

UNIVERSITY OF CALIFORNIA, SAN DIEGO

**Quantifying Locality In The Memory Access Patterns of HPC  
Applications**

A thesis submitted in partial satisfaction of the  
requirements for the degree  
Master of Science

in

Computer Science

by

Jonathan Weinberg

Committee in charge:

Professor Allan Snavely, Chair  
Professor Sid Karin, Cochair  
Professor Lawrence Carter

2005

Copyright  
Jonathan Weinberg, 2005  
All rights reserved.

The thesis of Jonathan Weinberg is approved:

---

---

Co-Chair

---

Chair

University of California, San Diego

2005

To Marylee, whose boundless love and tireless faith could move mountains but inexplicably chose to produce this thesis instead.

## TABLE OF CONTENTS

Signature Page . . . . .	iii
Dedication . . . . .	iv
Table of Contents . . . . .	v
List of Figures . . . . .	vi
List of Tables . . . . .	vii
Acknowledgements . . . . .	viii
Abstract of the Thesis . . . . .	x
1 Introduction . . . . .	1
2 Quantifying Locality . . . . .	4
2.1 Spatial Locality . . . . .	5
2.2 Temporal Locality . . . . .	7
3 Measuring Locality . . . . .	10
3.1 Performance . . . . .	11
4 Results . . . . .	14
4.1 HPCC . . . . .	14
4.2 Apex-MAP . . . . .	17
4.2.1 Locality Scoring of Apex-MAP . . . . .	18
4.2.2 Apex-MAP To Other Codes . . . . .	22
4.3 Large Parallel Applications . . . . .	24
5 Other Uses . . . . .	30
6 Related Work . . . . .	33
7 Conclusions and Future Work . . . . .	35
Bibliography . . . . .	37

## LIST OF FIGURES

3.1	Original overheads of tracing CG.A.1 with variously sized caches. . .	12
3.2	Optimized overheads of tracing CG.A.1 with variously sized caches	13
4.1	Locality scores of the HPCC and NPB benchmarks . . . . .	15
4.2	Temporal locality functions of HPCC and NPB benchmarks . . . . .	16
4.3	Apex-MAP performance surface on a superscalar machine . . . . .	17
4.4	Apex-MAP performance surface on a vector machine . . . . .	18
4.5	As K increases, the temporal locality of Apex-MAP decreases . . . . .	19
4.6	Mapping of K to temporal Score . . . . .	20
4.7	Mapping of L to spatial score . . . . .	20
4.8	Apex-MAP shows increased performance with increased locality . . . . .	22
4.9	The temporal reuse curves of each phase of AVUS . . . . .	27
4.10	The temporal reuse curves of each phase of HYCOM . . . . .	27
4.11	The temporal reuse curves of AVUS and HYCOM compared to the HPCC Benchmarks . . . . .	28
4.12	The locality scores of AVUS and HYCOM are within the bounds of the HPCC benchmarks . . . . .	28
5.1	Locality scoring captures how the memory work of CG changes with problem size . . . . .	31
5.2	Temporal reuse functions of each processor in a 16 processor run of HPL . . . . .	32

## LIST OF TABLES

4.1	Locality scores and performance of Apex-MAP as $\mathbf{L}$ and $\mathbf{K}$ vary. Each of the three vertical sections corresponds to $\mathbf{L} = \{1,2,4\}$ respectively while the rows inside each section correspond to $\mathbf{K} = \{.001, .01, .05, .1, .5, 1\}$ respectively. . . . .	21
4.2	Mapping Apex-MAP to CG.A Yields Similar Performance . . . . .	24

## ACKNOWLEDGEMENTS

I am very much indebted to Allan Snavely who has closely advised me on this work and whose thoughtful guidance and friendship has pulled me up this hill.

I would like to also thank Michael McCracken for his selflessness, insight, and friendship which has helped immeasurably in developing these ideas and the tools by which they were exercised.

I also owe a great thanks to Arun Jagatheesan without whose support, encouragement, and good-heartedness this work would have ended before ever beginning.

Laura Carrington has been instrumental in maintaining and deploying the production version of Metasim, enabling the collection of the data presented here. Without her diligence and professionalism, many of the results presented in this work would not have been possible.

I am indebted to Xiaofeng Gao for sharing his expansive expertise and experience with the tools utilized in this study.

This thesis was much influenced by Larry Carter's thoughtful input, creativity, and signature frankness.

I would further like to thank Sid Karin for his service on my thesis committee and his genuine interest in this work, as well as Erich Strohmaier, Bob Lucas, John McAlpin, David Koester, Jeffrey Vetter, and Pedro Diniz for their interest and invaluable input.

This work was funded in part by NSF awards CNS-0406312, the DOE Office of Science through the award entitled HPCS Execution Time Evaluation, by NSF NGS Award #0406312: Performance Measurement & Modeling of Deep Hierarchy Systems, and by the Department of Energy Office of Science through SciDAC award High-End Computer System Performance: Science and Engineering. Computer time was provided by the Pittsburgh Supercomputing Center via an NSF NRAC award.

This thesis, in part, is a reprint of the material as it appears in: J. Weinberg, M. O. McCracken, A. Snavely, E. Strohmaier. Quantifying Locality In The Memory

Access Patterns of HPC Applications. In *Supercomputing 2005, Seattle, WA., November 12-16, 2005*. The thesis author was the primary researcher and author of this publication.

ABSTRACT OF THE THESIS

**Quantifying Locality In The Memory Access Patterns of HPC  
Applications**

by

Jonathan Weinberg

Master of Science in Computer Science

University of California San Diego, 2005

Professor Allan Snavely, Chair

Several benchmarks for measuring the memory performance of HPC systems along dimensions of spatial and temporal memory locality have recently been proposed. However, little is understood about the relationships of these benchmarks to real applications and to each other. We propose a methodology for producing architecture-neutral characterizations of the spatial and temporal locality exhibited by the memory access patterns of applications. We demonstrate that our results track intuitive notions of locality on several synthetic and application benchmarks. We employ the methodology to analyze the memory performance components of the HPC Challenge Benchmarks, the Apex-MAP benchmark, and their relationships to each other and other benchmarks and applications. We show that our methodology can be applied to scoring real large-scale parallel applications and that this analysis can be used to both increase understanding of the benchmarks and enhance their usefulness by mapping them, along with applications, to a 2-D space along axes of spatial and temporal locality.

# Chapter 1

## Introduction

Machine performance in the arena of supercomputing has traditionally been tracked by The Top500 List [2], a ranking of the world's fastest supercomputers based on the peak floating point operations per second that each can achieve on the LINPACK benchmark [17]. This ranking system has dominated performance comparisons of supercomputers for over a decade and today serves as a unique historical record of the technological evolution of supercomputing.

Despite its longevity, the Top500 list has garnered an increasingly vocal and growing set of critics who justifiably protest that the LINPACK benchmark only stresses machines in a very limited and unrealistic manner, that is, by measuring the peak floating point rate on a compute bound code. Since 1993 when the Top500 project was established, the exponentially widening gap between processor and memory speeds has only exacerbated the problem. Application runtimes today are increasingly dominated by memory operations, a phenomenon that the LINPACK benchmark has failed to capture.

To provide a more complete understanding of comparative machine performance, particularly memory subsystem performance, several new benchmarks have been proposed including components of the HPC Challenge (HPCC) Benchmarks [1] and Apex-Map [30]. A guiding principle in the design of these benchmarks is that the memory subsystem should be stressed along two dimensions, representing

various degrees of spatial and temporal memory locality. The HPCC benchmark suite addresses this requirement with multiple memory benchmarks, each of which presumably represents extreme combinations of spatial and temporal locality in its memory access patterns. Apex-MAP is a synthetic probe that directly tackles the locality concept by performing data movement operations in accordance with parameterized degrees of spatial and temporal locality.

These benchmarking efforts represent important progress towards the establishment of performance probes that more accurately mimic the behavior of today’s applications. However, little is known concretely of the relationships between these probes and the applications they are intended to help us understand. While some components of the HPCC benchmark suite are intended to exhibit “low” spatial locality and another “high”, we cannot quantify “how much” each exhibits or how that quantity compares to a real application of interest. While Apex-MAP can hypothetically be parameterized to imitate the spatial and temporal locality of any application, no satisfactory method exists for easily determining what those parameters might be for a given code. The ability to measure the degree of spatial and temporal locality exhibited by an application or a benchmark, and to relate that to these benchmarks, could go a long way towards understanding these relationships and making the benchmarks more practically useful.

In addition to enabling benchmark evaluation, measurable metrics of spatial and temporal locality can have much to offer the greater understanding of application performance, particularly with respect to the deep memory hierarchies of today’s machines [4, 3, 7, 25]. Such analysis can lend insight into the memory requirements of given applications as input parameters or processor counts scale, be used as part of a performance model to predict memory subsystem performance, or even expose load imbalance issues with respect to workload difficulty on parallel codes.

Section 2 of this paper proposes quantifications of spatial and temporal locality scores whereby we start with classic definitions and arrive at concrete metrics that can be measured of real parallel applications.

