2005-07-14

Cache Characterization and Performance Studies Using Locality Surfaces

Elizabeth Schreiner Sorenson
Brigham Young University - Provo

CACHE CHARACTERIZATION AND PERFORMANCE STUDIES USING LOCALITY SURFACES

by

Elizabeth S. Sorenson

A dissertation submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Brigham Young University

August 2005

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a dissertation submitted by

Elizabeth S. Sorenson

This dissertation has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	J. Kelly Flanagan, Chair
Date	Michael A. Goodrich
Date	Bryan S. Morse
Date	Dan R. Olsen
Date	William A. Barrett

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the dissertation of Elizabeth S. Sorenson in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

J. Kelly Flanagan
Chair, Graduate Committee

Accepted for the Department

Parris K. Egbert
Graduate Coordinator

Accepted for the College

G. Rex Bryce, Associate Dean
College of Physical and Mathematical Sciences

ABSTRACT

CACHE CHARACTERIZATION AND PERFORMANCE STUDIES USING LOCALITY SURFACES

Elizabeth S. Sorenson

Department of Computer Science

Doctor of Philosophy

Today's processors commonly use caches to help overcome the disparity between processor and main memory speeds. Due to the principle of locality, most of the processor's requests for data are satisfied by the fast cache memory, resulting in a signficant performance improvement. Methods for evaluating workloads and caches in terms of locality are valuable for cache design.

In this dissertation, we present a locality surface which displays both temporal and spatial locality on one three-dimensional graph. We provide a solid, mathematical description of locality data and equations for visualization. We then use the locality surface to examine the locality of a variety of workloads from the SPEC CPU 2000 benchmark suite. These surfaces contain a number of features that represent sequential runs, loops, temporal locality, striding, and other patterns from the input trace.

The locality surface can also be used to evaluate methodologies that involve locality. For example, we evaluate six synthetic trace generation methods and find that none of them accurately reproduce an original trace's locality.

We then combine a mathematical description of caches with our locality definition to create cache characterization surfaces. These new surfaces visually relate how references with varying degrees of locality function in a given cache. We examine how varying the cache size, line size, and associativity affect a cache's response to different types of locality.

We formally prove that the locality surface can predict the miss rate in some types of caches. Our locality surface matches well with cache simulation results, particularly caches with large associativities. We can qualitatively choose prudent values for cache and line size. Further, the locality surface can predict the miss rate with 100% accuracy for some fully associative caches and with some error for set associative caches.

One drawback to the locality surface is the time intensity of the stack-based algorithm. We provide a new parallel algorithm that reduces the computation time significantly. With this improvement, the locality surface becomes a viable and valuable tool for characterizing workloads and caches, predicting cache simulation results, and evaluating any procedure involving locality.

ACKNOWLEDGMENTS

I give heartfelt thanks to both my husband, Frank, and my son, Daniel for sharing me with this work for a time. I truly needed all their support, patience, and love to finish.

I am grateful to Kelly, the best advisor possible, for his wisdom and counsel. His knowledge, his commitment to the research, his financial backing, his time, and his vision have all been invaluable. I am also grateful to members of the lab, especially Myles, who over the years have offered suggestions, conversation, and critiques. I am thankful for Niki's wonderful work in creating the traces. I am grateful to all the members of my committee for their time and their edits.

Most of all, I am grateful to my Father in Heaven. It was His idea for me to finish, and He did what was necessary to buoy me up and make it happen.

Thank you all.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Value of Locality

Adherence to Moore's Law enables processor speeds to double every 18 months. Memory density is increasing at a similar rate, but DRAM memory speeds increase at the much slower rate of about 7% per year [43, page 391]. This means that the time needed to access memory is an increasing bottleneck. To help overcome this mismatch in speed, computer architects take advantage of the fact that smaller memories placed close to the processor are significantly faster than main memory [73].

These small, fast memories between the processor and main memory are called **caches**. Caches contain a subset of the data in main memory. If a considerable portion of the data the processor requires is found in the cache, the processor accesses slower main memory significantly less often. Cache access speeds are on the order of a single CPU cycle, while main memory access speeds are on the order of hundreds of cycles [43]. When the contents of an accessed memory location are in the cache, it is termed a **hit**. When the contents are not in the cache it is termed a **miss**.

Large caches are more likely to have hits but are slower and more expensive.

For this reason, most processors today have multiple levels of caches. Level-one (L1) caches are next to the processor and are the smallest and fastest kind of cache. L1 caches are typically split into two pieces, one for intructions and one for data. Level-two caches (L2) caches are between the L1 cache and main memory. They are larger and slower than the L1 cache. There may also be more levels of cache between the L2 cache and main memory. The **memory hierarchy** refers to the entire range of memory storage devices, from the smallest and fastest (L1 cache) to main memory to magnetic disk. Researchers do significant amounts of cache studies to find appropriate trade-offs between performance and expense at each level of the hierarchy.

Caches typically store the most recently accessed data from main memory. Due to the **principle of locality**, these recently accessed parts of memory are the most likely to be re-used by the processor in the near future. Caches improve performance because most programs run on processors have large amounts of locality [28, 43]. Locality means that after a processor accesses location x in memory, x and other locations close to x in memory tend to be accessed soon. Typically, researchers divide locality into two types. **Temporal locality** occurs when the processor reuses the same location in memory shortly after a previous use. **Spatial locality** occurs when items close together in memory are used by the processor soon after one another [43]. Some researchers have divided locality into more types [21, 58, 86], but these two are the most common.

Cache performance is frequently evaluated in terms of **miss rate**, or the number of misses for a given workload divided by the total number of memory references. Most caches are described in terms of their **size** (how much data the cache can hold), their **line size** (what size chunks of data are pulled from main memory at a given time, sometimes termed block size) and their **associativity** (how many places

in the cache a given piece of data can reside) [43]. Larger cache sizes, line sizes, and associativities tend to yield lower miss rates, but with slower access times.

Caches also have varying types of replacement policies. The replacement policy controls which block of memory is evicted from the cache when a new memory block must be inserted in the cache. The three primary policies are **random**, **least-recently used** (LRU), and **first in, first out** (FIFO) [43, pages 399-400]. LRU, or an approximation of LRU, is the most common case in today's caches [9]. In this work we focus exclusively on the LRU replacement policy.

Researchers generally agree that locality and miss rate are closely related [19, 43, 50, 64, 89]. Miss rates for a given workload are usually determined via trace-driven simulation. To do this, a trace of the workload is first created by recording the memory requests made by the processor while the workload runs on a given computer system. This process is termed **tracing.** There are a number of methods in use for tracing programs [7, 17, 33, 63, 83].

Once a trace is recorded, the trace is submitted to a cache simulation program for a given cache configuration. The simulator returns the miss rate, or some other metric that reflects how well the cache would have performed on the given program. This can be time consuming, especially if a large number of cache configurations are evaluated. In addition, the space necessary to store a trace of reasonable length can be quite large [17].

One replacement for trace-driven simulation is a cache independent locality metric that can be used to predict the miss rate for any cache configuration. (Some researchers refer to cache independent locality as **intrinsic locality** [50, 64].) Even if the calculation of the locality metric takes longer than simulations for several cache configurations, the trade-off is advantageous if the locality metric can accurately predict the miss rate for any cache. Many such locality metrics have been

proposed over the years [64, 66, 67].

Researchers have used their locality metrics for a variety of reasons beyond merely predicting miss rate. The conclusion is that locality is useful for a number of applications. However, no one locality metric is universally accepted by the community, primarily because each metric is specifically tailored to a given application. We now briefly introduce a number of locality metrics from the computer science literature and examine one set of them in detail.

1.2 Previous Locality Metrics

A wide variety of locality metrics, each with a slightly different purpose, have been proposed over the years. Most researchers develop locality metrics for use at some level of the memory hierarchy. Conte and Hwu use multiple two-dimensional graphs to characterize workloads in terms of their memory access behavior [24]. In [86] Wolf and Lam use locality metrics to help modify a compiler to improve loop nest locality. Thiebaut et al. use a locality metric as an input for creating synthetic traces [79]. In [66] Salsburg uses a scalar definition of locality to help predict cache performance. Fiat and Karlin use locality to study paging issues in [32]. In [82] Truong et al. use locality to help rearrange data in memory to improve program performance. Sanchez and Gonzalez, in [67], use a different set of two-dimensional graphs to characterize workloads and analyze cache misses. In [62] McKinley and Temam use their own set of two-dimensional graphs of various aspects of locality to help make future "architecture and software cache optimizations." Luo and John compress traces using principles of locality [60]. However, each of these locality metrics has limited application.

In the above examples, the researchers usually evaluate locality as either a scalar

value [66, 79] or a series of two-dimensional graphs [62, 67]. A scalar value oversimplifies locality. A value such as the miss rate may help one understand how a given workload would function in a given cache but gives little indication how the same workload would function in caches of other sizes or configurations. There is no single two-dimensional graph that includes all aspects of locality, so multiple graphs are necessary. For some applications, such as real-time locality analysis [20], only one aspect of locality is of interest, and therefore simpler locality metrics may be valuable. However, even in such situations it would be of value to have an all-encompassing metric with which to evaluate the simple metric's effect on the other aspects of locality.

To demonstrate the deficiencies of two-dimensional locality metrics, we now describe both the temporal and spatial locality metrics from Conte and Hwu's paper [24]. Most of the metrics we have mentioned are used exclusively by researchers from the same research groups. Conte and Hwu's metrics were used by two other groups, in [50] and [64], making them the closest to a standard in the field of locality. We describe the metrics defined by Conte and Hwu using both their equations and their words as presented in their paper. We then show graphs of their functions for one workload, specifically the instruction fetches of *twolf* from the SPEC CINT2000 benchmark suite. This trace was taken using the BACH tracing method [33], as were all the traces used in this dissertation. This trace contains 50,191,887 memory references, of which 21,988 are unique. Full details about the trace may be found in Appendix A.

Recall that the purpose of Conte and Hwu was to characterize benchmarks in terms of their memory access behavior. They divide locality into two forms, the **interreference temporal density function** and the **interreference spatial density function**, i.e. temporal and spatial locality. They define temporal locality as

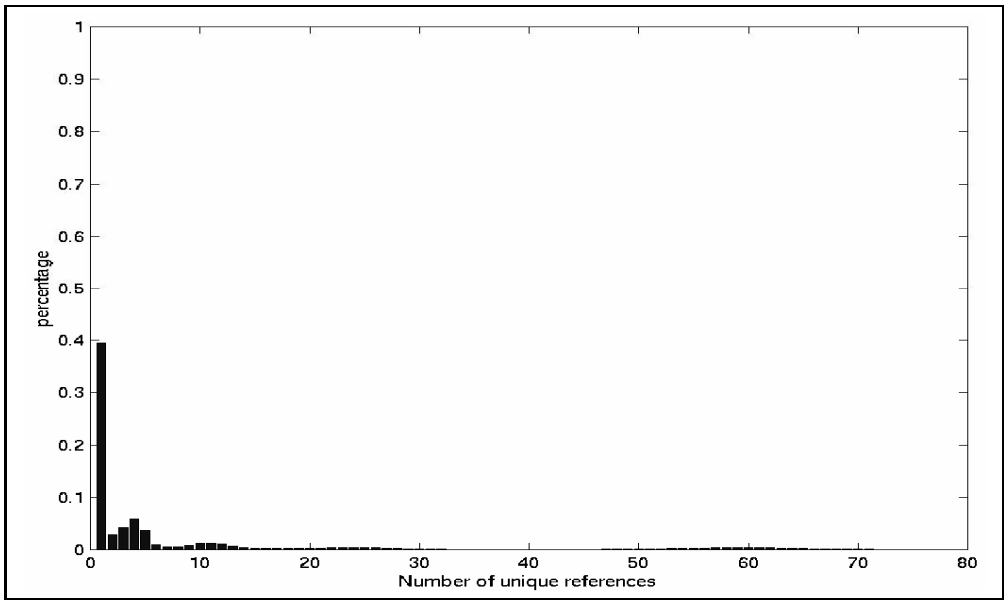


Figure 1.1: Conte and Hwu's interreference temporal density function for the instruction fetches of *twolf*.

the "probability of there being x unique references between successive references to the same item" [24]. They then present the following equation:

$$f^T(x) = \sum_t P[u(w(t)) = x] \tag{1.1}$$

where w is the trace, $w(t)$ is the memory reference in the trace at time t, $u(w(t))$ is the number of unique references between $w(t)$ and the next instance of $w(t)$.

In Figure 1.1 we see Conte and Hwu's interreference temporal density function for the instruction fetches of *twolf*. We can immediately see that almost 40% of the references in the instruction fetches of *twolf* are immediate repeats of the previous reference. There are also a fair amount of repeats to the same memory location after less than fifteen unique references. We can see a slight lump of data around sixty unique references, indicating that there are a few repeats after around sixty unique references. Overall, we can see that the instructions of *twolf* have good temporal

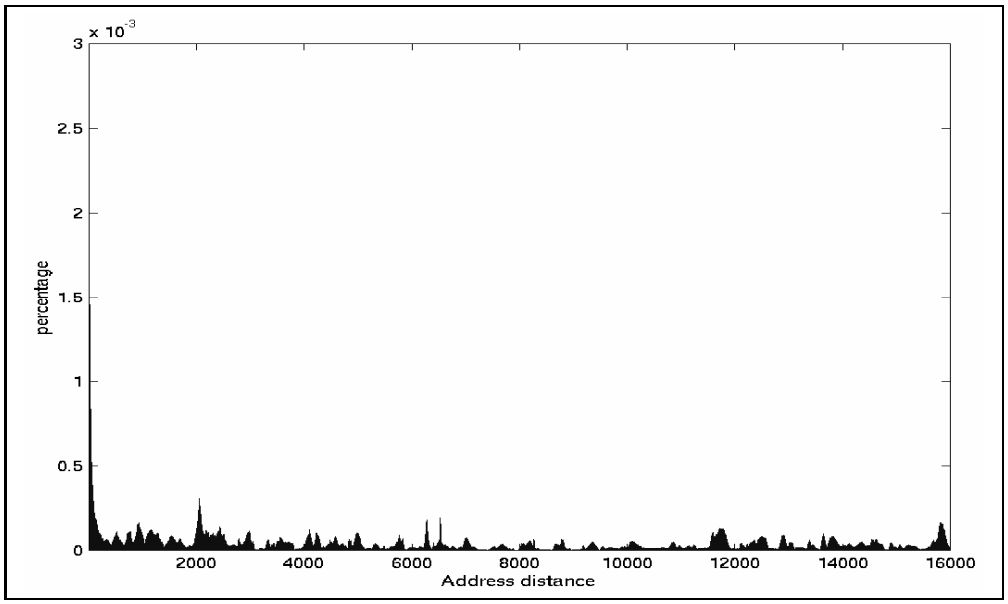


Figure 1.2: Conte and Hwu's interreference spatial density function for the instruction fetches of *twolf*.

locality. But other then saying that a cache should be able to contain at least fifteen items from memory, we learn little about optimal cache size.

Spatial locality is the "probability that between references to the same item, a reference to an item x units away occurs" [24] and uses the following equation:

$$f^S(x) = \sum_t \sum_{k=1}^{next(w(t))} P[|w(t) - w(t+k)| = x] \tag{1.2}$$

where w is the trace, $w(t)$ is the memory reference in the trace at time t, and $next(w(t)) = i$ if i is the smallest number such that $w(t) = w(t+i)$.

In Figure 1.2 we see Conte and Hwu's interreference spatial density function for the instruction fetches of *twolf*. This graph covers a greater range of values, making it harder to see any data. The first bar of the graph, at an address distance of one, is completely masked by the y axis. This bar should be at a height of 0.003. This means that 0.3% of memory addresses between two repeated references are at a distance of one memory location away from the repeated references.

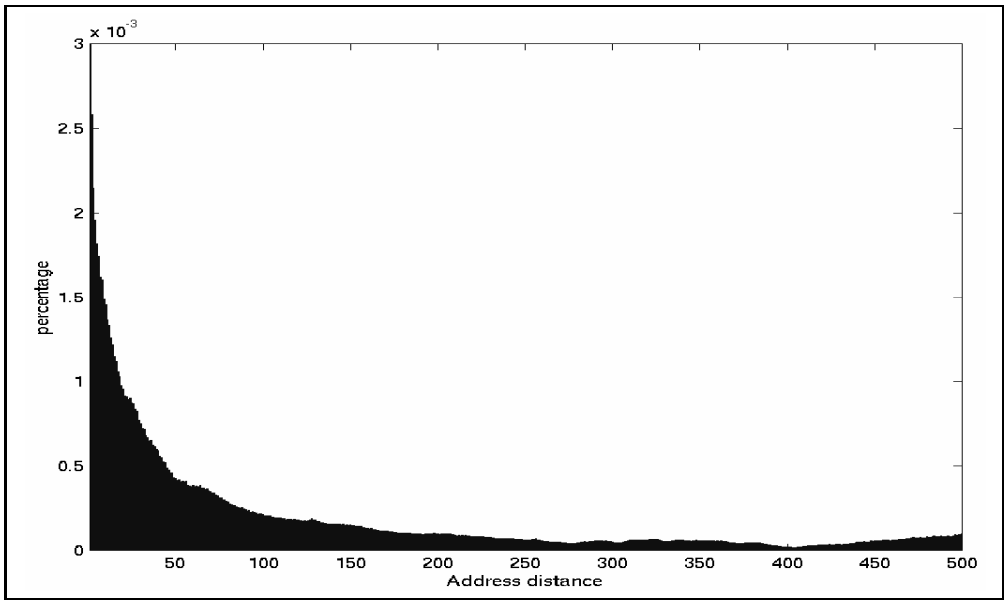


Figure 1.3: Conte and Hwu's interreference spatial density function for the instruction fetches of *twolf*, with a smaller maximum address distance shown.

To get a better feel for what is going on at small address distances, Figure 1.3 shows the same data with a smaller range for the x axis. Now we can see that, comparatively speaking, memory addresses between two repeated references tend to be close in memory. But the percentages are too small to get a clear picture of what is going on. Many aspects of spatial locality that would be of interest to cache designers cannot be seen here. We see how close in memory various references are, but have no feel for how close in time they are. Perhaps all the references that are close in memory are far enough apart in time that cache designers cannot take advantage of it. We get no feel for the distribution of sequential runs of intructions. To see all of this information, even more spatial locality graphs would be needed.

While there are advantages to Conte and Hwu's locality metrics, they do miss important aspects of locality. For example, if a, b, and c are all memory addresses, both *abcabc* and *abcabcabcabcabcabc* would have the same locality graphs but dif-

ferent miss rates. A locality metric would be more useful for cache studies if such differences were observed.

In addition, this spatial locality metric assumes a significant amount of temporal locality exists in the trace. If we had a trace that was one long sequential run, such as 1, 2, 3, 4, ..., 999, 1000, we should see significant spatial locality. However, since there is no temporal locality, Conte and Hwu's interreference spatial density function would produce no information. In short, when using two-dimensional graphs as locality metrics, either large numbers of graphs are needed for each workload, or some aspects of locality are missed. Perhaps a three-dimensional graph would be more valuable.

We have found three locality metrics that use three-dimensional graphs: the memory mountain [18, 76], the value reuse profile (VRP) [47], and the locality surface [38]. Both the memory mountain and the VRP use time as one of their dimensions. This means their metrics are tied with particular systems that have given latencies and are not useful for predicting miss rate. The locality surface, however, has several interesting advantages that give it a more general application than the other metrics we have mentioned. We now describe the original locality surface in more detail.

The locality surface, originally developed by Grimsrud [38, 40], incorporates both temporal and spatial locality in a single, three-dimensional, graph. It is essentially a histogram of the number of stride/delay occurrences that are found between values in a list. For any two values, the delay is the number of values between them in the list and the stride is the difference between the values. Stride can be positive or negative, but delay is only counted in one direction and is therefore always positive.

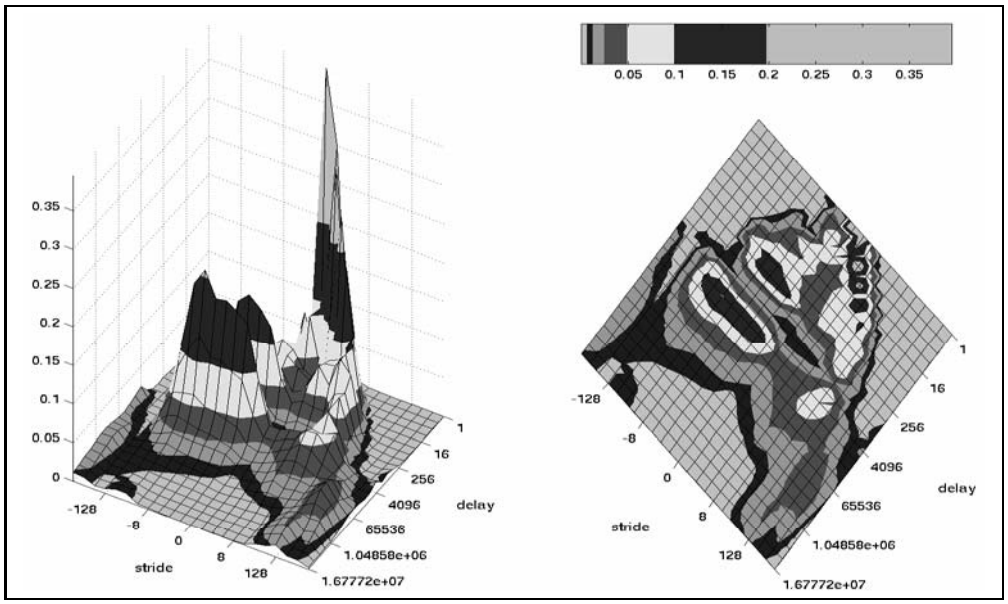


Figure 1.4: Locality surface for the instruction fetches of *twolf* as defined by [40].

Grimsrud defined his surface using the following equation:

$$L(\vec{T}, s, d) = Pr\big(\vec{T}[t_0]+s = \vec{T}[t_0+d] \wedge \vec{T}[t_0]+s \notin \{\vec{T}[t_0+1], \ldots, \vec{T}[t_0+d-1]\}\big) \quad (1.3)$$

where $\vec{T}$ is the trace, s is the stride, d is the delay, and $\vec{T}[t_0]$ is the reference at time t_0 in the trace [40]. Figure 1.4 shows Grimsrud's locality surface for the instruction fetches of *twolf*.

For visualization purposes, the surface is displayed on a log scale in both the stride and delay axis. Two views of the same surface are typically displayed for easier identification of the height and location of various features. The most dramatic difference between Grimsrud's locality surface and other locality metrics is the unification of temporal and spatial locality into one locality function. Numerous two-dimensional graphs would be required to see the same information. On the locality surface, temporal locality becomes a special case of locality that occurs when the stride equals zero. Information about spatial locality may be seen wherever the stride is not zero. The locality surface shows much more information regarding the

spatial locality of the input workload than can ever be seen by one two-dimensional graph.

There are a few problems with the locality surface described by Grimsrud, especially when used for evaluating cache memories. For example, Grimsrud's delay is the total number of values between two given values in a list. On his locality surface, the maximum delay is therefore a function of the length of the trace. When using the locality surface for cache studies, Grimsrud was limited. Caches may have exactly the same number of misses, or the same miss rate, for multiple traces of different lengths. LRU caches are more stack based. In fact, other researchers employing two-dimensional graphs used the number of unique values between two given values as a delay measure [24, 49] and claim that the unique count is better for analyzing caches than the total reference count [16, 30].

We need a better locality metric than the ones here mentioned. In [50], John et al. say one goal of a good locality metric should be to be "useful in predicting cache performance without detailed simulations." To do this, a locality metric should include all aspects of both temporal and spatial locality but also use the unique count that is more useful for cache studies.

1.3 Dissertation Overview

In this dissertation, we re-introduce our improved locality surface originally presented in [74]. In that work, we improved Grimsrud's locality surface by using a unique count rather than total reference count for delay. This better tailors the locality surface for LRU cache studies. We then used it to characterize a few benchmarks from the SPEC CINT2000 suite and qualitatively predicted cache performance. In this dissertation we expand the introduction by adding a detailed mathematical de-

scription of our locality surface. We again characterize benchmarks and perform qualitative cache predictions, only with a different group of workloads that includes both the SPEC CFP2000 suite as well as the SPEC CINT2000 suite.

In [74] we also introduced the cache characterization surface and attempted to quantitatively evaluate cache performance. Cache characterization surfaces provide a visual overview of how the cache performs in terms of locality and independent of any particular workload. However, we were only able to characterize caches with a cache size up to 256 Kbytes and the quantitative prediction had large errors. In this work we create cache characterization surfaces for much larger cache sizes (up to 64 Mbytes). In addition, we model cache performance, which allows us to mathematically show why, for example, quantitative predictions work better for fully-associative caches than for direct-mapped caches. We also examine how two different traces may have the same locality information, and how that affects cache performance prediction. In this work, we focus on L1 caches. The techniques we use, however, may be used for any level of the memory hierarchy that uses an LRU replacement policy and for which traces are available.

Another addition in this work is the evaluation of the effectiveness of a variety of synthetic trace models in terms of locality. One of the few criticisms of the locality surface is the length of time for its calculation. In answer to this, we also present a new parallel algorithm that significantly improves the time to calculate the locality surface for workloads with poor locality.

We first introduce our new locality surface in Chapter 2 with a precise mathematical description. Next we use the locality surface for characterizing a number of workloads from the SPEC C2000 suite in Chapter 3. Chapter 4 demonstrates the value of the locality surface for qualitatively predicting cache simulation results. In Chapter 5 we describe caches mathematically and introduce the improved cache

characterization surface. We next mathematically determine some of the limits of the locality surface in Chapter 6. Then we use the locality surface for quantitative cache simulation prediction in Chapter 7. In Chapter 8 we use the locality surface to evaluate the effectiveness of a number of synthetic trace methods. Chapter 9 details the algorithms used for creating the locality surface, including a new parallel algorithm. In Chapter 10 we conclude and enumerate a number of areas for future work.

Chapter 2

A New Definition of Locality

2.1 Introduction

In the previous chapter, we discussed locality, why it is useful, and the limitations of current locality metrics. In this chapter, we describe a new locality surface using the mathematics of sets and bags and briefly compare our new surface with Grimsrud's locality surface. We treat a trace of memory address references as a string, or a list of integers, which we input into our locality functions.

We assume that the reader is familiar with normal set notation and operations. The notation and definitions for bags are inconsistent in the literature [10, 27, 42, 55, 56] and not as commonly known. Books that contain large sections on set theory usually only mention bags, or multisets, as an aside [35, 71], if at all [77]. Therefore, we first give some basic definitions of terms and notation and some background definitions of a number of bag functions.

2.2 Basic Definitions

Let a **string**, or **vector**, be a list of (not necessarily unique) integers where the order is significant. The **length** of the string is the number of elements in the string. The length of string v is written $|v|$. A string **element** is any individual integer within the string. When referring to the ith element of string v, we write $v[i]$ where $1 \leq i \leq |v|$. Let V indicate the set of all possible such strings.

We use letters near the end of the alphabet, such as v, w, x, y, and z, to indicate strings. Letters near the beginning of the alphabet, such as a, b, and c, usually indicate integers, which may be either indices or individual elements of a string. Capital letters, such as B, C, and T, indicate sets, bags, cache configurations, or surfaces. Hence $|v|$ indicates the length of string v, $|a|$ indicates the absolute value of integer a, and $|S|$ indicates the cardinality of the set S.

Example 2.1. *Let us define a string, v_1, to be equal to 2, 7, 5, 10, 5, 2, 8. Then we write $v_1 = 2, 7, 5, 10, 5, 2, 8$. Also, $|v_1| = 7$, $v_1[1] = 2$, $v_1[2] = 7$, $v_1[3] = 5$, $v_1[4] = 10$, $v_1[5] = 5$, $v_1[6] = 2$, and $v_1[7] = 8$.*

The elements of a string may be any integer, whether positive, negative, or zero. Hence $v_2 = 47, -8, 3, 0, -62, 99, 3, -8, 0, 24$ is also a valid string where $|v_2| = 10$.

We now introduce our own function on a string: the **reverse** of a string. We write $rev(v)$ to indicate the reverse of the string v. The reverse of a string is just like it sounds: the original string is recorded in reverse order. Formally, $w = rev(v)$ when $|w| = |v|$ and $w[i] = v[|v| - i + 1]$ where $1 \leq i \leq |w|$. We now show that taking the reverse twice of a string yields the original string. This result is specifically used in later chapters.

Theorem 2.1. *For any string x, $rev(rev(x)) = x$.*

Proof. Let $y = rev(x)$ and $z = rev(y) = rev(rev(x))$. From the definition of the reverse, we know that $|x| = |y| = |z|$. For convenience, let us define $k = |x|$.

Therefore, for $1 \leq i \leq k$, $y[i] = x[k - i + 1]$ and $z[i] = y[k - i + 1] = x[k - (k - i + 1) + 1] = x[k - k + i - 1 + 1] = x[i]$. Since for $1 \leq i \leq k$ $z[i] = x[i]$, we see that $z = x$ and $rev(rev(x)) = x$. □

Theorem 2.2. *For any string v, if w is a string such that $v = rev(w)$, then $w = rev(v)$.*

Proof. We know that $v = rev(w)$. We reverse both strings to yield $rev(v) = rev(rev(w))$. By Theorem 2.1, we may now write $rev(v) = w$ or $w = rev(v)$. □

2.2.1 Bag Definitions

In the following sections we make heavy use of **bags**, which are sometimes called **multisets** in the literature. A bag has similar notation and operations as a set but allows duplicates. For example, $\{1, 4, 5\}$ is both a set and a bag, whereas $\{1, 4, 1, 7\}$ is a bag but is not a set (since it has a duplicate element). Like a set, the elements of a bag have no order. We now discuss more formal descriptions of bags from the literature, and some functions on bags, using the notation that works best for our applications.

Albert defines a bag as "a collection of elements that may contain duplicates" [10]. Kuchen and Gladitz say that a bag is "a variant of sets, where multiple occurrences of each element are allowed" [57]. Klausner and Goodman say that the number of copies of an element in a bag is called the **multiplicity** of the element in the bag [56]. Albert notes that "while a set is characterized by its membership, a bag is characterized by the multiplicity of its elements." Both Kuchen and Gladitz, in [57], and Klausner and Goodman, in [56], use the notation $\#(b, B)$ to denote the number of occurrences of the element b in the bag B. Further, $b \in B$ is equivalent to $\#(b, B) > 0$.

Example 2.2. *A is a bag such that* $A = \{x, x, y, z\}$. *B is another bag such that* $B = \{(1,7),\ (2,3),\ (2,4),\ (2,4),\ (2,4)\}$. *Then* $\#(x, A) = 2$, $\#(y, A) = 1$, *and* $\#(z, A) = 1$. *Also,* $\#\big((1,7), B\big) = 1$, $\#\big((2,3), B\big) = 1$, *and* $\#\big((2,4), B\big) = 3$. *Note that* $\#\big((1,7), A\big) = 0$ *and* $\#\big((1,3), B\big) = 0$.

2.2.2 Bag Operations

We now define a number of operations that we use in the definition of the locality surface and later in this dissertation. First we formally define **bag equality**. If B_1 and B_2 are bags, then $B_1 = B_2$ iff $\forall b[\#(b, B_1) = \#(b, B_2)]$. Note that the order of the elements in the bag is not important.

Example 2.3. $\{(1,2),\ (2,2),\ (2,2)\} = \{(2,2),\ (1,2),\ (2,2)\} = \{(2,2),\ (2,2),\ (1,2)\}$. *However* $\{(1,2),\ (2,2),\ (2,2)\} \neq \{(1,2),\ (2,2)\}$.

Grumbach and Milo define an operation, **additive union** or $\uplus$ [42], which we here describe using our notation. If B_1 and B_2 are both bags, then $B_1 \uplus B_2$ is a bag such that any element b has the property $\#(b, B_1 \uplus B_2) = \#(b, B_1) + \#(b, B_2)$. In words, the additive union of two bags contains all the elements of the first bag as well as all the elements of the second bag without removing any duplicates.

For the purposes of this work, we say that every set is also a bag. This allows us to take the additive union of two or more sets and obtain a bag.

Example 2.4. *Let us say that* $B_1 = \{(1,7),\ (2,4)\}$ *and* $B_2 = \{(2,3),\ (2,4),\ (2,4)\}$, *then* $B_1 \uplus B_2 = \{(1,7),\ (2,3),\ (2,4),\ (2,4),\ (2,4)\}$.

Next we define **bag subtraction**. When we subtract one bag from another, we proper-subtract the number of occurrences of every element in the second bag from the number of occurrences of that element in the first bag [84]. If B_1 and B_2 are both bags, then $B_1 - B_2$ is a bag such that any element b has the property $\#(b, B_1 - B_2) = max\big(0, \#(b, B_1) - \#(b, B_2)\big)$.

Example 2.5. *Let* $B_1 = \{(-2,1),\ (0,1),\ (0,1),\ (2,1),\ (0,2)\}$. *Let* $B_2 = \{(0,1),\ (9,1)\}$. *Then* $B_1 - B_2 = \{(-2,1),\ (0,1),\ (2,1),\ (0,2)\}$ *and* $B_2 - B_1 = \{(9,1)\}$.

Next we define a function for removing the duplicates from a bag. We may term the result as either a bag or a set. Albert calls this the **duplicate elimination** function [10]. Both Albert [10] and Dayal et al. [27] use δ to indicate duplicate elimination. Formally, $\delta(B) = \{b | b \in B\}$.

Example 2.6. *Let* B_1 *be as defined in Example 2.5. Let* $B_2 = \{4,\ 3,\ 2,\ 4,\ 3,\ 4\}$. *Then* $\delta(B_1) = \{(-2,1),\ (0,1),\ (2,1),\ (0,2)\}$ *and* $\delta(B_2) = \{4,\ 3,\ 2\}$

Next we define **bag select**. In [31], Dyreson writes "The selection operation selects [elements] from a [bag] that fit some criteria, creating a new [bag] with the selected [elements]." We use the standard symbol for select, σ. We put our select criteria in the sigma's subscript. So $\sigma_\lambda(B)$ is a bag that contains all the elements of B that satisfy the condition λ. Since B is a bag, we specify that the number of each element in $\sigma_\lambda(B)$ is the same as the number of that element in B. Formally, if B is a bag and λ is a condition, then $\sigma_\lambda(B)$ is a bag such that $\#(b, \sigma_\lambda(B)) = \#(b, B)$ if b satisfies λ and $\#(b, \sigma_\lambda(B)) = 0$ if b does not satisfy λ.

