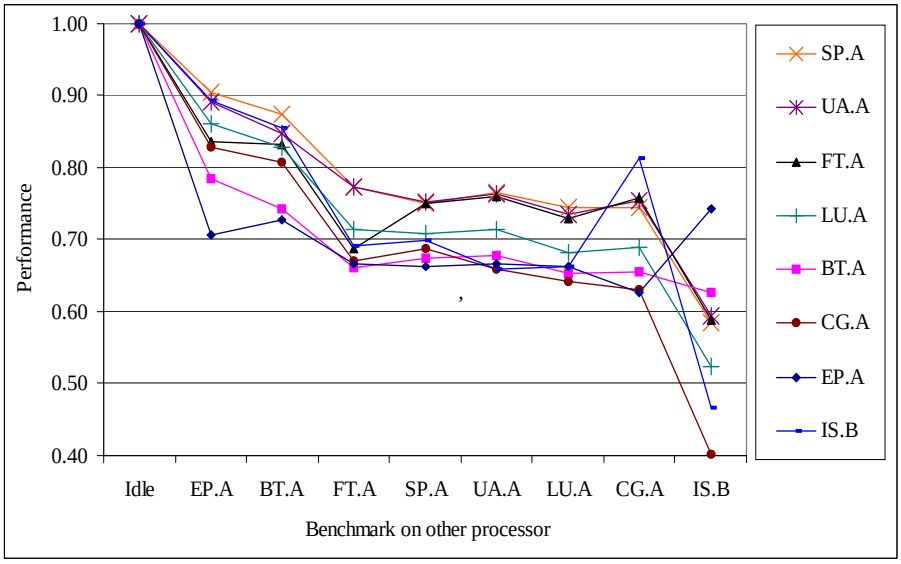

Figure 6.11: NPB performance while space-sharing on Centrino

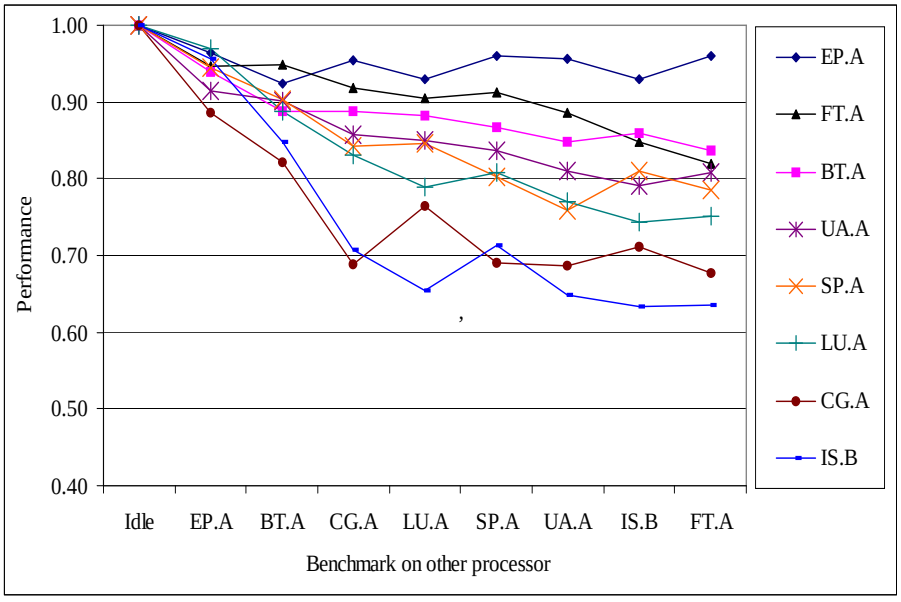

Figure 6.12: NPB performance while space-sharing on Power4

cisions. If a scheduler could obtain these signatures, it could eventually learn to identify symbiotic combinations, irrespective of the volatility of the actual workload.

Instead of relying only on the workload stream however, it may be possible to train a scheduler using parameter sweeps of the Chameleon benchmark itself. To illustrate this, let us take Chameleon's approximation of LU, which has the smallest performance error of the tests reported in Section 5.8.3. Tables 6.7 and 6.8 compare the sensitivity and interference numbers of LU.A and the Chameleon benchmark targeting that application on the Pentium D820. These measurements convey that the symbiotic properties of LU.A and its Chameleon counterpart are similar and should motivate future research investigating the possibility of using Chameleon as a space-sharing proxy.