Section 3 describes how we actually measure the locality of applications using the Metasim tracer and describes the performance of these techniques.

Section 4 displays results from our locality analysis performed on several application-based and synthetic benchmarks; included are spatial and temporal scores for memory benchmarks from HPCC and then a “recipe” for finding the mapping between an arbitrary HPC program and Apex-MAP input parameters.

Section 5 explores additional uses for locality measurement of HPC applications.

Section 6 reviews and summarizes the area’s rich literature on which we have built. Finally, we discuss our conclusions and ongoing work.

This chapter, in part, is a reprint of the material as it appears in: J. Weinberg, M. O. McCracken, A. Snavely, E. Strohmaier. Quantifying Locality In The Memory Access Patterns of HPC Applications. In *Supercomputing 2005, Seattle, WA., November 12-16, 2005*. The thesis author was the primary researcher and author of this publication.

## Chapter 2

# Quantifying Locality

To analyze applications for memory locality, we desire simple, measurable, and architecture-independent metrics for quantifying both spatial and temporal locality. Locality is well defined as a notion in the computer architecture literature. Bunt recently summarized findings from his 20- year career [12], during which he and colleagues have defined locality metrics and also proposed ways to measure them. We start from these definitions and are guided by much previous work further described in the related work section. Nevertheless, it will become apparent that some concrete and arbitrary choices have to be made when implementing the formal definitions about how to count locality statistics, and some approximations have to be made to make the data acquisition process tractable for HPC applications. We work through those details.

In our study of parallel applications, we are focused on the memory reference patterns of individual processors to their local memory (addresses sent through the load-store units). It is true that spatial and temporal locality exist in messages and inter-processor communication and some components of the HPCC test these. We do not treat that here, although a straightforward extension can be considered for future work.

After gathering detailed statistics about spatial and temporal locality, there is a temptation to reduce this information into a single score per-loop or even per

application so that one can make broad comparisons such as “application A has more temporal locality than B”. This reduction can be useful but may potentially oversimplify things and throw information away. We propose single-number spatial and temporal locality scores and show their uses and limitations in what follows.

Philosophically, we think it is important to suppress the urge to customize these metrics to proportionately track some observable phenomena such as application performance or theoretical cache hit rates. This sort of mapping between locality and observable phenomena on a particular machine can be performed more effectively at a later stage; the metrics need only track intuitive notions of locality in an internally consistent fashion. We explore the implication below.

## 2.1 Spatial Locality

Spatial locality is the tendency of applications to access memory addresses near other recently accessed addresses [11, 19]. We use a definition similar to that found in [11] to define a spatial locality metric based on the average length of non-zero strides performed by an application. The first practical, arbitrary choice we have to make stems from the interleaved nature of address streams under dynamic execution. For example, a loop could have a stride 1 reference to an index array followed by a random access (indirect) reference to another array using the index; if we only look at two consecutive dynamic addresses for spatial locality, we may miss the fact that the stride 1 address stream (every-other address) has spatial locality. We therefore define the stride of a memory access to be the minimum absolute distance, in 64-bit words, of that memory reference to its nearest neighbor among the  $\mathbf{W}$  previously accessed addresses. The “look back window” that contains these previous  $\mathbf{W}$  addresses should be sized large enough to capture the memory behavior of most reasonably sized application loops but small enough that we can search it with each new reference in a reasonable amount of time. In our study, we have chosen to set  $\mathbf{W} = 32$ . This choice is admittedly arbitrary and small but improves the performance of our data collection methods described in section 3.

Once we have defined stride, we can formulate the following simple summation to represent a single-number spatial locality score where  $stride_i$  denotes the fraction of total dynamic memory operations that are of stride length  $i$ :

$$\sum_{i=1}^{\infty} stride_i/i \quad (2.1)$$

The idea is simply to generate a normalized score in the range  $[0,1]$  that is inversely proportional to the average stride length. An application that performs only stride 1 references receives a score of 1, an application that performs only stride 2 references receives a score of .5, an application whose memory references are evenly split among stride 1 and 2 receives a score of .75, and so forth.

Notice that the summation does not include stride 0 references. By this definition, spatial locality is the tendency of an application to reference memory addresses near *other* recently referenced addresses, not the same ones. We consider stride 0's to be the simplest case of temporal locality, not a degenerate case of spatial. This is another concrete but arbitrary choice.

Even though there is technically no termination for the series, it does converge at some  $i$  where all subsequent terms in the series are zero for real applications. The summation is a harmonic series that, in practice, can actually be terminated arbitrarily if we accept the maximum possible error to be  $(1-score)/(i+1)$ , as shown in Proof 2.1.

*Proof. 2.1. The score,  $S$ , obtained by terminating the spatial scoring summation at  $i = t$  has a maximum possible error of  $S' = (1 - S)/(t + 1)$ .*

Let  $F = \sum_{i=1}^t stride_i$  and  $S = \sum_{i=1}^t stride_i/i$ . It is trivial to see that  $S \leq F$ .

Let  $F' = \sum_{i=t+1}^{\infty} stride_i$  and  $S' = \sum_{i=t+1}^{\infty} stride_i/i$ . It is likewise trivial to see that  $S' \leq F'/(t + 1)$ .

Because  $F + F' = 1$ , it follows that  $S' \leq (1 - F)/(t + 1)$ . Since  $S \leq F$ , the total possible error  $S' \leq (1 - S)/(t + 1)$ .

□

For the studies presented in this paper, we have chosen to terminate the summation at  $\mathbf{i}=8$ , a value with whose error bound we are comfortable. Further, in our experience, few real codes regularly exhibit much longer strides. We henceforth refer to this variable, the maximum stride length considered in the summation, as **T**.

## 2.2 Temporal Locality

Temporal locality is the tendency of an application to reference the same memory addresses that it referenced recently [19]. Many memory locality studies, further detailed in the related work section, have focused on this type of locality by analyzing statistics related to reuse distance. The reuse distance of some reference to memory address A is the number of unique memory addresses that have been accessed since the last access to A.

We begin our analysis by collecting information about the distribution of reuse distances in an application run. We graph this distribution in what we call the application’s temporal reuse function. The reuse function plots reuse distances against the percentage of an application’s dynamic memory operations with reuse distances less than or equal to that distance. Section 4 contains several examples of such graphs, such as Figure 4.2.

Producing a single score from this data is somewhat less straightforward than is the case for spatial locality. While in the spatial case the arrangement of memory suggests intuitive meanings and consequently a natural weighting for “nearest-neighbor” and “next-nearest neighbor”, no such natural meaning attaches to reuse distance bins. One approach is to choose a point on the graph that corresponds to a particular size of cache and formulate the temporal score as the fraction of total memory accesses with reuse distance less than that size. However, we would then be bound to a specific architecture and forced to always qualify an application’s temporal score with a cache size. This may be of some limited use for predicting cache hit rates [22] but does not adequately summarize the whole distribution.

Further, such a metric is difficult to apply to cache-less, vector-based machines.

An application's reuse function contains information that is inevitably lost if that function is summarized into a single score. This is certainly also true of stride information with spatial locality scoring, but in that case, we can utilize a natural weighting to summarize the information intuitively. In the absence of such a natural weighting, we have generally chosen to maintain the entire distribution. However, in the interest of a single normalized metric for comparing reuse functions, we formulate a metric similar to that of spatial locality where the cumulative fraction of memory operations at each reuse distance is scaled by some increasing function of that reuse distance and summed. For illustrative purposes, one can visualize this as the area under the temporal reuse function, plotted on some scale, and normalized by total area on the graph. The fundamental idea here is that since the reuse function is monotonically increasing, each memory access increases the application's total score in a manner inversely proportional to its reuse distance. The result is a metric that recognizes more temporal locality as an application contains more memory references with lower reuse distances. Again the range of these scores is  $[0,1]$  with 0 indicating no temporal locality and 1 indicating that all memory references are within the shortest reuse distance measured.

An important parameter is the weighting to assign to memory references at each reuse distance. A best weighting is not immediately obvious. There are many intriguing choices for this scale, including one that scales references at each reuse distance by its respective expected or typical cache latency. Our goal however, need not be to create metrics that proportionately track some observable phenomenon like performance or cache hits, but ones that simply track our intuitive perceptions of locality. The mapping onto observable phenomena can always be performed at a later stage. For simplicity this initial study employs a log scale where each memory reference is weighed by the log of its reuse distance with respect to the largest distance considered. A simple mechanism for visualizing this is the space under the reuse curve, where the reuse distance axis is plotted on a log scale. Again, this choice is subjective and we have provided full temporal analyses should the

reader choose to investigate an alternative scale.

The other variable we must consider before practically measuring or approximating such a metric is the size of the largest reuse distance to include. This choice will affect the scoring because it dictates the normalizing value, i.e. the total area on the graph. We must choose to integrate from 0 to some reuse distance  $N$ , where there is no compelling value at which to standardize  $N$ . This consideration is of only limited concern when we compare applications on the same graph, but it does preclude our using the metric as an unqualified application attribute if no standard exists.