Example 2.7. *Let* $B_3 = \{(-2,1),\ (0,1),\ (0,1),\ (2,1),\ (0,2)\}$, *where each ordered pair contains an* x *component and a* y *component designated* (x, y). *Then* $\sigma_{x=0}(B_3) = \{(0,1),\ (0,1),\ (0,2)\}$, $\sigma_{y=2}(B_3) = \{(0,2)\}$, *and* $\sigma_{x=0 \wedge y=1}(B_3) = \{(0,1),\ (0,1)\}$.

Now we define **projection**. Most of the bags we use in this dissertation contain ordered pairs as elements. We can use the projection function to create a new bag that contains only the first or second part of each ordered pair in the original bag. Gersting writes that "The project operation creates a new relation made up of certain attributes from the original relation, eliminating any duplicate tuples" [35, page 282]. Our projection operation does *not* remove duplicates. In our terminology,

a relation is a bag that contains elements that have multiple parts, such as a bag with ordered pairs as elements. We use the following notation to indicate a projection, $\pi_L(B)$, where L indicates which half of the ordered pair is desired and B indicates the bag of ordered pairs [84].

Example 2.8. *Let B_3 be defined as in Example 2.7. Then $\pi_x(B_3) = \{-2, 0, 0, 0, 2\}$ and $\pi_y(B_3) = \{1, 1, 1, 1, 2\}$.*

We now define an operation of our own, called **manipulation**. There are times where we have a bag of ordered pairs and we wish to perform some mathematical operation on each of the first elements of the ordered pairs, or each of the second elements. We use the following notation to designate a mathematical manipulation, $\mu_E(B)$, where E indicates some expression that indicates which part of the ordered pair is to be manipulated and how, and B indicates the bag of ordered pairs.

Example 2.9. *Let B_3 be as defined in Example 2.7. Then $\mu_{x=2x}(B_3) = \{(-4, 1), (0, 1), (0, 1), (4, 1), (0, 2)\}$ and $\mu_{x=-x \wedge y=y+1}(B_3) = \{(2, 2), (0, 2), (0, 2), (-2, 2), (0, 3)\}$.*

2.3 Locality Definitions

Now that our background definitions are complete, we are ready to build our definition of locality. We begin by defining stride and delay.

The **stride** is the difference between two separate string elements, where the former element is subtracted from the latter. Formally,

$$stride(v[a], v[b]) = v[b] - v[a] \tag{2.1}$$

where $v \in V$, a and b are both valid indices of the string, and $a < b$. $stride(v[a], v[b])$ is undefined when $a \geq b$. Since the elements of the string may be any integer, the stride may also be any integer, whether positive, negative, or zero.

Example 2.10. *Let* $v_1 = 2, 7, 5, 10, 5, 2, 8$ *as in Example 2.1. Then* $stride(v_1[1], v_1[2]) = 7 - 2 = 5$, $stride(v_1[3], v_1[5]) = 5 - 5 = 0$, *and* $stride(v_1[2], v_1[6]) = 2 - 7 = -5$. $stride(v_1[4], v_1[4])$ *is undefined, since* $4 = 4$. $stride(v_1[3], v_1[1])$ *is undefined, since* $1 < 3$.

A **delay** exists between two separate elements of a string, as long as neither element is equal to another element between the two. Formally, $delay(v[a], v[b])$ is defined where $v \in V$, a and b are valid indices of the string, $a < b$, and $\forall i(a < i < b)(v[a] \neq v[i] \wedge v[b] \neq v[i])$. Note that $v[a]$ may be equal to $v[b]$. When defined, the delay is the number of unique elements between two string elements, inclusive of the earlier element and exclusive of the later. Formally,

$$delay(v[a], v[b]) = \begin{cases} |\delta(\{v[a] \cdots v[b-1]\})| & \text{if } (a < b) \wedge \big(\forall i(a < i < b)(v[a] \neq v[i] \wedge v[b] \neq v[i])\big), \\ \text{undefined} & \text{otherwise,} \end{cases} \tag{2.2}$$

where $v \in V$ and both a and b are valid indices of v. On the right-hand side of the equation, we put all the elements of the string v between $v[a]$ and $v[b-1]$, inclusive, into a bag, then remove any duplicates and take the cardinality of the resulting set. This yields the number of unique elements in v between $v[a]$ and $v[b-1]$, inclusive.

Property 2.1. *For any string* v *and valid indices* a *and* b, $1 \leq delay(v[a], v[b]) \leq b - a$ *when the delay is defined.*

For the delay to be defined, we require $a < b$. *This means that the number of unique elements between* $v[a]$ *and* $v[b-1]$, *inclusive, is always at least one.*

The delay is at its maximum when every element is unique. Since we count all the elements between $v[a]$ *and* $v[b-1]$, *inclusive, the maximum value is* $b - a$.

Example 2.11. *Using* v_1 *from Example 2.1,* $delay(v_1[1], v_1[5])$ *is not defined because* $v_1[3] = v_1[5]$, $delay(v_1[1], v_1[7])$ *is not defined because* $v_1[1] = v_1[6]$, *and*

$delay(v_1[4], v_1[3])$ is not defined because $3 < 4$. The following is a list of all the delays that are defined in v_1, any delay not listed is undefined:

$$\begin{gathered}
delay(v_1[1], v_1[2]) = 1, \\
delay(v_1[1], v_1[3]) = 2, \\
delay(v_1[1], v_1[4]) = 3, \\
delay(v_1[1], v_1[6]) = 4, \\
delay(v_1[2], v_1[3]) = 1, \\
delay(v_1[2], v_1[4]) = 2, \\
delay(v_1[2], v_1[6]) = 3, \\
delay(v_1[2], v_1[7]) = 4, \\
delay(v_1[3], v_1[4]) = 1, \\
delay(v_1[3], v_1[5]) = 2, \\
delay(v_1[4], v_1[5]) = 1, \\
delay(v_1[4], v_1[6]) = 2, \\
delay(v_1[4], v_1[7]) = 3, \\
delay(v_1[5], v_1[6]) = 1, \\
delay(v_1[5], v_1[7]) = 2, \textit{ and} \\
delay(v_1[6], v_1[7]) = 1.
\end{gathered}$$

The **stride/delay combination** is an ordered pair containing the stride and delay for two separate elements of a string. It exists if and only if the stride for the two elements and the delay for the two elements are defined. If the stride/delay combination exists, it is displayed as: $(stride(v[a], v[b]), delay(v[a], v[b]))$ where $v \in V$ and a and b are valid indices of v. When the stride/delay combination does not exist, for example when the delay is undefined, we say that the stride/delay combination is undefined. For notational convenience, we write $s/d(v[a], v[b])$ to indicate the stride/delay combination for the two elements $v[a]$ and $v[b]$. Formally,

$$s/d(v[a], v[b]) = \begin{cases} (stride(v[a], v[b]), delay(v[a], v[b])) & \text{if } delay(v[a], v[b]) \text{ is defined,} \\ \text{undefined} & \text{otherwise.} \end{cases} \tag{2.3}$$

Notice that the stride is always defined when the delay is defined, so it is sufficient to say that the stride/delay combination is defined when the delay is defined. Note also that while the stride may be any integer, the delay is always greater than or equal to one. So $(2, -1)$ is not a valid stride/delay combination.

Example 2.12. *Using v_1 from Example 2.1, the stride/delay combination for $v_1[1]$ and $v_1[2]$ is $(5, 1)$, the stride/delay combination for $v_1[2]$ and $v_1[6]$ is $(-5, 3)$, and the stride/delay relationship for $v_1[2]$ and $v_1[5]$ is undefined since delay$(v_1[2], v_1[5])$ is undefined. As mentioned earlier, we can also write this as $s/d(v_1[1], v_1[2]) = (5, 1)$, $s/d(v_1[2], v_1[6]) = (-5, 3)$, and $s/d(v_1[2], v_1[5])$ is undefined.*

Since the stride/delay combination is an ordered pair, one stride/delay combination is equal to another only if both the stride and delay are equal. Formally, $(a, b) = (c, d)$ iff $(a = c) \wedge (b = d)$.

Example 2.13. *The stride/delay combination $(0, 4)$ is not equal to the stride/delay combination $(0, 3)$ since $4 \neq 3$. The stride/delay combination $(-2, 7)$ is equal to the stride/delay combination $(-2, 7)$ since $-2 = -2$ and $7 = 7$.*

The **locality data** for a particular element of a string, $v[a]$, is a set of all the valid stride/delay combinations with an earlier element in the string. Formally,

$$\ell(v[a]) = \{s/d(v[i], v[a]) \mid (1 \leq i < a) \wedge \textit{delay}(v[i], v[a]) \text{ is defined}\}. \tag{2.4}$$

We now mention a couple of properties of the locality data of individual elements of strings. These properties are true for any string $v \in V$.

Property 2.2. *For any string v, $\ell(v[1]) = \emptyset$.*

The first element of a string never has any earlier elements with which to have stride/delay relationships.

Theorem 2.3. *The delay portions of the locality data for a particular string element is always a set, i.e. a multiset with no duplicates.*

Proof. More formally, for any string v, if we let $D = \pi_d\big(\ell(v[i])\big)$ then for any delay $d \in D$ we know that $\#(d, D) = 1$. In other words, we never see a duplicate delay within the locality data for an individual string element. We now show this by contradiction.

Assume that a particular string element, $v[i]$, has the same delay with two earlier string elements, namely $v[a]$ and $v[b]$. We formally write this $\mathit{delay}(v[a], v[i]) = \mathit{delay}(v[b], v[i])$ where $a < b < i$. This means that the number of unique elements between $v[a]$ and $v[i]$ is equal to the number of unique elements between $v[b]$ and $v[i]$. Formally, we write this as

$$|\delta(\{v[a] \cdots v[b] \cdots v[i-1]\})| = |\delta(\{v[b] \cdots v[i-i]\})|.$$

For this to be true, all the elements from $v[a]$ to $v[b-1]$ must be elements that are already between $v[b]$ and $v[i-1]$. More formally, $\{v[a] \cdots v[b-1]\} \subseteq \{v[b] \cdots v[i-1]\}$. Therefore we know that $\{v[a]\} \subseteq \{v[a+1] \cdots v[i-1]\}$. In other words, $v[a]$ is equal to some element between $v[a]$ and $v[i]$, meaning that $\mathit{delay}(v[a], v[i])$ is undefined.

We have now reached a contradiction, which means that the delay between a particular element and any earlier element in the string is unique. We may therefore conclude that the delay portions of the locality data for a particular string element is always a set. □

Theorem 2.4. *The stride portions of the locality data for a particular string element is always a set, i.e. a multiset with no duplicates.*

Proof. More formally, for any string v, if we let $S = \pi_s\big(\ell(v[i])\big)$ then for any stride $s \in S$ we know that $\#(s, S) = 1$. In other words, we never see a duplicate stride within the locality data for an individual string element.

Recall from Equation 2.4 that a stride/delay relationship is only included in the locality data for a given element if the delay is defined. We now prove, by contradiction, that the stride is never duplicated.

Let us assume that we have two stride/delay relationships in $\ell(v[i])$ such that the stride is the same. We let these two relationships be (s, d_1) and (s, d_2). Note

that $d_1 \neq d_2$, from Theorem 2.3. We let $d_1 > d_2$. We let $v[j]$ and $v[k]$ be the earlier elements of v that cause these stride/delay relationships. Specifically, $s/d(v[j], v[i]) = (s, d_1)$ and $s/d(v[k], v[i]) = (s, d_2)$. Since $d_1 > d_2$, we know that $j < k < i$. From Equation 2.1, we can write that $v[i] - v[j] = s$ and $v[i] - v[k] = s$. Algebra now tells us that $v[j] = v[k]$. We can now see that $delay(v[j], v[i])$ is undefined, since $v[k]$ is between $v[j]$ and $v[i]$ and is equal to $v[j]$. From Equation 2.4, $s/d(v[j], v[i])$ is not included in $\ell(v[i])$, which is a contradiction.

Since any duplicate stride leads to a contradiction, we have shown that the stride portions of the locality data for a particular string element is always a set. □

Theorem 2.5. *The locality data for a particular string element is always a set, i.e. a multiset with no duplicates.*

Proof. In other words, for any string v and stride/delay combination (s, d), if $(s, d) \in \ell(v[i])$ then $\#\big((s, d), \ell(v[i])\big) = 1$. This is easy to see, given either Theorem 2.3 or Theorem 2.4. If either all the delays are unique or all the strides are unique, then surely all the stride/delay relationships are also unique. Since the bag of stride/delay relationships has no duplicates, then it is also a set. □

We now give an example showing the locality data for each element of a particular string. Notice the validity of Property 2.2 and Theorems 2.3 – 2.5.

Example 2.14. *Using v_1 from Example 2.1,*
$\ell(v_1[1]) = \emptyset$,
$\ell(v_1[2]) = \{(5, 1)\}$,
$\ell(v_1[3]) = \{(-2, 1), (3, 2)\}$,
$\ell(v_1[4]) = \{(5, 1), (3, 2), (8, 3)\}$,
$\ell(v_1[5]) = \{(-5, 1), (0, 2)\}$,
$\ell(v_1[6]) = \{(-3, 1), (-8, 2), (-5, 3), (0, 4)\}$, *and*
$\ell(v_1[7]) = \{(6, 1), (3, 2), (-2, 3), (1, 4)\}$.

Remember that the elements in a set are not ordered, hence $\{(-4, 1), (3, 2)\} = \{(3, 2), (-4, 1)\}$. For convenience, in this work we display sets of stride/delay relationships ordered by the delay. However, this is only to simplify determining if a particular stride/delay relationship is a member of the set and should not imply ordering.

Theorem 2.6. *The locality data for a given string element yields the same result as using an LRU stack to compute delays if stack traversal stops when an element on the stack with the same value is reached.*

Proof. Recall that $v[a]$ is the ath member of the string v. If $a = 1$, then there is no stack and no locality data for $v[a]$ (Property 2.2). Therefore, assume that $a > 1$. The top of the LRU stack at this point is the immediately previous element in the string, i.e. $v[a-1]$. We can easily see that $delay(v[a-1], v[a])$ is always defined, since there is nothing between $v[a-1]$ and $v[a]$. Also, the delay is 1, which is also the depth in the LRU stack of the top element.

Let $v[i]$ be any earlier element of v. If $v[a]$ is equal to some element between $v[i]$ and $v[a]$, then the stack traversal ceases when the element equal to $v[a]$ is reached, and the delay between $v[i]$ and $v[a]$ is undefined. If $v[i]$ is equal to some element between $v[i]$ and $v[a]$, then $v[i]$'s position in the LRU stack was replaced by that other element, and the delay is undefined. If neither $v[a]$ nor $v[i]$ are equal to any elements between them, then the depth in the stack is equivalent to the number of unique elements between $v[i]$ and $v[a-1]$, inclusive. Hence our definition of delay is equivalent to computing delay using an LRU stack, if stack traversal ceases when the same element is reached on the stack. □

The locality data for an entire string v is a bag, or multiset, of all the stride/delay combinations that exist in v. We can also think of it as the additive union of the

locality data for each element of the string. Formally,

$$\ell(v) = \biguplus_{i=1}^{|v|} \ell(v[i]), \tag{2.5}$$

where $v \in V$.

Example 2.15. *Using v_1 from Example 2.1,* $\ell(v_1) = \{(-5,1),\ (-3,1),\ (-2,1),\ (5,1),\ (5,1),\ (6,1),\ (-8,2),\ (0,2),\ (3,2),\ (3,2),\ (3,2),\ (-5,3),\ (-2,3),\ (8,3),\ (0,4),\ (1,4)\}$.

Remember that the elements of a bag are not ordered. As with sets, we display bags of stride/delay relationships ordered primarily by the delay. Again, this simplifies determining if a particular stride/delay relationship is a member of the bag, and also helps us enumerate how many of a given relationship are in the bag. This ordering for convenience should not imply that elements of a bag are ordered.

We now prove that the locality data of a reversed string is the same as the locality data of the original string except with the sign of the stride reversed.

Theorem 2.7. *For any string v,*

$$\ell(rev(v)) = \mu_{s=-s}\ell(v).$$

Proof. Let v be any string and $w = rev(v)$. From Theorem 2.2, we also know that $v = rev(w)$. First we show that $\ell(w) \subseteq \mu_{s=-s}\ell(v)$ by showing that any stride/delay relationship in $\ell(w)$ is also in $\ell(v)$ if the stride portion is multiplied by -1. Let (s,d) be some stride/delay relationship in $\ell(w)$. Then (s,d) must be a member of the locality data for some element of w. We call this element $w[i]$, so that $(s,d) \in \ell(w[i])$. We now show that there exists some j such that $(-s,d) \in \ell(v[j])$.

Since $(s,d) \in \ell(w[i])$, there must be an earlier element of w, let us call it $w[a]$, such that $a < i$, $stride(w[a], w[i]) = w[i] - w[a] = s$, and $delay(w[a], w[i]) = d$. Since v and w are reverses of each other, we know that $|v| = |w|$. Let $k = |v|$. Therefore,

$w[a] = v[k-a+1]$ and $w[i] = v[k-i+1]$. Since $a < i$, we know that $k-i+1 < k-a+1$. Then $stride(v[k-i+1], v[k-a+1]) = v[k-a+1] - v[k-i+1] = w[a] - w[i] = -s$.

Since v and w are reverses of each other, we know that the same elements that are between $w[a]$ and $w[i]$ are are also between $v[k-i+1]$ and $v[k-a+1]$, only in reverse order. Since $delay(w[a], w[i])$ is defined, we know that neither $w[a]$ nor $w[i]$ is equal to an element between $w[a]$ and $w[i]$. This means that neither $v[k-i+1]$ nor $v[k-a+1]$ is equal to an element between $v[k-i+1]$ and $v[k-a+1]$. Recall that $k-i+1 < k-a+1$. Therefore, we know that $delay(v[k-i+1], v[k-a+1])$ is defined. For the same reason, we can also say that $delay(w[a], w[i]) = delay(v[k-i+1], v[k-a+1])$. Since the elements that exist between $w[a]$ and $w[i]$ are the same elements that are between $v[k-i+1]$ and $v[k-a+1]$, the number of unique elements between them is also the same. So $delay(v[k-i+1], v[k-a+1]) = d$.

If we let $j = k-a+1$, then we may say that $(-s, d) \in \ell(v[j])$. We have now shown that $\ell(w) \subseteq \mu_{s=-s}\ell(v)$, if $w = rev(v)$.

Since $v = rev(w)$, we may also say that $\ell(v) \subseteq \mu_{s=-s}\ell(w)$. If we multiply the stride portion of each of the stride/delay relationships on both sides of this equation by $-s$, we get $\mu_{s=-s}\ell(v) \subseteq \mu_{s=-(-s)}\ell(w)$. We may rewrite this as $\mu_{s=-s}\ell(v) \subseteq \ell(w)$.

Since $\ell(w)$ and $\mu_{s=-s}\ell(v)$ are both subsets of each other, we may conclude that $\ell(w) = \mu_{s=-s}\ell(v)$. Replacing w with $rev(v)$, we finish with $\ell(rev(v))\mu_{s=-s}\ell(v)$, as desired. □

We believe that our definition of locality data incorporates all interesting aspects of temporal and spatial locality. For each element of a string, its stride/delay relationship with the most recent instance of each earlier element value is recorded in the locality bag. Now we need a useful way to view, store, and analyze all this data.

2.4 Visualizing Locality Data

When dealing with strings that are millions, even billions, of integers long, the locality data bag is quite large. For example, the locality data for about 50 million instructions of *twolf*, from the SPEC CINT 2000 suite, contains over 289 million stride/delay relationships. We represent this information visually using several steps. First, we generate a three-dimensional histogram by counting how many of each stride/delay combination exist in $\ell(v)$. For large strings, there is still the potential for millions of stride/delay combinations, so we group some of them together. We use the visualization technique developed by Grimsrud [38] to display his locality surfaces. While the computation of our locality surfaces is significantly different, Grimsrud's display method works well for our data and also allows easy comparisons with his work.

Since we want extra detail near the origin, where $stride = 0$ and $delay = 1$, we group logarithmically in both the stride and delay direction. Now the problem is that those bins at large delays and large strides contain many more stride/delay combinations, making it difficult to view the data near the origin. Grimsrud found that the best way to even this out for the type of input strings he used was to divide each bin by the size of the range of strides that are in the bin. Since our input strings are very similar to Grimsrud's, we have found his method to be useful. In order to view the results as a percentage, we also used Grimsrud's method of dividing by the total number of elements in the original trace minus one.

In later chapters, we use this visualization method for other bags besides a bag of locality data. For this reason, our visualization functions operate on what we term a **stride/delay bag**. This is a bag that consists entirely of stride/delay combinations. So far, the locality data of a string is the only stride/delay bag that

		Stride																
		-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8
Delay	1				1		1	1							2	1		
	2	1								1			3					
	3				1			1										1
	4									1	1							

Table 2.1: The locality histogram for the string v_1 from Example 2.1. We write $h(\ell(v_1))$ to indicate the entire histogram.

we have encountered. In Chapter 5, however, we define a **miss bag** that is another stride/delay bag. When the following visualization functions are used with bags of locality data, the final result is called the **locality surface** of the input string. We now describe each visualization step in detail.

2.4.1 Making the Histogram

Let $h(B)$ represent the three-dimensional **histogram** that counts the number of occurrences of each stride/delay relationship in the bag B. When B is the locality data for some string v, we may term $h(B)$ the **locality histogram**. Let $h(B, s, d)$ represent an individual entry in the histogram, namely the count of how many of the specific stride/delay combination (s, d) are in the bag B. Note that s may be any integer, positive or negative, while $d > 0$. Formally,

$$h(B, s, d) = \#\big((s, d), B\big) \tag{2.6}$$

where B is a locality bag.

Example 2.16. *Recall from Examples 2.1 and 2.15 that* $v_1 = 2, 7, 5, 10, 5, 2, 8$ *and* $\ell(v_1) = \{(-5, 1), (-3, 1), (-2, 1), (5, 1), (5, 1), (6, 1), (-8, 2), (0, 2), (3, 2), (3, 2), (3, 2), (-5, 3), (-2, 3), (8, 3), (0, 4), (1, 4)\}$. *Then* $h(\ell(v_1), 0, 2) = 1$, $h(\ell(v_1), 3, 2) = 3$, *and* $h(\ell(v_1), -6, 1) = 0$. *Table 2.1 shows the complete results.*

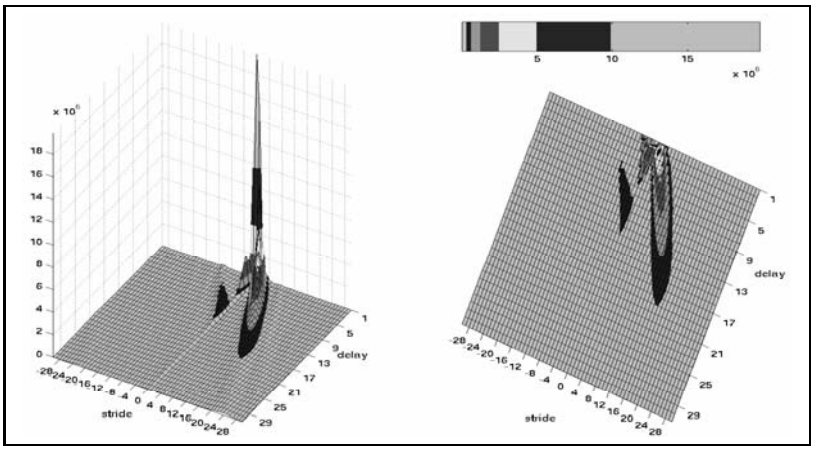


Figure 2.1: Example of the raw locality histogram for a real trace, the instruction trace of *twolf*. This figure merely shows a small part of the actual histogram.

Example 2.17. *Figure 2.1 shows a small part of the histogram for the instruction trace of the real workload twolf. (Twolf is a place and route simulator from the SPEC CINT 2000 benchmark suite.) There is not enough room to display the entire histogram, since the histogram ranges from* 1 *to over* 4000 *in the delay direction and from* -128 *to* 128 *in the stride direction. Figure 2.1 limits the delay direction to a maximum of* 31 *and the stride direction to a maximum absolute value of* 30*.*

Theorem 2.8. *For any string* v*,*

$$\sum_{s=-\infty}^{\infty} h(\ell(v), s, 1) = |v| - 1.$$

Proof. This says that the locality data for any string v always has $|v|-1$ stride/delay relationships where the delay is 1. This is because every element of the string, except the first element, has exactly one stride/delay relationship where $delay = 1$. The first element of the string does not, due to Property 2.2.

For any element of an arbitrary string, v, if $2 \leq i \leq |v|$, then $delay(v[i-1], v[i]) = 1$. This delay is always defined because $i - 1 < i$ and there are no elements between $v[i-1]$ and $v[i]$ that either element can equal. The delay is always 1 since we are counting the number of unique elements between $v[i-1]$ and $v[i]$, inclusive of $v[i-1]$

and exclusive of $v[i]$. In this case, there is always exactly one element to count, i.e. $v[i-1]$. □

Theorem 2.9. *For any string v and delay $d \geq 2$,*

$$\sum_{s=-\infty}^{\infty} h(\ell(v), s, d) = max\left(0, \left(\sum_{s=-\infty}^{-1} h(\ell(v), s, d-1) + \sum_{s=1}^{\infty} h(\ell(v), s, d-1) - 1\right)\right). \tag{2.7}$$

Proof. In words, this property means that counting all the stride/delay relationships at a particular delay is equivalent to zero or one less than the number of stride/delay relationships with a delay one smaller where the stride is not zero, whichever is greater.

Let d be the delay we are concerned with. Therefore, $d-1$ is the delay one smaller. If a particular element of the given string v has a stride/delay relationship with an earlier element where the delay is $d-1$, there are only two circumstances where that same element of v does not have a stride/delay relationship where the delay is d. Formally, if $(s_1, d-1) \in \ell(v[i])$, there are only two reasons for $(s_2, d) \notin \ell(v[i])$ where s_1 and s_2 are unknown stride values. Both reasons relate to the fact that the stride/delay relationships that are in $\ell(v[i])$ represent a stack traversal of earlier elements in v, if the stack traversal stops when an element with equal value is reached. (See Theorems 2.6 and 2.5.)

The first reason occurs if $s_1 = 0$. This means that an element with the same value as $v[i]$ is found in the stack, at depth $d-1$. Stack traversal stops at this point, and no stride/delay relationships are computed for $v[i]$ for any larger delays. Hence $v[i]$ cannot have a stride/delay relationship where the delay is d. Each time this situation occurs there is a $(s_1, d-1)$ stride/delay relationship in $\ell(v[i])$. We know that there are $\#\big((0, d-1), \ell(v)\big)$ less stride/delay relationships where delay is d than where delay is $d-1$ due to this reason.

The second reason occurs if $(s_1, d-1)$ represents the stride/delay relationship between the element at the bottom of the stack and $v[i]$. This means that the stack is not yet deep enough to create a d delay because there are only $d-1$ elements in the stack. When the delay minus one stride/delay relationships have non-zero strides, this situation may occur only once for a given delay value. If the value of $v[i]$ was already in the stack, this situation would never occur because a stride zero would have been found. Therefore, $v[i]$ must be a new value. When it is added to the stack, the depth of the stack is increased by one and now able to create a delay of d for future elements in v. This accounts for the subtraction of one in Equation 2.7.

When the delay minus one stride/delay relationships have only zero strides, i.e. there is no stride/delay relationship $(s_1, d-1)$ where $s_1 \neq 0$, then this situation does not occur at all. In this case, counting all the non-zero-stride stride/delay relationships results in zero. Subtracting one in Equation 2.7 yields an answer of -1. We correct this by taking the maximum with zero, so the final answer is 0, as if one were not subtracted at all. □

2.4.2 Binning the Data

We now reduce the resolution of the locality surface as stride and delay become large by binning. Let $H(B)$ represent the **binned histogram** of the stride/delay bag B. When B is the locality data for some string v, we may term $H(B)$ the **binned locality histogram**. We decided to center our binning around the stride/delay combination $(0, 1)$, which is the smallest absolute value stride and the smallest delay. Since we group in two dimensions (stride and delay), each bin receives a stride label and a delay label for indexing the bin. The following paragraphs describe the binning in English, and Equation 2.8 describes the binning mathematically.

We group in the delay direction by leaving $delay = 1$ and $delay = 2$ as separate bins, with delay labels of 1 and 2 respectively. We bin $3 \leq delay \leq 4$ together and give it a delay label of 3, bin $5 \leq delay \leq 8$ together and give it a delay label of 4, etc. The $delay = 1$ case is addressed in Lines 1-3 of Equation 2.8. The other delay cases are addressed in Lines 4-6 of Equation 2.8.

We group in the stride direction similarly, but it is a bit more complex due to the negative strides. We leave $stride = 0$, $stride = \pm 1$, and $stride = \pm 2$ as five separate bins, and give them stride labels 0, 1, -1, 2, and -2 respectively. We then group in both the positive and negative directions, binning $3 \leq stride \leq 4$ together and giving it a stride label of 3, binning $-3 \geq stride \geq -4$ together and giving it a stride label of -3, binning $5 \leq stride \leq 8$ together and giving it a stride label of 4, binning $-5 \geq stride \geq -8$ together and giving it a stride label of -4, etc. Lines 1 and 4 of Equation 2.8 address the cases when $-1 \leq stride \leq 1$. Lines 2 and 5 of Equation 2.8 address the cases when $stride > 1$. Lines 3 and 6 of Equation 2.8 address the cases when $stride < -1$.

Let $H(B, a, b)$ represent a bin where a is the stride bin label and b is the delay bin label. Then,

$$H(B,a,b) = \begin{cases} h(B,a,b) & \text{if } (-1 \leq a \leq 1 \wedge b = 1), \\ \sum_{s=2^{a-2}+1}^{2^{a-1}} h(B,s,b) & \text{if } (a > 1 \wedge b = 1), \\ \sum_{s=-(2^{-a-1})}^{-(2^{-a-2}+1)} h(B,s,b) & \text{if } (a < -1 \wedge b = 1), \\ \sum_{d=2^{b-2}+1}^{2^{b-1}} h(B,a,d) & \text{if } (-1 \leq a \leq 1 \wedge b > 1), \\ \sum_{s=2^{a-2}+1}^{2^{a-1}} \sum_{d=2^{b-2}+1}^{2^{b-1}} h(B,s,d) & \text{if } (a > 1 \wedge b > 1), \\ \sum_{s=-(2^{-a-1})}^{-(2^{-a-2}+1)} \sum_{d=2^{b-2}+1}^{2^{b-1}} h(B,s,d) & \text{if } (a < -1 \wedge b > 1). \end{cases} \tag{2.8}$$

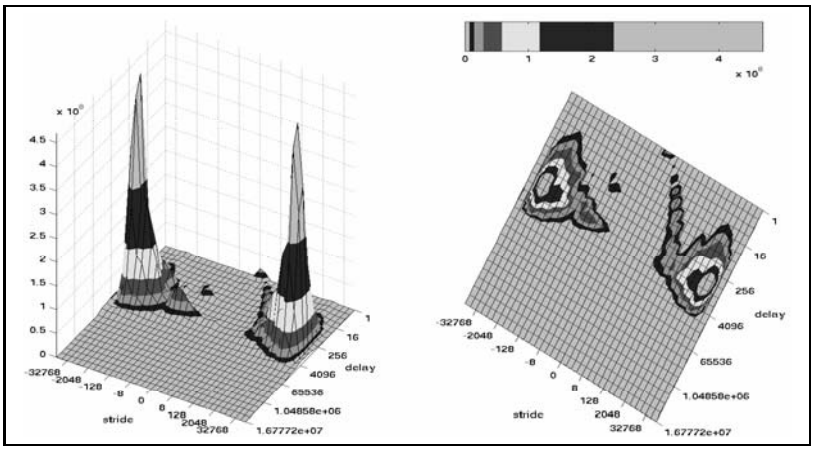


Figure 2.2: Example of the binned locality histogram for a real trace, the instruction trace of *twolf*.

Stride

Delay	-8 − -5	-4 − -3	-2	-1	0	1	2	3 − 4	5 − 8
1	1	1	1	0	0	0	0	0	3
2	1	0	0	0	1	0	0	3	0
3 - 4	1	0	1	0	1	1	0	0	1

Table 2.2: The binned locality histogram for the string v_1 from Example 2.1. We write $H(\ell(v_1))$ to indicate the binned locality histogram for v_1.