Table 6.7: Relative sensitivities of LU and Chameleon on Pentium D820

	EP.A	BT.A	FT.A	SP.A	UA.A	LU.A	CG.A	IS.B
LU.A	0.99	0.94	0.85	0.80	0.76	0.75	0.72	0.59
Chameleon	1.00	0.94	0.88	0.78	0.76	0.74	0.73	0.57

Table 6.8: Relative interference of LU and Chameleon on Pentium D820

Benchmark	LU.A	Chameleon
EP.A	1.00	1.00
BT.A	0.94	0.93
FT.A	0.84	0.83
SP.A	0.81	0.80
CG.A	0.76	0.75
LU.A	0.77	0.79
UA.A	0.73	0.72

Chapter 7

Related Work

As described in the opening chapter, an incredible breadth of work has addressed locality modeling over the past 40 years. While a complete survey would merit its own publication, this chapter touches on some important contributions.

7.1 Locality Models and Synthetic Traces

One of the earliest reference models, the *independent reference model*, was introduced in 1971 by Denning [9, 25]. 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 an extremely popular basis for quantifying locality. Reuse distance was first studied by Mattson et. al around 1970 [45]. 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 [11, 18, 26, 32, 33]. Many works have also used reuse distance analysis for program diagnosis and compiler optimization [26, 49, 72]. The Chameleon framework distinguishes itself by eliminating error when block widths are known and by 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 [16, 15]. His model is a histogram of reference distances with a fixed cache width. The Chameleon framework also 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 [57]. Thiebaut later refined this idea by observing that stride distributions exhibit a fractal pattern governed by a hyperbolic probability function [63, 62, 64]. In recent years, the PMaC framework has focused on spatial locality but added a temporal element by including a lookback window [22, 51].

As discussed in Chapter 2, an interesting hybrid approach that fuses spatial and temporal locality into *locality surfaces* was introduced by Grimsrud [31, 30]. Sorenson later studied a refinement of this idea extensively [54, 53, 37, 56, 55]. 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 [24]. 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 [67].

7.2 Measuring Reuse Distance

Techniques for collecting reuse distance distributions have been studied extensively over the past four decades. Mattson's original approach was to use a stack [45]. Bennet and Kruskal improved this technique by using a tree structure to search for elements [14]. Later proposals include Olken's AVL tree [46], Sugumar and Abraham's splay tree [61], and Almasi's "hole" algorithms [10].

In 1991, Kim et al proposed a method similar to that used by Chameleon's tracer [39]. They used a single hashtable to locate elements in the stack and marked each element with the smallest sized cache that holds it.

Ding et al provide a useful summary of the runtime complexity of these and other proposals [26].

7.3 Tunable Synthetic Benchmarks

Work on tunable synthetic benchmarks has been somewhat scarce. Wong and Morris argued mathematically that benchmarks can be synthesized to match the LRU cache hit function when block widths are known [71]. 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 a practical framework.

More recently, Strohmaier and Shan developed the tunable memory benchmark Apex-Map [59, 60]. The benchmark accepts one spatial and one temporal parameter, allowing users to compare architectures via large parameter sweeps. However, as discussed in Section 5.2, 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.

The MultiMaps benchmark [3], also discussed in Section 5.2, is similarly able

to exhibit a range of memory access patterns. However, because it is not grounded in a general characterization of memory behavior, its relationships to other applications is nebulous. It is also difficult to determine the breadth or completeness with which it covers memory behavioral space. The Chameleon benchmark addresses these issues by basing its memory behavior directly on an observable characterization of reference locality that is capable of describing the full breadth of cache-observable memory behavior.

Chapter 8

Future Work

There are several studies and extensions to Chameleon that constitute important future work. Of these, an elegant extension to address ILP is foremost. At its core, Chameleon's memory characterization quantifies locality in memory access patterns. As such, it can predict the cache hit rates of applications. As this work has shown however, hit rates alone are not always sufficient for forecasting performance. Memory dependencies that dictate the degree of instruction-level parallelism exposed by an application to the machine can commonly alter performance by 2-4x. What is needed is a sound methodology for quantifying and extracting the level of ILP from an application. Whether gleaned from runtime or static analysis, this quantification should be added to the memory signature and integrated seamlessly with Chameleon's synthetic address trace generator.