We consequently formulate the following summation, where  $reuse_i$  denotes the fraction of dynamic memory operations with reuse distance less than or equal to  $i$ .

$$\frac{\sum_{i=0}^{\log_2(N)-1} ((reuse_{2^{i+1}} - reuse_{2^i}) * \log_2(N) - i)}{\log_2(N)} \quad (2.2)$$

One factor to remember is that this metric does not *necessarily* track memory subsystem performance. It is indeed possible for two applications to yield identical temporal scores but perform very dissimilarly on a particular machine. For example, on a cache-based machine, the application whose reuse function has a higher value at the reuse distance corresponding to the machine's cache size would have a performance edge. To predict performance on a known cache-based architecture, a better approach might be to examine the value of each application's reuse curve at the point corresponding to that architecture's cache size, as is done in [22].

This chapter, in part, is a reprint of the material as it appears in: J. Weinberg, M. O. McCracken, A. Snively, E. Strohmaier. Quantifying Locality In The Memory Access Patterns of HPC Applications. In *Supercomputing 2005, Seattle, WA., November 12-16, 2005*. The thesis author was the primary researcher and author of this publication.

# Chapter 3

## Measuring Locality

We gather memory access characteristics using the Metasim tracer [27], a tool for dynamic analysis of a program’s memory references, developed as part of a framework for memory-centric performance prediction. The tracer collects statistics about memory strides and simulates cache behavior as the program runs.

In Metasim, instrumentation is added around each memory reference using a binary rewriting tool such as Atom [29] or Dyninst [10]. As the instrumented binary runs, the tracer compares each address in a basic block to a window of previous addresses to capture stride information. The stride of an address is defined as in section 2.1. The size of the look-back window,  $\mathbf{W}$ , is configurable.

Metasim produces a detailed report of the collective stride information in each basic block. We use this count to generate the spatial locality score as discussed in section 2.1 for each basic block in the program. The score for the whole program is a sum of each block’s score, weighed by the percentage of dynamic memory references the block had generated.

In addition to the stride calculation described above, Metasim runs each address through a set of cache simulators as each is encountered. For performance prediction, this is used to generate cache statistics for each target architecture. We exploit this feature to collect reuse distance distributions by simulating a series of variously sized *temporal caches*, caches with a 1-word block size. The percentage

of memory operations with reuse distance less than or equal to  $\mathbf{N}$  is the hit rate in an  $\mathbf{N}$ -line temporal cache. One should note that our implementation is somewhat inexact because Metasim simulates caches with a random replacement policy instead of LRU (Least Recently Used). This may create some small noise in the data, i.e. it is possible that an address is evicted before  $\mathbf{N}$  memory references to other addresses have transpired. Practically however, we observed the hit rates of these caches are not very different on real address streams whether LRU or random replacement policy is used while random is faster and uses less memory in our simulations.

### 3.1 Performance

The slowdown yielded by the Metasim tracer has been shown to be within two orders of magnitude of the original code [26]. We benefit from existing work done to improve tracing speed in Metasim via sampling methods that reduce the overhead of memory instrumentation [13]. Current research on Metasim performance enhancement is continually lowering tracing time and today, traces are usually within a single order of magnitude. The Performance Modeling and Characterization (PMaC) group at the San Diego Supercomputer Center regularly employs Metasim to trace large-scale, highly parallel, scientific codes [14].

During our studies however, we found that simulating temporal caches did introduce some difficulty for the existing Metasim implementation. More specifically, temporal caches are fully associative and it is time consuming for Metasim to identify a cache hit or miss. To compound the problem, temporal caches often contain many more lines than realistic caches. For example, the largest temporal cache we use to produce the results in Chapter 4 contains 131,072 cache lines in its single set while the largest sets Metasim had been simulating prior were of length 128, and the second largest a quarter of that. A linear search through hundreds of thousands of elements for each memory reference is prohibitively expensive and quickly dominates the tracing runtimes as shown in Figure 3.1.

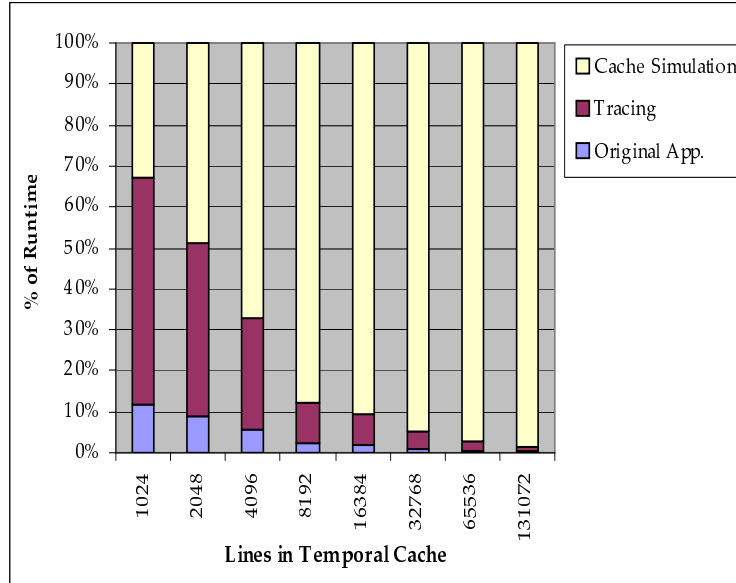


Figure 3.1: Original overheads of tracing CG.A.1 with variously sized caches.

To avoid this linear search, we introduce an optimization to Metasim aimed at flattening high-associativity caches. We use a hashtable to store the cache line number of any block in a cache, keyed by its block address. To decrease the size of the memory working set, all high-associativity caches share a single hashtable instance and the cache’s name is included in the key. Once the caches fill, the hashtable always holds exactly one entry for each cache line in the simulated caches and therefore remains comparatively small. To remove the overhead introduced by the dynamic memory management of the hashtable, we exploit the fact that, except on a compulsory miss, memory addresses are never added to the hashtable without another being removed. When this happens, the hashtable implementation uses the previously released node to store the next entry, eliminating the need to allocate or deallocate memory on non-compulsory misses. The optimized overheads are far more palatable as shown in Figure 3.2.

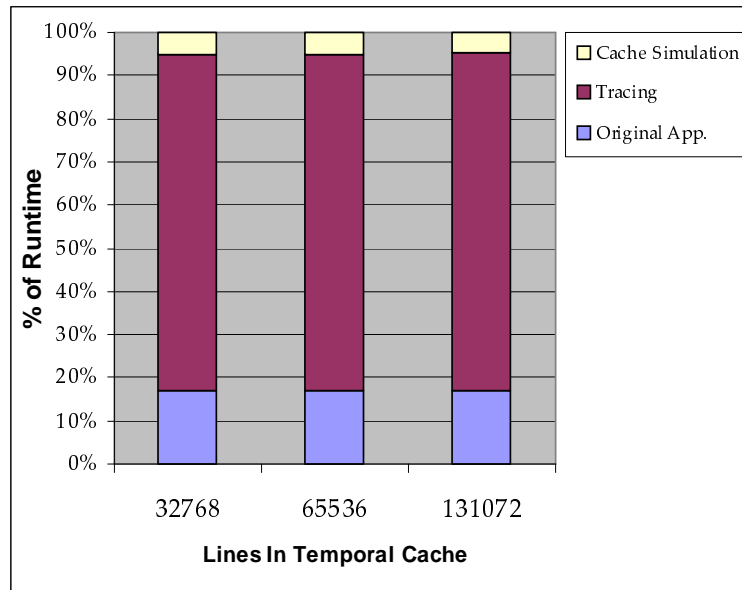


Figure 3.2: Optimized overheads of tracing CG.A.1 with variously sized caches

This chapter, in part, is a reprint of the material as it appears in: J. Weinberg, M. O. McCracken, A. Snavey, E. Strohmaier. Quantifying Locality In The Memory Access Patterns of HPC Applications. In *Supercomputing 2005, Seattle, WA., November 12-16, 2005*. The thesis author was the primary researcher and author of this publication.

# Chapter 4

## Results

To evaluate the methodology, we have applied it to the analysis of relevant serial benchmarks from the HPCC and Apex-MAP. We used an Atom version of Metasim to instrument all binaries and performed the traces on *Lemieux*, an Alpha SC45 machine at the Pittsburgh Supercomputer Center. To obtain the following results, we configured Metasim to use a look back window of size  $\mathbf{W}=32$  and a maximum detectable stride length of  $\mathbf{T}=8$ . For our temporal analysis, we measure reuse distances from  $\mathbf{N}$  in the range 16 to 131072 8-byte words by doubling, distances that correspond to temporal cache sizes of 128Bytes to 1MB. When we present an application’s temporal score, we refer to the integral of its logscaled locality curve from 0 to 131072, normalized as a percentage of total area as given by Formula 2.2. We performed no custom tuning on any of the benchmarks.

### 4.1 HPCC

To analyze the spatial and temporal locality exercised by the HPCC benchmarks, we examine the following four benchmarks from the suite:

**Stream** - a simple synthetic benchmark program that measures sustainable memory bandwidth and the corresponding computation rate for a simple vector kernel

**RandomAccess (GUPS)** - measures the rate of integer random updates of memory

**FFT** - measures the floating point rate of execution of double precision complex one-dimensional Discrete Fourier Transform.