Example 2.18. *Table 2.2 shows the binned locality histogram for the string v_1 from Example 2.1.*

Example 2.19. *Figure 2.2 shows the binned locality histogram for the instruction trace of twolf. We can now see the entire range of data instead of the small section shown in Figure 2.1. We typically label each bin using largest (s, d) values assigned to the bin rather than the (a, b) values, since this is more descriptive of the input data. We have allowed the stride range to be large enough to see all the visible features. Notice the high spikes at large strides. The binned locality histogram appears as two such spikes for all workloads we have seen. To get information that allows us to characterize the differences of various workloads, we need one more step.*

Theorem 2.10. *For any string v,*

$$\sum_{a=-\infty}^{\infty} H(\ell(v), a, 1) = |v| - 1.$$

Proof. We can show this by using the definition of $H(B)$ and Property 2.8:

$$\begin{aligned}
\sum_{a=-\infty}^{\infty} H(\ell(v), a, 1) &= \sum_{a=-\infty}^{-2} \sum_{s=-(2^{-a-2})}^{-(2^{-a-2}+1)} h(\ell(v), s, 1) + \sum_{a=-1}^{1} h(\ell(v), a, 1) + \\
&\quad \sum_{a=2}^{\infty} \sum_{s=2^{a-2}+1}^{2^{a-1}} h(\ell(v), s, 1) \\
&= \sum_{a=-\infty}^{-2} h(\ell(v), a, 1) + \sum_{a=-1}^{1} h(\ell(v), a, 1) + \sum_{a=2}^{\infty} h(\ell(v), a, 1) \\
&= \sum_{a=-\infty}^{\infty} h(\ell(v), a, 1) \\
&= |v| - 1.
\end{aligned}$$

□

2.4.3 Normalizing the Bins

Now we normalize the bins in the stride direction by dividing each bin by the size of the range of strides that it contains. The bin with a stride label of 0 contains only one stride value (i.e. 0), so we divide it by one. The bin with a stride label of -5 is assigned eight stride values (i.e. $-16 \leq \mathit{stride} \leq -9$), so we divide this bin by eight. In addition, we normalize the entire surface by dividing every bin by one less than the length of the input string, resulting in a percentage. We term the result a **surface**, represented by $\mathcal{S}(B, v)$ where B is a locality bag and v is the string whose length we wish to use. $\mathcal{S}(B, v)$ is the entire surface, and $\mathcal{S}(B, v, a, b)$ is a specific bin on the surface.

$$\mathcal{S}(B, v, a, b) = \begin{cases} \frac{H(B,a,b)}{(|v|-1)\cdot(2^{|a|-2})} & \text{if } |a| > 1, \\ \frac{H(B,a,b)}{|v|-1} & \text{if } |a| \leq 1 \end{cases} \tag{2.9}$$

		Stride								
		$-8 - -5$	$-4 - -3$	-2	-1	0	1	2	$3 - 4$	$5 - 8$
Delay	1	0.0417	0.0833	0.1667	0	0	0	0	0	0.1250
	2	0.0417	0	0	0	0.1667	0	0	0.2500	0
	3 - 4	0.0417	0	0.1667	0	0.1667	0.1667	0	0	0.0417

Table 2.3: The locality surface for the string v_1 from Example 2.1. We write either $\mathcal{S}(\ell(v_1), v_1)$ or $\mathcal{L}(v_1)$ to indicate this locality surface.

where B is a locality bag, $v \in V$, and a and b are integers such that $1 \leq b$. Note that for a bag of locality data $|v| - 1$ is the number of stride/delay relationships where $delay = 1$, as we saw in Properties 2.8 and 2.10. So $\mathcal{S}(\ell(v), v, 0, 1)$ is the percentage of $delay = 1$ stride/delay relationships where $stride = 0$.

What type of surface $\mathcal{S}(B, v)$ is depends on what type of stride/delay bag is used. When B is a bag of locality data, we call $\mathcal{S}(B, v)$ a locality surface. Because we refer to locality surfaces frequently, we now define a function that takes a string and returns the locality surface of that string. Formally, $\mathcal{L}(v) = \mathcal{S}(\ell(v), v)$ and $\mathcal{L}(v, a, b) = \mathcal{S}(\ell(v), v, a, b)$. This new function simplifies the notation.

Example 2.20. *Table 2.3 shows the locality surface for the string v_1 from Example 2.1.*

Example 2.21. *Figure 2.3 shows the locality surface for the instruction trace of twolf. Now we can see significant features, identify the size and relative frequency of loops, and make better comparisons with the locality surfaces of other workloads. Compare this surface with Grimsrud's surface for the same workload, Figure 1.4. Notice that our new locality surface has sharper, more well-defined features. When we examine the cache results for this workload in Chapter 4, it will be readily apparent to the reader that the cache results directly match our locality surface and not Grimsrud's. The maximum stride and delay were chosen to be consistent with the locality surfaces of all the instruction traces in this dissertation. In Section 3.3 we explain why these values were chosen.*

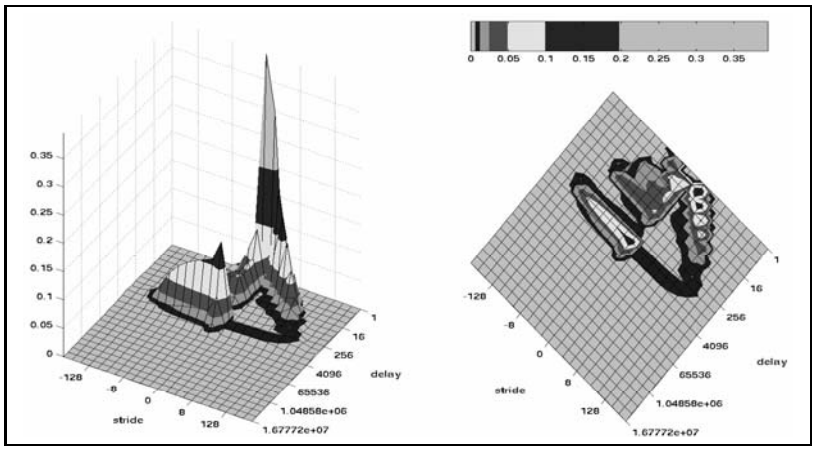


Figure 2.3: Example of the locality surface for a real trace, the instruction trace of *twolf*.

We now show that creating a locality surface from the reverse of a string yields the same result as flipping the locality surface over the $stride = 0$ axis.

Theorem 2.11. *For any string v,*

$$\mathcal{L}(rev(v), a, b) = \mathcal{L}(v, -a, b).$$

Proof. Let $w = rev(v)$. Let (s, d) be any particular stride/delay combination in $\ell(w)$. We then know, from Theorem 2.7, that $(-s, d) \in \ell(v)$. Let (a_1, b_1) be the particular bin of the locality surface that (s, d) falls in. We first show that $(-s, d)$ falls into the bin labeled $(-a_1, b_1)$.

The delay bin label depends entirely on the value of the delay. Since the delay values of (s, d) and $(-s, d)$ are the same, they both have the same delay bin label. Therefore we may say that $(-s, d)$ has a delay bin label of b_1.

The absolute value of the stride bin label depends entirely on the absolute value of the stride. Since the absolute value of the stride portions of (s, d) and $(-s, d)$ matches, we know that the absolute value of the stride bin labels also match. Therefore, we may say that $(-s, d)$ has a stride bin label with $|a_1|$ as its absolute value.

The sign of the stride bin label matches the sign of the stride value. Since the sign of the stride component of (s, d) and $(-s, d)$ are opposite, we may conclude that the sign of the stride bin labels is also opposite. Therefore the stride bin label for $(-s, d)$ is $-a_1$. Hence we know that $(-s, d)$ falls into the bin labeled $(-a_1, b_1)$. This shows us that $H(w, a, b) = H(v, -a, b)$.

When we compute the locality surface from the binned histogram, the computations are all based on the length of the string $|v|$ and the absolute value of the stride label. If the computations on the bins labeled (a, b) and $(-a, b)$ are equivalent, then reversing the string does not change the computations. From the definition of reverse, we know that $|v| = |w|$. Basic math tells us that $|a| = |-a|$. So both (a, b) and $(-a, b)$ have the same computations performed on them. Therefore, the changes made to convert from the binned histogram to the locality surface do not affect the equality already proved. Hence, $\mathcal{S}(\ell(w), w, a, b) = \mathcal{S}(\ell(v), v, -a, b)$ and $\mathcal{L}(w, a, b) = \mathcal{L}(v, -a, b)$. □

When computing the locality surface for an input string, it is easiest in terms of programming time to use a simple stack algorithm, thanks to Theorem 2.6. However, this may take a lot of compute time, depending on the locality of the input string. To solve this problem, we have written a parallel version of the stack program that significantly reduces compute time when multiple processors are available. We discuss these two algorithms and compare compute time for both in Chapter 9. In general, when we refer to "the locality program" we mean any program that computes the locality data defined in Equation 2.5.

2.5 Summary

We have here described in solid, mathematical terms, our definition of locality. We have further shown how the locality data can be transformed into a locality surface that yields a visually pleasing result of large amounts of data. Next, we demonstrate what common locality surface features represent and examine a number of locality surfaces of workloads from the SPEC CINT2000 and SPEC CFP2000 benchmark suites.

Chapter 3

The Locality of Workloads

In this chapter, we synthetically create strings of numbers that have characteristics that are known to exist in the memory accesses of real workloads, such as sequential and looping elements. We also create synthetic strings that produce specific features on the locality surface that we later see when examining the surfaces of real workloads. This helps us understand what a given feature on the locality surface indicates about the input string.

We then describe the traces we have taken from real workloads and look at a number of their locality surfaces. All of the real workloads examined in this dissertation are from the SPEC CPU 2000 benchmark suite [3]. This suite is usually split into two sub-suites, the integer benchmarks (CINT2000) and the floating point benchmarks (CFP2000). We describe the general characteristics of instruction versus data traces and integer versus floating point workloads and the effects of various inputs on the same workload. We also compare the same workload under differing operating systems.

Our intent is to give an overview of the locality of the workloads in the SPEC C2000 suite, not to provide a detailed workload characterization study. We do not

attempt to explain why a given workload has the locality it does. We wish to give the reader a feel for locality surfaces and what features (including location and size of the features) are typical.

3.1 Locality Surface Features

We now examine the locality surfaces for a number of artificial traces. We created each synthetic trace for one of two reasons. One reason was to discover what the locality for a given pattern of numbers looks like. For example, it is well known that many workloads contain sequential memory accesses and/or loops. We deliberately created strings with these characteristics and examined the resulting locality surfaces. We sometimes used random references to increase the delay between various elements of a synthetic trace, so we also examined a trace made entirely of random references. Another reason we made synthetic traces was to determine what kind of string created a specific feature on the locality surface. There are several curious features seen in a number of locality surfaces (which we examine in Section 3.3). For some of these features, we contrived artificial traces that created the features when sent through the locality surface program.

For each of these artificial traces, we display the code that creates the trace and display the locality surface that is produced when the trace is run through the locality program. In several cases, we also describe other types of traces that may create similar locality surface features. This helps us when we examine surfaces of actual workloads; we are better able to determine what the surface features indicate about the input string.

As mentioned in Chapter 1, we show two views of each locality surface, one from the side and one from the top. The labels on both axes reflect the maximum

stride and delay values in each bin rather than the bin labels, since this is more descriptive of the input trace. This also allows for a more intuitive understanding of the location of various features and simplifies our cache studies.

3.1.1 Sequential References

First we examine what a locality surface looks like for a simple sequence of memory references. The code fragment in Figure 3.1 creates a synthetic trace of sequential references, and Figure 3.2 shows the resulting locality surface. Sequential references create a ridge on the locality surface where stride = delay, i.e. $\mathcal{L}(v, a, a)$. The length of the ridge represents the length of the sequential run, and the height of the ridge indicates the percentage of the trace involved in the run. Recall that the interreference spatial density function described in Chapter 1 would have given us no information about the sequentiality of this trace.

Real workloads generally contain several different sequential runs of various lengths. The rate at which the ridge decays as stride and delay increase demonstrates the distribution of the lengths of the various sequential runs. Because this feature is so common in real workloads, we call the data where $\mathcal{L}(v, a, a)$ the **sequential ridge**.

3.1.2 Random References

The code fragment in Figure 3.3 creates a uniformly distributed string of random references. The locality surface of these references is shown in Figure 3.4. Most of the volume of the surface is around a delay of 1 million, about the same number as the number of unique references in the trace. This is because of the binning and dividing the bins that was done in Sections 2.4.2 and 2.4.3. When v is a string of

```
void main()
{
    ulong addr = 0;
    for (int i = 0; i < 100000; i++)
    {
        ProduceReference(addr);
        addr++;
    }
}
```

Figure 3.1: Code fragment that creates a synthetic trace of sequential memory references.

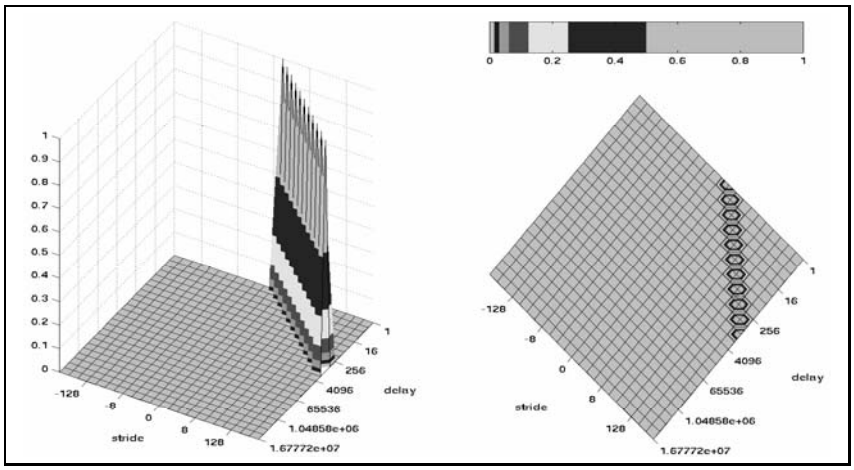


Figure 3.2: Locality surface for the sequential references generated by the code in Figure 3.1.

uniformly distributed random references, then $h(v)$, i.e. the unbinned histogram, looks flat.

Notice the slight spike of the surface along the temporal axis, at the bin labeled $(0, 21)$. This is an artifact of computing the locality data of random references. Let v be a string of random numbers. Let $v[b]$ be an element of the string v. Let $v[a] = v[b]$ such that $a < b$ and there does not exist a $v[i]$ where $a < i < b$ and $v[i] = v[b]$. When computing $\ell(v[b])$, we do not compute $s/d(v[i], v[b])$ for all $i < b$. We only compute for $a < i < b$, due to the restrictions on when the delay exists. This results in a slight preference for temporal locality in the final tally of locality data, and a small spike on the random hump where $stride = 0$.

3.1.3 Temporal References

The code fragment in Figure 3.5 creates a synthetic trace of references with varying amounts of temporal locality. One memory location is referenced repeatedly with varying numbers of random references between the repetitions. The resulting locality surface is shown in Figure 3.6. There are two basic features in this surface. Because of the random references used to create different amounts of temporal locality, we have a random reference hump around a delay of 64,000. This is because there are between 32,000 and 64,000 unique references in the trace. There is also a ridge along the temporal axis from delay = 1 to delay = 64. This indicates repeated references with between 1 and 64 unique references between the repetitions.

Where $\mathcal{L}(v, 0, b)$, the stride is zero, meaning that the axis $\mathcal{L}(v, 0, b)$ represents the delays between repeated instances of the same input number. Researchers commonly call this temporal locality. Hence we call the axis where $\mathcal{L}(v, 0, b)$ the **temporal axis**. The data along the temporal axis is roughly equivalent to Conte and Hwu's

```
void main()
{
    for (int i = 0; i < 1000000; i++)
        ProduceRandomReferences(1);
}
```

Figure 3.3: Code fragment that creates a synthetic trace of uniformly distributed random memory references.

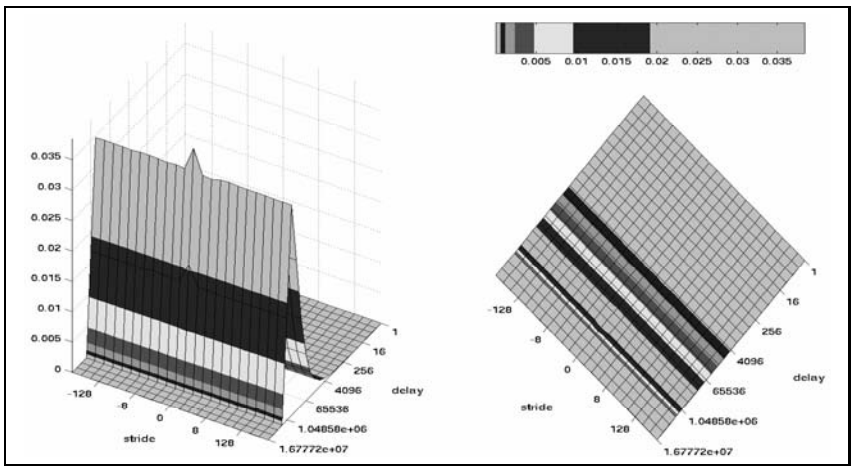


Figure 3.4: Locality surface for a series of uniformly distributed random numbers generated by the code in Figure 3.3.

```
void main()
{
    ulong addr = 0;
    for (int i = 1; i < 64; i *= 2)
    {
        for (int j = 0; j < 1000; j++)
        {
            ProduceReference(addr);
            ProduceRandomReferences(i-1);
        }
    }
}
```

Figure 3.5: Code fragment that creates a synthetic trace with varying amounts of temporal locality.

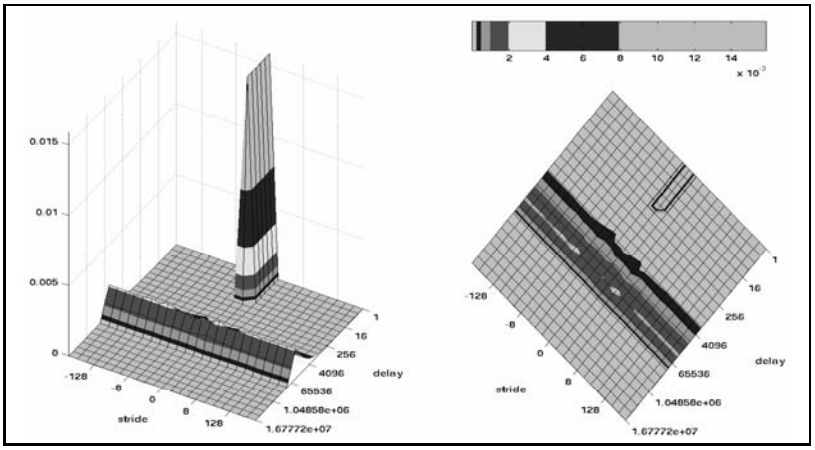


Figure 3.6: Locality surface for the synthetic trace with varying amounts of temporal locality generated by the code fragment in Figure 3.5.

interreference temporal density function [24] and most other temporal locality definitions that are based on a unique reference, or stack distance, count [16, 88].

3.1.4 Looping References

Another important feature to identify is a loop. Figure 3.7 shows a code fragment that creates five loops of equal frequency. The loops are 2, 16, 128, 1024, and 8192 references long. Figure 3.8 shows the resulting locality surface. We see a sequential ridge on this surface because each loop consists of sequential references. Notice the decay of the ridge due to the varying lengths of the sequential runs within the loops.

Looping structures that contain positive strides within the loop are featured between the line where $delay = -stride$ and the temporal axis. The location of the loop hump along the delay axis roughly indicates the number of unique references between the repetition of each element in the loop. We generally consider this to be equal to the number of unique elements in the loop, however, this may not be the case. For example, if a loop was 200 elements long and between each iteration of the loop 56 random references were used, there would be 256 unique elements between the repetition of each loop element, and the loop hump would appear at a delay of 256 on the locality surface. For the purpose of selecting optimal cache sizes, whether the loop is 256 unique elements long or 200 elements long with 56 random references in between each iteration does not matter. Either way, a cache would need to be at least 256 elements in size to ensure that the elements of the loop are in the cache during each repetition.

In Figure 3.8, the loop of length two is almost hidden next to the sequential ridge. The height of a loop hump indicates the relative frequency of loops of that size. Qualitative predictions of cache performance for a particular workload can be

```
void main()
{
    ulong addr = 0;
    int i, len, num;
    for (len = 2; len < 0x10000; len *= 8)
    {
        for (num = 0; num < (0x10000/len); num++)
        {
            addr = 100 * len;
            for (i = 0; i < len; i++)
            {
                ProduceReference(addr);
                addr++;
            }
        }
    }
}
```

Figure 3.7: Code fragment that creates five sizes of loops with equal frequency.

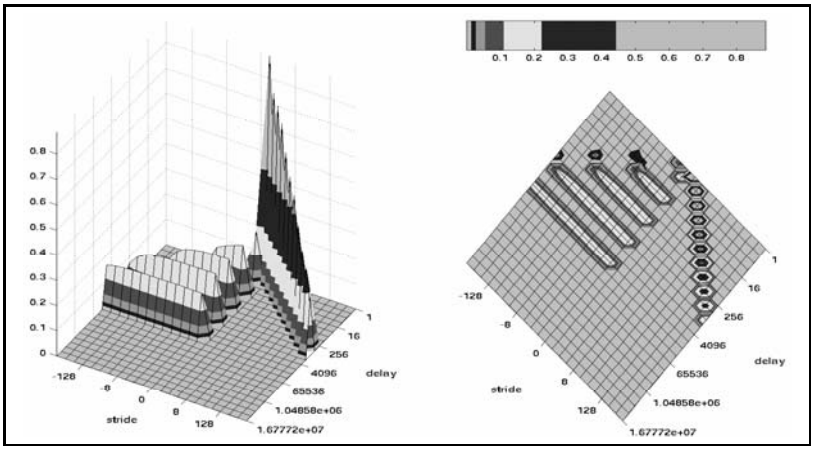


Figure 3.8: Locality surface that results from the synthetic trace of loops generated by the code in Figure 3.7.

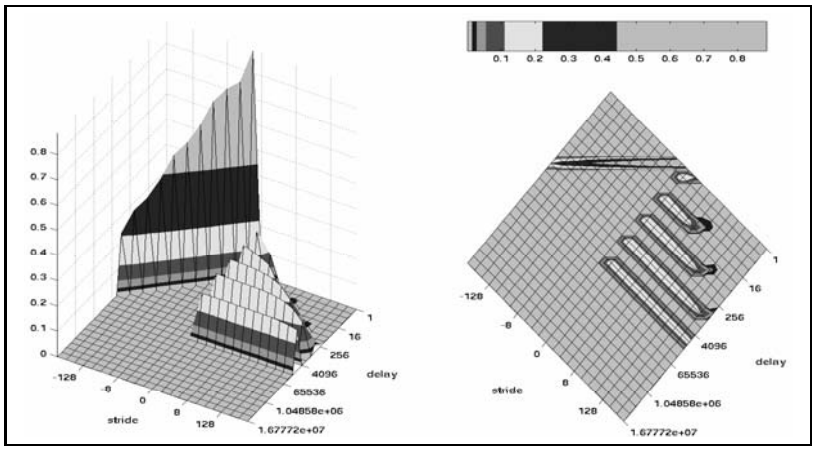


Figure 3.9: Locality surface output when the synthetic trace generated by the code in Figure 3.7 is put through the locality program in reverse order. Notice how this locality surface is equivalent to the surface in Figure 3.8 flipped over the temporal axis as predicted by Theorem 2.11.

performed by comparing the cache size with the delay location of the primary loop structures in a given workload. If the cache is not large enough to contain the major loops, cache performance will suffer.

Now we ask, what if the loop contains negative strides rather than positive strides? Figure 3.9 shows the resulting locality surface when we take the reverse of the synthetic trace generated by the code in Figure 3.7 and run it through the locality program. As predicted by Theorem 2.11, the reverse of the trace generated by Figure 3.7 results in the same locality surface merely flipped over the temporal axis. Notice how, with the negative strides within each loop, the loop structures are between the temporal axis and the line where $delay = stride$. This surface actually gives us a better view of how the sequential ridge decays.

Now we create a loop with both postive and negative strides and where the absolute value of the stride is not always one. Figure 3.10 shows a code fragment that creates a loop, repeated ten times, that has primarily sequential data, with periodic negative stride jumps. The loop contains 8192 unique references. Figure 3.11 shows

the resulting locality surface.

Again, we can see the sequential ridge and we can easily identify the loop at 8K unique words. However, the loop structure on the locality surface extends into both negative and positive strides. When examining real workloads, we see a number of loops with humps on both sides of the temporal axis. These loops often have a lack of data at small positive strides. We hypothesize that by looking at the distribution of stride values in the loop structure, we can deduce what each iteration of the loop does. In the case of Figure 3.11, there is both positive and negative strides, with the positive strides being nearly sequential and the negative strides being jumps.

In summary, looking at the delay location of a loop structure helps us know what cache size is needed to contain the loop. Looking at the distribution of strides in the loop structure helps us know what the loop contains. Negative strides in the loop hump indicates forward progress within the loop; positive strides in the loop hump indicate backwards jumps within the loop. To distinguish beween random references and loop structures, compare the locality surfaces containing random data (Figures 3.4 and 3.6) with the locality surfaces containing loops (Figures 3.8, 3.9, and 3.11). Notice that random references tend to make a uniform stride distribution across the entire surface for a number of delays while loops tend to have an uneven stride distribution at a single delay value.

3.1.5 Variable Striding

When examining the locality surfaces of real workloads, a few of the surfaces contain what appears to be a sequential ridge, only shifted either in the positive stride or delay direction. These shifted ridges are caused by one of two patterns.

The shift in the stride direction is due to **striding**. As the sequential references

```
void main()
{
    ulong addr = 0;
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 80; j++) {
            for (int k = j * 10000; k < j * 10000+100; k++) {
                ProduceReference(k);
            }
            for (int k = j * 10000-10; k > j * 10000-20; k--) {
                ProduceReference(k);
            }
        }
    }
}
```

Figure 3.10: Code fragment that creates one larger loop with both positive and negative strides.

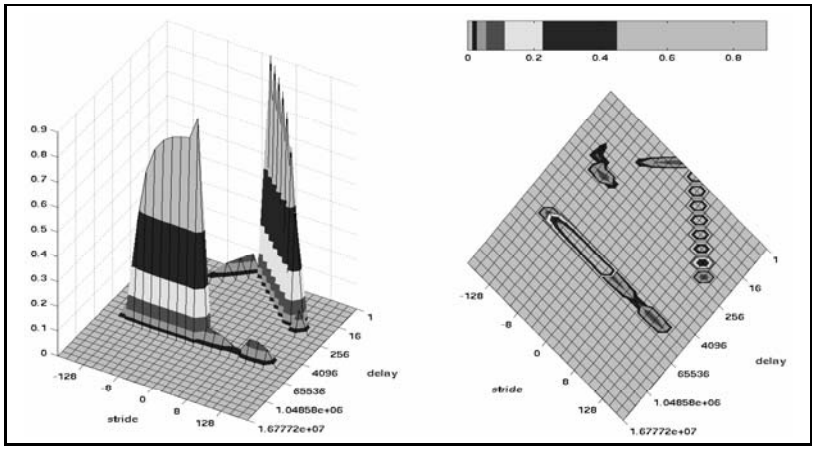


Figure 3.11: Locality surface that results from the synthetic trace of the loop generated by the code in Figure 3.10.

created by the code in Figure 3.1 progress with a stride of one, striding references progress with a larger, regular, stride. For example, the references 1, 4, 7, 10, 13 are striding references with a stride of three.

The shift in the delay direction is due to **delayed sequential** references. This occurs when otherwise sequential references have a regular number of random (or merely unrelated) references interspersed between each pair of elements. For obvious reasons, we refer to the number of interspersed references plus one as the delay. For example, the references $1, 2, 3$ in the string $1, 44, 98, 127, 2, 76, 102, 39, 3$ are delayed sequential references with a delay of four. Grimsrud refers to this as **fractional striding**. For convenience, we use the term **variable striding** to refer to both striding references and delayed sequential references.

The code in Figure 3.12 creates a trace with a sequential series (as created in Figure 3.1) for reference, a striding series (with a stride of 16), and a delayed sequential series (with a delay of 16). Figure 3.13 shows the resulting locality surface. In this case, the top view may be of more use to show the locations of each of these series. We purposely made the relative lengths of each series in such a way as to make the heights of the ridges approximately equal.

We can see the sequential series is the middle ridge on the locality surface. It begins at the bin labeled $(0, 1)$. The striding series is the ridge on the right, where $stride > delay$. It begins at the bin labeled $(5, 0)$, or where the stride equals 16 and the delay equals 1. The delayed sequential series is the ridge on the left, where $stride < delay$. It begins at the bin labeled $(1, 5)$, or where the stride equals 1 and the delay equals 16. Notice that adjusting the value of *stride* in the code of Figure 3.12 would shift the striding ridge in the stride direction by the amount of the change. A similar change in the value of *delay* shifts the delayed sequential ridge in the delay direction. Notice that even though random values were used, we

```
void main()
{
    ulong addr = 0;
    for (int i = 0; i < 1000; i++) {
        ProduceReference(addr++);
    }       // creates the sequential series
    addr += 10000;
    int stride = 16;
    for (int i = 0; i < 16000; i++) {
        ProduceReference(addr);
        addr += stride;
    }       // creates the striding series
    addr += 10000;
    int delay = 16;
    for (int i = 0; i < 1000; i++) {
        ProduceReference(addr++);
        ProduceRandomReferences(delay-1);
    }       // creates the delayed sequential series
}
```

Figure 3.12: Code fragment that makes a sequential series, a striding series with a stride of 16, and a delayed sequential series with a delay of 16.

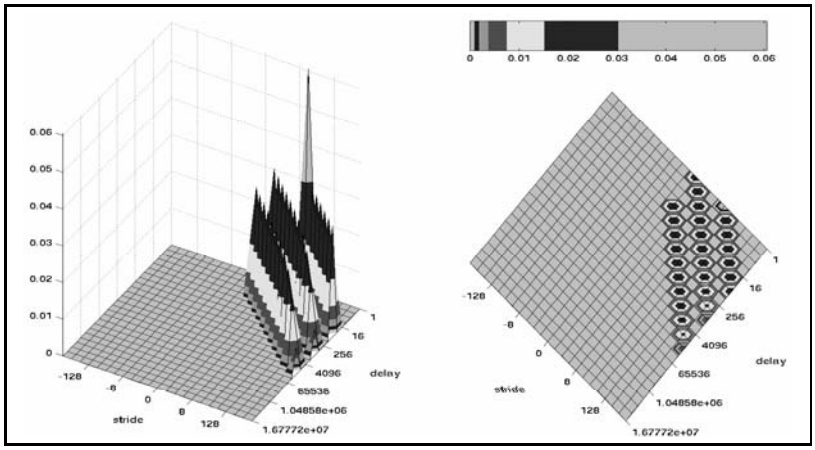


Figure 3.13: Locality surface that results from the synthetic trace generated by the code in Figure 3.12.

do not see a random hump at any delay. This is because the repeated patterns in the three ridges sufficiently dominated the small random relationships, making them effectively disappear.

One feature that sticks out in Figure 3.13 is the height of the initial point of the striding ridge. This is an artifact of how we chose to bin the data in Chapter 2. Recall from Section 2.4.3 that we divide the bins in the stride direction by the number of stride values that fall in that bin. Recall further that we do not divide the bins in the delay direction, we merely sum them. A regular sequential series has data at the bins labeled $(1,1)$, $(2,2)$, $(3,3)$, $(4,4)$, etc. A striding ridge has data at bins labeled $(a,1)$, $(a+1,2)$, $(a+2,3)$, $(a+3,4)$, etc., where a depends on the value of *stride* in the code. For the regular sequential series, the dividing in the stride direction matches the summing in the delay direction. For the striding sequence, however, $H(v,a+1,2)$ has about the same amount of data as $H(v,a,1)$, but contains two times as many stride values so when we compute the locality surface from the binned histogram, the data at $(a+1,2)$ is divided by twice as much as the data at $(a,1)$. Because of the summing in the delay direction, $H(v,a+2,3)$ has about twice as much data as $H(v,a+1,2)$. Therefore, when it is divided by twice as much as the previous bin, the final result is about the same. Hence a spike is seen where $delay = 1$ for the striding series and the rest of the ridge is approximately equal.

As mentioned before, a significant amount of sequential data is found in real workloads. Several real workloads also show evidence of delayed sequential series. This may occur, for example, when an array of numbers is summed and stored at each stage. Hence the array is accessed sequentially, with regular accesses between each array access to another location in memory where the running total is stored.

Striding is also found in a few real workloads. This may occur when data is stored in an array of records. If the array is traversed, but only one entry in each

record is accessed, the result is a striding pattern. The value of the stride used would be the size of the record. Another possible cause of striding is when the elements of a matrix are accessed in row order, despite the matrix being stored in column order. This mistake in coding is commonly fixed now by compilers using the Loop Interchange compiler optimization technique [43], so is not likely to be the cause of the striding found in workloads compiled with optimizations set.

3.1.6 The Jut

We now examine another feature seen in a number of real traces which we call the **jut**. The code in Figure 3.14 creates a trace that contains several runs of sequential code where the beginning of each run is a large negative jump from the beginning of the previous run. Figure 3.15 contains the resulting locality surface. We wish the reader to notice that the range shown on the stride direction is wider than shown in previous locality surfaces. We can adjust the location of the jut on the locality surface by changing the delay and stride values in the code segment.

Just as loops do not necessarily contain sequential elements, juts may not always be generated by sequential runs as we have done here. Any pattern of numbers may be repeated to cause a jut rather than a loop. In a loop, the pattern is repeated exactly the same. To create a jut, the start of each pattern has a large, regular, negative shift from the previous pattern start. For example, take the pattern: 1001, 1004, 1009, 1016, 1025. If we shift the start of each pattern by -500 from the previous start and repeat three times, we get: 1001, 1004, 1009, 1016, 1025, 501, 504, 509, 516, 525, 1, 4, 9, 16, 25. The locality surface for this short trace would contain a small jut. The start of the jut is determined by the size of the negative jump between each pattern repetition. The length of the jut is determined by the

```
void main()
{
    int stride = 1000;
    int delay = 100;
    int freq = 100;
    ulong addr = freq * stride;
    for (int i = 0; i < freq; i++) {
        for (int j = 0; j < delay; j++) {
            ProduceReference(addr);
            addr++;
        }
        addr = addr - delay - stride;
    }
}
```

Figure 3.14: Code fragment to create a jut.

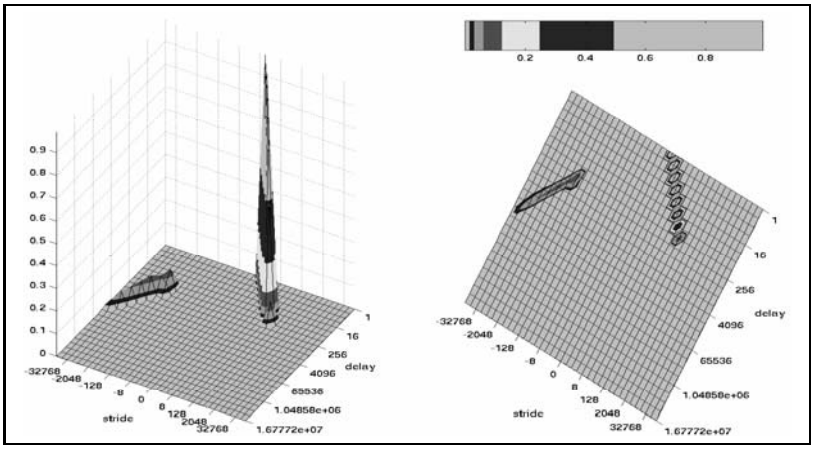


Figure 3.15: Locality surface that results from the synthetic trace generated by the code in Figure 3.14.

length of the pattern.