In conjunction with the locality characterization, an ILP quantification would enable a more textured evaluation of program and architectural optimizations and interactions. As importantly, its integration would allow the Chameleon benchmark to fulfill its promise as an accessible, executable benchmark proxy for applications. Chameleon's runtime capabilities could be equally applied to simplifying performance prediction, anticipating runtime interactions of space-shared applications, and studying the comparative ILP capabilities of architectures.

Another interesting refinement of this work would be a study of the cache surface characterization itself. Given a broad workload, what is the tradeoff between the model's granularity and accuracy? For example, do we need 17 LRU caches or do 8 suffice? Perhaps these should be collected at non-uniform points that conform to the particular curves and knees of each application. Also, the granularity of α values collected for this work is certainly more fine than is needed. How well can these be compressed without losing accuracy? It is most likely that they too can be chosen at application specific sizes. It may be possible to decrease the size of the characterization to a great extent, enabling simpler comparisons.

Characterization compression may help the development of a symbiotic space-sharing scheduler. This work has suggested that such a scheduler may employ a learning algorithm to compare application signatures. It would be interesting to investigate such an algorithm and determine the relationship between the required training set size and achieved accuracy. This study would also naturally require a fuller investigation for the locality and ILP quantifications' combined ability to predict runtime space-sharing interaction. Another interesting question would be how best to quantify similarity between signatures for this purpose and how disparate these surfaces truly are.

The study could employ parameter sweeps of the Chameleon benchmark to map the interactivity between applications of various signatures. Perhaps this could lend insight not only into the ability of memory signatures to forecast symbiosis, but also into the nature of interaction on specific architectures. Perhaps a certain *type* of memory hierarchy design is less prone to detrimental interactions between common application signatures than is another.

As a pure locality model Chameleon can be applied to many novel application studies. For example, the characterization can be used to compare and study scaling behavior for large-scale applications. Strong scaling-studies, whereby analysts investigate the effects of increasing processor counts on parallel codes while holding the input data set steady, necessitate a memory behavior characteriza-

tion. Observed performances are unsteady because of the stepwise delay penalties of cache hierarchy levels. Chameleon's locality characterization can provide such studies a smooth intermediate level of abstraction, enabling interpolation and extrapolation of runtime behavior.

Chapter 9

Conclusions

This work has presented practical solutions to three pervasive problems in memory performance analysis: memory behavior characterization, accurate synthetic address trace generation, and tunable memory benchmarking with clear relationships to applications. The major contributions are:

Unified Reference Locality Model - This work describes the state of reference locality analysis in the field of memory modeling and compares the relative strengths and shortcomings of various approaches. It argues that many seemingly disjoint proposals actually converge on a single unified memory reference model and describes this model's relationship to caches and cache hit rates.

This work introduces a new definition of spatial locality, α, that suffices to characterize a memory stream's behavior as defined by the unified locality model. The α properties of applications can be tractably captured using memory address tracing and leveraged to create synthetic address traces.

Fast Memory Tracing for Locality Analysis - This work describes the implementation of two memory tracers for collecting the memory signatures of applications with minimal slowdown. One tracer, built using the Pin

instrumentation library for x86 architectures, can capture the memory signatures of serial codes with approximately 30x slowdown. A second tracer, built using the PMaCInst instrumentation library for Power architectures, can capture the memory signatures of parallel codes with approximately 5x slowdown. The interval and basic block sampling techniques used by these tracers can reduce tracing overheads by as much as two orders of magnitude without significant loss of accuracy.

This dissertation also contributes to the current body of work regarding fast LRU cache simulation by describing an algorithm and accompanying data structures for fast simulation of multiple, fully-associative, LRU caches.

Accurate Synthetic Address Traces - This work describes the implementation of a synthetic address stream generator that can convert memory signatures into synthetic address traces with cache hit rates nearly identical to those of target applications. It introduces an iterative correction technique that enables the stream generator to mimic hit rates on caches of known width without error. The stream generation technique is also the first proposal to offer tunable hit rates, simultaneously on caches of disparate widths and depths.

Using the NAS benchmarks and 68 cache configurations, the results presented in this work verify that the synthetic address traces produce LRU cache hit rates that are, on average, within 2% of those produced by the original applications. Using a cache simulation of 46 commercially available, set associative cache hierarchies, they further verify that these traces produce hit rates on real-world machines that are highly comparable to those of target applications.

Tunable Memory Benchmark - This dissertation described the implementation of the Chameleon benchmark, a fully-tunable synthetic memory bench-

mark with memory behavior that is dictated by the framework's memory signatures. The benchmark concept calls for a modified synthetic address trace generator with extended accuracy adjustment capabilities, which are described as well.