**HPL** - the Linpack TPP benchmark that measures the floating point rate for solving a linear system of equations.

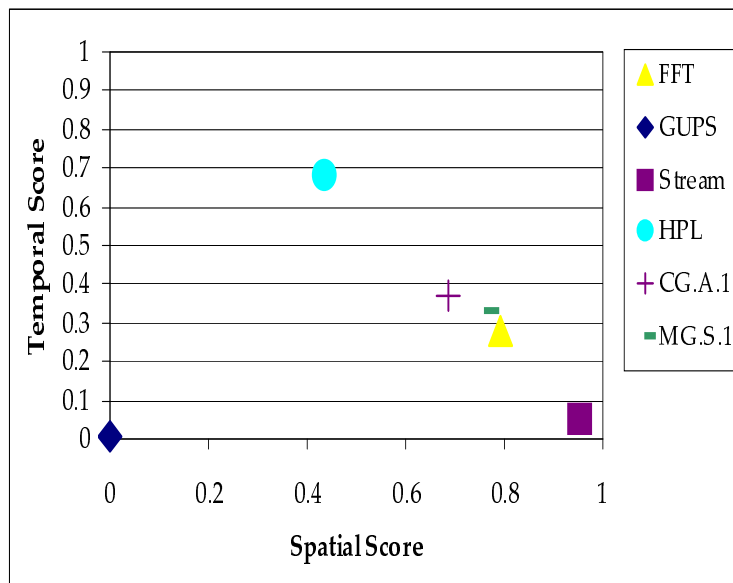


Figure 4.1: Locality scores of the HPCC and NPB benchmarks

Figure 4.1 shows the locality scores of the four applications and Figure 4.2 displays their temporal reuse functions. To place these scores within some context, we have performed a similar analysis on CG and MG, two of the NAS Parallel Benchmarks [8]. These benchmarks were run at problem class A and S respectively.

Notice that Stream and GUPS are scored intuitively. GUPS performs random updates to a large memory and therefore displays neither spatial nor temporal locality. Stream on the other hand, performs only regularly strided memory access to a large memory and therefore displays a great degree of spatial locality but very

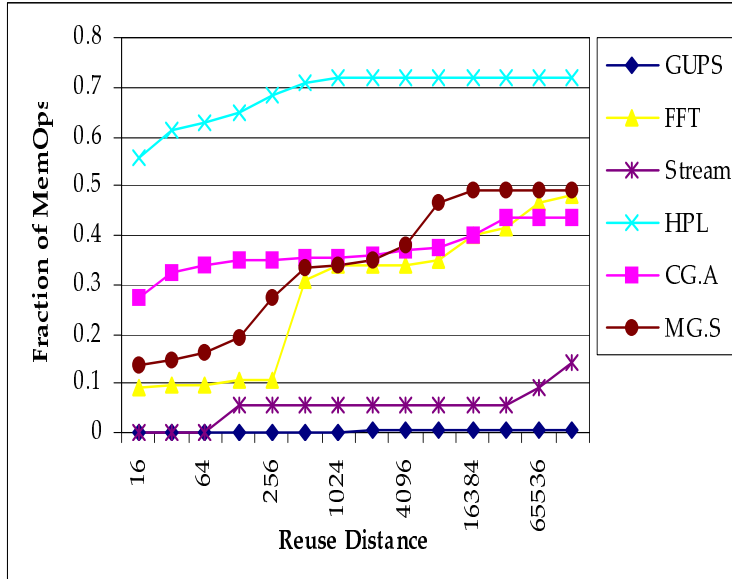


Figure 4.2: Temporal locality functions of HPCC and NPB benchmarks

little temporal. Stream’s access pattern exemplifies the motivation for differentiating locality along these dimensions. If we were to predict a cache hit rate for Stream based on reuse distance alone [22], then we would under-predict, having not accounted for cache hits induced by prefetching.

HPL and FFT are both meant to represent compute-bound codes and are therefore expected to exhibit high degrees of locality. The results bear this out with HPL exhibiting especially high levels of temporal locality while FFT does similarly with spatial locality.

The end result is that these metrics allow us to compare benchmarks via two single-number scores and make comparisons such as “Stream has more spatial locality than CG” or “FFT has lower temporal locality than HPL” in a straightforward and meaningful way.

## 4.2 Apex-MAP

Apex-MAP [30] is a synthetic benchmark that stresses a machine’s memory subsystem according to parameterized degrees of spatial and temporal locality. Along with other parameters, the user specifies  $\mathbf{L}$  and  $\mathbf{K}$ , parameters related to spatial locality and temporal reuse respectively. Apex-MAP then chooses a configurable number of indices into a data array that are distributed according to  $\mathbf{K}$ , using a non-uniform random number generator. The indices are most dispersed when  $\mathbf{K}=1$  (uniform random distribution) and become increasingly crowded as  $\mathbf{K}$  approaches 0. Apex-MAP then performs  $\mathbf{L}$  stride 1 references starting from each index. This process is repeated a configurable number of times.

Parameter sweeps of Apex-MAP have been used to map the locality space of certain systems with respect to  $\mathbf{L}$  and  $\mathbf{K}$ . Figures 4.3 and 4.4 show how performance in cycles per memory operation varies as a function of  $\mathbf{L}$  and  $\mathbf{K}$  on two very different architectures.<sup>1</sup>

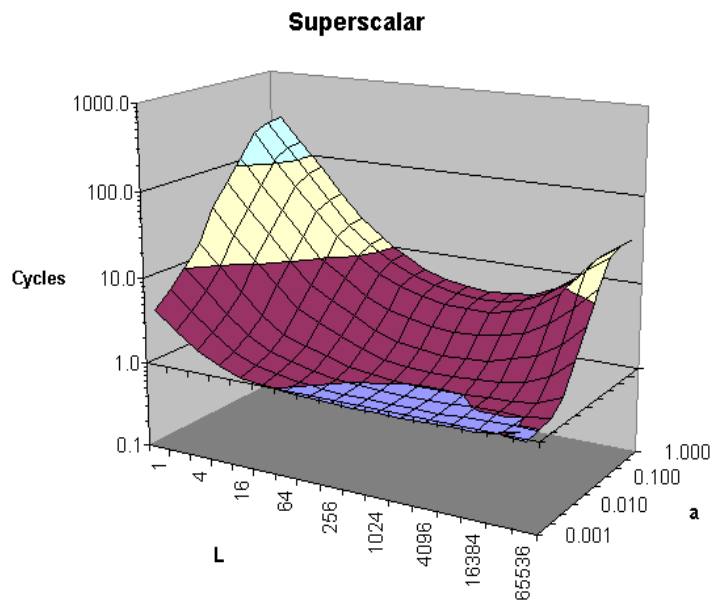


Figure 4.3: Apex-MAP performance surface on a superscalar machine

<sup>1</sup>The parameter  $\mathbf{K}$  is sometimes referred to as **alpha**.

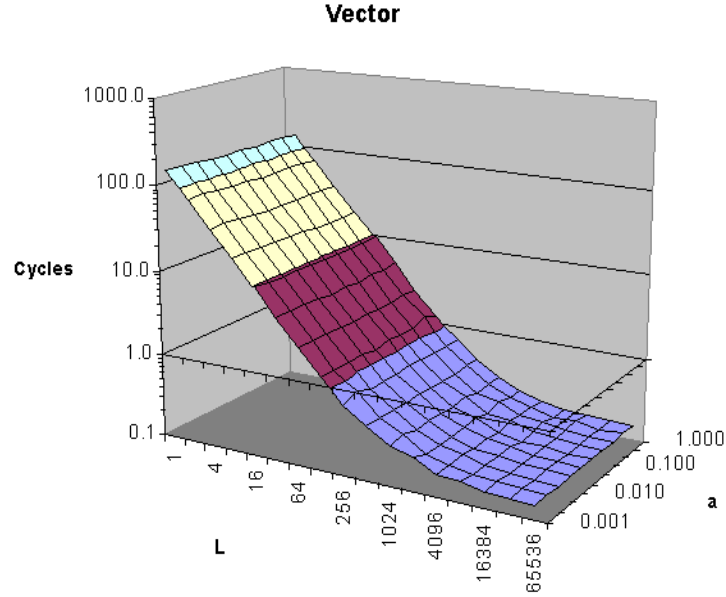


Figure 4.4: Apex-MAP performance surface on a vector machine

There has been some effort to determine the  $\mathbf{L}$  and  $\mathbf{K}$  values of certain applications using back fitting [30]. Theoretically, if we could more easily obtain the  $\mathbf{L}$  and  $\mathbf{K}$  value of some application, we could use Apex-MAP as a lightweight probe to mimic the memory access behavior of that application and give some indication of its possible memory performance on a target architecture.

#### 4.2.1 Locality Scoring of Apex-MAP

Our first results aim to determine the extent to which the locality parameters of Apex-MAP actually affect the benchmark’s memory behavior. To determine this, we performed a parameter sweep of Apex-MAP, tracing all combinations of  $\mathbf{K}$  and  $\mathbf{L}$  formed from the sets  $\{.001, .01, .05, .1, .5, 1\}$  and  $\{1, 2, 4, 8, 16, 32, 64, 128, 512, 1024\}$  respectively.