3.1.7 Other Features and Locality Events

As we examine locality surfaces of real traces in the next section, we see the need to examine some of these features in more detail to gain a better understanding of what causes various aspects of the feature. For example, we have already expressed a desire to better understand how the contents of a loop affects the distribution of strides within the loop structure. We could also determine how the distribution of sequential runs mathematically relates to the distribution of heights of the sequential ridge.

In addition, we see locality events on the locality surfaces of real workloads that are not part of the features we have described. We can make some general statements about such unlabeled features. For example, a bump at a positive stride indicates frequent forward strides at the given delay. However, more investigation is needed to make more detailed observations. There is considerable future work available in this area.

3.2 Real Traces

We now describe the traces used to create the locality surfaces of real workloads. A trace is an event-ordered list of addresses requested by the CPU from memory. (A synthetic or artificial trace is a list of numbers generated by any means that may be processed in the same way as a real trace.) One of the more accurate methods for tracing is to collect memory addresses directly from the pins of a CPU package. BYU Address Collection Hardware, or BACH, is one of the more successful methods for doing this [33, 41]. A large repository of such traces can be found at

http://traces.byu.edu [1, 81]. All of the traces used in this dissertation are from this trace library.

We believe these traces are the most correct available. In their well-known survey of tracing methods, Uhlig and Mudge identify hardware tracing as the most complete and accurate but the most costly in terms of both time and equipment [83]. Some researchers have questioned the accuracy of hardware tracing, claiming that the necessity of periodically stalling the processor introduces questionable memory references. However, a recent paper by Watson and Flanagan dispels this belief [85].

Note that all of the techniques presented in this dissertation are independent of how the trace is taken. Traces of memory addresses taken from any source, by any method, may be used. However, errors in the trace cause corresponding errors in the results, so we have chosen to use the most accurate traces freely available to the research community.

All of the traces used in this dissertation were taken from a single processor Pentium III 733 MHz machine with 16 G of disk and 1 G of RAM. The caches were turned off. Traces were taken under Redhat Linux 6.2, Windows NT Workstation 4.0, and Windows 2000. Details of the traces used in this chapter are found in Table 3.1. Details for all the SPEC workloads can be found in Appendix A. The locality surfaces of all the SPEC workloads can be found in Appendix B. In this dissertation we focus on L1 cache analysis. Since L1 caches are usually split into separate instruction and data caches, we have split all our traces into separate traces of instructions and data.

Prior to splitting each trace, the length of each trace was arbitrarily fixed at about 90 million references. Hence the lengths of the split traces, shown in Table 3.1, gives a rough indication of the instruction/data mix. For example, the instruction trace for the workload *eon* with the *kajiya* input is 48,518,645 references long. We can

then guess that the length of the data trace for *eon.kajiya* is 41,481,355 references, meaning that the instructions are approximately 53.9% of the original, unsplit, trace.

3.3 Characterizing the Workloads in Terms of Locality

When creating the locality surface for a given trace, we must choose what **granularity** we wish to examine. The granularity determines at what level the locality relationships are determined [16]. For example, if our trace contains consecutive references to blocks that begin with memory byte 232 and then memory byte 240, our chosen granularity determines the stride. With a granularity of one byte, the stride would be 8. With a granularity of one eight-byte memory word, the stride would be 1.

The chosen granularity also affects the delay. If our trace consists of the following stream of references to memory bytes: 232, 240, 241, 242, 243, 248, then the delay between bytes 232 and 248 is 5 with a granularity of one byte. However, the delay is 2 with a granularity of one eight-byte word. In general, increasing the granularity decreases the value of many strides and delays.

For all of the locality surfaces in this dissertation, we chose a granularity of one eight-byte word. We do this for two reasons. First, the Pentium III has an eight-byte data bus width [2]. Second, eight bytes is the smallest line size of interest in a cache. (We discuss in Chapter 5 why it is easier to predict cache performance when the granularity is not larger than the line size.)

workload	suite	OS	type	total refs	unique refs	description
applu	FP	L	D	44,061,521	1,523,951	Parabolic/Elliptic Partial Differential Equations
bzip2.g7	INT	L	D	34,797,524	194,560	Compression
eon.cook	INT	L	I	48,100,350	29,999	Computer Visualization
eon.kajiya	INT	L	I	48,518,645	28,220	Computer Visualization
eon.rush	INT	L	I	48,298,304	28,451	Computer Visualization
galgel	FP	L	D	35,138,856	1,177,014	Computational Fluid Dynamics
gap	INT	L	D	31,714,213	923,381	Group Theory, Interpreter
gap	INT	NT	D	33,386,414	1,093,040	Group Theory, Interpreter
gap	INT	2k	D	33,242,917	991,604	Group Theory, Interpreter
gzip.source	INT	L	I	54,496,031	20,448	Compression
gzip.source	INT	NT	I	59,161,078	29,290	Compression
gzip.source	INT	2k	I	59,111,271	86,664	Compression
mcf	INT	L	D	33,151,617	1,385,873	Combinatorial Optimization
mgrid	FP	L	D	37,351,118	5,598,803	Multi-grid Solver: 3D Potential Field
mgrid	FP	NT	D	41,755,272	5,570,840	Multi-grid Solver: 3D Potential Field
mgrid	FP	2k	D	39,023,447	5,090,300	Multi-grid Solver: 3D Potential Field
perlbmk.diffmail	INT	L	D	35,496,885	1,875,818	PERL Programming Language
perlbmk.perfect	INT	L	D	38,424,060	97,950	PERL Programming Language
twolf	INT	L	D	39,611,264	332,193	Place and route simulator
twolf	INT	L	I	50,191,887	21,988	Place and route simulator
wupwise	FP	L	D	51,477,244	8,404,245	Physics/Quantum Chromodynamics
wupwise	FP	NT	D	36,372,267	936,409	Physics/Quantum Chromodynamics
wupwise	FP	2k	D	36,316,042	874,529	Physics/Quantum Chromodynamics

Table 3.1: Description of the traces used in this chapter.

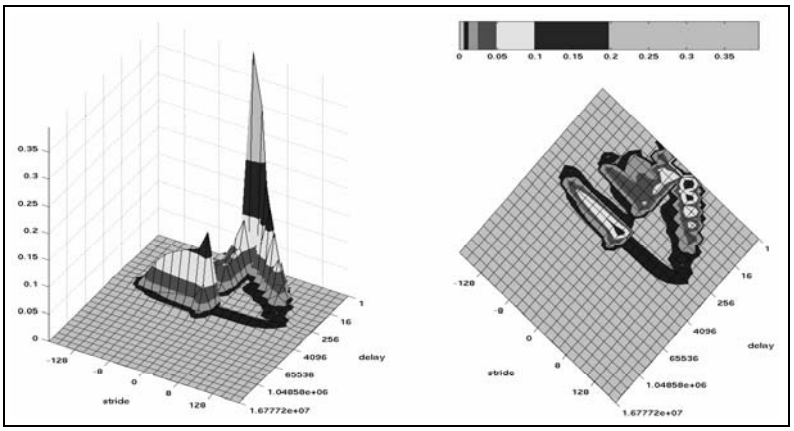
3.3.1 Instructions versus Data

First we examine some of the general characteristics of instruction fetches versus data reads and writes. This also show us some characteristics general to all SPEC CPU 2000 workloads. Recall that we have split all our traces into separate instruction and data traces to facilitate focus on L1 caches. The first workload is *twolf*. The instructions are shown in Figure 3.16(a) and the data are in Figure 3.16(b).

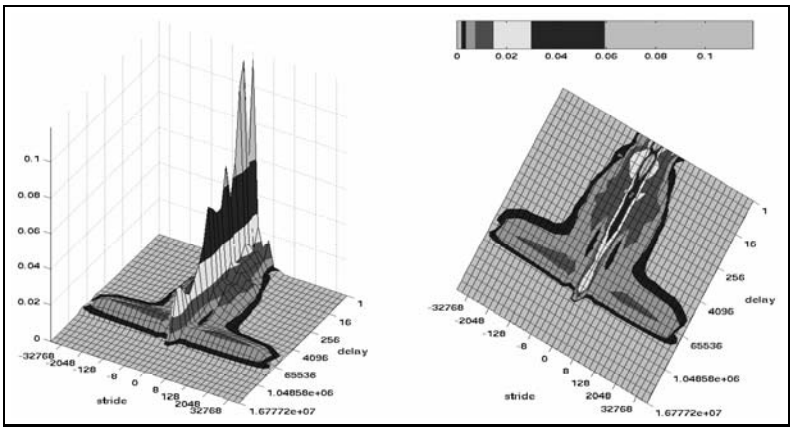
As mentioned in Chapter 2, we show two views of each locality surface, one from the side and one from the top. Since the granularity is one eight-byte word, the labels on both axis are in words. For example, in Figure 3.16(b), the maximum delay with visible data is 256 Kwords.

When examining locality surfaces, we often refer to the largest delay at which a feature is visible as the **effective working set size** of the input trace. The working set size may actually be much larger, but if a number of the references are accessed much less frequently than others, they do not have a significant impact on the system performance and do not create visible locality features. The true working set size would be the effective working set size plus these infrequent references. For the instructions of *twolf*, the effective working set size is 4 Kwords. For the data of *twolf*, the effective working set size is 256 Kwords.

Similarly, we refer to the largest stride (or the absolute value of the smallest stride) at which a feature is visible as the **effective memory range** of the input trace. For example, if the largest stride with visible data is 16 words and the smallest stride with visible data is -256 words, we would say that the effective memory range is 256 words. The actual memory range is the largest memory address used by the workload minus the smallest memory address. (Note that subtracting two address yields a stride.) However, if the very large or very small valued memory addresses



(a) Locality surface for the instruction trace.



(b) Locality surface for the data trace.

Figure 3.16: The locality surfaces for the *twolf* trace: 3.16(a) shows the instructions of *twolf* and 3.16(b) shows the data of *twolf*. We can here see many of the typical differences between instruction and data traces. The data trace of *twolf* is representative of Category 1 data traces, described later in this chapter.

are seldom used, they have little impact on performance and the strides to these locations do not appear on the locality surface. Hence the effective memory range is the range of memory frequently accessed by the workload. For the instructions of *twolf*, the effective memory range is 128 words. For the data of *twolf*, the effective memory range is 64 Kwords.

One general difference between instructions and data is the effective memory range. The locality surfaces for data traces tend to have larger effective memory ranges then instruction traces. For this reason, we display all the locality surfaces for instruction traces with a stride range from -1024 words to 1024 words. We display all the locality surfaces for data traces with a stride range from $-131,072$ words to $131,072$ words. We keep these ranges constant throughout the dissertation. The reader should remember the differences in range if comparing the locality of an instruction trace with the locality of a data trace.

Recall from Section 3.1.3 that the slice of the locality surface where $stride = 0$ is referred to as the temporal axis. The location where stride is zero and delay is one, i.e. the bin labeled $(0, 1)$, is of more specific interest since this indicates hits in any cache. We term this locality the **temporal spike**. The temporal spike is usually the tallest portion of the locality surface, indicating that the stride/delay relationship $(0, 1)$ occurs more frequently than any other.

Part of the reason for this is our choice of granularity. The larger the granularity, the more bits are shifted off, and the more likely two values are the same. In the case of the Pentium III, it is likely that the processor requests the first two bytes in a word, then the next two bytes, etc. Each request requires the transfer of the entire eight-byte word on the bus when the caches are turned off. With caches on, however, the performance impact is minimal. Even the smallest cache results in a hit with two immediately repeating references to the same word. Therefore, the

temporal spike gives us an indication of the worst-case cache scenario. If the height of the temporal spike is 0.60, we know that 60% of the references are hits.

In Figure 3.16(a), the temporal spike reaches a height of 0.395. In Figure 3.16(b), the temporal spike reaches 0.115. Therefore, 39.5% of the instructions of *twolf* are immediate repeats of the previous word and 11.5% of the data references are immediate repeats. In Figure 3.16(b), the temporal spike is not the tallest point. The tallest point is actually at the bin labeled $(0, 3)$, or a delay of 4 words, with a value of 0.119. We allow the maximum height of the surface to be dictated by the maximum value on the surface. This allows us to see greater detail. However, care should be taken to note the maximum values when comparing two different surfaces.

We see some significant features in the instructions of *twolf*. We see several looping structures, from the smallest delay up to 1 Kword. The loops are almost entirely on the negative stride side of the surface, meaning they contain positive strides almost exclusively. This probably matches with the significant sequential ridge seen in the surface. These features fit with the typical nature of instruction patterns. Instructions tend to be sequential and tend to be repeated in loops. Loops and sequential ridges can be seen to varying degrees in all of the instruction trace locality surfaces we have made.

The data trace for *twolf* shows more locality events along the temporal axis than we see with the instruction trace, resulting in a larger effective working set size. This trend holds for all our traces, i.e. the data traces tend to have more locality events on their locality surfaces and larger effective working set sizes. However, features and shapes on the data locality surfaces are not as consistent as for instruction traces. As mentioned earlier, all instruction traces are primarily loops and sequential ridges. We have placed each of the SPEC CPU 2000 data traces into one of five categories based on the primary features, effective working set size, and general appearance

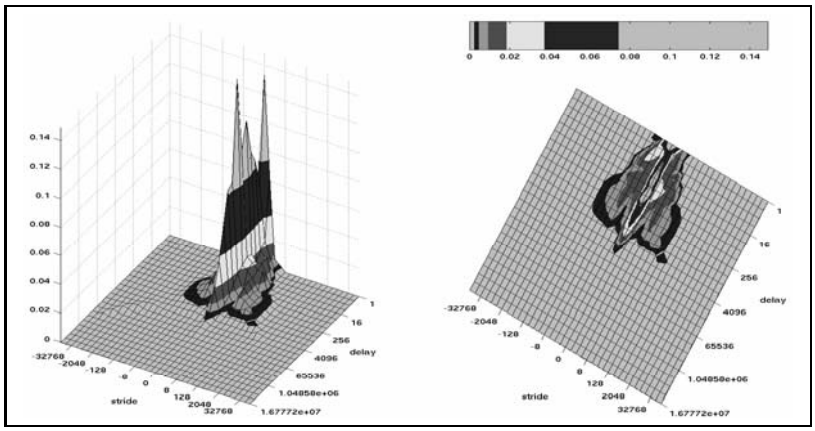


Figure 3.17: The locality surface for the data trace of *bzip2* with the *g7* input. This surface is representative of the Category 2 data traces.

of the surface. We now show a representative locality surface for each of the five categories.

Category 1 is represented by the data trace of *twolf*, shown in Figure 3.16(b). These traces have primarily temporal locality, i.e. most of the features are focused around the temporal axis, that merges into a random hump.

Category 2 is represented by the data trace for *bzip2* with the *g7* input, shown in Figure 3.17. These traces are also primarily temporal locality, as Category 1. However, the Category 2 surfaces have much smaller effective working set sizes than Category 1 and do not have random humps.

Category 3 is represented by the data trace for *galgel*, shown in Figure 3.18. Again, these surfaces have primarily temporal locality, but with larger effective working set sizes than in Category 2. The effective working set size of workloads in Category 3 is similar to that found for workloads in Category 1, however the Category 3 workloads do not have a random hump. In addition, the Category 3 surfaces may have a jut, as described in Section 3.1.6, and have smaller effective memory ranges.

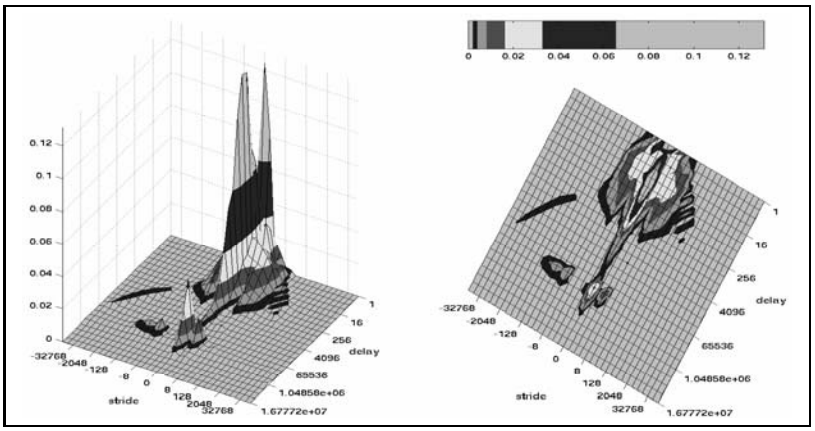


Figure 3.18: The locality surface for the data trace of *galgel*. This surface is representative of the Category 3 data traces.

Category 4 is represented by the data trace for *gap*, shown in Figure 3.19. These surfaces are very similar to the ones in Category 3; they have primarily temporal locality and large effective working set sizes. However, the juts in Category 4 are more pronounced, and the effective memory range is larger than for Category 3.

Category 5 is the most variable and interesting of the categories. We represent it with the data trace for *wupwise*, shown in Figure 3.20. This is the most interesting locality surface we have created in terms of the number and size of the features. The workloads in Category 5 have large effective working set sizes and large effective memory ranges. Each surface also has several features from the following list: sequential ridges, juts, loops, striding ridges, and/or delayed sequential ridges. For example, Figure 3.20 has large loops, a significant jut, and a striding ridge with a stride of 2 words. (The striding ridge is so close to where the sequential ridge would be that it is difficult to notice the change in position.)

It should be obvious to the reader that the lines between these categories are not rigidly defined. A number of workloads fall somewhere between the average of two categories. In addition, some categories are more well defined than others. Specif-

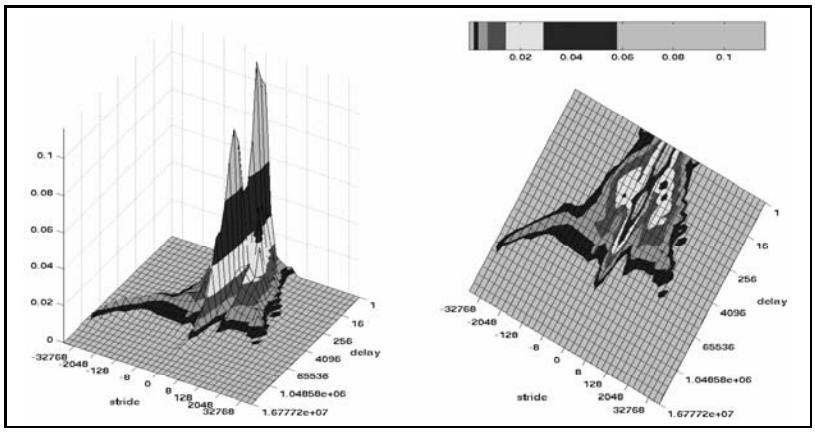


Figure 3.19: The locality surface for the data trace of *gap*. This surface is representative of the Category 4 data traces.

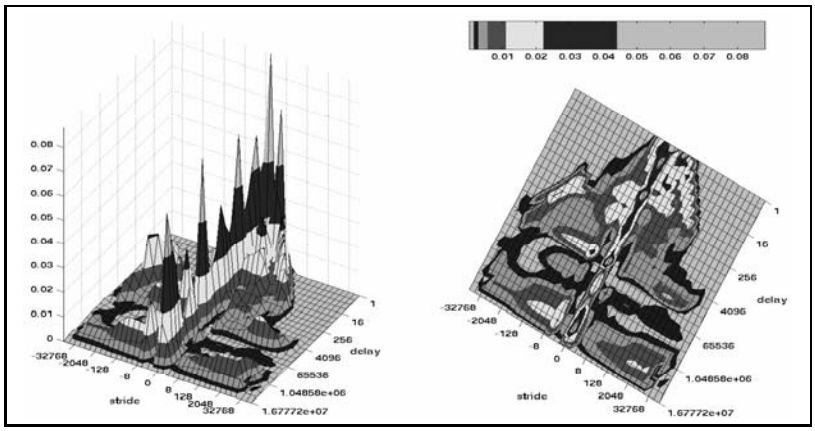


Figure 3.20: The locality surface for the data trace of *wupwise*. This trace is representative of the Category 5 data traces.

ically, Category 5 contains all the large, wild looking, locality surfaces. However, splitting the data traces into categories allows us to present a series of surfaces that covers, fairly well, the range of localities that may be seen in the data traces of the SPEC CPU 2000 benchmark suite.

3.3.2 Integer Versus Floating Point Workloads

As might be expected, there are some significant differences in locality between the integer and floating point SPEC CPU benchmarks. In general, the instruction traces from both the integer and floating point component suites have the same trends: primarily loops and sequential features.

However, the data traces show some significant differences between the two subsuites. In general, the floating point benchmarks have worse locality, meaning more features, larger effective working set sizes, and larger effective memory ranges. If we can see features at large delays on the locality surface, that means memory references tend to be repeated further apart and references close in memory are used further apart in time. Features at large strides indicate that memory references tend not to be close in memory. Therefore, large effective working set sizes and large effective memory ranges indicate worse locality.

The worst locality of all of the SPEC CPU 2000 workloads is the data trace of the floating point workload, *wupwise*, already shown in Figure 3.20. The worst locality shown in the integer suite is the data trace of *mcf*, shown in Figure 3.21. At first glance, it is easy to see that the locality surface for *wupwise* contains more features throughout the surface than the locality surface for *mcf*. Notice that the effective working set size for the data of *mcf* is 2 Mwords. The effective working set size for the data of *wupwise* is 8 Mwords, meaning that the largest workload in the

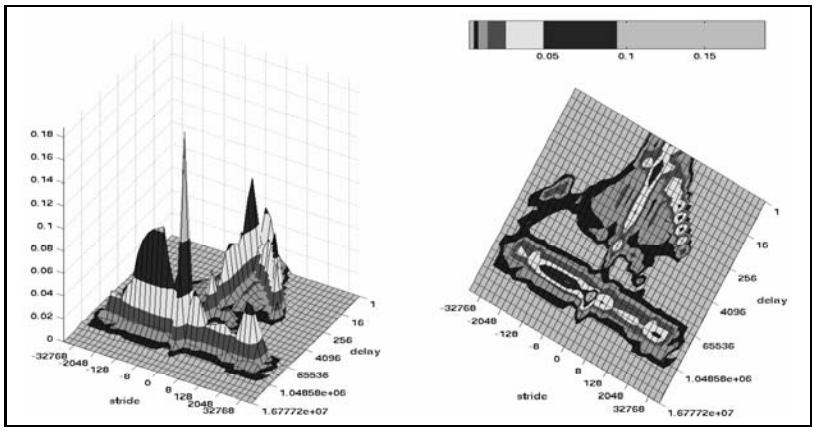


Figure 3.21: Locality surface for the data trace of *mcf*.

floating point suite is about four times the size of the largest workload in the integer suite.

We see the same integer versus floating point trends when comparing workloads from each subsuite that have much smaller effective working set sizes. The locality surface for the data trace of *bzip2.g7*, already seen in Figure 3.17, has one of the smallest effective working set sizes seen in the integer subsuite, namely 4 Kwords. Figure 3.22 shows the locality surface for the data of *applu*, which has the smallest effective working set size of all the surfaces in the floating point subsuite, namely 512 Kwords. Again, we can easily observe a significant difference in effective working set size. We also see more features on the floating point surface, i.e. a significant jut, a loop, and temporal locality events at much larger delays. The effective memory range is also considerably different, 128 words for *bzip2.g7* versus 32 Kwords for *applu*.

3.3.3 How different inputs affect the locality

A number of the integer benchmarks have several possible input files. The *bzip2* workload has six possible inputs, the *eon* workload has three, *gcc* has five, *gzip* has

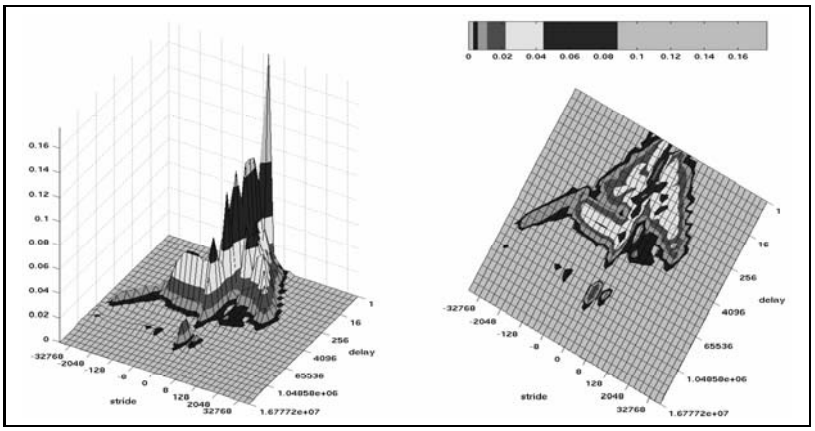
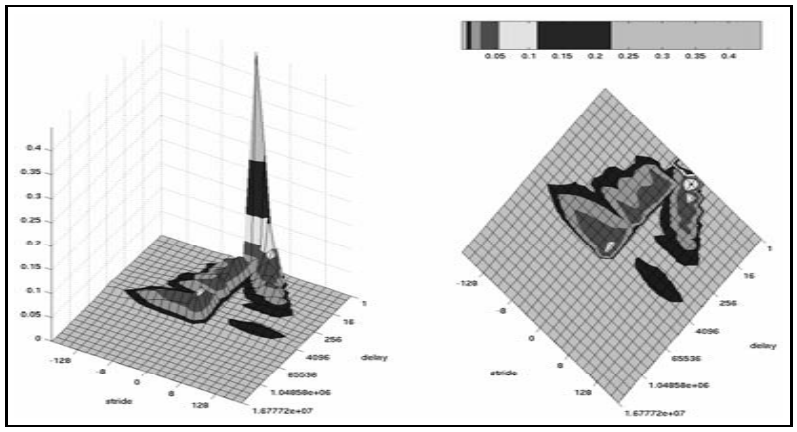


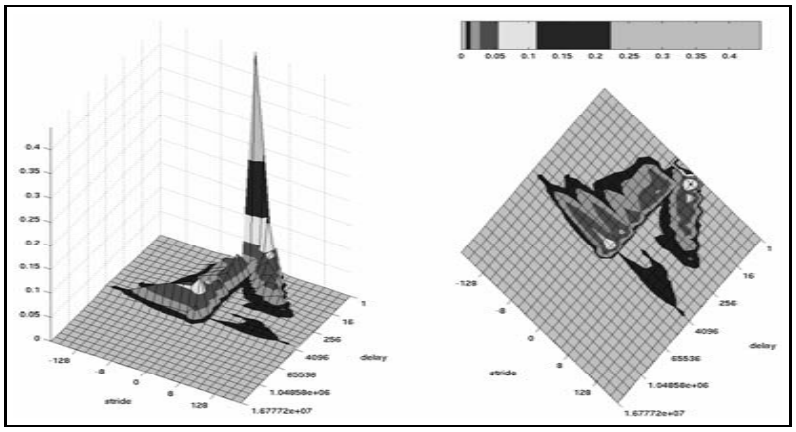
Figure 3.22: Locality surface for the data trace of *applu*.

five, *perlbmk* has four, *vortex* has three, and *vpr* has two. A good question to ask when characterizing these workloads is whether the locality of the workload changes with differing inputs? If all the inputs yield the same locality, there is little need for more than one of the inputs when analyzing memory performance.

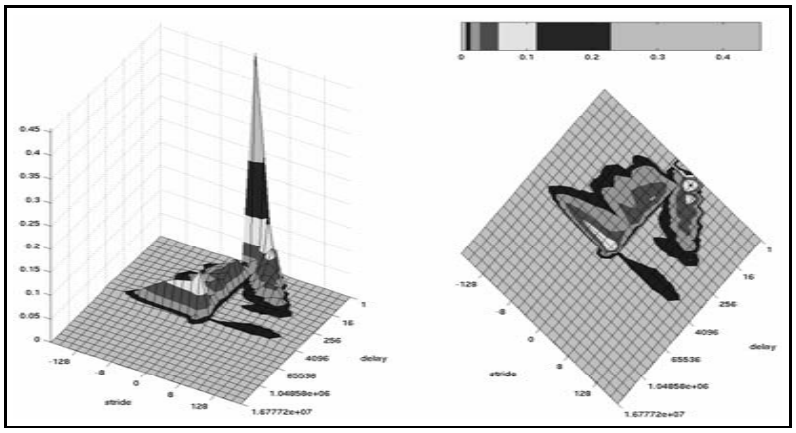
In general, when comparing the locality surface for the same workload with differing inputs, we see the same general trends, but with varying magnitudes. This is true for both the instruction and data traces. One of the more subtle examples of this is the instruction traces for the *eon* workload, which has three different inputs: *eon.rush* is shown in Figure 3.23(a), *eon.kajiya* is shown in Figure 3.23(b), and *eon.cook* is shown in Figure 3.23(c). The changing trend is best seen when examining the height of the loop where $stride = 0$ at a delay of 4 Kwords. First, note that the scales for these three surfaces are almost identical. Then one may notice that the height of the noted loop rises from *eon.rush* to *eon.kajiya* to *eon.cook*. In fact, the values at these points is 0.067 for *eon.rush*, 0.086 for *eon.kajiya*, and 0.106 for *eon.cook*.



(a) *rushmeier* input



(b) *kajiya* input



(c) *cook* input

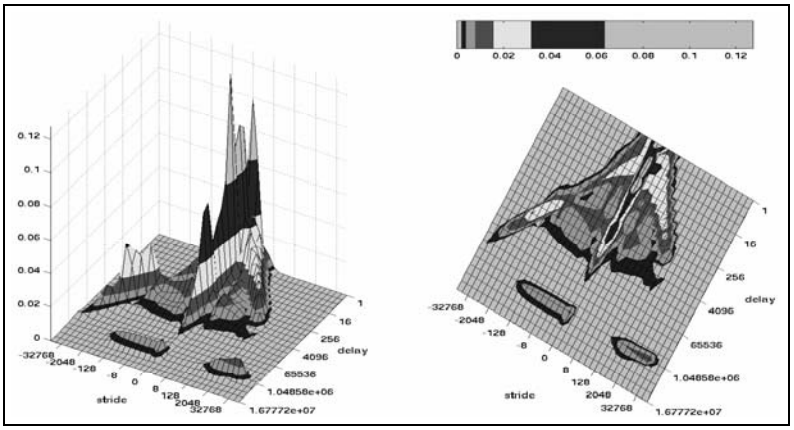
Figure 3.23: The locality surfaces for the instruction traces of the *eon* workload under three different inputs: 3.23(a) uses the *rushmeier* input, 3.23(b) uses the *kajiya* input, and 3.23(c) uses the *cook* input. Here we see that locality surfaces for the same workload with different inputs may appear similar, but have slightly different values.

There are one or two examples where one of the inputs causes a significant difference in locality, not merely trend exageration. One of these is the data traces for the workload *perlbmk*, which has four inputs. Three of the inputs, *diffmail*, *makerand*, and *splitmail*, have very similar locality. However, the locality for the *perfect* input deviates significantly. We here show only one example of the first three alongside the *perfect* input. Figure 3.24(a) shows the locality surface for the data trace of *perlbmk.diffmail* and Figure 3.24(b) shows the locality surface for the data trace of *perbmk.perfect*.

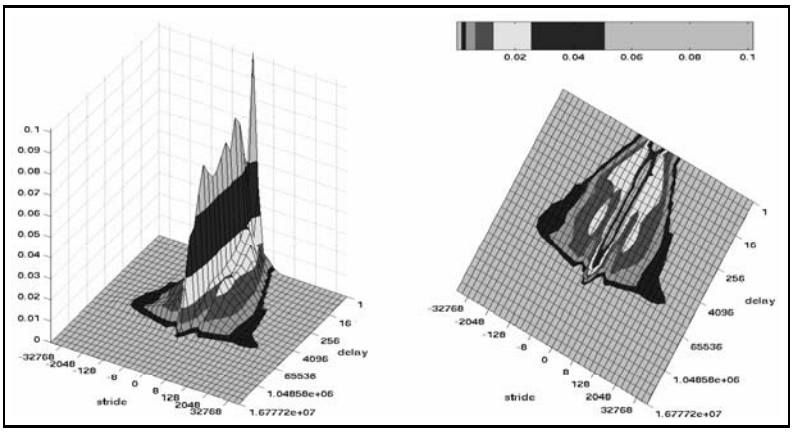
It is interesting that all the features seen in Figure 3.24(b) are also in Figure 3.24(a). But there are some significant features in Figure 3.24(a) not it Figure 3.24(b), namely the jut and the large loop at a delay around 512 Kwords. These features are found in the other two *perlbmk* inputs and appear very similar to the ones seen in Figure 3.24(a). For the readers interested in examining more of these trends, all the locality surfaces are found in Appendix B.

3.3.4 Trends of the OS

In the trace repository, each workload in the SPEC CPU 2000 was traced under RedHat Linux 6.2, Windows NT Workstation 4.0, and Windows 2000. The compiler chosen depended on both the operating system and the language the benchmark was written in. Workloads written in C/C++ were compiled for Linux using gcc and compiled for Windows NT and Windows 2000 using the command line version of MS Visual Studio C. All of the Fortran workloads were compiled using the Lahey Fortran Compiler under all three operating systems. Comparing the locality surfaces for the same workload under different operating systems yields interesting results. The results are a little different for the instruction traces verses the data traces.



(a) *diffmail* input



(b) *perfect* input

Figure 3.24: The locality surfaces for the data traces of the *perlbmk* workload under two different inputs: 3.24(a) uses the *diffmail* input and 3.24(b) uses the *perfect* input. Here we see that locality surfaces for the same workloads may appear significantly different.

The instruction traces have less variance than the data traces. In a few cases there are small features under one or two operating systems not seen on the other one or two. The differences are small enough to be generally ignored. The most interesting feature to compare between operating systems on the instruction traces is the effective working set size. In general, the effective working set size of the Windows 2000 instruction trace is larger than for either the Linux or Windows NT instruction traces. The relationship between the effective working set size of the Linux and Windows NT traces varies, sometimes Linux is greater, sometimes Windows NT, and sometimes they are about equal.