Using the NAS benchmarks, measurements reveal that Chameleon's hit rates are comparable to those of target applications on the same 68 LRU caches used to evaluate the framework's synthetic streams. Further, readings from the PAPI performance counter library confirm that the benchmark also produces comparable hit rates on a non-trivial, real world cache hierarchy.

Lastly, this work describes the implementation and evaluation of an automated scheme for exposing various degrees of instruction-level parallelism to the Chameleon benchmark through a compile-time flag.

The Chameleon Framework outlined in this dissertation, can be used to describe, compare, and mimic the memory access patterns of arbitrary serial or parallel 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 even 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.

Appendix A

Example Memory Signature

This is a sample memory signature in the format output by the Chameleon tracer. The metadata at the top reports the number of instructions simulated, the sampling rate, the maximum block width used to calibrate the temporal locality parameters. Size of word is the smallest load unit assumed by the tracer and the "using access sizes" flag designates whether the tracer performed multiple reads when more than one word was requested or treated all reads as a single word.

The bin number corresponds to the basic block sequence id for block-sampled codes. Those that were not block-sampled, such as this example, contain only a single bin.

The temporal values are reported as a series of hit rates on fully associative LRU caches of increasing size. The spatial locality parameters are reported next. $P(2^N)$ is shorthand for $\alpha(2^N, 2^{N+1})$. The reported average value is the α value for that working set size.

The value is also broken down by reuse distances to provide higher granularity. For example, the line "Reuse(2^2) = 0.497281 (3932/7907)" under "P(2^4)" indicates that memory accesses with reuse distance=R, $\{R|2^1 < R \leq 2^2\}$, have a 49.7% probability of reusing working sets of size 2^4 and that this number was derived by observing 3932 reuses out of 7907 trials.

Instructions Seen : 0x429*10^7 + 1652644414

Instructions Simulated : 55000000

Simulation limit : 4294967285

Sample rate : 0.1

Line Length (bytes) : 512

Size of Word (bytes) : 4

Using access sizes : yes

Bin: 0

TEMPORAL LOCALIY (Reuse Distance CDF):

HitRate(2^0) = 0.392172

HitRate(2^1) = 0.693671

HitRate(2^2) = 0.693817

HitRate(2^3) = 0.694109

HitRate(2^4) = 0.694694

HitRate(2^5) = 0.695869

HitRate(2^6) = 0.698217

HitRate(2^7) = 0.702929

HitRate(2^8) = 0.712338

HitRate(2^19) = 0.731167

HitRate(2^10) = 0.768718

HitRate(2^11) = 0.843716

HitRate(2^12) = 0.9887

HitRate(2^13) = 0.997397

HitRate(2^14) = 0.997397

HitRate(2^15) = 0.997505

HitRate(2^16) = 0.997505

SPATIAL LOCALITY (Probability of Set Reuse):

P(2^2):

Reuse(2^0) = 0.953847 (20078766/21050290)

Reuse(2^1) = 0.000230828 (1980/8577821)

Reuse(2^2) = 0.497281 (3932/7907)

Reuse(2^3) = 0.492609 (7798/15830)

Reuse(2^4) = 0.503193 (15916/31630)

Reuse(2^5) = 0.502397 (31966/63627)

Reuse(2^6) = 0.498101 (63348/127179)

Reuse(2^7) = 0.4999 (127630/255311)

Reuse(2^8) = 0.499974 (254791/509608)

Reuse(2^9) = 0.49957 (509334/1019545)

Reuse(2^10) = 0.500176 (1017036/2033357)

Reuse(2^11) = 0.500326 (2031886/4061126)

Reuse(2^12) = 0.500253 (3927421/7850876)

Reuse(2^13) = 0.500014 (235502/470991)

Reuse(2^14) = 0 (0/1)

Reuse(2^15) = 0.00236486 (14/5920)

Reuse(2^16) = NA (0/0)

Avg = 0.614295 (2.83073e+07/4.6081e+07)

P(2^3):

Reuse(2^0) = 0.977263 (20823354/21307827)

Reuse(2^1) = 0.654317 (8231385/12580114)

Reuse(2^2) = 0.500502 (3992/7976)

Reuse(2^3) = 0.503574 (8031/15948)

Reuse(2^4) = 0.501802 (16009/31903)

Reuse(2^5) = 0.501139 (32132/64118)