Figure 4.5 plots the temporal reuse functions of all combinations of  $\mathbf{K}$  while  $\mathbf{L}$  is held constant at 1 (i.e. no runs of stride 1). The results clearly indicate that our metrics track the temporal locality of Apex-MAP as a monotonic function of

**K**. The smaller **K** is, the faster a larger fraction of memory operations falls into smaller temporal caches due to small reuse distances.

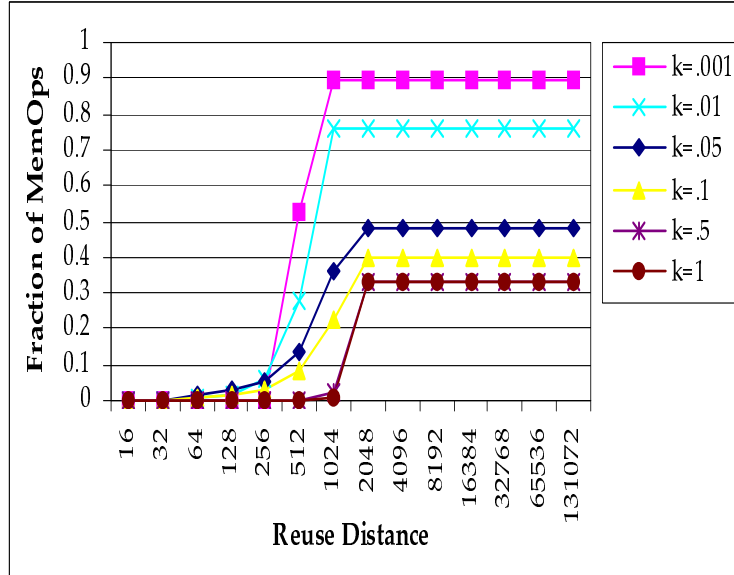


Figure 4.5: As **K** increases, the temporal locality of Apex-MAP decreases

Figure 4.6 plots the mapping functions between **K** and our temporal scores for each value of **L**. The reason that different values of **L** yield separate functions is that as **L** increases, it introduces more strided, non-temporal, references which in turn lower the percentage of total memory operations that the temporal hits constitute at each point on the curve. This phenomenon is most clear in the case of **K**=1 where temporal scores are in proportion to  $1/L$ . As the value of **K** increases however, the relationship between the curves is blurred, which could indicate some interdependence between **L** and **K** with respect to temporal reuse.

Figure 4.7 charts the mapping function between **L** and our spatial score under various assumptions for **K**. These results indicate that the spatial score is, for the most part, an increasing function of **L** that asymptotically approaches a value near .9, chiefly independent of **K**. The sporadically anomalous behavior starting at **L**=8 stems from a compiler optimization that begins to exercise a new basic block only

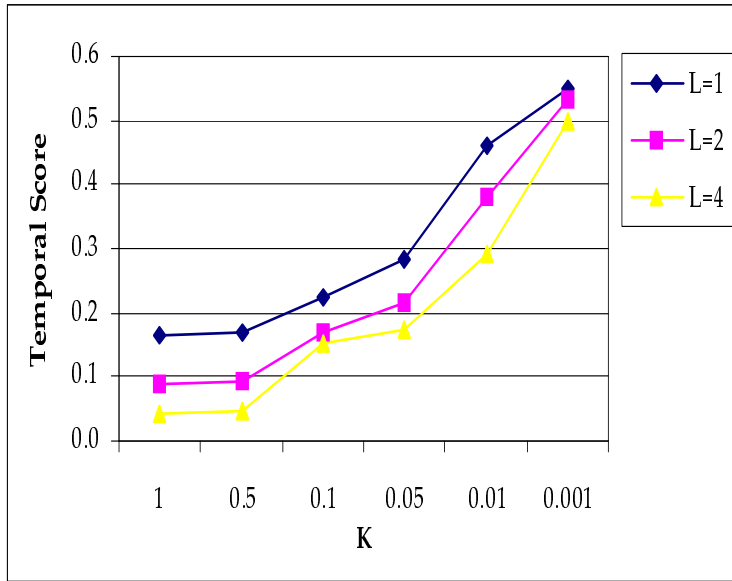


Figure 4.6: Mapping of K to temporal Score

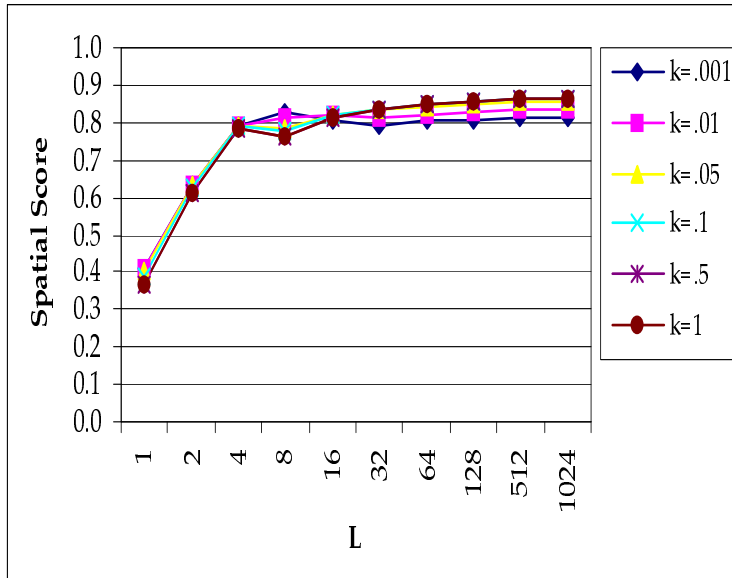


Figure 4.7: Mapping of L to spatial score

Spatial Score	Temporal Score	Performance (x10 <sup>8</sup> memops/s)
.40	.55	4.11
.41	.46	1.42
.40	.28	.508
.39	.22	.304
.37	.17	.168
.37	.16	.165
.63	.53	5.61
.63	.38	2.36
.63	.22	.895
.63	.17	.583
.61	.09	.330
.61	.08	.306
.79	.50	8.52
.79	.29	4.01
.79	.17	1.57
.79	.15	1.04
.78	.05	.625
.78	.04	.591

Table 4.1: Locality scores and performance of Apex-MAP as  $\mathbf{L}$  and  $\mathbf{K}$  vary. Each of the three vertical sections corresponds to  $\mathbf{L} = \{1,2,4\}$  respectively while the rows inside each section correspond to  $\mathbf{K} = \{.001, .01, .05, .1, .5, 1\}$  respectively.

when  $\mathbf{L}$  grows larger than four.

Based on these results, Apex-MAP covers a spatial score range of approximately .35-.85 and a temporal score range of approximately .02-.55. It is possible that the high end of the temporal score could be increased using yet smaller values of  $\mathbf{K}$  with which we have not experimented.

To confirm that our locality scoring relates intuitively to that of Apex-MAP and consequently to performance, we benchmarked Apex-MAP runs on our SC45 system, parameterized by all combinations of  $\mathbf{L}=\{1, 2, 4\}$  and  $\mathbf{K}=\{.001, .01, .05, .1, .5, 1\}$ . Table 4.1, shows the locality scores and achieved performance for each of these runs. The maximum theoretical peak on the 1GHz system should be near  $1 \times 10^9$  memory operations per second.

As shown in Figure 4.6 and verified by Table 4.1, the value of  $\mathbf{L}$  impacts the temporal score of Apex-MAP while  $\mathbf{K}$  is held constant. Consequently, our data points are not uniformly distributed, but we can nevertheless use them to interpolate a performance surface. Figure 4.8 plots the performance of Apex-MAP on the SC45 as a function of its spatial and temporal scores. The surface confirms that our spatial and temporal scores relate intuitively to the performance of Apex-MAP and its notions of locality.