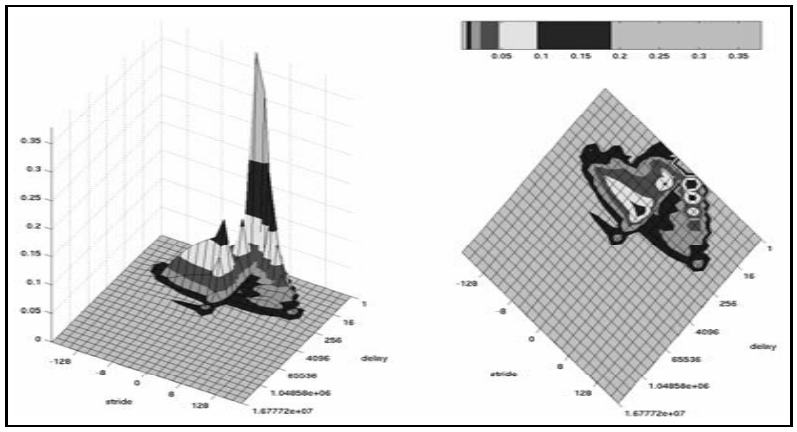
For most of the data traces, the NT and 2000 traces look almost identical. The Linux data traces frequently have a jut not seen in any trace for either of the Windows operating systems. (We do not know what characteristic of Linux creates this jut that is not created by the Windows operating systems, but the point of this dissertation is cache studies, so we leave such investigations to future work.) Also, the Linux data traces tend to have larger effective memory ranges and the Windows data traces tend to have larger effective working set sizes. In data traces with large loops, such as *mcf*, the Linux data trace tends to have loops on both sides of the temporal axis while the Windows data traces tend to be only on the negative side. We now look at a few examples.

First we look at the instruction traces of *gzip* with the *source* input file as one example of instruction trace differences. Figure 3.25 contains the surfaces for the instructions of *gzip.source* under Linux, Windows NT, and Windows 2000. Here we can see some of the minor differences mentioned for instruction traces, the Windows NT surface shows two blips at larger positive and negative strides than seen for the Linux surface. The Windows 2000 surface shows only one blip, on the positive stride side. This workload has one of the larger differences between the three different

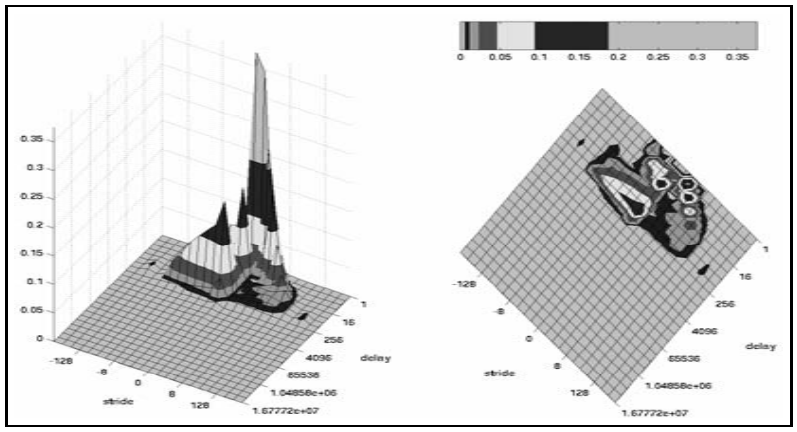
operating systems seen in the instruction traces. The differences are obviously minor, and may generally be ignored. We can also see here the effective working set size trend. The Windows NT trace has the smallest effective working set size, followed by the Linux trace. The Windows 2000 trace has the largest effective working set size. We speculate that this means that Windows 2000 generally uses more unique instructions than either Linux or Windows NT.

We now examine data trace differences across operating systems. Our first selected data trace is the data reads and writes of *gap*. We have already seen the locality surface for the data trace of *gap* under Linux in Figure 3.19, however we repeat the surface in Figure 3.26 for comparison purposes. Figure 3.26 also contains the locality surfaces for the data of *gap* under Windows NT and Windows 2000. Before comparing the surfaces, we first note that the scales are almost identical. We now note that the shape of the two Windows surfaces are almost identical. The Linux surface has a different shape along the top of the temporal axis and has the jut already mentioned. The effective working set sizes of the three surfaces are the same, but thanks to different features. Without the jut on the Linux surface, its effective working set size would be much smaller than for the other two surfaces. The effective memory range is significantly different. For the Linux trace, the effective memory range is 64 Kwords. However, the Windows NT and 2000 traces have effective memory ranges of 512 words and 256 words, respectively.

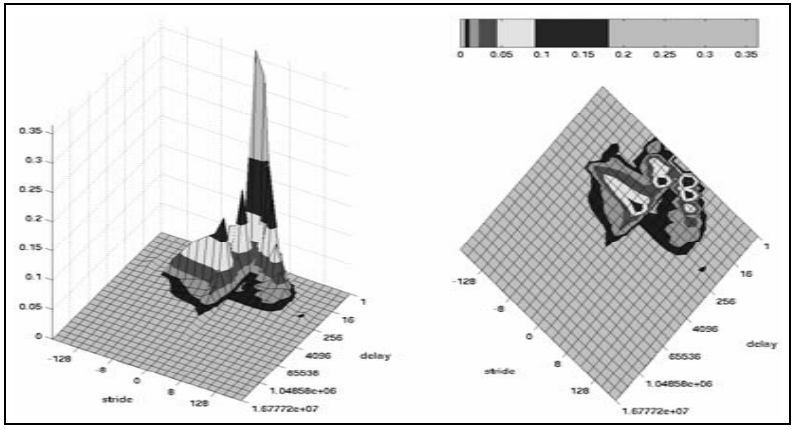
We now look at the data traces for *mgrid*. This is an example of the few traces where the surfaces for Windows NT and Windows 2000 are significantly different. Figure 3.27 shows the locality surfaces for the data traces of *mgrid* under Linux, Windows NT, and Windows 2000. This time we note that the scales are significantly different for each of the three surfaces; we must take this into consideration when comparing them.



(a) Linux

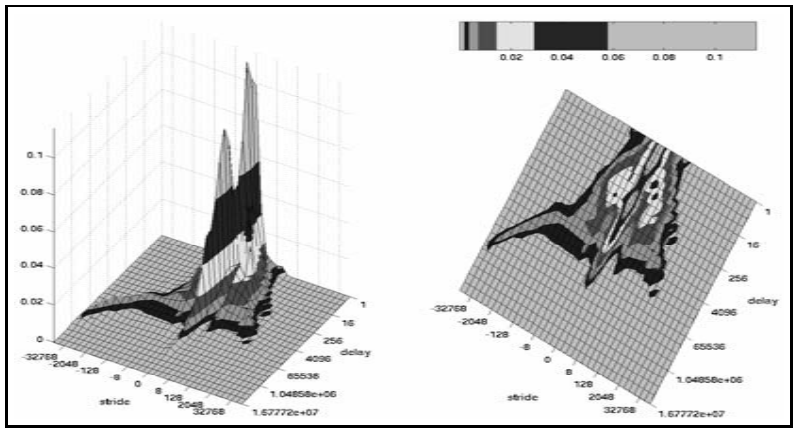


(b) Windows NT

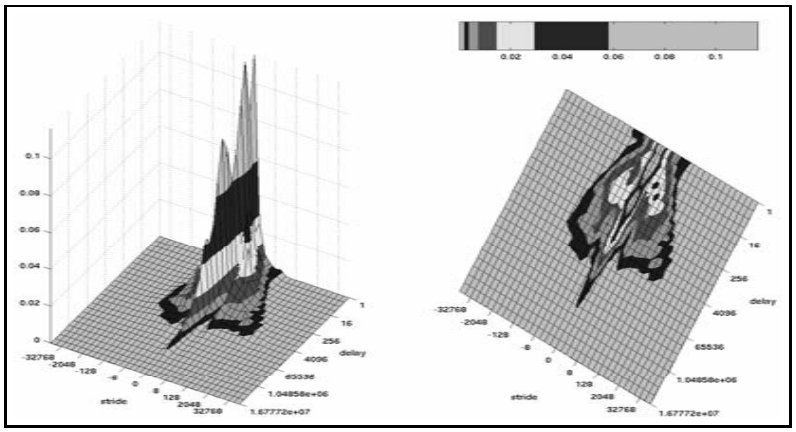


(c) Windows 2000

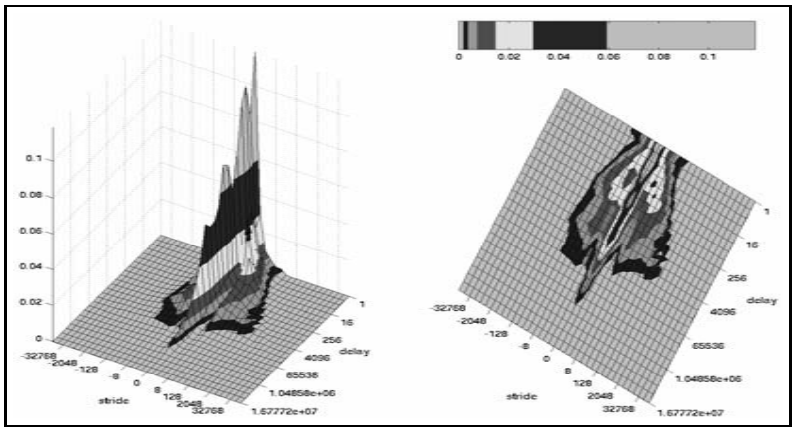
Figure 3.25: The locality surfaces for the instruction traces of the *gzip* workload with the *source* input file under three different operating systems: 3.25(a) is under Linux, 3.25(b) is under Windows NT, and 3.25(c) is under Windows 2000.



(a) Linux



(b) Windows NT



(c) Windows 2000

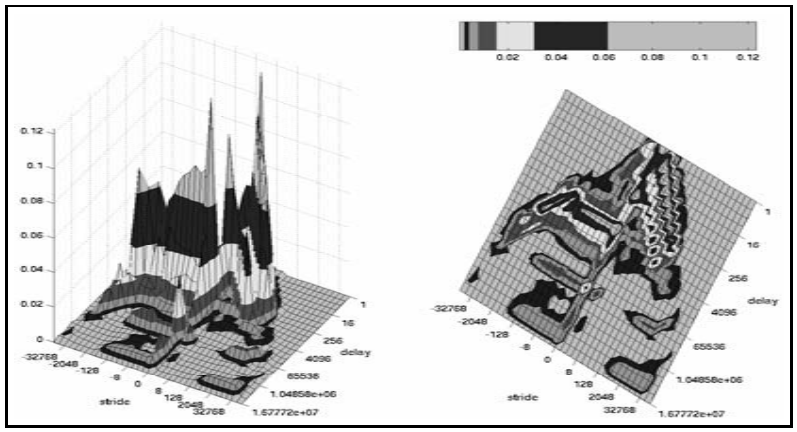
Figure 3.26: The locality surfaces for the data traces of the *gap* workload under three different operating systems: 3.26(a) is under Linux, 3.26(b) is under Windows NT, and 3.26(c) is under Windows 2000.

We first notice that the Linux surface has the characteristic jut that is in neither of the Windows surfaces. Next, we see that both the Linux surface and the Windows 2000 surface have loop features on both sides of the temporal axis while the Windows NT surface has loop features only on the negative stride side. This is true even if we match the scales. In addition, the loop at about 1 Kwords is much more dominant in the Linux surface than in either of the Windows surfaces. All three surfaces have a significant loop at about 128 Kwords and another at about 4 Mwords. The Windows 2000 trace, however, has loops of various sizes between these two. An interesting side note is that *mgrid* shows a delayed sequential ridge under all three operating systems.

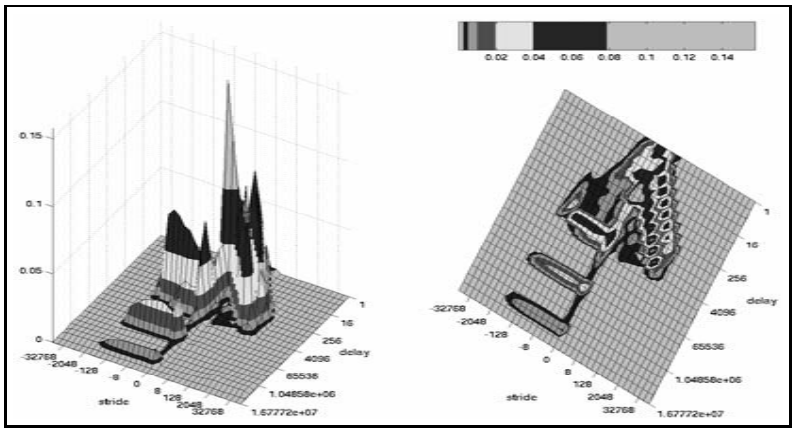
For almost all of the workloads in the SPEC CPU 2000 suite, the general shapes of the locality surfaces between the different operating systems are similar. For one workload, however, this is not true. Compare the locality surface for the data of *wupwise* under Linux (Figure 3.28(a)) with the surfaces for the data of *wupwise* under Windows NT (Figure 3.28(b)) and Windows 2000 (Figure 3.28(c)).

The Linux surface has a completely different shape from the two Windows surfaces, enough to firmly place them in separate general shape categories. Figure 3.28(a) is definitely Category 5, while Figures 3.28(b) and 3.28(c) are closer to Category 4, without the jut. Table 3.1 tells us that the number of unique references in the Linux trace is about ten times the number in either of the Windows traces. Again, this is the only workload in the SPEC CPU 2000 suite that did this.

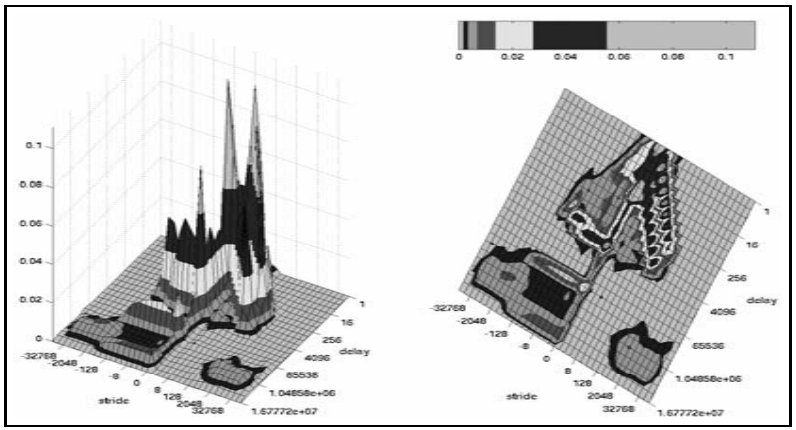
Some have noticed the differing lengths in the Linux *wupwise* trace verses either of the Windows *wupwise* traces (51 million verses 36 million, see Table 3.1). This suggests that perhaps the first 36 million references of all three traces creates the features seen in Figures 3.28(b) and 3.28(c), while the last 15 million references of the Linux trace add the extra features seen in Figure 3.28(a). We have checked this



(a) Linux

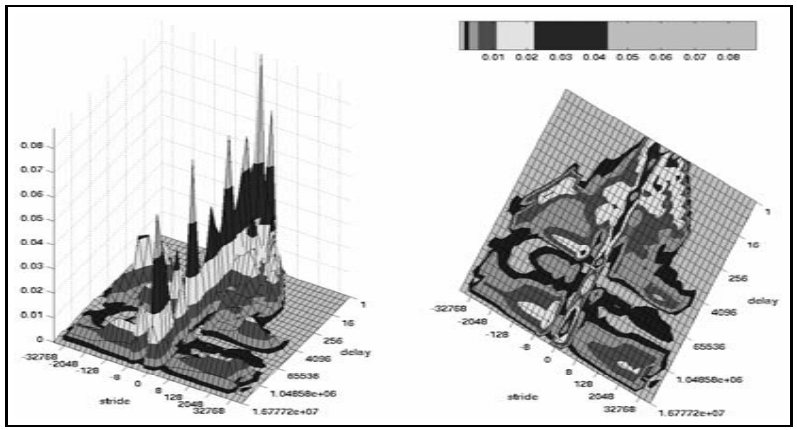


(b) Windows NT

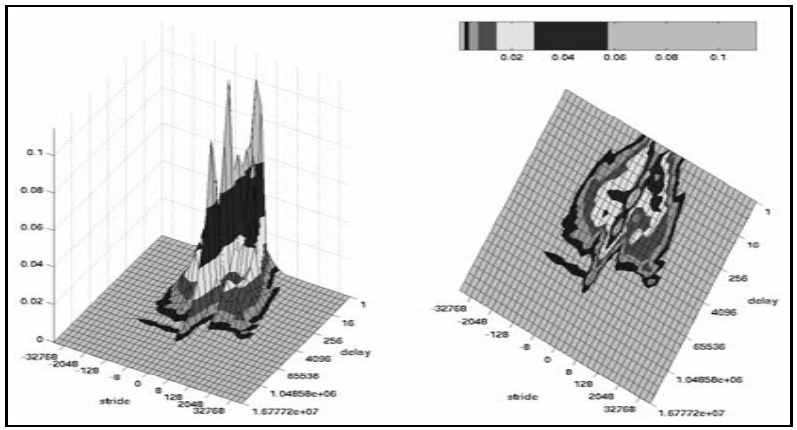


(c) Windows 2000

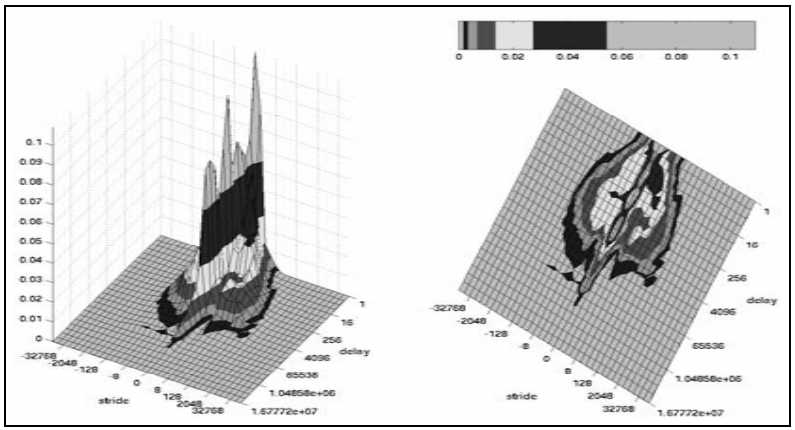
Figure 3.27: The locality surfaces for the data traces of the *mgrid* workload under three different operating systems: 3.27(a) is under Linux, 3.27(b) is under Windows NT, and 3.27(c) is under Windows 2000.



(a) Linux



(b) Windows NT



(c) Windows 2000

Figure 3.28: The locality surfaces for the data traces of the *wupwise* workload under three different operating systems: 3.28(a) is under Linux, 3.28(b) is under Windows NT, and 3.28(c) is under Windows 2000.

hypothesis by creating the locality surface for only the first 36 million references of the Linux *wupwise* trace. The locality surface of the truncated Linux *wupwise* trace has all of the features seen in Figure 3.28(a) clearly displayed to the point that displaying the truncated trace's locality surface is redundant.

We believe that there is a simpler explanation for the differing lengths of the *wupwise* traces. Whatever caused the difference in locality between the Linux and Windows traces also altered the instruction/data mix, affecting the length of our split traces as mentioned in Section 3.2.

Conjectures such as this should be checked by a serious study of how the locality of a given workload varies across multiple traces of the same workload before being stated as solid conclusions. It is possible that the deviations of the Linux trace of *wupwise* was caused by some anomaly of the tracing system evident in that particular tracing run. It is less likely that the two Windows traces were caused by some unknown anomaly, since they are so similar to each other. We do not think it likely that the different locality evidenced by the Linux trace of *wupwise* was merely an anomaly, since several other floating point data traces have similar locality (see Figure 3.27 and the data locality surface for *swim* in Appendix B).

Other unexplained variations in locality, such as seen in Figure 3.24, may also be caused by unknown factors in the tracing method. Again, we do not believe this is likely, however it is possible. Before the observations made in this chapter become conclusive statements, such a possibility should be investigated. We leave this to future work.

3.4 Summary

In this chapter, we have created synthetic traces that produce a variety of features on the locality surface. We have also shown the locality surfaces for a number of traces from the SPEC CPU 2000 benchmark suite. The locality surfaces created from synthetic traces have helped us understand what the features on a surface indicate about the input trace. Further investigation in this area would yield a greater understanding of how locality surface features and input trace patterns relate.

In addition, we have described how instruction traces and data traces typically appear in terms of locality. We have also compared the general features of integer workloads versus floating point workloads and the trends for workloads with different inputs. Lastly, we compared the traces for workloads taken under differing operating systems. Next we use a number of locality surfaces to qualitatively predict the cache performance of the input trace data.

Chapter 4

Qualitative Cache Performance Prediction Using Locality Surfaces

In this chapter, we look at a number of locality surfaces and qualitatively predict optimal cache size and whether larger or smaller line sizes perform better. We examine a range of actual cache simulation results and check the accuracy of our predictions. We do this to demonstrate how well our locality surface matches cache performance, in terms of miss rate, and its value in qualitatively predicting cache simulation results. This allows researchers to better focus their simulation efforts on interesting cache configurations and to avoid investigating obviously impractical cache configurations.

Table 4.1 shows the statistics for the traces used in this chapter. Some of the traces and locality surfaces were used in earlier chapters. We show these locality surfaces again to make life easier for the reader. These same six traces are also used for quantitative cache performance predictions in Chapter 7. We selected these traces because they represent both integer and floating point benchmarks, instruction and data traces, and a variety of features on the locality surface. In

workload	suite	type	total refs	uniq refs	description
applu	FP	D	46,261,474	1,524,041	Parabolic/Elliptic Partial Differential Equations
crafty	INT	I	50,020,348	30,338	Game Playing: Chess
galgel	FP	D	37,070,561	1,255,136	Computation Fluid Dynamics
perlbmk.diffmail	INT	I	54,083,478	34,648	PERL Programming Language
swim	FP	D	42,031,084	7,988,204	Shallow Water Modeling
twolf	INT	I	50,191,887	21,988	Place and Route Simulator

Table 4.1: Description of the traces used in this chapter. All of these traces were taken under the Linux operating system.

short, we believe these six traces to cover the range of locality among the SPEC benchmarks.

4.1 Predicting Optimal Cache Size

We begin by using the locality surface to predict optimal cache size. It should be obvious that increasing cache size never decreases cache performance, in terms of miss rate. However, there frequently is a point beyond which increasing the cache size improves performance very little. Optimal cache size can generally be predicted by noting at what point along the delay axis the majority of the locality surface volume is contained. For many locality surfaces, this involves noting the location of the major looping structures.

We now demonstrate this ability using three different workloads. For each workload, we compare the locality surface with the cache performance of each trace. The

first surface has a small effective working set size, the second has a larger effective working set size, and the third has one of the largest effective working set sizes.

The first trace we examine is the instructions of *twolf*. The locality surface is shown in Figure 4.1. The effective working set size for this trace is 4 Kwords. The most significant loop on this surface is at a delay of 1 Kword. There is little data on the surface at a delay of 256 words. Therefore, we expect to see dramatic cache performance improvements as the cache size increases to about 128 words, or 1 Kbytes. We then expect little improvement as the cache size further increases in size to 512 words, or 4 Kbytes. We expect a more dramatic improvement as the cache size reaches 1 Kword, or 8 Kbytes, further improvement as the cache size reaches 4 Kwords, or 32 Kbytes, and very little improvement as the cache size increases further.

Figure 4.2 shows the cache performance, obtained from cache simulations, for the instruction trace of *twolf*. The x axis shows the cache size, increasing by powers of two. The y axis shows the miss rate, i.e. the number of misses in the given cache divided by the total references submitted to the cache. Four different associativities are shown, from direct mapped to 8-way associative caches. All the caches simulated for Figure 4.2 have 8-byte line sizes.

The 8-way associative cache line most closely matches our predictions. We see consistent improvement in cache performance until the cache size reaches about 1 Kbyte, then a plateau. We see the most dramatic cache performance improvement as the cache size moves from 4 Kbytes to 8 Kbytes, and maximal cache performance is reached between 32 Kbytes and 64 Kbytes. The other associativities show similar trends, but at larger cache sizes. For example, the direct-mapped cache line reaches maximal cache performance at about 256 Kbytes. We discuss in Chapter 5 why caches with greater associativities match more closely with the locality surface.

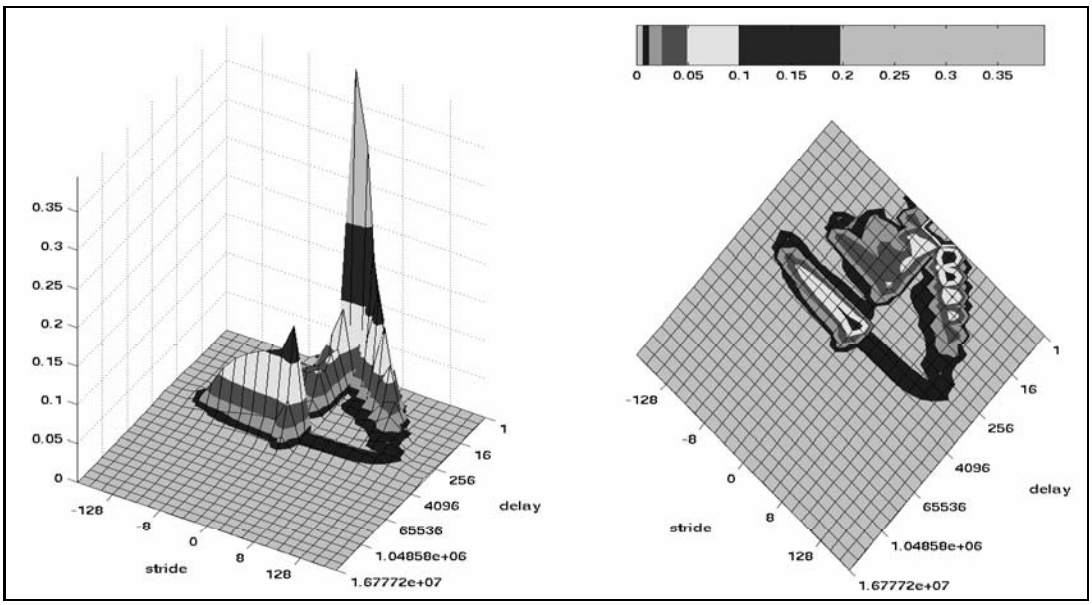


Figure 4.1: Locality surface for the instruction trace of *twolf*.

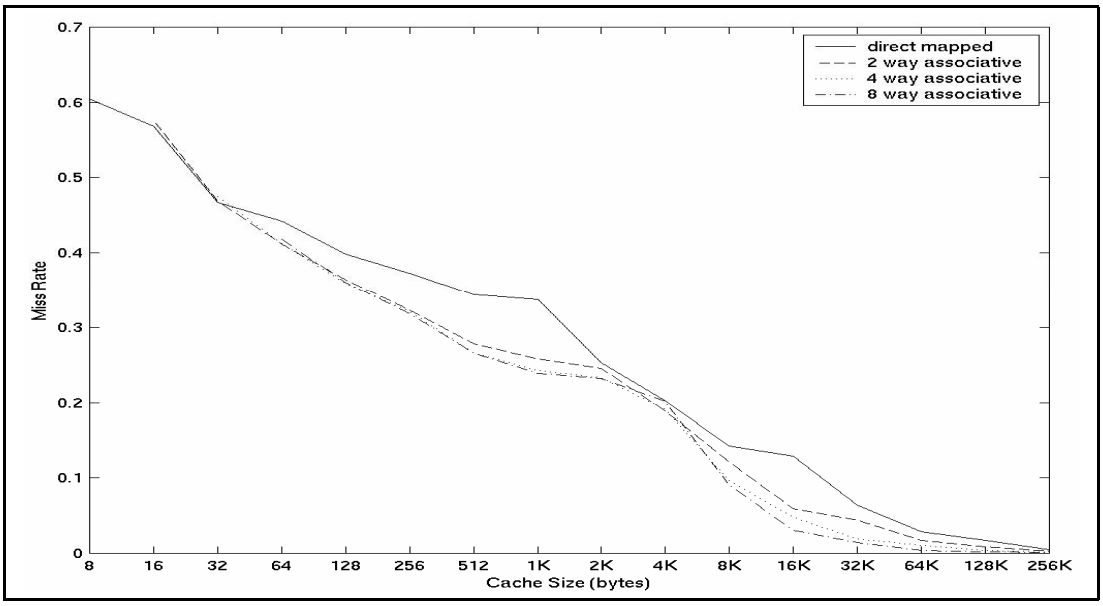


Figure 4.2: Cache simulation results for the instruction trace of *twolf*. All caches have an 8-byte line size.

Figure 4.3 shows the locality surface for the data of *applu*. The effective working set size for this trace is 512 Kwords. There appears to be only one loop in this trace, with a delay of 2 Kwords. The vast majority of locality surface features are at a smaller delay of 2 Kwords, hence we expect much more cache performance improvements at cache sizes smaller than 16 Kbytes, and diminishing improvements at larger cache sizes. There is a lack of temporal locality data between 32 Kwords and 128 Kwords. We therefore expect little cache performance improvement as the cache size increases from 16 Kwords, or 128 Kbytes, to 128 Kwords, or 1 Mbyte.

Figure 4.4 shows the cache simulation results for the data trace of *applu*. We see several associativities and all the caches have an 8-byte line size. Again, the largest associativity shown (8-way associative) has the best match with our predictions. The miss rate decreases sharply until about 16 Kbytes, and then plateaus until 2 Mbytes. As the cache size increases from 2 Mbytes to 4 Mbytes, we see another sharp improvement in miss rate. At this point, the optimum cache performance is reached, and we see almost no further improvement.

One of the advantages of qualitative cache performance predictions is to give cache designers an idea of the likelihood of significant returns when increasing the cache size. For example, Figure 4.4 shows us that increasing the cache size from 256 Kbytes to 512 Kbytes, or from 32 Mbytes to 64 Mbytes, yields almost no performance improvements. This can also be observed from the locality surface, without the need for any cache simulations.

Figure 4.5 shows the locality surface for the data of *swim*. The effective working set size for this trace is 8 Mwords. One can see a number of looping structures on this surface. Due to the large amount of locality data at all delays, we expect somewhat consistent improvements at each step as we increase the cache size up to about 8 Mwords, or 64 Mbytes. We expect a somewhat sharper improvement in

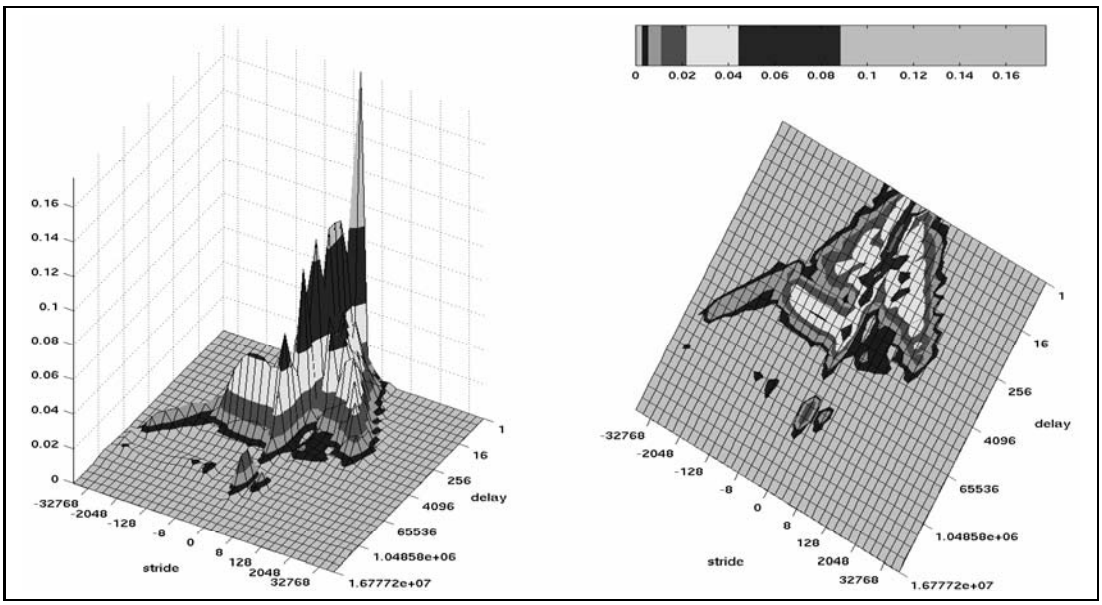


Figure 4.3: Locality surface for the data trace of *applu*.

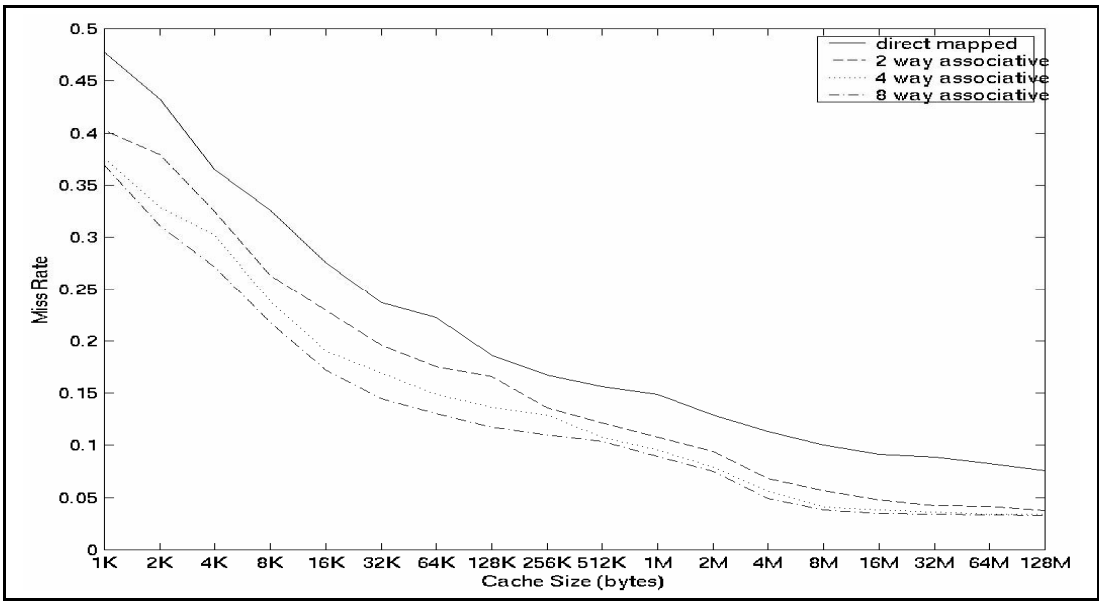


Figure 4.4: Cache simulation results for the data trace of *applu*. All caches have an 8-byte line size.

cache performance as we reach a cache size of 16 Kwords, or 128 Kbytes.