Reuse(2^6) = 0.497909 (63815/128166)

Reuse(2^7) = 0.500949 (128868/257248)

Reuse(2^8) = 0.499742 (256688/513641)

Reuse(2^9) = 0.500734 (514651/1027794)

Reuse(2^10) = 0.499773 (1024338/2049607)

Reuse(2^11) = 0.499873 (2046199/4093439)

Reuse(2^12) = 0.499824 (3955321/7913431)

Reuse(2^13) = 0.500105 (237428/474756)

Reuse(2^14) = 0 (0/1)

Reuse(2^15) = 0.00253378 (15/5920)

Reuse(2^16) = NA (0/0)

Avg = 0.739862 (3.73422e+07/5.04719e+07)

P(2^4):

Reuse(2^0) = 0.988752 (21195483/21436596)

Reuse(2^1) = 0.850998 (12408627/14581264)

Reuse(2^2) = 0.495258 (3969/8014)

Reuse(2^3) = 0.495632 (7942/16024)

Reuse(2^4) = 0.503933 (16146/32040)

Reuse(2^5) = 0.499705 (32170/64378)

Reuse(2^6) = 0.498454 (64147/128692)

Reuse(2^7) = 0.500246 (129187/258247)

Reuse(2^8) = 0.499964 (257799/515635)

Reuse(2^9) = 0.499792 (515725/1031880)

Reuse(2^10) = 0.499668 (1028133/2057634)

Reuse(2^11) = 0.499533 (2052928/4109696)

Reuse(2^12) = 0.5001 (3973178/7944769)

Reuse(2^13) = 0.500283 (238404/476538)

Reuse(2^14) = 1 (1/1)

Reuse(2^15) = 0.00253378 (15/5920)
Reuse(2^16) = NA (0/0)

Avg = 0.796013 (4.19239e+07/5.26673e+07)

P(2^5):
Reuse(2^0) = 0.994458 (21381819/21500981)
Reuse(2^1) = 0.930395 (14497269/15581839)
Reuse(2^2) = 0.497072 (3989/8025)
Reuse(2^3) = 0.502553 (8070/16058)
Reuse(2^4) = 0.506447 (16261/32108)
Reuse(2^5) = 0.501907 (32374/64502)
Reuse(2^6) = 0.500849 (64571/128923)
Reuse(2^7) = 0.499277 (129180/258734)
Reuse(2^8) = 0.501486 (259066/516597)
Reuse(2^9) = 0.499946 (516876/1033864)
Reuse(2^10) = 0.499989 (1030828/2061703)
Reuse(2^11) = 0.500061 (2059119/4117737)
Reuse(2^12) = 0.50026 (3982350/7960567)
Reuse(2^13) = 0.500632 (239044/477484)
Reuse(2^14) = 1 (1/1)
Reuse(2^15) = 0.00456081 (27/5920)
Reuse(2^16) = NA (0/0)

Avg = 0.822483 (4.42208e+07/5.3765e+07)

P(2^6):
Reuse(2^0) = 0.997288 (21474782/21533173)
Reuse(2^1) = 0.966385 (15541521/16082129)

Reuse(2^2) = 0.502802 (4038/8031)

Reuse(2^3) = 0.495054 (7958/16075)

Reuse(2^4) = 0.499347 (16052/32146)

Reuse(2^5) = 0.496872 (32082/64568)

Reuse(2^6) = 0.500899 (64642/129052)

Reuse(2^7) = 0.499836 (129444/258973)

Reuse(2^8) = 0.499123 (258120/517147)

Reuse(2^9) = 0.50067 (518126/1034865)

Reuse(2^10) = 0.499495 (1030862/2063808)

Reuse(2^11) = 0.499827 (2060175/4121773)

Reuse(2^12) = 0.500315 (3986662/7968307)

Reuse(2^13) = 0.500869 (239383/477935)

Reuse(2^14) = 1 (1/1)

Reuse(2^15) = 0.00422297 (25/5920)

Reuse(2^16) = NA (0/0)

Avg = 0.835217 (4.53639e+07/5.43139e+07)

P(2^7):

Reuse(2^0) = 0.9987 (21521263/21549269)

Reuse(2^1) = 0.983556 (16063704/16332273)

Reuse(2^2) = 0.500062 (4017/8033)

Reuse(2^3) = 0.497045 (7990/16075)

Reuse(2^4) = 0.497917 (16015/32164)