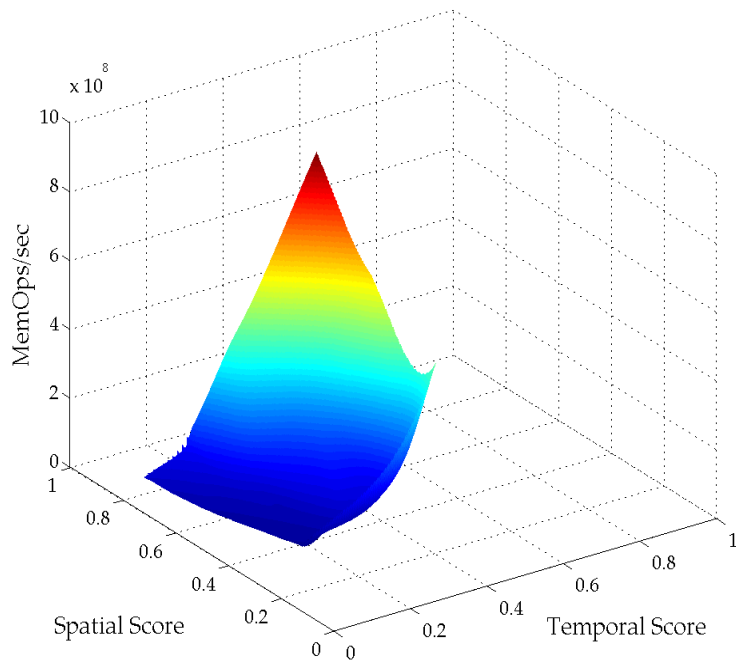


Figure 4.8: Apex-MAP shows increased performance with increased locality

### 4.2.2 Apex-MAP To Other Codes

In the previous section, we developed a mapping between the abstract spatial and temporal parameters of Apex-MAP and our observable locality metrics. These mappings enable us to measure the locality score of a given application and find

the corresponding values of  $\mathbf{L}$  and  $\mathbf{K}$ . To find the  $\mathbf{L}$  and  $\mathbf{K}$  values for an application of interest:

1. Use Metasim on the code of interest to trace and calculate its temporal and spatial locality scores.
2. Consult the interpolations of Figure 4.7 to find the  $\mathbf{L}$  value corresponding to the application's spatial score.
3. Select the curve on Figure 4.6 that corresponds to the  $\mathbf{L}$  value derived in the previous step. To find  $\mathbf{K}$ , select the point on the curve that corresponds to the application's temporal score.

Using the above procedure, we can configure Apex-MAP to perform memory accesses with the locality of an application of interest, creating a lightweight memory performance probe to represent it.

The tracer code for Metasim, and instructions for running it, along with source code and instructions for Apex-MAP are at [www.sdsc.edu/pmac](http://www.sdsc.edu/pmac).

One should not assume that an application will get the same performance in memory operations per second as its Apex-MAP representative. This may be true though if memory performance is the limiting factor.

Table 4.2 displays an analysis of CG.A, a serial benchmark we assume may be memory bound, using Apex-MAP. The spatial and temporal scores of this application are .68 and .33 respectively, as displayed in Figure 4.1. Using the linear interpolation shown in Figure 4.7, we can interpolate a spatial score of .68 to map to an  $\mathbf{L}$ -value of 3. Based on the  $\mathbf{L}=2$  and  $\mathbf{L}=4$  curves presented in Figure 4.6, we could choose  $\mathbf{K}=.01$  as the approximate value for  $\mathbf{K}$ . On our test system, a single 1-GHz Alpha processor completed CG.A at a memory operation rate of  $3.75 \times 10^8$  operations per second. Apex-MAP at  $\mathbf{L}=3$  and  $\mathbf{K}=.01$  performs at a rate of approximately  $3.25 \times 10^8$  operations per second, a 7.5% discrepancy.

The end-result of this section is that we now have a recipe for determining  $\mathbf{L}$  and  $\mathbf{K}$  parameter settings for Apex-MAP that should cause it to mimic the

S,T	L,K	Apex-MAP Performance (memOps/s)	CG.A Performance (memOps/s)	Error
.68, .33	3, .01	$3.25 \times 10^8$	$3.75 \times 10^8$	7.5%

Table 4.2: Mapping Apex-MAP to CG.A Yields Similar Performance

temporal and spatial locality behavior of a chosen application.

### 4.3 Large Parallel Applications

The extension of our methodology to full scale applications presents some interesting new challenges and opportunities. First, we must consider the increased runtime of these full applications. While the runtimes of the benchmarks we analyze in Section 4.1 do not exceed a few minutes, real HPC applications can require several hours to complete full runs. If we use Metasim to trace the entire application at once, the full runtime would far exceed the maximum allowed by most batch scheduling systems.

To deal with this, we break the tracing of large applications into *phases*. A phase is simply a trace where only some subset of the memory references is instrumented. If an application’s binary defines  $\mathbf{B}$  basic blocks and we chose to break the tracing into  $\mathbf{P}$  phases, then phase 1 contains the first  $\mathbf{B}/\mathbf{P}$  blocks in the static binary, phase 2 contains the next  $\mathbf{B}/\mathbf{P}$  basic blocks and so on. The instrumented binary of each phase is then run in a tractable time.

There are several items to note in the approach. First, even though the blocks in each phase constitute a contiguous set in the static binary, this does not mean that they necessarily constitute anything contiguous in the application’s dynamic execution. When the application executes, basic blocks are exercised in an order that we do not undertake to predict. It is therefore not correct to say that the first phase of some application trace occurs chronologically before the second phase or vice versa in the dynamic execution.

Secondly, we must decide how to combine the trace information at the end into a single score. To do this, we derive the locality scores of each phase separately and then compute a weighted average for the application, weighting each phase by the total number of dynamic memory operations it executed. To graph collective temporal reuse functions, we can calculate weighted averages for each reuse distance point.

This approach does sacrifice some accuracy however. This is because some arbitrary and unpredictable number of memory references are ignored between every instrumented reference. In the case of calculating spatial locality, there are no ill-effects because we calculate stride information on a per basic block scope and so the ordering, existence, or absence of basic blocks does not affect the scoring.

In the case of temporal locality however, reuse distances are calculated on an entire program level and so it is possible for the existence or absence of some basic blocks to affect the total outcome. Reuse distances of some memory references may appear either greater than or less than their true distance. For instance, if a memory reference is reused following an uninstrumented basic block, we may attribute it a shorter reuse distance than it truly has. Alternatively, if that uninstrumented block had reused the memory address, the reference would appear to have a longer reuse distance.

We accept this measurement noise for now and hope that error is mitigated by grouping basic blocks together that are contiguous in the static binary. The partitioning of phases by contiguous regions of the binary tends to keep entire functions together and so the large work loops that often characterize HPC codes are more likely to be traced in entirety. We refer the reader to Chapter 7 for ideas on how to minimize this source of error in future work.

To demonstrate the applicability of this methodology to full-scale parallel codes, we analyze the following two codes as defined by the DoD HPCMP 2005 Technology Insertion (TI-05).

**AVUS Standard** - AVUS was developed by the Air Force Research Laboratory

(AFRL) to determine the fluid flow and turbulence of projectiles and air vehicles. Its standard test case calculates 400 time-steps of fluid flow and turbulence for a wing, flap, and end plates using 7 million cells. Its large test case calculates 150 time-steps of fluid flow and turbulence for an unmanned aerial vehicle using 24 million cells. We ran this test case using 32 processors.

**HYCOM Standard** - The Naval Research Laboratory (NRL), Los Alamos National Laboratory (LANL), and the University of Miami developed HYCOM as an upgrade to MICOM (both well-known ocean modeling codes) by enhancing the vertical layer definitions within the model to better capture the underlying science. HYCOM's standard test case models all of the world's oceans as one global body of water at a resolution of one-fourth of a degree when measured at the Equator. We ran this test case using 59 processors.

Figures 4.9 and 4.10 show the temporal reuse curves for each phase of AVUS and HYCOM respectively. Because these statistics were gathered as part of a large-scale production effort on behalf of the DoD's HPCMO program, we were limited in the number of temporal caches we were permitted to simulate. The temporal scores of these applications along with those of the benchmarks presented earlier have been adjusted to reflect this change.

As the graphs show, AVUS was broken into six phases while HYCOM was broken into three. To produce the collective temporal reuse curves displayed in Figure 4.11 we compute the weighted average of these curves. Each curve is weighting by the total number of memory operations performed in that phase. The HPCC applications are also plotted in figure 4.11 to provide a frame of reference. AVUS shows a relatively high temporal score but we should not make a blanket conclusions like, AVUS has almost as much temporal locality as HPL. We must keep in mind that these locality scores are always in the context of workload. If the same HPL workload were broken up among multiple processors, the temporal score would jump significantly as can be inferred from Figure 5.2 in the following Chapter.