Figure 4.6 shows the cache simulation results for the data of *swim*. The cache performance does improve at each increase of cache size, with the sharpest increase reached as we move to 128 Kbytes. For the 8-way associative cache, we see the optimal cache performance reached at about 64 Mbytes, as predicted. Looking at the miss rates in Figure 4.6, we see that increasing the cache size at any point would be advantageous. This can also be seen from the locality surface in Figure 4.5, without the need for numerous cache simulations.

4.2 Predicting Optimal Line Size

We now attempt to qualitatively predict optimal line sizes using the locality surface. We can predict whether smaller or larger line sizes are more optimal primarily based on the nature of the sequential ridge in a locality surface. Traces with large amounts of sequential references perform better with larger line sizes, and also have larger sequential ridges. We compare three other locality surfaces with their associated cache simulation results. The first surface has a large sequential ridge, the second has a small sequential ridge, and the third has no sequential ridge.

The locality surface for the instruction trace of *crafty* is found in Figure 4.7. This trace produces the most pronounced sequential ridge of any of the traces from the SPEC CPU2000 suite. There appear to be a number of sequential runs that are at least 64 words long in this trace. We therefore expect to see significant cache performance improvement as the line size is increased. The effective working set size for this surface is 8 Kwords. We should see improvements as we increase cache size until reaching 8 Kwords, or 64 Kbytes, where nearly optimal performance would be reached.

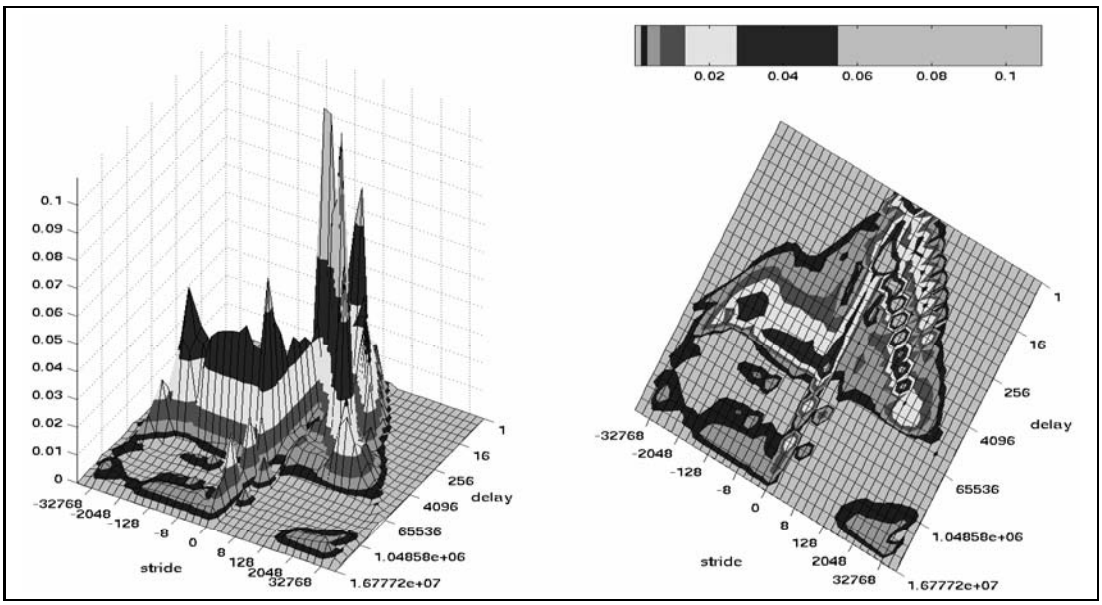


Figure 4.5: Locality surface for the data trace of *swim*.

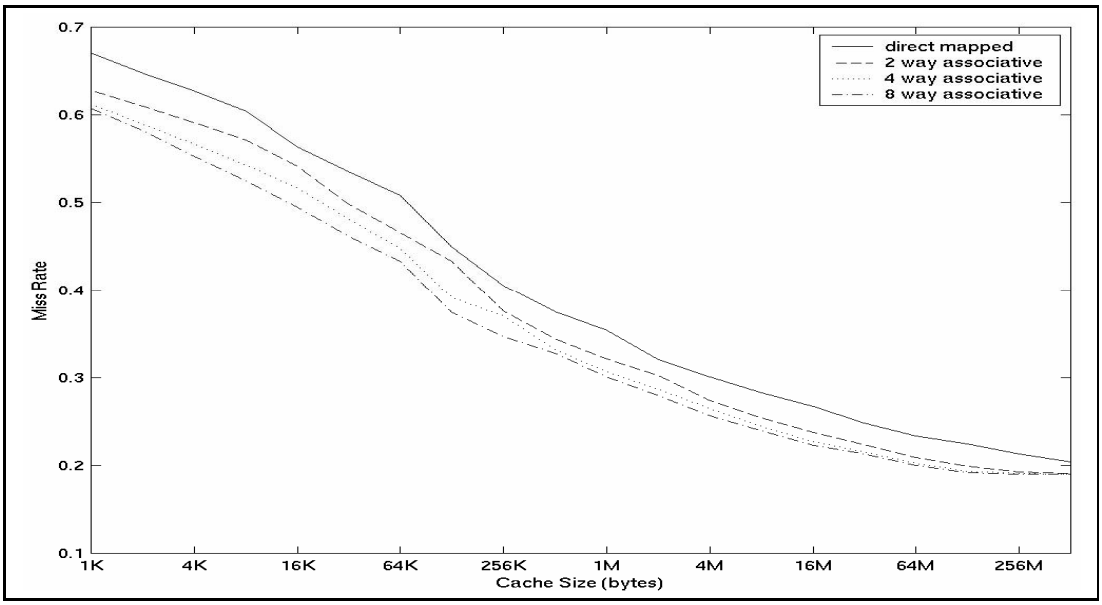


Figure 4.6: Cache simulation results for the data trace of *swim*. All caches have 8-byte line sizes.

Figure 4.8 shows the cache results obtained from simulation. In this figure, each line of the graph indicates a different cache line size. We see exactly the predicted trends. Increasing the line size significantly increases the overall cache performance. In addition, optimal cache performance is reached at about 64 Kbytes, as predicted.

Figure 4.8 shows that increasing the line size by one or two factors generally improves cache performance more than by increasing the cache size by one or two factors. For a 1 Kbyte cache, merely changing from an 8-byte line size to a 16-byte line size improves the miss rate from 53.97% to 37.95%. If instead of doubling the line size we quadruple the cache size, we only see the miss rate improving from 53.97% to 42.01%. These results are predicted by the locality surface. When the surface shows a large sequential ridge and an effective working set size not much larger than the maximum sequential run, increasing the line size would achieve much better performance than increasing the cache size.

The locality surface for the instruction trace of *perlbmk* with the *diffmail* input is shown in Figure 4.9. The locality surface shows a much smaller sequential ridge, reaching a maximum delay of 16 words. We therefore expect that the longest sequential run in the trace is 16 words long. The effective working set size is 4 Kwords. Unlike the instruction trace of *crafty*, *perlbmk.diffmail* has a much larger effective working set size than the length of the maximum sequential run. This means that the instructions of *perlbmk.diffmail* reach the point where increasing the line size creates more misses than hits faster than the instructions of *crafty* did.

The cache results for the instruction trace of *perlbmk.diffmail* is in Figure 4.10. As predicted, we see improvements as the line size is increased, but much smaller improvements than seen for the instructions of *crafty* in Figure 4.8. (Note different scales when comparing the graphs.) This shows that, indeed, the surface with the more dominate sequential ridge has greater improvements with increased line size

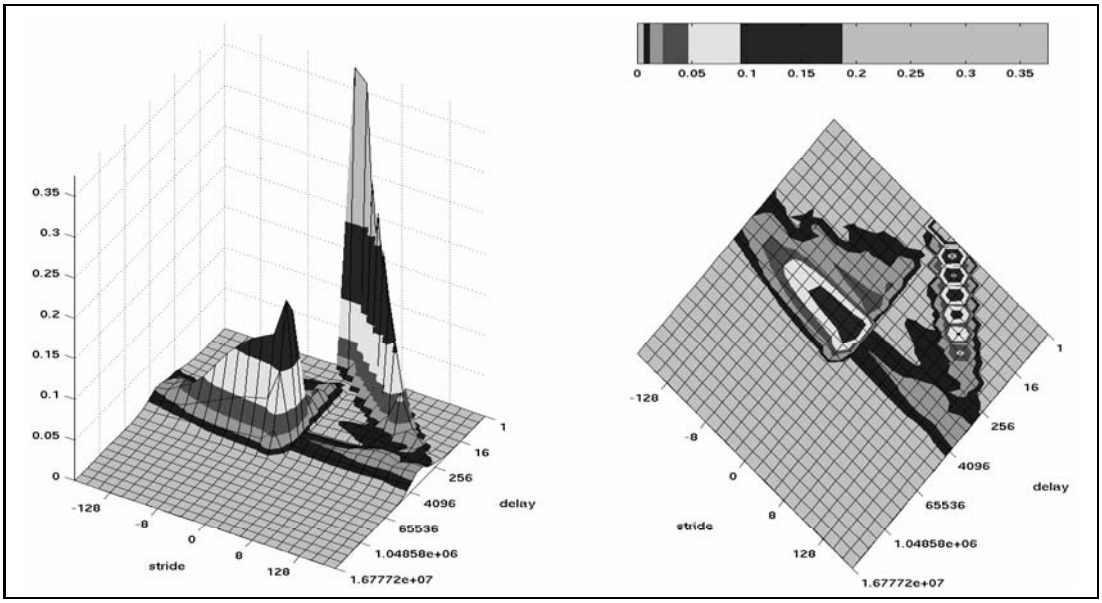


Figure 4.7: Locality surface for the instruction trace of *crafty*.

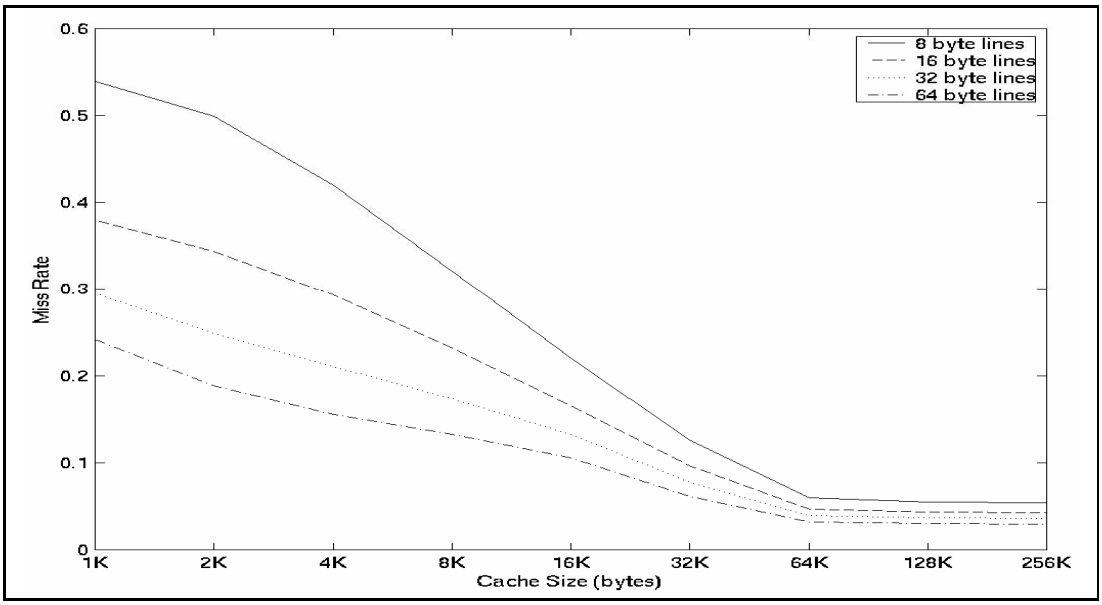


Figure 4.8: Cache simulation results for the instruction trace of *crafty*. All caches are direct mapped.

than the surface with the smaller sequential ridge.

Because this time the effective working set size is much larger than the maximum sequential run, increasing the cache size is now more effective than increasing the line size. Merely looking at the graph shows us that, beginning with a 1 Kbyte cache with 8-byte lines, doubling the cache size improves performance more than increasing the line size to 64 bytes.

The locality surface for the data trace of *galgel* is found in Figure 4.11. Here we have an example of a trace that consists primarily of temporal locality. When examining the cache results, we should see sharp improvements as we increase the cache size up to about 1 Kwords, or 8 Kbytes, continued but lessening improvements up to a cache size of 64 Kwords, or 512 Kbyte, and then larger improvements as we increase the cache size to 1 Mword, or 8 Mbytes. There is a small amount of sequential locality data, however it is dominated by the temporal locality. Therefore we expect less performance improvements with increased line size.

Figure 4.12 shows some cache simulation results for direct mapped caches with varying line sizes. First we notice that varying the line size changes the cache results very little. In fact, 8-byte and 16-byte line size results are almost identical, as are 32-byte and 64-byte line size results.

4.3 Summary

In this chapter, we have seen how our locality surface matches well with cache simulation results, particularly with caches with larger associativies. We have also seen how optimal cache size and optimal line size can be predicted with the use of the locality surface. In the next chapter, we mathematically describe caches,

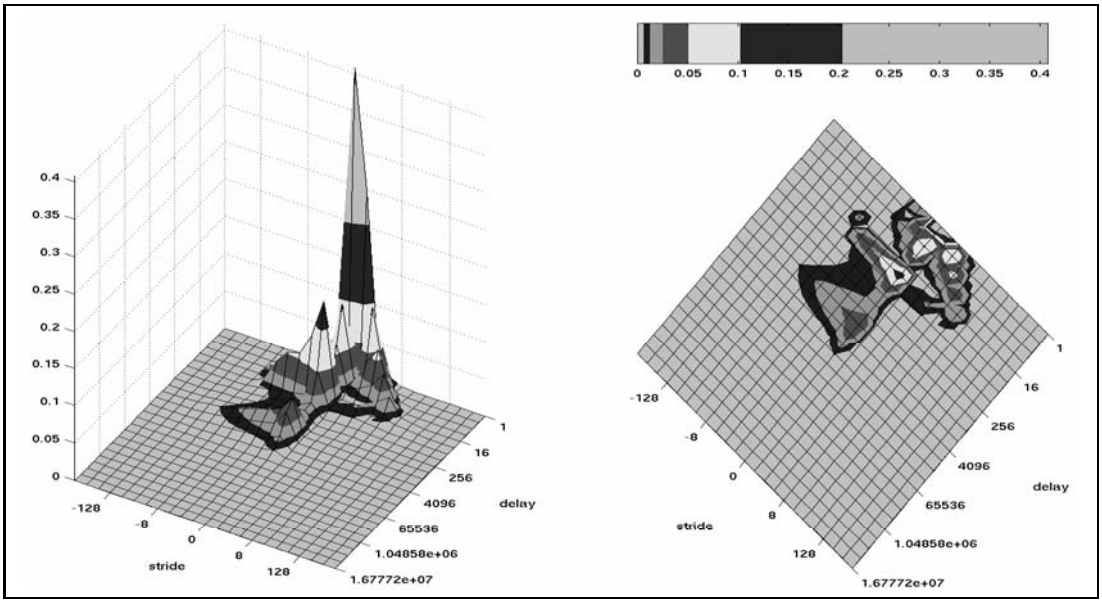


Figure 4.9: Locality surface for the instruction trace of *perlbmk.diffmail*.

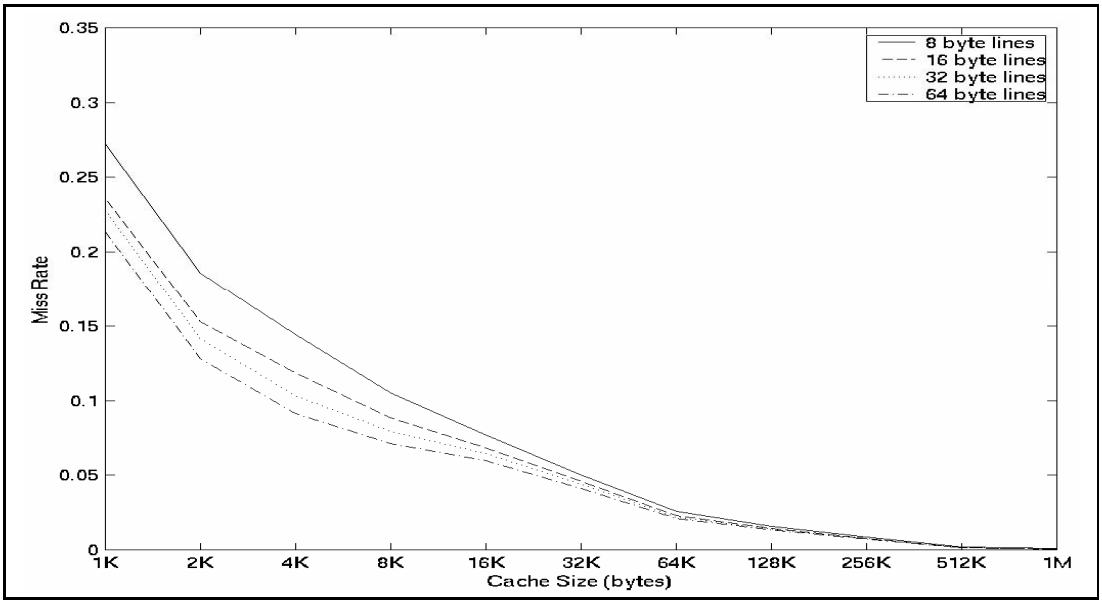


Figure 4.10: Cache simulation results for the instruction trace of *perlbmk.diffmail*. All caches are direct mapped. When comparing with the results in Figure 4.8, note the difference in scale on the y-axis.

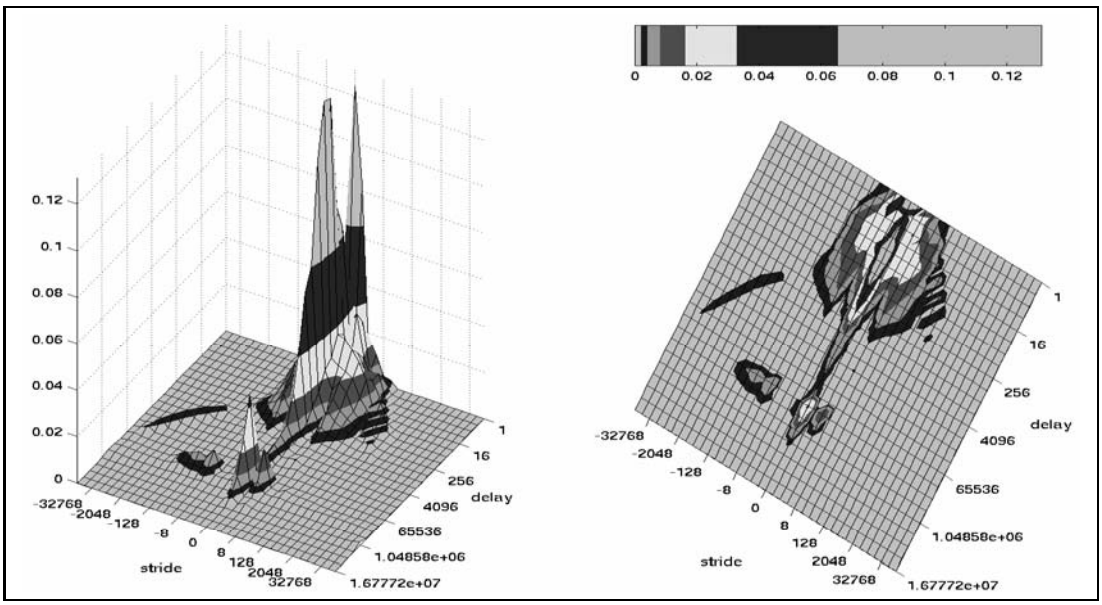


Figure 4.11: Locality surface for the data trace of *galgel*.

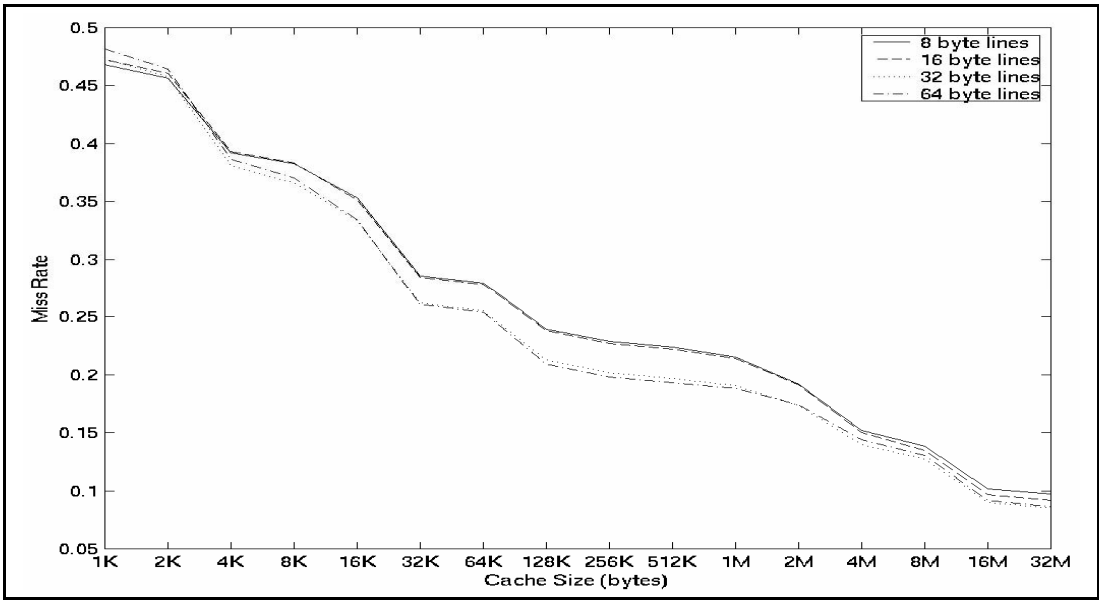


Figure 4.12: Cache simulation results for the data trace of *galgel*. All caches are direct mapped.

present a cache characterization surface, and answer the question as to why caches with larger associativities correlate better with locality.

Chapter 5

Caches and Locality

We have just seen that locality and cache miss rate seem correlated with respect to cache size and line size. Now we ask, can we quantify the correlation and verify these results for the general case? How does associativity fit into the equation? To answer these questions, it would help if we could first describe caches in terms of locality without tying the results to a particular workload. In this chapter, we do just that for traditional caches.

We begin by writing an equation for the miss rate of a cache, based on the cache configuration and the input string. This equation mathematically represents what a cache simulator does. Next, we introduce three new surfaces, the miss surface, the miss rate surface, and finally the cache characterization surface.

This last surface is our desired description of a given cache in terms of locality that visually represents how various stride/delay relationships impact a given cache, independent of any workload. This chapter focuses exclusively on traditional caches, i.e. caches that may be described entirely by their size, line size, and associativity. However, the techniques described may be applied to any cache with a LRU replacement policy, such as column-associative [6] and skewed associative [68]

caches.

To our knowledge, no other group of researchers have attempted to describe caches in terms of locality. The closest that other researchers have come is the memory mountain [18, 76], a three-dimensional graph with working set size, stride, and throughput as the dimensions. The mountain essentially graphs throughput versus locality rather than miss rate versus locality as we do. In addition, Bryant and O'Halloran used their memory mountain to characterize the entire memory system of an actual computer, rather than the miss rate of one level of the system. We also make our cache characterization surfaces for any cache we have a simulator for, not just caches that have been built.

5.1 A Description of Traditional Caches

We begin by describing traditional caches and creating an equation which, given a cache configuration and input string, returns the miss rate for that string in the given cache. Let C indicate the parameters necessary to describe a specific cache. We use subscripts to indicate specific parameters. For traditional caches, the cache size, line size, and associativity completely describe the cache. Therefore, we let C_s represent the cache size, C_l represent the line size, and C_a represent the associativity. Notice that the number of lines in a cache is C_s/C_l, and the size of a cache set is C_aC_l. When a cache is fully associative, then $C_a = C_s/C_l$. We describe a direct mapped cache as a 1-way associative cache, i.e. where $C_a = 1$. We also use the term **set associative** to refer to any cache that is not fully associative.

We wish to create a function where given a cache configuration, a string, an index to a specific element of the string, and the granularity of the string, we can determine if the specific string element results in a hit or miss in the given cache.

(Recall from Section 3.3 that granularity is defined as the level at which the locality relationship is determined.) Let $miss(C, v, i, g)$ be such a function, where C is the cache configuration, v is the string, $v[i]$ is the specific element, and g is the granularity of the string. $miss$ is a boolean function that returns 1 if the element is a miss in the cache and 0 if it is a hit. Formally,

$$miss(C, v, i, g) = \begin{cases} 1 & \text{if } v[i] \text{ is a miss in cache } C, \\ 0 & \text{otherwise.} \end{cases}$$

Now we must mathematically describe how to determine when $v[i]$ misses in the given cache. We consider four cases: 1) when the cache is fully associative and the line size equals the string granularity, 2) when the cache is fully associative and the line size does not equal the string granularity, 3) when the cache is set associative and the line size equals the string granularity, and 4) when the cache is set associative and the line size does not equal the string granularity.

5.1.1 Case One

When the cache is fully associative and the line size equals the string granularity, we can determine if a particular element, $v[i]$, of the string is a hit or miss by finding the last time that value was in the string and counting the number of unique references between them. If the number of unique references, i.e. the delay, is less than the number of lines in the cache, then it is a hit, otherwise it is a miss. If the delay is greater than the number of cache lines or $v[i]$ is the first instance of the value in the string, then $v[i]$ is a miss.

In other words, given $C_a = C_s/C_l$ and $C_l = g$, $miss(C, v, i, g)$ is 0 if there exits an integer d such that $d \leq C_a$ and $(0, d)$ is in the locality bag for $v[i]$. Recall that $(0, d)$ is only in $\ell(v[i])$ if $v[i]$ is a repeated value and the most recent instance of that

value in v was d unique references earlier. In all other instances, $miss(C, v, i, g)$ is 1. Formally, when $(C_a = C_s/C_l) \wedge (C_l = g)$, then:

$$miss(C, v, i, g) = \begin{cases} 0 & \text{if } (\exists d)\big[(0, d) \in \ell(v[i]) \wedge d \leq C_a\big], \\ 1 & \text{otherwise.} \end{cases} \tag{5.1}$$

Example 5.1. *Let* $v = 194, 35, 193, 57, 290, 259, 66, 310, 118, 222, 158, 57, 194, 130, 150, 345, 194, 246, 310, 67, 66, 57, 162, 54, 193, 67, 89, 98, 226, 257$. *Note that* $|v| = 30$. *Also, let* $g_v = 8$.

For this example, let $C_s = 64$, $C_l = 8$, *and* $C_a = 8$. *Note that* $C_a = C_s/C_l$ *and* $C_l = g_v$. *We now calculate* $miss(C, v, i, g)$ *for several of the elements of* v *in cache* C *using Equation 5.1:*

$$\begin{aligned} miss(C, v, 1, g) &= 1, \\ miss(C, v, 4, g) &= 1, \\ miss(C, v, 12, g) &= 0, \quad \textit{since } (0, 8) \in \ell(v[12]) \textit{ and } 8 \leq 8, \\ miss(C, v, 21, g) &= 1, \\ miss(C, v, 22, g) &= 1, \\ miss(C, v, 26, g) &= 0, \quad \textit{since } (0, 6) \in \ell(v[26]) \textit{ and } 6 \leq 8. \end{aligned}$$

5.1.2 Case Two

We now consider the case where the cache is fully associative, but the line size and string granularity are not equal. We assume that the granularity is never greater than the line size, and that both the line size and granularity are always powers of two. Given these assumptions, we can write $C_l = 2^h g$, where $h \geq 1$. (If $h = 0$, then the line size and granularity are equal. If $h < 0$, then the granularity is greater than the line size; the elements of the string are too large to fit in a cache line.)

We wish to create a new string that is equivalent to v but with the granularity adjusted to match C_l. We define a function, $zoom(v, h)$, that returns the desired string, where h is the factor necessary to adjust the granularity appropriately, i.e. h

is a positive integer such that $C_l = 2^h g$. To increase the granularity of a particular string element, we divide that element by the zoom factor. Since we always adjust the granularity by a power of two, we simply shift the string element right by h bits. Therefore, $zoom(v, h) = w$ where $w[k] = v[k] >> h$, $|w| = |v|$, and $1 \leq k \leq |v|$.

Now, given $C_a = C_s/C_l$ and $C_l = 2^h g$,

$$miss(C, v, i, g) = miss(C, zoom(v, h), i, g') \tag{5.2}$$

where $h = \log_2(C_l/g)$ and $g' = C_l$. We have now transformed Case Two into an instance of Case One and may use Equation 5.1.

Example 5.2. *Let* $v = 194, 35, 193, 57, 290, 259, 66, 310, 118, 222, 158, 57, 194, 130, 150, 345, 194, 246, 310, 67, 66, 57, 162, 54, 193, 67, 89, 98, 226, 257$. *Note that* $|v| = 30$. *Also, let* $g_v = 8$.

For this example, let $C_s = 64$, $C_l = 32$, *and* $C_a = 2$. *Note that* $C_a = C_s/C_l$ *and* $C_l \neq g_v$. *Before we can use Equation 5.1, we must first calculate* $zoom(v, h)$ *where* $C_l = 2^h g_v$. *We let* $h = 2$. *To calculate* $zoom(v, 2)$, *we simply shift each element of* v *right two places. Therefore,* $zoom(v, 2) = 48, 8, 48, 14, 72, 64, 16, 77, 29, 55, 39, 14, 48, 32, 37, 86, 48, 61, 77, 16, 16, 14, 40, 13, 48, 16, 22, 24, 56, 64$.

We now calculate $miss(C, zoom(v, 2), i, C_l)$ *for several of the elements of* v *in cache* C *using Equation 5.1:*

$$\begin{aligned}
miss(C, zoom(v, 2), 1, C_l) &= 1, \\
miss(C, zoom(v, 2), 4, C_l) &= 1, \\
miss(C, zoom(v, 2), 12, C_l) &= 1, \\
miss(C, zoom(v, 2), 21, C_l) &= 0, \\
miss(C, zoom(v, 2), 22, C_l) &= 1, \\
miss(C, zoom(v, 2), 26, C_l) &= 1.
\end{aligned}$$

5.1.3 Case Three

When the cache is set associative and the line size equals the string granularity, we can determine if a particular element, $v[i]$, of the string is a hit or miss by first

selecting all the elements of the string that map to the same cache set as $v[i]$ and then performing the same operations as in Case One, only using the number of lines in the cache set rather than the number of lines in the entire cache.

First, we need a way to create a new string consisting entirely of the elements of v that map to the same cache set as $v[i]$. We define a function, $cacheset(v, i, C)$, that returns the desired new string given the input string, v, the index to an element in v that maps to the desired cache set, and the cache configuration, C. We compute $cacheset(v, i, C)$ using several intermediary functions. Intermediary functions are functions that are only used for computing the result of the equation or function they are attached to. We let $p_x()$ indicate an intermediary function, where x designates which one. Functions that are used in several places, i.e. non-intermediary functions, are given names, e.g. $cacheset(v, i, C)$.

We let $p_1(v, i, k, C)$ be our first intermediary function. It determines if $v[k]$ is in the same cache set as $v[i]$. We define a function, $count(v, i, k, C)$, that tells us how many elements in v, up to index k, map to the same cache set as $v[i]$. Using both $p_1(v, i, k, C)$ and $count(v, i, k, C)$, we can then create another intermediary function, $p_2(v, i, k, C)$ that returns the index into v of the kth element of v that is in the same cache set as $v[i]$. Using $p_2(v, i, k, C)$, we can create the desired string of elements that map to the same cache set as $v[i]$. Note that while all three of these funtions have the same inputs, i.e. v, i, k, and C, k has a different meaning in each.

While performing all of these operations, we execute a couple of computations repeatedly. To simplify notation, we here introduce two variables, n and t, as shortcuts. Since the granularity of v equals the line size of the cache in Case Three, we know the lower order bits of each element indicate the cache set. The number of bits necessary is equal to the $\log_2$ of the number of cache sets. Let n represent the

number of cache sets, so

$$n = C_s/(C_a C_l).$$

Then the number of bits equals $\log_2(n)$. To use n, a function must have a cache configuration as input.

We may determine which cache set element $v[i]$ belongs to by calculating the element mod the number of cache sets. Hence $v[i]$ belongs to cache set $v[i] \bmod n$. We let t indicate the cache set that $v[i]$ belongs to, i.e.

$$t = v[i] \bmod n.$$

To use t, a function must have a string named v, an index i into v where $v[i]$ is in the desired set, and a cache configuration as input.

Example 5.3. *Let* $v = 194, 35, 193, 290, 57, 259, 66, 310, 118, 222, 57, 158, 194, 130, 150, 345, 194, 246, 310, 67, 66, 57, 162, 54, 193, 67, 89, 98, 226, 257$. *Note that* $|v| = 30$. *Let* C *be defined such that* $C_s = 1024$, $C_l = 8$, *and* $C_a = 4$. *We wish to compute* n *and* t *for index* 28.

By the definitions above,

$$\begin{aligned} n &= C_s/(C_a C_l) \\ &= 1024/(4 * 8) \\ &= 32 \end{aligned}$$

and

$$\begin{aligned} t &= v[28] \bmod n \\ &= 98 \bmod 32 \\ &= 2. \end{aligned}$$

Now we define the intermediary function $p_1(v, i, k, C)$ to return 1 if $v[k]$ and $v[i]$ are in the same cache set and 0 otherwise. Formally,

$$p_1(v, i, k, C) = \begin{cases} 1 & \text{if } t = v[k] \bmod n, \\ 0 & \text{otherwise,} \end{cases}$$

where $1 \leq i \leq |v|$ and $1 \leq k \leq |v|$.