Reuse(2^5) = 0.498754 (32219/64599)

Reuse(2^6) = 0.501196 (64719/129129)

Reuse(2^7) = 0.500843 (129772/259107)

Reuse(2^8) = 0.499872 (258624/517380)

Reuse(2^9) = 0.499621 (517287/1035358)

Reuse(2^10) = 0.499422 (1031200/2064785)

Reuse(2^11) = 0.500088 (2062276/4123830)

Reuse(2^12) = 0.499808 (3984576/7972208)

Reuse(2^13) = 0.500159 (239176/478200)

Reuse(2^14) = 1 (1/1)

Reuse(2^15) = 0.00489865 (29/5920)

Reuse(2^16) = NA (0/0)

Avg = 0.841441 (4.59329e+07/5.45883e+07)

P(2^8):

Reuse(2^0) = 0.999411 (21544626/21557317)

Reuse(2^1) = 0.991941 (16324715/16457346)

Reuse(2^2) = 0.50504 (4058/8035)

Reuse(2^3) = 0.500435 (8046/16078)

Reuse(2^4) = 0.501943 (16148/32171)

Reuse(2^5) = 0.499745 (32287/64607)

Reuse(2^6) = 0.501208 (64733/129154)

Reuse(2^7) = 0.499639 (129494/259175)

Reuse(2^8) = 0.499385 (258431/517499)

Reuse(2^9) = 0.499831 (517625/1035600)

Reuse(2^10) = 0.499119 (1030813/2065266)

Reuse(2^11) = 0.499513 (2060434/4124889)

Reuse(2^12) = 0.49974 (3985005/7974158)

Reuse(2^13) = 0.500596 (239451/478332)

Reuse(2^14) = 1 (1/1)

Reuse(2^15) = 0.00743243 (44/5920)

Reuse(2^16) = NA (0/0)

Avg = 0.844503 (4.62159e+07/5.47255e+07)

References

[1] HPC Challenge: http://icl.cs.utk.edu/hpcc/.

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

[3] http://www.sdsc.edu/pmac/projects/mmaps.html.

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

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

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

[7] Top500: http://www.top500.org.

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

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

[10] 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.

[11] 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.

[12] J. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[13] 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.

[14] B. T. Bennett and V. J. Kruskal. Lru stack processing. *IBM Journal for Research and Development*, pages 353–357, July 1975.

[15] 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.

[16] 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.

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

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

[19] D. C. H. G. M. P. Browne, S. Papi: A portable interface to hardware performance counters. June 1999.

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

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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.

[27] J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark: Past, Present and Future. *Concurrency: Practice and Experience*, 15:803–820, 2003.

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

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

[34] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco, CA, 1990.

[35] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. Technical Report CS-TR-1994-1207, 1994.

[36] 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.

[37] L. K. John and A. M. G. Maynard, editors. *Using Locality Surfaces to Characterize the SPECint 2000 Benchmark Suite*. Kluwer Academic Publishers, 2001.

[38] A. K. Jones. Modernizing high-performance computing for the military. *IEEE Computational Science and Engineering*, 03(3):71–74, 1996.

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

[46] F. Olken. Efficient methods for calculating the success function of fixed space replacement policies. Master's thesis, University of California, Berkeley, Berkeley, California, May 1981.

[47] 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.

[48] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 24–35, Washington, DC, USA, 1996. IEEE Computer Society.

[49] 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.

[50] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A Framework for Application Performance Modeling and Prediction. In *Supercomputing 02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–17, Los Alamitos, CA, 2002. IEEE Computer Society Press.

[51] A. Snavely, N. Wolter, and L. Carrington. Modeling Application Performance by Convolving Machine Signatures with Application Profiles. In *Proceedings*

of IEEE 4th Annual Workshop on Workload Characterization, pages 128–137, December 2001.

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

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

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

[55] 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.

[56] 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.

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

[58] 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.

[59] 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.

[60] 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.

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

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

[63] 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.

[64] 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.

[65] 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.

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

[67] 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.

[68] 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.

[69] J. Weinberg and A. Snavely. User-Guided Symbiotic Space-Sharing of Real Workloads. In *The 20th ACM International Conference on Supercomputing (ICS '06)*, June 2006.

[70] 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.

[71] 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.

[72] Y. Zhong, C. Ding, and K. Kennedy. Reuse distance analysis for scientific programs. In *Proceedings of Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, Washington DC, March 2002.

[73] 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.