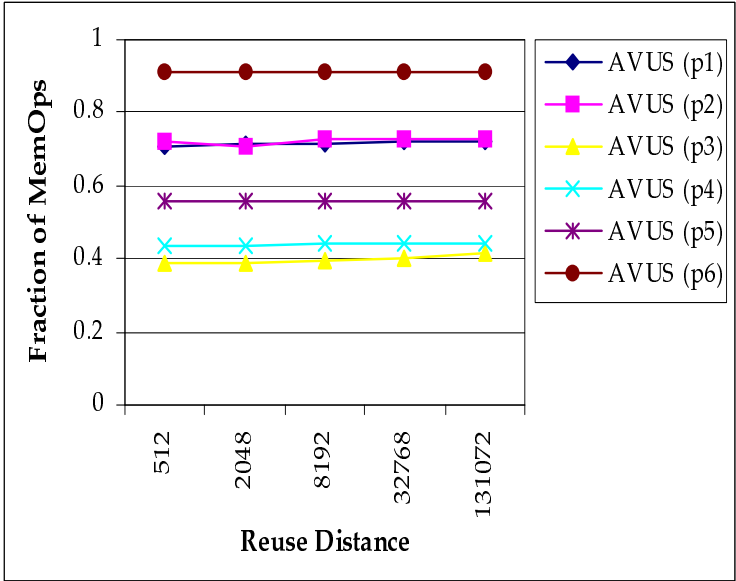


Figure 4.9: The temporal reuse curves of each phase of AVUS

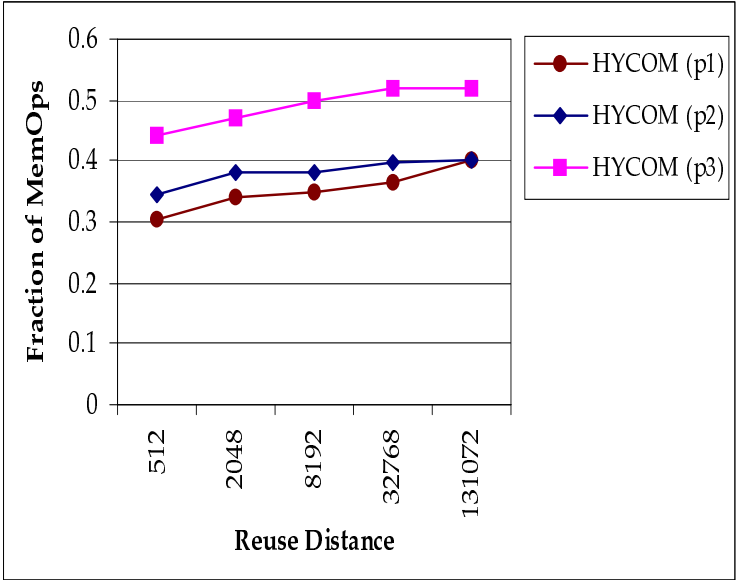


Figure 4.10: The temporal reuse curves of each phase of HYCOM

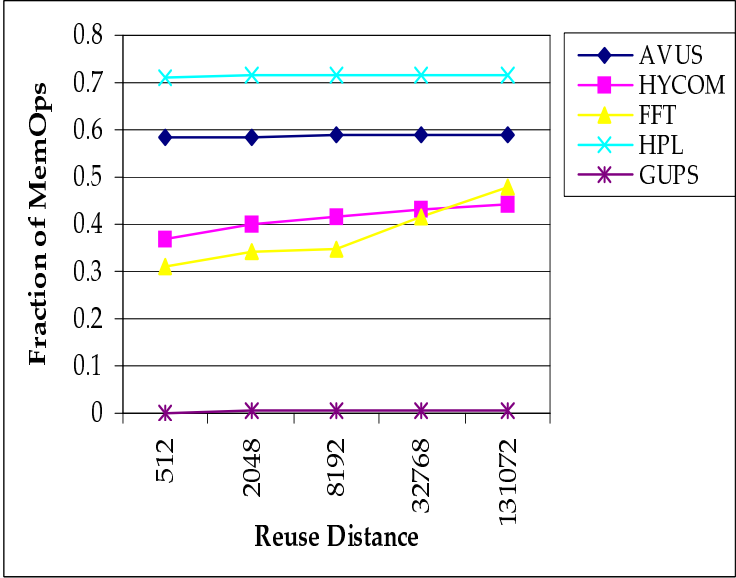


Figure 4.11: The temporal reuse curves of AVUS and HYCOM compared to the HPCC Benchmarks

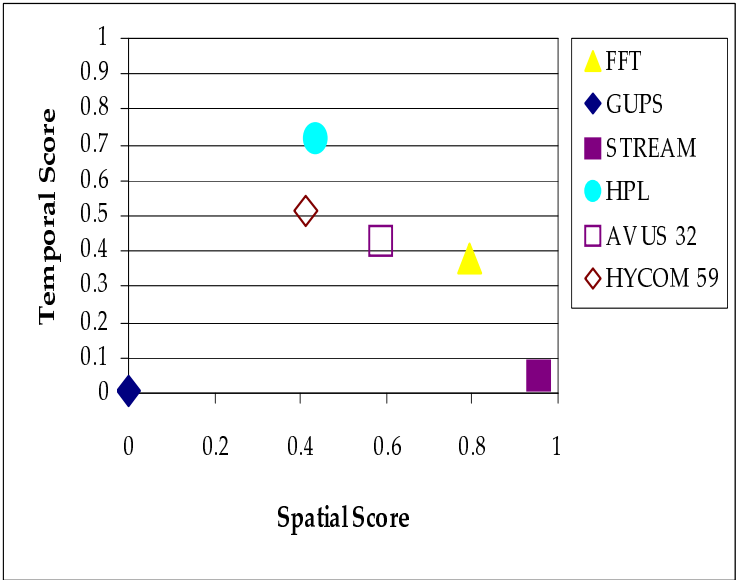


Figure 4.12: The locality scores of AVUS and HYCOM are within the bounds of the HPCC benchmarks

Figure 4.12 plots the temporal and spatial locality scores of AVUS and HYCOM in the context of the HPCC applications. We see that the scores of the large applications are within the region of score space bounded by the HPCC benchmarks.

This chapter, in part, is a reprint of the material as it appears in: J. Weinberg, M. O. McCracken, A. Snavely, E. Strohmaier. Quantifying Locality In The Memory Access Patterns of HPC Applications. In *Supercomputing 2005, Seattle, WA., November 12-16, 2005*. The thesis author was the primary researcher and author of this publication.

# Chapter 5

## Other Uses

The presented methodology for locality analysis has many potential uses for helping us understand application performance. One example is using the analysis to understand how applications scale as problem size or processor counts grow. Figure 5.1 plots the locality scores of CG as the problem size scales between classes S, W, and A. The graph also shows the average spatial and temporal locality of the memory work assigned to each processor as processor counts scale.

We see that as input size increases, temporal score decreases, because the array sizes and size of the working set increases. At the same time, spatial score increases with long runs of stride 1 references through bigger arrays. In this simple case, everything matches intuition although the behavior of large applications could be less intuitive.

Splitting the CG class A problem across 8 CPU's (comparing CG.A.1 to CG.A.8) reduces the size of the working set per-processor and thus increases the temporal score while spatial score drops due to diminishing array/loop bounds.

Another possible application of locality analysis is to quantify the difficulty of memory work across processors in a single parallel application. Equitable workload distribution is often considered in terms of quantity, as defined by software data structures or even floating point/memory operations assigned to each node. Using locality analysis, we can add a qualitative dimension that defines the *difficulty* of

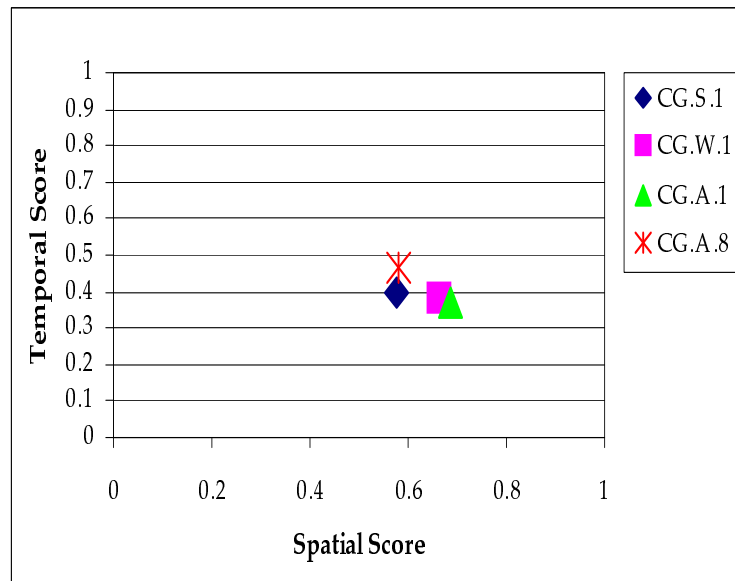


Figure 5.1: Locality scoring captures how the memory work of CG changes with problem size

each processor's workload.

Figure 5.2 graphs the temporal reuse function of each processor in a 16 processor run of HPL. This run is performed using the same input as that used to generate the results in Section 4.1. The graph verifies that the memory workload difficulty of each processor in HPL is relatively even, but not perfectly so. The temporal scores range from a low of .83 (processor 0) to a high of .87 (processor 8).

Figure 5.2 also demonstrates how the temporal score of HPL increases dramatically when the workload is split among 16 processors instead of only 1.

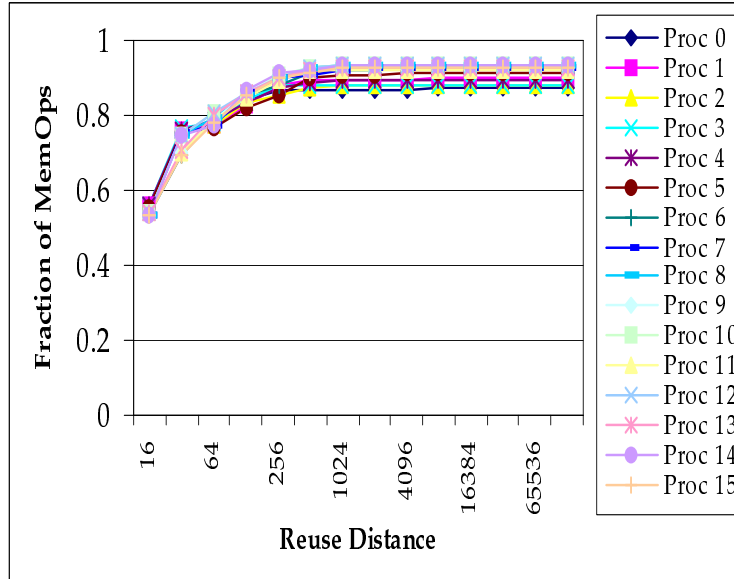


Figure 5.2: Temporal reuse functions of each processor in a 16 processor run of HPL

This chapter, in part, is a reprint of the material as it appears in: J. Weinberg, M. O. McCracken, A. Snaveley, E. Strohmaier. Quantifying Locality In The Memory Access Patterns of HPC Applications. In *Supercomputing 2005, Seattle, WA., November 12-16, 2005*. The thesis author was the primary researcher and author of this publication.