Example 5.4. *Let v and C be as defined in Example 5.3. We wish to compute $p_1(v, i, k, C)$ where $i = 28$ and $k = 1 \ldots 30$.*

$$
\begin{array}{lll}
p_1(v,28,1,C) = 1, & p_1(v,28,11,C) = 0, & p_1(v,28,21,C) = 1, \\
p_1(v,28,2,C) = 0, & p_1(v,28,12,C) = 0, & p_1(v,28,22,C) = 0, \\
p_1(v,28,3,C) = 0, & p_1(v,28,13,C) = 1, & p_1(v,28,23,C) = 1, \\
p_1(v,28,4,C) = 1, & p_1(v,28,14,C) = 1, & p_1(v,28,24,C) = 0, \\
p_1(v,28,5,C) = 0, & p_1(v,28,15,C) = 0, & p_1(v,28,25,C) = 0, \\
p_1(v,28,6,C) = 0, & p_1(v,28,16,C) = 0, & p_1(v,28,26,C) = 0, \\
p_1(v,28,7,C) = 1, & p_1(v,28,17,C) = 1, & p_1(v,28,27,C) = 0, \\
p_1(v,28,8,C) = 0, & p_1(v,28,18,C) = 0, & p_1(v,28,28,C) = 1, \\
p_1(v,28,9,C) = 0, & p_1(v,28,19,C) = 0, & p_1(v,28,29,C) = 1, \\
p_1(v,28,10,C) = 0, & p_1(v,28,20,C) = 0, & p_1(v,28,30,C) = 0.
\end{array}
$$

Let *count* indicate the number of elements of v earlier than $v[k]$ that are in the same cache set as $v[i]$. Formally,

$$count(v, i, k, C) = \sum_{j=1}^{k} p_1(v, i, j, C),$$

where $1 \leq i \leq |v|$ and $1 \leq k \leq |v|$. Notice that if we wish to determine how many elements of v are in the same cache set as $v[i]$ and earlier than i, we compute $count(v, i, i-1, C)$.

Example 5.5. *Let v and C be as defined in Example 5.3. We wish to compute $count(v, i, k, C)$ where $i = 28$ and $k = 1 \ldots 30$.*

$$
\begin{array}{lll}
count(v,28,1,C) = 1, & count(v,28,11,C) = 3, & count(v,28,21,C) = 7, \\
count(v,28,2,C) = 1, & count(v,28,12,C) = 3, & count(v,28,22,C) = 7, \\
count(v,28,3,C) = 1, & count(v,28,13,C) = 4, & count(v,28,23,C) = 8, \\
count(v,28,4,C) = 2, & count(v,28,14,C) = 5, & count(v,28,24,C) = 8, \\
count(v,28,5,C) = 2, & count(v,28,15,C) = 5, & count(v,28,25,C) = 8, \\
count(v,28,6,C) = 2, & count(v,28,16,C) = 5, & count(v,28,26,C) = 8, \\
count(v,28,7,C) = 3, & count(v,28,17,C) = 6, & count(v,28,27,C) = 8, \\
count(v,28,8,C) = 3, & count(v,28,18,C) = 6, & count(v,28,28,C) = 9, \\
count(v,28,9,C) = 3, & count(v,28,19,C) = 6, & count(v,28,29,C) = 10, \\
count(v,28,10,C) = 3, & count(v,28,20,C) = 6, & count(v,28,30,C) = 10.
\end{array}
$$

Now we define another intermediary function, $p_2(v, i, k, C)$, that returns the index into v of the kth instance of an element in the same set as $v[i]$:

$$p_2(v, i, k, C) = j \text{ where } p_1(v, i, j, C) = 1 \wedge \mathit{count}(v, i, j, C) = k,$$

where $1 \leq i \leq |v|$ and $1 \leq k \leq \mathit{count}(v, i, |v|, C)$. Requiring $p_1(v, i, j, C) = 1$ means that $v[j]$ must be in the same cache set as $v[i]$. Requiring $\mathit{count}(v, i, j, C) = k$ means that just after $v[j]$ we have seen k string elements of v in the same cache set as $v[i]$.

Example 5.6. *Let v and C be as defined in Example 5.3. We wish to compute $p_2(v, i, k, C)$ where $i = 28$ and $k = 1 \ldots 10$. We do this using the values for $p_1(v, 28, k, C)$ from Example 5.4 and the values for $\mathit{count}(v, 28, k, C)$ from Example 5.5:*

$p_2(v, 28, 1, C) = 1$ *because* $p_1(v, 28, 1, C) = 1$ *and* $\mathit{count}(v, 28, 1, C) = 1$,
$p_2(v, 28, 2, C) = 4$ *because* $p_1(v, 28, 4, C) = 1$ *and* $\mathit{count}(v, 28, 4, C) = 2$,
$p_2(v, 28, 3, C) = 7$ *because* $p_1(v, 28, 7, C) = 1$ *and* $\mathit{count}(v, 28, 7, C) = 3$,
$p_2(v, 28, 4, C) = 13$ *because* $p_1(v, 28, 13, C) = 1$ *and* $\mathit{count}(v, 28, 13, C) = 4$,
$p_2(v, 28, 5, C) = 14$ *because* $p_1(v, 28, 14, C) = 1$ *and* $\mathit{count}(v, 28, 14, C) = 5$,
$p_2(v, 28, 6, C) = 17$ *because* $p_1(v, 28, 17, C) = 1$ *and* $\mathit{count}(v, 28, 17, C) = 6$,
$p_2(v, 28, 7, C) = 21$ *because* $p_1(v, 28, 21, C) = 1$ *and* $\mathit{count}(v, 28, 21, C) = 7$,
$p_2(v, 28, 8, C) = 23$ *because* $p_1(v, 28, 23, C) = 1$ *and* $\mathit{count}(v, 28, 23, C) = 8$,
$p_2(v, 28, 9, C) = 28$ *because* $p_1(v, 28, 28, C) = 1$ *and* $\mathit{count}(v, 28, 28, C) = 9$,
$p_2(v, 28, 10, C) = 29$ *because* $p_1(v, 28, 29, C) = 1$ *and* $\mathit{count}(v, 28, 29, C) = 10$.

Now we can write that $\mathit{cacheset}(v, i, C) = w$ where $w[k] = v[p_2(v, i, k, C)]$, $|w| = \mathit{count}(v, i, |v|, C)$, and $1 \leq k \leq |w|$.

Example 5.7. *Let v and C be as defined in Example 5.3. We wish to compute $\mathit{cacheset}(v, 28, C)$. We do this using the values for $p_2(v, 28, k, C)$ from Example 5.6. If $w = \mathit{cacheset}(v, 28, C)$, then*

$w[1] = v[p_2(v, 28, 1, C)] = v[1] = 194$, $w[6] = v[p_2(v, 28, 6, C)] = v[17] = 194$,
$w[2] = v[p_2(v, 28, 2, C)] = v[4] = 290$, $w[7] = v[p_2(v, 28, 7, C)] = v[21] = 66$,
$w[3] = v[p_2(v, 28, 3, C)] = v[7] = 66$, $w[8] = v[p_2(v, 28, 8, C)] = v[23] = 162$,
$w[4] = v[p_2(v, 28, 4, C)] = v[13] = 194$, $w[9] = v[p_2(v, 28, 9, C)] = v[28] = 98$,
$w[5] = v[p_2(v, 28, 5, C)] = v[14] = 130$, $w[10] = v[p_2(v, 28, 10, C)] = v[29] = 226$.

Finally, we write the desired results: $\mathit{cacheset}(v, 28, C) = 194, 290, 66, 194, 130, 194, 66, 162, 98, 226$.

Now, given $C_a \neq C_s/C_l$ and $C_l = g$,

$$miss(C, v, i, g) = miss(C', cacheset(v, i, C), count(v, i, i, C), g) \tag{5.3}$$

where $C'_s = C_a C_l$, $C'_l = C_l$, and $C'_a = C_a$. Our new cache configuration, C', is equivalent to a fully associative cache that is the same size as the cache set size of C since $C'_a = C_a = (C_a C_l)/C_l = C'_s/C'_l$. Because our new cache is fully associative, we may now use Equation 5.1.

5.1.4 Case Four

We now consider the case where the cache is set associative and the line size does not equal the string granularity. As before, if we can transform the cache and string into an instance of a case we have already defined, we can use an earlier equation. We have three choices. First, we can adjust the granularity and remove all the elements of the string not in the given cache set and then use Equation 5.1. Second, we can remove all the elements of the string not in the given cache set and then use Equation 5.2. Or, third, we can adjust the granularity and use Equation 5.3.

The first option involves the most work. The second option involves performing operations similar to what we did in Case Three, only the computations are more complex because the granularity and line size do not match. Therefore, it seems that the third option would be the simplest choice. When presented with a Case Four situation, we adjust the granularity and reduce the problem to a Case Three situation. We can do this by using Equation 5.2, which transforms Case Four into Case Three. To summarize the whole process for this case, when a cache is set associative and does not have matching line size and granularity, we use Equation 5.2 to transform Case Four into an instance of Case Three and then use Equation 5.3 to transform into Case One.

We may now combine Equations 5.1, 5.2, and 5.3 and write the definition of *miss* as one equation that covers all four cache cases:

$$
\begin{aligned}
&miss(C, v, i, g) \\
&\quad = \begin{cases}
miss(C, zoom(v, h), i, g') & \text{if } (C_l \neq g), \\
miss(C', cacheset(v, i, C), & \\
\qquad count(v, i, i, C), g) & \text{if } (C_l = g) \wedge (C_a \neq C_s/C_l), \\
\qquad 0 & \text{if } (C_a = C_s/C_l) \wedge (C_l = g) \wedge \\
& \qquad (\exists d)\big[(0, d) \in \ell(v[i]) \wedge d \leq C_a\big], \\
\qquad 1 & \text{otherwise,}
\end{cases}
\end{aligned}
\tag{5.4}
$$

where $h = \log_2(C_l/g)$, $g' = C_l$, $C'_s = C_a C_l$, $C'_l = C_l$, and $C'_a = C_a$. The first line of Equation 5.4 represents Case Two and the first step of Case Four. The second line of Equation 5.4 represents Case Three and the second step of Case Four. The third and fourth lines of Equation 5.4 represent Case One.

5.1.5 Miss Rate

We can now mathematically write the miss rate for a given trace and cache combination. To do this, we count the number of misses in the trace and divide by the length of the trace. We let $Miss(v, C)$ indicate the miss rate of the string v in the cache C. Formally,

$$
Miss(v, C) = \frac{\sum_{i=1}^{|v|} miss(C, v, i, g_v)}{|v|}, \tag{5.5}
$$

where g_v is the granularity of the string v.

Now that we have formally described caches and written an equation that determines if a given string element is a miss in the given cache, we are ready to tie locality and cache performance together for particular strings.

5.2 Miss and Miss Rate Surfaces

In this section, we present two new surfaces that help us understand, from a locality perspective, how a given workload functions in a particular cache. This brings us closer to describing caches in terms of locality for all workloads in general.

5.2.1 Miss Surface

We first introduce the **miss surface**. A miss surface is based on a bag of stride/delay relationships called the **miss data**. The miss data is converted into a miss surface using the visualization steps described in Section 2.4. Recall from Equation 2.5 that the locality bag for a given string is the additive union of the locality data for each element of the string. In contrast, the miss data for a given string and cache configuration is the additive union of the locality data for each element of the string that causes a miss in the given cache. We let $m(v, C)$ indicate the **miss data** for string v in cache C. We compute the miss data as follows:

$$m(v, C) = \biguplus_{i=1}^{|v|} \big[\ell(v[i]) * miss(C, v, i, g)\big]. \tag{5.6}$$

Notice that in this equation we are multiplying a bag (i.e. $\ell(v[i])$) with an integer (i.e. $miss(C, v, i, g)$). In this case, the integer is either a 0 or a 1. So we formally declare what occurs in these two cases. Let B be a bag. Then $B * 0 = \emptyset$ and $B * 1 = B$.

Example 5.8. *Let $v_1 = 2, 7, 5, 10, 5, 2, 8$ as in Example 2.1. Let C be a cache configuration where $C_s = 32$ bytes, $C_l = 8$ bytes, and $C_a = 4$. Note that C is fully associative. Recall from Example 2.14 that*

$\ell(v_1[1]) = \emptyset$,
$\ell(v_1[2]) = \{(5, 1)\}$,
$\ell(v_1[3]) = \{(-2, 1), (3, 2)\}$,

$\ell(v_1[4]) = \{(5,1),(3,2),(8,3)\}$,
$\ell(v_1[5]) = \{(-5,1),(0,2)\}$,
$\ell(v_1[6]) = \{(-3,1),(-8,2),(-5,3),(0,4)\}$, *and*
$\ell(v_1[7]) = \{(6,1),(3,2),(-2,3),(1,4)\}$.

Now we compute $miss(C, v, i, g)$ *for each element of* v_1*:*

$miss(C, v_1, 1, g) = 1$,
$miss(C, v_1, 2, g) = 1$,
$miss(C, v_1, 3, g) = 1$,
$miss(C, v_1, 4, g) = 1$,
$miss(C, v_1, 5, g) = 0$,
$miss(C, v_1, 6, g) = 0$, *and*
$miss(C, v_1, 7, g) = 1$.

Using Equation 5.6, we calculate that $m(v_1, C) = \ell(v_1[1]) \uplus \ell(v_1[2]) \uplus \ell(v_1[3]) \uplus \ell(v_1[4]) \uplus \ell(v_1[7]) = \{(-2,1), (5,1), (5,1), (6,1), (3,2), (3,2), (3,2), (-2,3), (8,3), (1,4)\}$.

Recall that in Section 2.4 we made the visualization functions use stride/delay bags as input, rather than locality bags. Since the miss bag is a stride/delay bag, we are now allowed to use miss bags as inputs without changing the functions to create the miss surface. Hence $h(m(v, C))$ is the miss histogram, $H(m(v, C))$ is the binned miss histogram, and $\mathcal{S}(m(v, C), v)$ is the miss surface. Let $\mathcal{M}$ be the miss surface fuction that takes a string and a cache configuration as input and returns a miss surface. Formally, $\mathcal{M}(v, C) = \mathcal{S}(m(v, C), v)$ and $\mathcal{M}(v, C, a, b) = \mathcal{S}(m(v, C), v, a, b)$.

Figure 5.1 shows an example miss surface. Specifically, it shows the instruction trace of *twolf* filtered by a 1 Kbyte direct mapped cache with an 8-byte line size. As with the locality surface, we show two views of the miss surface. We have scaled the maximum value of the graph to be equivalent to the maximum value on the locality surface for the instruction trace of *twolf*, last seen in Figure 4.1 and reshown here in Figure 5.2. Note that the grayscale color map in Figure 5.1 does not match the one in Figure 5.2 but is relative to the maximum height in Figure 5.1. This allows us to better determine the relative heights of the features on the miss surface.

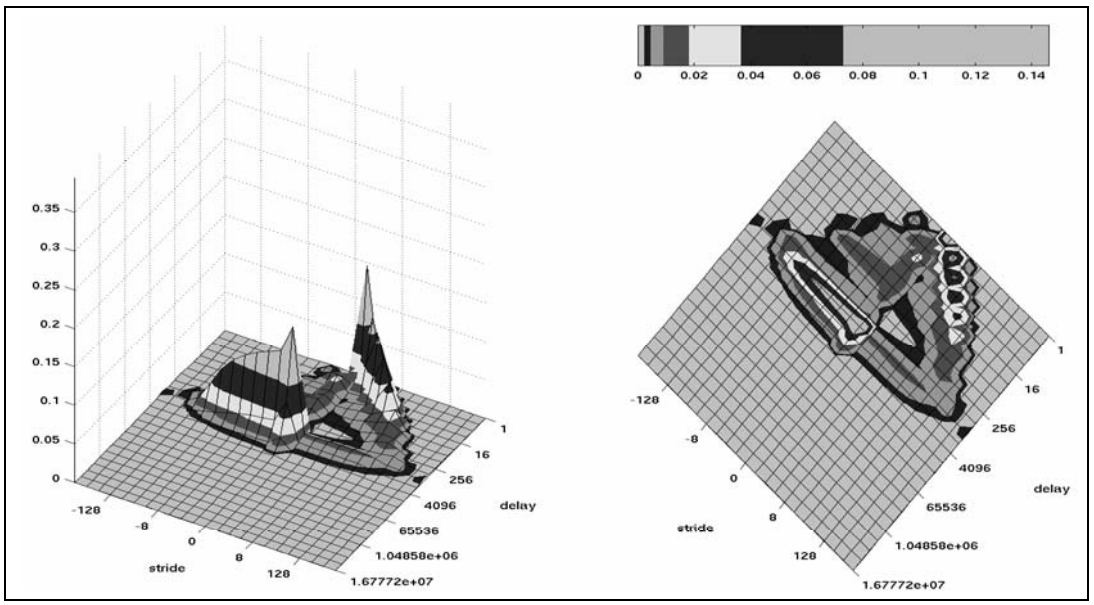


Figure 5.1: The miss surface for the instruction trace of *twolf* filtered by a 1 Kbyte direct mapped cache with an 8-byte line size.

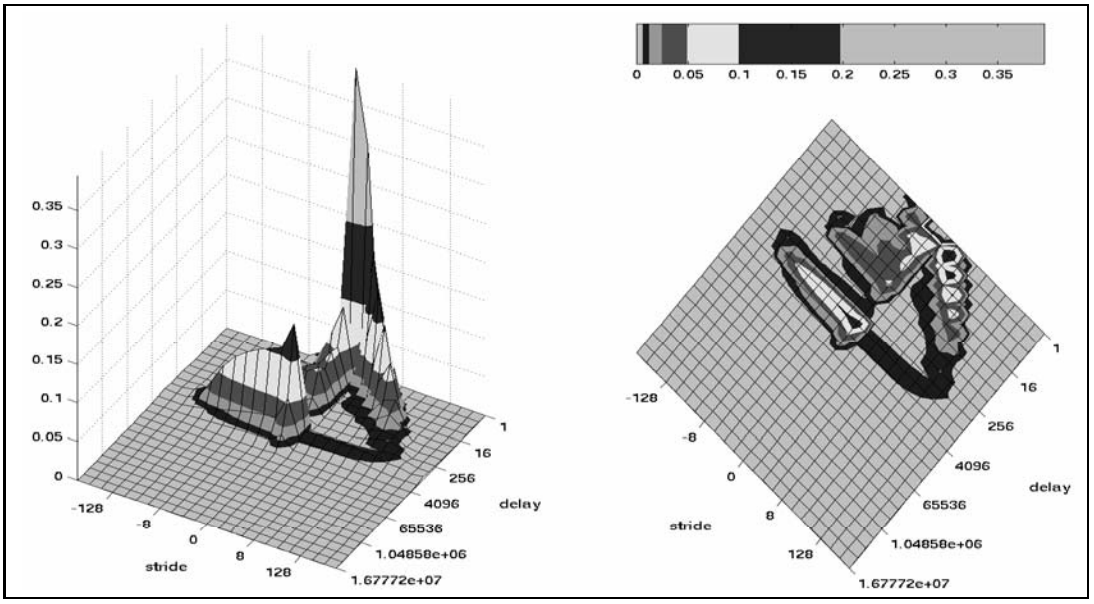


Figure 5.2: The locality surface for the instruction trace of *twolf*.

The size of the cache used for the miss surface in Figure 5.1 is 1 Kbytes, or 128 words. When comparing the locality surface for the same trace in Figure 5.2 with the miss surface in Figure 5.1, note that the height of all the features is dramatically reduced where the delay is less than 128 but hardly reduced at all where the delay is greater than 128. For example, consider the loop at 1 Kwords. The exact value of the bin where the loop crosses the temporal axis is 0.145242 on the locality surface and 0.144540 on the miss surface. Hardly a reduction at all.

Recall that the miss surface shows us the locality of the elements of the trace that missed in the cache. Another thing we may notice on the miss surface in Figure 5.1 is the absence of a temporal spike. Specifically, $\mathcal{M}(v, C, 0, 1) = 0$ for any trace and cache configuration. Even the smallest cache has a hit any time a value is immediately repeated.

5.2.2 Miss Rate Surface

The miss surface for a particular workload and cache can help us understand how the given workload function in the given cache, but we find ourselves constantly referring back to the locality surface of the workload. To remove this necessity, we define the **miss rate surface**. Each bin on this surface is equivalent to the corresponding bin on the miss surface divided by the corresponding bin on the locality surface. Before we formally define the miss rate surface, we define division of surfaces.

Given two surfaces, T_1 and T_2, we may divide them, bin by bin, to obtain a third surface, T_3. Formally, if $T_3 = T_1/T_2$, then $T_3(a, b) = T_1(a, b)/T_2(a, b)$ for any valid bin (a, b).

Let $\mathcal{R}$ be the miss rate function that takes a string and a cache configuration, computes the miss surface and locality surface, divides them bin by bin, and re-

turns the miss rate surface. Formally, $\mathcal{R}(v, C) = \mathcal{M}(v, C)/\mathcal{L}(v)$ and $\mathcal{R}(v, C, a, b) = \mathcal{M}(v, C, a, b)/\mathcal{L}(v, a, b)$. Let us follow the math to see what drops out as we perform these calculations:

$$\mathcal{R}(v, C, a, b) = \frac{\mathcal{M}(v, C, a, b)}{\mathcal{L}(v, a, b)}$$

$$= \frac{\mathcal{S}(m(v, C), v)}{\mathcal{S}(\ell(v), v)}$$

$$= \begin{cases} \frac{\frac{H(m(v,C),a,b)}{(|v|-1)\cdot(2^{|a|-2})}}{\frac{H(\ell(v),a,b)}{(|v|-1)\cdot(2^{|a|-2})}} & \text{if } |a| > 1 \\ \\ \frac{\frac{H(m(v,C),a,b)}{|v|-1}}{\frac{H(\ell(v),a,b)}{|v|-1}} & \text{if } |a| \leq 1 \end{cases}$$

$$= \frac{H(m(v, C), a, b)}{H(\ell(v), a, b)}. \tag{5.7}$$

We remove the problem of dividing by zero by declaring that when $H(\ell(v), a, b) = 0$, we define $\mathcal{R}(v, C, a, b) = 0$. As the miss rate surface is computed, the dividing done by the Surface Function $\mathcal{S}$ drops out. Dividing the miss surface by the locality surface is equivalent to dividing the binned miss histogram by the binned locality histogram.

Figure 5.3 shows the miss rate surface for the instruction trace of *twolf* and a 1 Kbyte direct mapped cache with an 8-byte line size. By definition, this is the miss surface in Figure 5.1 divided by the locality surface in Figure 5.2. As with the locality and miss surfaces, we again show two views of the miss rate surface. Unlike the locality and miss surfaces, we use a different angle. Since miss rate surfaces tend to have lower values close to the origin and higher values away from the origin, it is

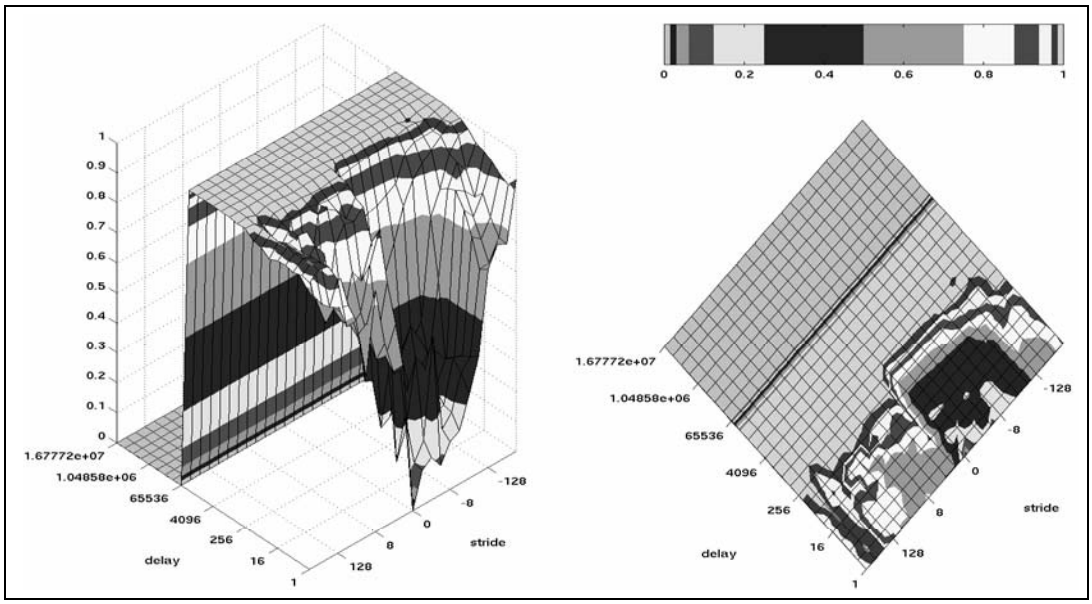


Figure 5.3: The miss rate surface for the instruction trace of *twolf* filtered by a 1 Kbyte direct mapped cache with an 8-byte line size. Essentially, this is the miss surface in Figure 5.1 divided by the locality surface in Figure 5.2.

of more value to have the origin in the foreground. Each bin now shows the average miss rate for all trace elements that have the given stride and delay.

Let T be the miss rate surface in Figure 5.3. The bin labeled $(3, 2)$ refers to the stride/delay relationships where the stride is 3 or 4 and the delay is 2. $T(3, 2) = 0.538$ means that 53.8% of all stride/delay relationships assigned to the $(3, 2)$ bin belong to the locality data of a reference in the instructions of *twolf* that was a miss in the given cache.

Notice that bins near the origin have low miss rates. In general, the further the bin is from the origin, the worse the miss rate. Note that the miss rate is zero at the $(0, 1)$ bin. As mentioned before, any cache has a hit when a reference is immediately repeated.

This miss rate surface tells us interesting things about how the instruction trace of *twolf* functions in a 1 Kbyte direct mapped cache with 8-byte lines, but little else.

We could conjecture that other workloads are likely to have similar miss rates for each stride/delay relationship with this cache, but intuition tells us this is highly suspect. Notice how the miss rate is not symmetric across the temporal axis. Logic tells us that traditional caches are not likely to favor negative strides over positive strides. Instead, we guess that the disparity is due to a lack of sufficient data at a number of bins. If the instruction trace of *twolf* has only a few stride/delay relationships that contribute to a given bin, it is hard to believe that the miss rate for that bin is representative of all traces in that cache.

The miss rate surface for a particular workload and cache can help us understand how the given workload functions in the given cache, but little other information. In general, the miss rate surface is more difficult to create than it is useful. It involves both cache simulation for the desired cache and computation of the locality data for the given workload, and yields information only about the specific workload and cache combination. However, if we could create a miss rate surface that is independent of any particular workload, we would have a surface that visualizes a cache in terms of locality.

5.3 Cache Characterization Surface

We have said that we wish to characterize caches in terms of locality. To do this, we wish to know the miss rate associated with each given stride/delay combination. If we can create a miss rate surface where the input trace is independent of any workload, we would have the desired **cache characterization surface**. We first discuss the notation for such a creation and then investigate the practicality of creating a true cache characterization surface. Finally, we display a number of cache characterization surfaces created using the method we chose.

Let $\mathcal{C}$ be a the cache characterization function that takes a cache configuration. Formally, $\mathcal{C}(C) = \mathcal{R}(v, C)$ and $\mathcal{C}(C, a, b) = \mathcal{R}(v, C, a, b)$ where v is a string that is independent of any workload. The question is, what v would fit the requirements?

5.3.1 Picking v

One idea is to use an infinite number of strings for v. For every workload that exists, create the miss surface for the given cache and the locality surface. Add, bin by bin, all the miss surfaces together into a composite miss surface. Similarly, add, bin by bin, all the locality surfaces together into a composite locality surface. Divide the composite miss surface by the composite locality surface to get a cache characterization surface.

There are several drawbacks to this plan. First, it is obvious to anyone who has ever considered benchmark selection that there is no list of all possible workloads, and the field is hotly debated as to which workloads are best at representing the entire workspace. In addition, it is well understood that current workloads cannot pretend to be representative of future workloads. Current workloads may seldom use the bin $(11, 1)$, but future workloads may use it frequently. The few times it is used by current workloads may not be representative of all its possible uses in a given cache.

Another idea for creating a cache characterization surface for a given cache, which we used here, is to submit random data to a miss rate surface program. This avoids the debate as to which workloads are representative and whether current workloads are adequate to predict future workloads. The random stream should be long enough that all stride/delay relationships are represented.

The random data should not only contain every stride/delay combination of

interest but should contain every combination a number of times. As the reader will see when examining actual cache characterization surfaces, a given stride/delay relationship may sometimes be a hit and sometimes a miss in a given cache. We wish to determine, on average, the percentage of times each stride/delay relationship is a hit versus a miss. To do this, we need each stride/delay relationship to occur a number of times.

The primary drawback to using a random trace as input is how time intensive it is. The locality program is stack based and therefore is $O(n^2)$. (This is discussed more in Chapter 9.) With random data, the stack grows large quickly and the compute time grows with the square of the stack size. One way to minimize this is to determine the maximum desired delay. For example, if we determine that a maximum delay of one million is adequate, there is no need to let the stack grow larger than that. The smaller the maximum delay, the faster the computation. However, the larger the maximum delay desired, the more applicable the resulting cache characterization surface is to any workload. Determing what is the optimal maximum delay is a balance between how general the cache characterization surface should be and available processing power/time.

A method for improving the speed of computing the cache characterization surface data for a random input is to notice that bins with large delay and large stride absolute value have a lot of possible stride/delay relationships that fall in the bin. In addition, stride/delay relationships that fall in the same bin tend to have the same response in a given cache. So we really do not need the occurrences of each stride/delay relationship to be equal, we need the number that fall in each bin to be roughly equal. This means we need more stride/delay relationships that are small, meaning small absolute value of both the stride and delay, because the bins with small stride/delay relationships have fewer stride/delay relationships per bin than

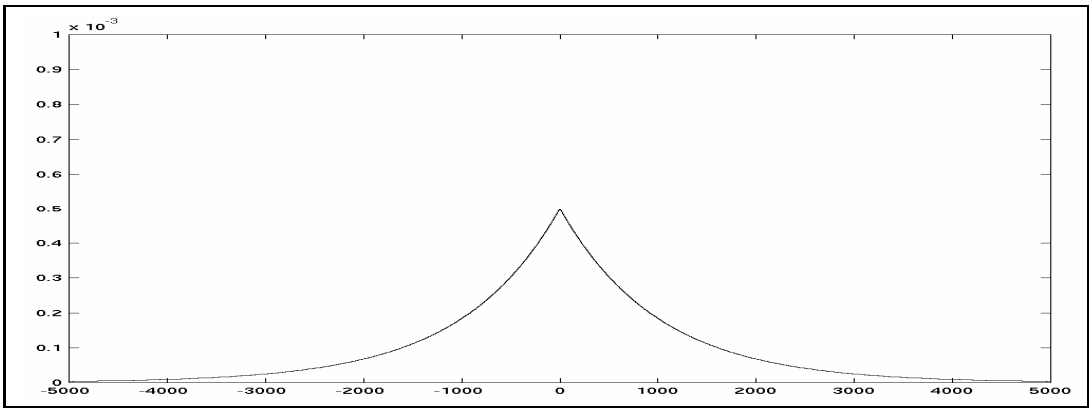


Figure 5.4: The Laplacian distribution with parameters $\mu = 0$ and $\lambda = 1000$.

the large ones.

We create this preference by using a different random distribution than the uniform distribution. The Laplace, or double exponential, distribution is shown in Figure 5.4. This distribution takes two parameters, μ and λ. The parameter μ determines the mean and λ determines how steep the dropoff is and hence the spread. A larger λ causes the central peak to be lower and the horizontal spread to increase. A smaller λ raises the central peak and decreases the horizontal spread. By carefully picking λ, we can find a balance point where bins with many entries, such as $(-11, 8)$ with $32,768$ entries, have approximately the same number of stride/delay relationships as bins with few entries, such as $(1, 2)$ with one entry.

Another point that reduces the time to compute miss surface/locality data for a random input string involves determining how long to make the random string. We first notice that after a point the value in each bin does not change. For example, after one stride/delay relationship in the $\ell(0, 1)$ bin, which always is a hit, the value never changes. We have noticed, for the caches we computed for this dissertation, that bins tend to hold constant after about 100 entries. Therefore, we attempt to cut the length of the random stream at the point where every bin has at least 100

entries.

Another method to improve the time to compute random locality data is to improve the speed of the program computation itself. A number of researchers have developed improvements to stack based programs [11, 80], but these are not useful when every level of the stack needs to be accessed each time, as is needed to compute the locality data. Our solution is a parallel algorithm, which we discuss in Chapter 9. In addition, we can compute a number of cache characterization surfaces at the same time. We use the same stream of random numbers for each cache characterization surface, meaning we can use one stack for all the surfaces. The only difference in the computations is the cache simulation and hit/miss results.

We are willing to spend a lot of time processing the cache characterization surfaces partly because we can compute a number of them at the same time. The biggest reason we are willing to spend weeks on the computation is that each surface need only be computed once. The only time new cache characterization surfaces need to computed is if we wish one for a new cache or if we wish to have a larger maximum delay.

5.3.2 Actual Cache Characterization Surfaces

We have created a number of cache characterization surfaces using a stream of random references generated by the Laplacian distribution. We selected a maximum delay goal of 8 Mwords, or 64 Mbytes. This meant that our random stream needed between $4,194,305$ and $8,388,608$ unique numbers. In addition, we wished to have a roughly equivalent number of entries in each bin of the binned histogram. Through trial and error, we created a trace with $500,000,000$, or 500 million, entries. Each value was generated by the Laplacian distribution with $\lambda = 600,000$. The trace had

7,931,968 unique references and roughly met the other requirements.

We created cache characterization surfaces using this random trace for 24 different caches with varying cache size, line size, and associativity. When performing the actual computations, we used the parallel algorithm discussed in Chapter 9 and created all 24 cache characterization surfaces at the same time. The time for this computation, using all 64 processors of the parallel machine we have, was just over seven weeks.