# Chapter 6

## Related Work

Temporal locality, perhaps because it is a little subtle to reason about, has a rich history in the literature. In 1970, Mattson et al. studied stack algorithms in cache management and defined the concept of stack distance [1]. Reuse distance is simply the same as LRU stack distance or stack distance using LRU replacement policy. In particular, Bunt has produced a host of publications studying both kinds of locality ranging from his seminal papers [11, 24, 23] through various investigations of metrics and measurement techniques and recently summarized in [12], 20 years after [23]. Our approach is to adapt these ideas and make them practical and suitable for analysis of HPC parallel applications via dynamic tracing.

In the early 1980s Smith's seminal work points out the opportunities for cache to take advantage of both kinds of locality [25]. This either presaged or, to some extent, sparked the cache-based system revolution. Subsequently, in the late 1980s Agarwal and Snir published several papers modeling deepening memory hierarchies using metrics of locality [4, 3]. Likewise, Carter and Alpern [7] made contributions in this area around the same time. We reviewed those works and consulted with Carter in adapting those ideas to this work.

Trace driven simulation and analysis has its own rich literature; we point the reader to [32] for a good summary up progress until late 1990s. Our work as in [27] has leveraged and extended the state-of-the-art to make parallel HPC applications

tracing fast enough to be practical.

Concepts of temporal and spatial locality are now so well developed they are taught to undergraduates in the classic Hennessy and Patterson text [19]. Nevertheless, as mentioned, the details of exactly how to measure them are left to the reader. We drew on more detailed ideas as of Bunt, Agarwal, Carter etc. above to devise metrics that are concrete and practical to measure for HPC applications.

Using reuse distance to reason about and improve performance is a well developed area as for example in [9, 16] and there is also work in exploiting spatial locality for performance as in the work of Torellas [31], Johnson [20], and Kumar [21]. From the standpoint of this work these provide more evidence that the capability of measuring and benchmarking locality of HPC applications is important.

Work such as Song's [28] is representative of a whole area that uses compilers to improve code locality, especially temporal locality, to improve performance. We refer the reader to [5] for indepth background on this area; there simply is not room here to cite all the good compiler analysis for locality papers.

Darema et al. in [15] characterized a scientific workload with respect to memory access patterns in 1987. This work aims to update that kind of capability. Harrison in [18] did similar work on pointer-chasing and numeric programs of the day. Ding et al. in [16] predict locality based on reuse distance; our focus is rather to measure locality directly.

Almasi et al. in [6] propose ways to calculate stack distance efficiently. Our approach is somewhat similar in spirit; we propose a tractable approximation.

This chapter, in part, is a reprint of the material as it appears in: J. Weinberg, M. O. McCracken, A. Snavely, E. Strohmaier. Quantifying Locality In The Memory Access Patterns of HPC Applications. In *Supercomputing 2005, Seattle, WA., November 12-16, 2005*. The thesis author was the primary researcher and author of this publication.

# Chapter 7

## Conclusions and Future Work

We have proposed and implemented a concrete methodology whereby benchmarks and applications can be scored for spatial and temporal locality. We used it to confirm that the HPC Challenge Benchmarks cover an interesting space along these dimensions. We provided a recipe for tracing an arbitrary application and determining the  $\mathbf{L}$  and  $\mathbf{K}$  values that can be used to configure Apex-MAP into a succinct benchmark proxy for the spatial and temporal characteristics of the application. We have further shown that our methodology can be applied to measuring the locality of large-scale HPC applications.

A possible line for future work could be to enhance the spatial scoring by considering the size of the look back window when calculating an application’s spatial score. This can be done in much the same manner as reuse distance is folded into its temporal score. Scoring each memory reference as a function of both its stride length and the size of the look back window required to identify such lengths would allow us to plot “spatial reuse” functions in a manner very much analogous to the temporal reuse functions we have presented here. It is possible to modify the Metasim tracer to collect this information with theoretically no palpable slowdown in the tracing performance.

Another line of future work might seek to answer how we can break up large applications into phases that are better suited to cache simulation. One way is to

perform a static analysis of the application binary and break the blocks along some best effort heuristic. An alternative approach might be to perform a preliminary round of tracing to dynamically determine a good partitioning. In these ways, we can create phases that contain few jumps to blocks outside of the phase. In either case, any new study of the effects of phasing on the Metasim traces would be pioneering.

In terms of performance prediction, one may argue that the transformation of our locality scores to Apex-MAP's locality parameters is unnecessary and simply introduces a gratuitous source of inaccuracy. This is true and better would be to create a benchmark based directly on the scoring principles presented here. The benchmark could be dialed in much the same manner as Apex-MAP, but could also contain other parameters to control *how* certain scores are reached. For example, a spatial score of .5 can be attained by an application that performs only stride 2 accesses or by one that performs half stride 1's and half random access. Investigating the variation in performance as these other parameters are tweaked could help determine the error bounds of using locality scoring to predict performance on memory-bound applications. Further, such an investigation could identify what other locality-related parameters are necessary to predict performance or to what degree this is possible altogether.

# Bibliography

- [1] Hpc challenge: <http://icl.cs.utk.edu/hpcc/>.
- [2] Top500: <http://www.top500.org>.
- [3] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 305–314, 1987.
- [4] A. Aggarwal, A. Chandra, and M. Snir. Hierarchical memory with block transfer. In *FOCS '87: Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 204–216, 1987.
- [5] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, San Francisco, CA, 2001.
- [6] G. Almasi, C. Cascaval, and D. Padua. Calculating stack distances efficiently. In *Proceedings of the first ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, 2002.
- [7] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. In *Tech. Rep. Cornell University*, pages 93–119, 1993.
- [8] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The nas parallel benchmarks. *International Journal of Supercomputer Applications*, 5(2):63–73, 1991.
- [9] K. Beyls and E. Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of PDCS*, pages 617–662, 2001.
- [10] B. Buck and J. Hollingsworth. An api for runtime code patching. *Journal of High Performance Computing Applications*, 14(4), 2000.
- [11] R. Bunt and J. Murphy. Measurement of locality and the behaviour of programs. *The Computer Journal*, 27(3):238–245, 1984.

- [12] R. Bunt and C. Williamson. Temporal and spatial locality: A time and a place for everything. In *Proceedings of the International Symposium in Honour of Professor Guenter Haring's 60th Birthday*, 2003.
- [13] L. Carrington, A. Snaveley, X. Gao, and N. Wolter. Performance prediction framework for scientific applications. *Lecture Notes in Computer Science*, 2659:926–935, January 2003.
- [14] L. Carrington, N. Wolter, A. Snaveley, 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.
- [15] F. Darema-Rogers, G. Pfister, and K. So. Memory access patterns of parallel scientific programs. In *SIGMETRICS 87: Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 46–58, New York, NY, 1987. ACM Press.
- [16] 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.
- [17] J. Dongarra, P. Luszczek, and A. Petitet. The linpack benchmark: Past, present and future. *Concurrency: Practice and Experience*, 15:803–820, 2003.
- [18] L. Harrison. Examination of a memory access classification scheme for pointer-intensive and numeric programs. In *Proceedings of the 10th International Conference on Supercomputing*, pages 131–140, New York, NY, 1996. ACM Press.
- [19] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco, CA, 1990.
- [20] T. Johnson, M. Merten, and W. Hwu. Run-time spatial locality detection and optimization. In *Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture*, pages 57–64, 1997.
- [21] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *ISCA 98: Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 357–368. IEEE Computer Society, 1998.

- [22] 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.
- [23] J. Peachey, R. Bunt, and C. Colbourn. Towards an intrinsic measure of program locality. In *Proceedings of the Sixteenth Annual Hawaii International Conference on System Sciences*, pages 128–137, 1983.
- [24] J. M. R.B. Bunt and S. Majumdar. A measure of program locality and its application. In *Proceedings of the 1984 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 28–40, 1984.
- [25] A. Smith. Cache memories. *ACM Computing Survey*, 14(3):473–530, 1982.
- [26] 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.
- [27] 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.
- [28] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *PLDI 99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 215–228. ACM Press, 1999.
- [29] 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.
- [30] E. Strohmaier and H. Shan. Architecture independent performance characterization and benchmarking for scientific applications. In *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Volendam, The Netherlands, 2004.
- [31] J. Torrellas, M. Lam, and J. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

- [32] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29(2):128–170, 1997.