We now show several surfaces here and discuss what we learn about how changing the cache size, the line size, and/or the associativity affects the cache's response to inputs with various locality. All cache characterization surfaces that we computed are shown in Appendix C.

First we examine how changing the cache size affects various stride/delay relationships. To do this, we hold the line size constant at 8-bytes and examine caches that are fully associative. Figure 5.5 shows a 16 Kbyte cache, Figure 5.6 shows a 128 Kbyte cache, and Figure 5.7 shows a 1 Mbyte cache.

As with all our surfaces, we display two views of the cache characterization surface. As with the miss rate surface, we display the cache characterization surface rotated so that the origin is in the foreground. Each bin on the surface indicates the percentage of references that have a stride/delay relationship in the bin and are misses in the given cache.

When creating the surface, whenever a bin is equal to zero on the locality surface, we define the equivalent bin on the cache characterization surface to be zero. This explains why the surface displays zero where the delay equals 16 Mwords. We picked the maximum delay shown on the surface to be one bin larger than the maximum delay seen on any of our cache characterization surfaces on purpose. This allows a clear indication of the color mapping and at what percentages the colors change. It

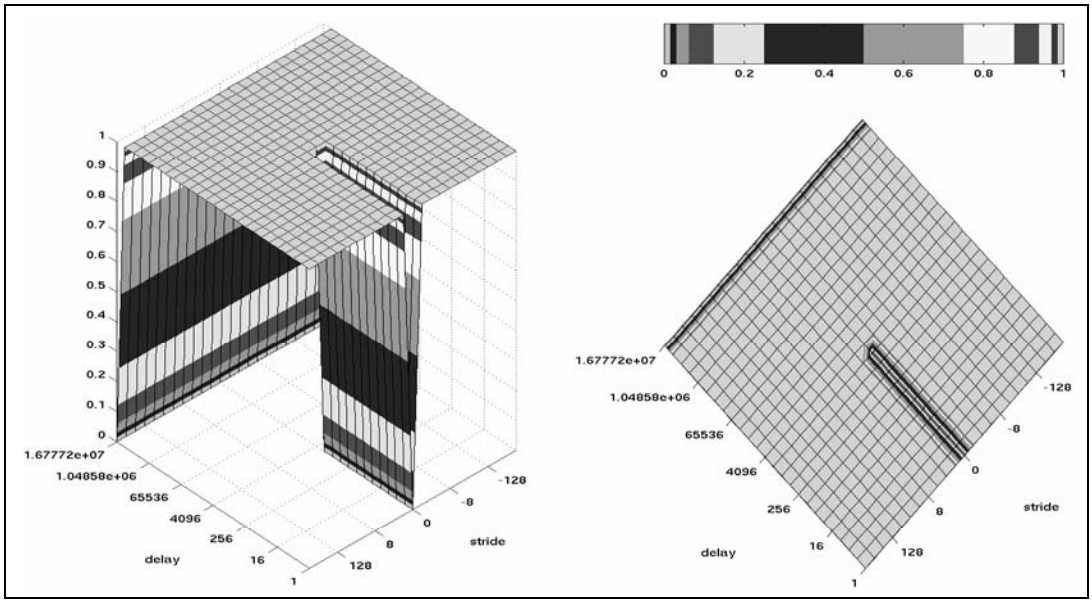


Figure 5.5: The cache characterization surface for a 16 Kbyte fully associative cache with 8-byte lines.

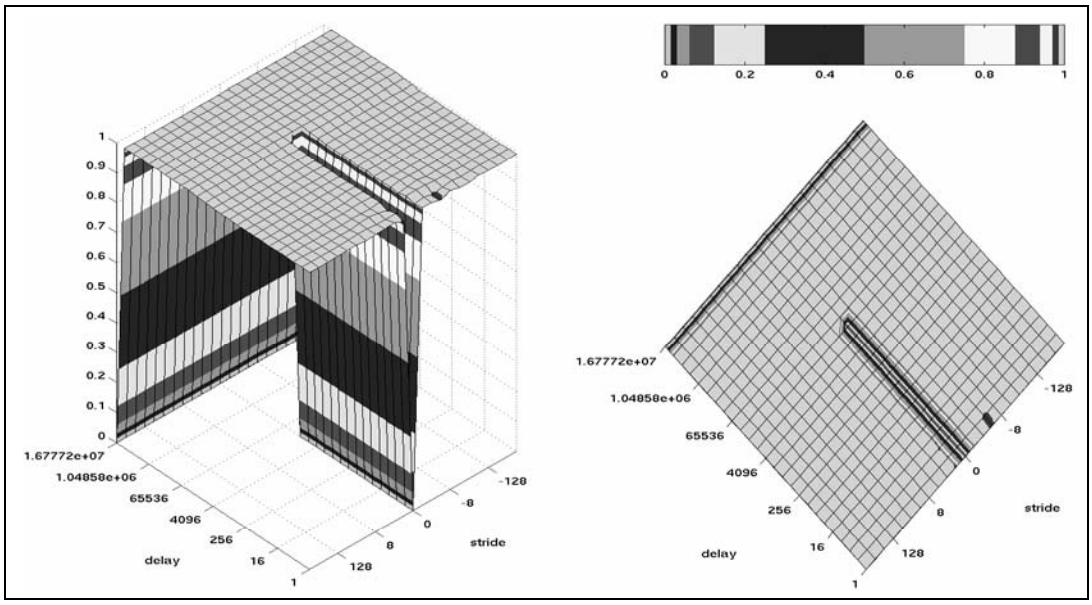


Figure 5.6: The cache characterization surface for a 128 Kbyte fully associative cache with 8-byte lines.

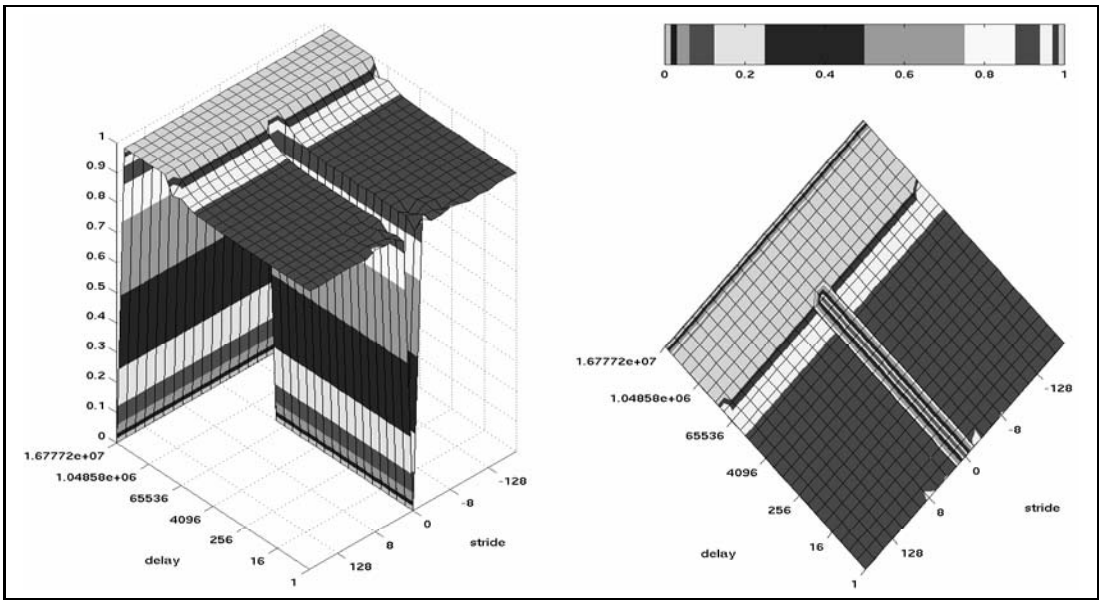


Figure 5.7: The cache characterization surface for a 1 Mbyte fully associative cache with 8-byte lines.

also visually connects the surface to the coordinates.

The first significant feature we see on the surfaces in Figures 5.5 – 5.7 is a trough down the temporal axis. We term this the **temporal trough**. At the bottom of the temporal trough the miss rate is 0%, indicating that references that have the given stride/delay relationships are always hits in the cache. Everywhere else on the cache characterization surface the miss rate is nearly 100%. For Figure 5.5, the temporal trough extends to a delay of 2 Kwords, or 16 Kbytes. For Figure 5.6, the temporal trough extends to 16 Kwords, or 128 Kbytes. And for Figure 5.7 the temporal trough extends to 128 Kwords, or 1 Mbyte. In other words, the length of the temporal trough matches the size of the given cache. More specifically, the delay at which the temporal trough ends is equal to the number of lines in the cache.

This makes logical sense given what we know of fully associative caches where the line size and granularity match. If d represents the number of lines in the given

fully associative cache, any value repeated within a delay of d from the previous reference to the value is a hit, any value repeated at a greater delay is a miss. The points on the temporal axis that are close to 100% represent the capacity misses in the cache. Remember that there are no conflict misses for fully associative caches. Note that compulsory misses are not represented on the temporal axis. Values cause compulsory misses when they have never been seen before in the trace, and values that are seen for the first time have no earlier reference to the same value to cause stride to be zero. Hence compulsory misses cannot occur on the temporal axis.

Another feature seen on some cache characterization surfaces is the overall slump across all strides as the delay decreases. This feature is evident in Figure 5.7 but not in Figures 5.5 or 5.6. This overall slump occurs when the cache size is large in comparison to the working set size of the trace input into the cache characterization program. As the cache size approaches the working set size the chance that any given reference is already in the cache increases, regardless of the locality of the given reference. Caches larger than 1 Mbyte have an overall slump on the cache characterization surface that is even lower. If cache characterization surfaces in the middle of your cache size range already exhibits an overall slump, then you know that you need an input trace with a larger working set size to make the resulting surfaces useful.

Now notice that we are able to create cache characterization surfaces for fully associative caches with 8-byte lines and cache size up to 64 Mbytes. Using our previous cache characterization program, used in [74], the maximum size we could create was 256 Kbytes. We have improved our maximum cache size by a factor of 256.

Now we examine how changing the line size affects the cache characterization surface. To do this, we hold the cache size consistent at 128 Kbytes and use only

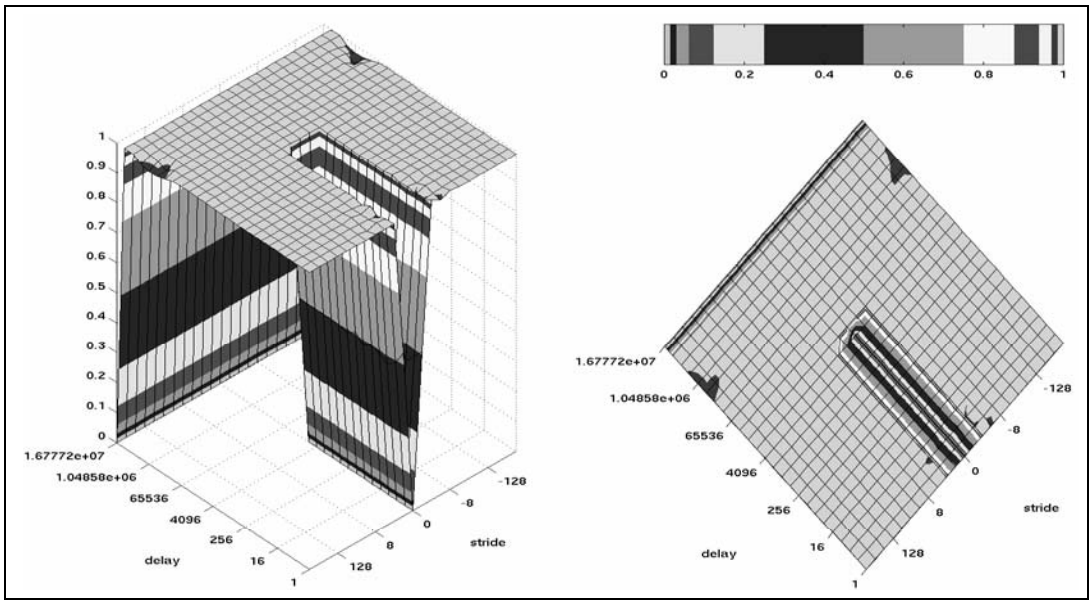


Figure 5.8: The cache characterization surface for a 128 Kbyte fully associative cache with 16-byte lines.

fully associative caches. We saw earlier, in Figure 5.6, the cache with 8-byte lines. Figure 5.8 shows a cache with 16-bytes lines, Figure 5.9 shows a cache with 32-byte lines, and Figure 5.10 shows a cache with 64-byte lines.

It should be clear from this series of figures that increasing the line size increases the width and decreases the length of the temporal trough. In fact, the width of the temporal trough is directly related to the line size.

Note that the 8-byte line cache in Figure 5.6 has miss rate nearly 100% except where stride is zero. The 16-byte line cache, in Figure 5.8, has miss rate nearly 100% except where the stride is ± 1 or zero. When the stride is ± 1, the miss rate is about 50%; where the stride is zero, the miss rate is 0%. Again, this makes logical sense. For a cache with 16-byte lines, if a value of the given string is ± 1 from the immediately previous value, there is a 50/50 chance that the new value was in the same line as the previous value and therefore is a hit.

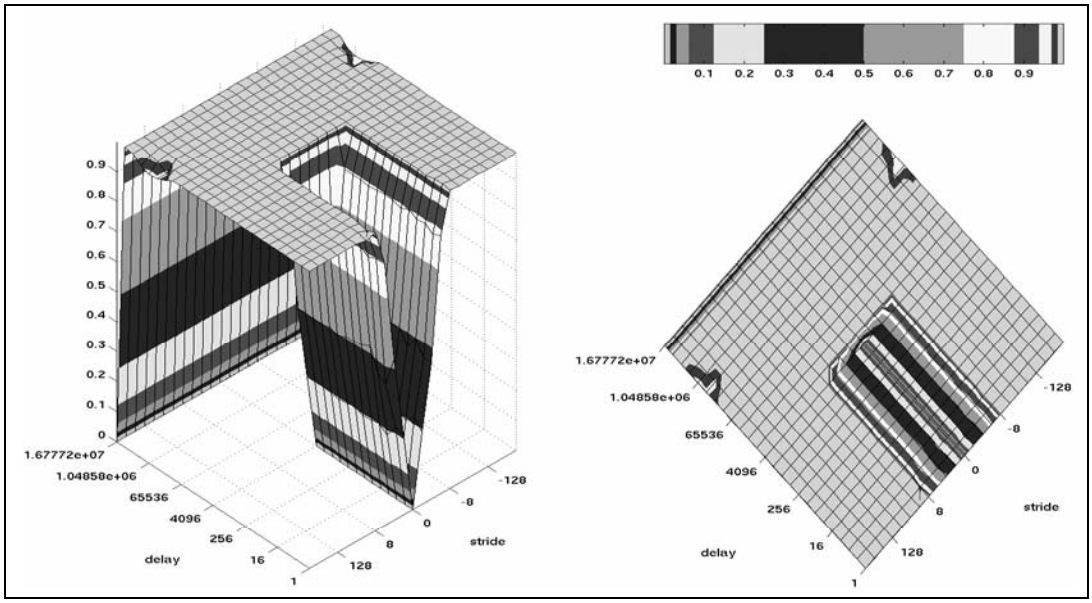


Figure 5.9: The cache characterization surface for a 128 Kbyte fully associative cache with 32-byte lines.

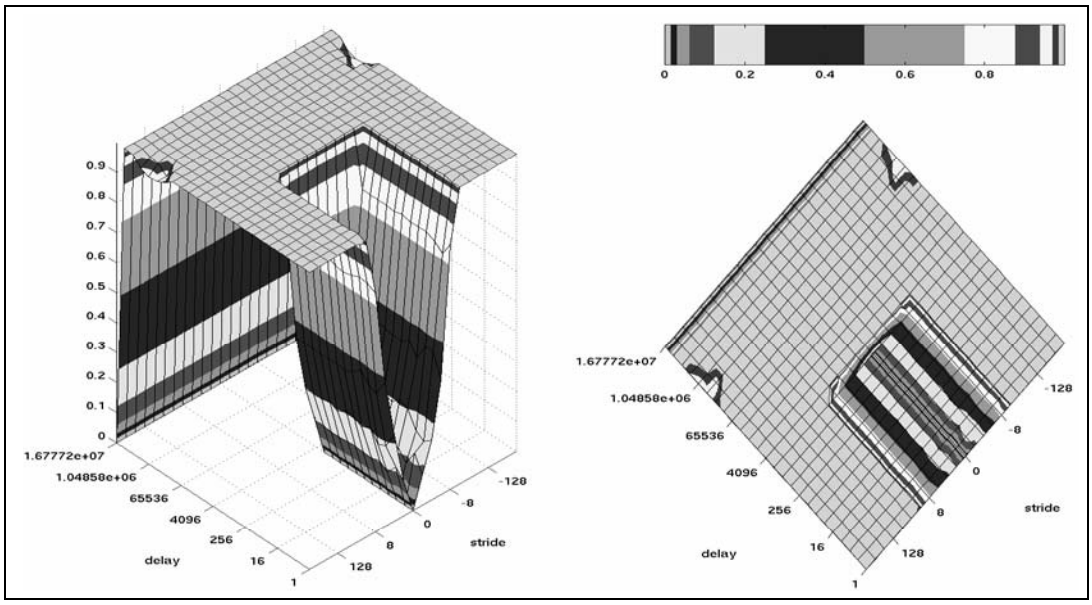
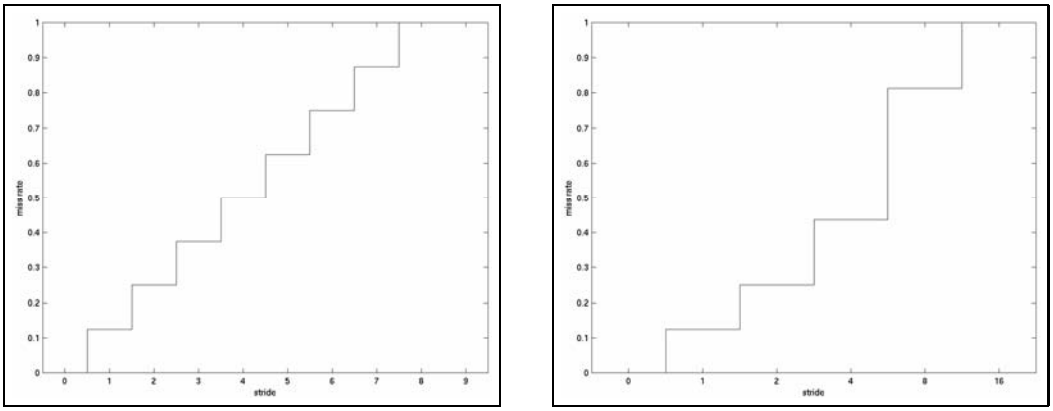


Figure 5.10: The cache characterization surface for a 128 Kbyte fully associative cache with 64-byte lines.



(a) unbinned (b) binned

Figure 5.11: The miss rate for various strides when $C_l = 8g$. Figure 5.11(a) shows the unbinned results and Figure 5.11(b) shows the binned results.

Now notice that the sides of the trough do not always rise linearly as the stride increases. This is particularly evident in Figure 5.10. This occurs because of the binning. We can best explain this visually using a line size of $8g$. Figure 5.11(a) shows the unbinned results and Figure 5.11(b) shows the binned results. A reference that is stride ± 3 from the previous reference has 37.5% chance of not being in the same line. A reference stride ± 4 has a 50% chance of being in the same line. However, when the binning occurs, strides 3 and 4 are averaged together, and the result is 43.75%. The final binned result, as seen in Figure 5.11(b), is not linear.

Another result of this binning is a non-linear relationship between the trough width and cache line size. For example, no standard cache line size results in a trough width of ± 4. Table 5.1 contains the lookup table that relates trough width to cache line size. To use this table, determine the stride at which the miss rate nearly reaches 100% along the $delay = 1$ axis. This is the trough width. For example, in Figure 5.10, the trough width is ± 16. Then lookup the trough width in Table 5.1 to obtain the result that $C_l = 8g$ for the cache in Figure 5.10.

trough width	C_l
± 1	g
± 2	$2g$
± 8	$4g$
± 16	$8g$

Table 5.1: Lookup table relating trough width to C_l.

Recall how we said that the length of the temporal trough indicates the number of lines in the cache. This becomes more significant now that we have differing line sizes. Actually, we can now write an equation for fully associative caches that tells us the cache size, given a few values from the cache characterization surface. For fully associative caches,

$$C_s = d * C_l, \tag{5.8}$$

where d is the length of the temporal trough. C_l may either be known, or obtained using Table 5.1.

We should mention that we processed the input random trace as if it has a granularity of 8 bytes. In reality, artificially generated traces may be defined to have arbitrary granularity. The declared granularity of the random stream is important because Figure 5.6, for example, is not really a cache characterization surface for a cache with 8-byte lines, it truly is a cache characterization surface for a cache where the granularity of the input trace and the line size of the cache match. Figure 5.8 is actually a cache characterization surface for a cache where the granularity of the input trace is half the size of the line size of the cache. Therefore, if a locality surface were computed for a given trace with a granularity of 32 bytes, and the cache we wished to examine had a line size of 32 bytes, we would use the cache characterization surface in Figure 5.6 rather than the one in Figure 5.9. For this reason, Table 5.1 determines the relationship between g and C_l and not the absolute

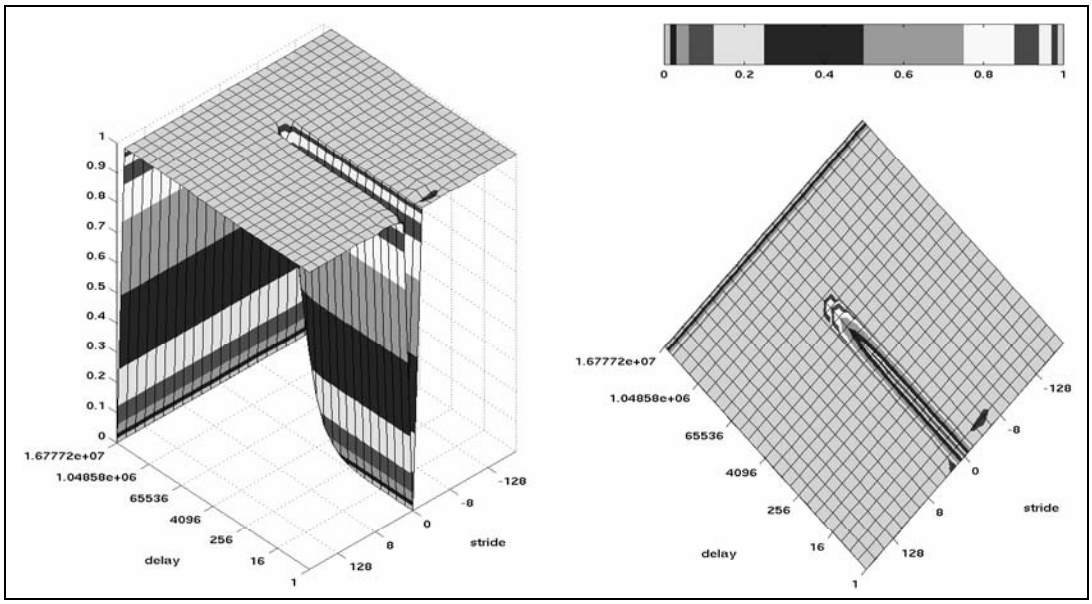


Figure 5.12: The cache characterization surface for a 128 Kbyte direct mapped cache with 8-byte lines.

cache line size.

Now we examine how changing the associativity affects the cache characterization surface. In this case, we hold the cache size constant at 128 Kbytes and the line size constant at 8-bytes and adjust the associativity. Figure 5.12 shows a direct mapped cache and Figure 5.13 shows a 4-way associative cache. The effects of associativity can be seen more clearly when we compare these surfaces with the related fully associative 128 Kbyte cache with 8-byte lines in Figure 5.6.

Notice in the fully associative cache, in Figure 5.6, the bottom of the temporal trough lies flat at a miss rate of zero. As we change the associativity to 4-way and then direct mapped, the bottom of the trough begins to curve upward as the delay increases. The point on the temporal axis where the value of the bin reaches 100% is also at a larger delay.

This curve at the back of the temporal trough occurs because there are stride/delay

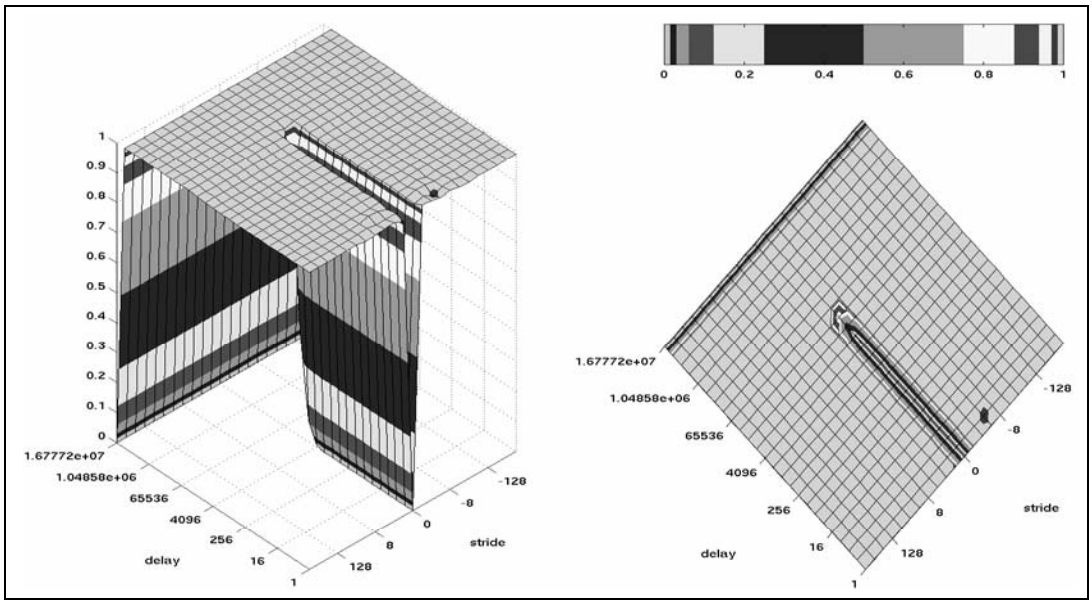


Figure 5.13: The cache characterization surface for a 128 Kbyte 4-way associative cache with 8-byte lines.

events that would normally result in hits in a fully associative cache, but have been evicted by conflict misses due to the set associativity. The lengthening of the trough results from the other half of the story; stride/delay events that would normally be evicted as capacity misses by a fully associative cache are sometimes found in the cache because of the set associativity.

Notice how the curve at the back of the temporal trough is most dramatic in Figure 5.12, which is the direct mapped cache. The slant of the back of the temporal trough for the 4-way associative cache, in Figure 5.13 is almost as sharp as for the fully associative cache in Figure 5.6. It is interesting how increasing the associativity by only two factors so closely approximates a fully associative cache. When we take into consideration that smaller associative caches are faster and easier to build [44], we can now see why small associativies are a common choice for the caches of today's processors. In essence, a 4-way or 8-way associative cache is almost equivalent to a

fully associative cache but has faster access times and is easier to build.

It is difficult to quantitatively determine the associativity of a cache based on its cache characterization surface. A practiced eye can generally differentiate between direct mapped, 4-way associative, and fully associative, however more specific determinations are difficult.

Regardless, it would be useful to determine the cache size for caches that are set associative. To do this, we may continue to use Equation 5.8 and merely change how the value for d is read on the surface. Instead of letting d be equal to where the temporal trough reaches nearly 100%, we let d be equal to the delay where the temporal trough reaches 50%. For fully associative caches, the back line of the trough reaches from 0% to 100% within one cache size, so d does not change. For caches that are set associative, the random data used to create the cache characterization surfaces has an equal chance of being found in the cache when it would not have been for a fully associative cache and of being a conflict miss, i.e. not found in the cache when it would have been for a fully associative cache. Due to this equality, the 50% point on the temporal trough is equivalent to the nearly 100% point for a fully associative temporal trough.

Using this information, we now see if we can determine the cache configuration for an unknown cache, given its cache characterization surface. Figure 5.14 shows a cache characterization surface for an unknown cache configuration. We first observe that the trough width is ± 8. We use Table 5.1 to determine that $C_l = 4g$. We now recall that our granularity is always 8 bytes in this dissertation, and therefore $C_l = 4g = 32$ bytes.

We now observe that the back of the temporal trough is significantly curved. It is as curved as any we have seen. We therefore conjecture that this cache is direct mapped. The back of the trough reaches the 50% point at a delay of 4K, hence

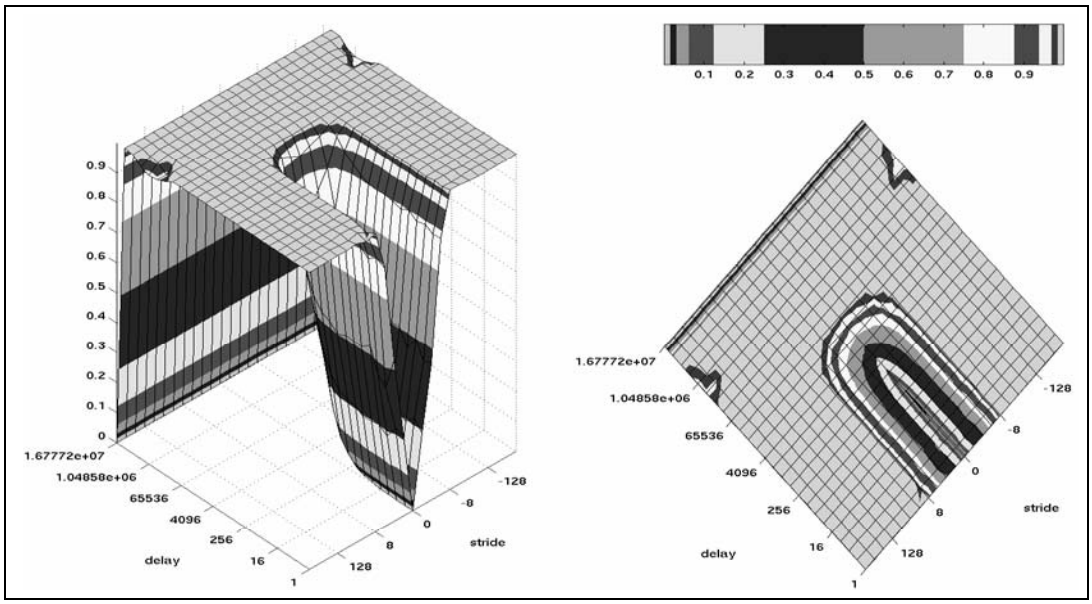


Figure 5.14: The cache characterization surface for a 128 Kbyte direct mapped cache with 32-byte lines.

$d = 4$K. We use Equation 5.8 to calculate $C_s = 4\text{K} * 32$ bytes $= 128$ Kbytes.

In fact, Figure 5.14 shows a direct mapped 128 Kbyte cache with 32-byte lines, as predicted. It is interesting to observe that lessening the associativity not only affects the back of the temporal trough but also the corners of the top of the trough. Compare the corners in Figure 5.14 with the corners in Figure 5.9. Notice how the fully associative cache has sharp, well defined corners. The direct mapped cache has rounded, soft corners.

We can get a feel for how a particular workload would function in a given cache by comparing the locality surface for the workload with the cache characterization surface for the given cache. If the locality surface has a lot of volume where the cache characterization surface has large miss rates, then performance is worse than if the volume of the locality surface is entirely in the temporal trough of the cache characterization surface. This ad hoc method confirms the qualitative assumptions

we made in Chaper 4. For example, we assumed that the amount of data on the temporal axis less than the cache size were hits, and the amount of data on the temporal axis greater than the cache size were misses. The relationship of the length of the temporal trough to cache size validates this assumption.

Further, we can now visually see how the cache line size relates to increasing stride, now known as the width of the temporal trough. We can now visually see how increasing line size helps any references that are physically close to recent references, not just references that contribute to a large sequential ridge. When we observe the softening of the back of the temporal trough for caches with less associativity, we understand why caches with larger associativities have more predictable results.

5.4 Summary

In this chapter, we related caches and locality. We developed an equation that calculates whether a specific element of a given string is a hit or miss in a given cache. Researchers can use this equation to calculate the miss rate for the entire string in the given cache. We then proved whether or not the miss rate can be computed from the locality data for given types of cache configurations.

We also presented the miss surface and the miss rate surface. Since their usefulness is limited, we decided that they are more trouble than they are worth. However, they did allow us to develop the cache characterization surface which has great value. The cache characterization surface directly related how references with varying degrees of locality function in a given cache. Another use of the cache characterization surface would be to validate the claims of alternative cache configurations. For example, the column associative cache creators claim that their cache has the same

performance as a 2-way associative cache [6]. It would be interesting to see if the cache characterization surface for each cache is equivalent to the other cache.

We now have two useful surfaces. The locality surface describes a workload in terms of locality, and the cache characterization surface describes caches in terms of locality. Perhaps there is a way to combine the two surfaces to yield quantitative cache results. Before attempting this in a practical setting, we first examine the theoretical limitations of using locality to quantitatively predict cache miss rate.

Chapter 6

Theoretical Limitations on Locality

In Chapter 4 we showed that our locality function seems to qualitatively match cache performance. It would be nice to use our locality surface to quantify cache performance, but first we ask, "Is this possible?" Is there a function such that, given a bag of locality data (or a locality surface) and a cache configuration, returns the miss rate? Using the precise definitions of locality from Chapter 2 and caches from Chapter 5, we show that we are limited in how well our locality functions predict cache performance, depending on the cache configuration.

We do this by first examining under what circumstances two strings may have the same locality data. We then examine, for various cache types, if there are situations where two strings have different miss rates but equivalent locality data.

6.1 When Two Strings Have the Same Locality Data

We now investigate when two strings have the same locality data. This is of interest when determining how valuable our locality metric is for predicting cache results. If two strings have the same locality data but access a particular cache in different ways, locality cannot precisely predict cache performance for that cache. However, if any two strings with the same locality data always access a particular cache in the same manner, our locality metric is more useful.

In this section, we examine when two strings have the same locality data, i.e. when $\ell(v) = \ell(w)$. If the locality data is the same, the locality count, the binned locality count, and the locality surface are also the same.

Property 6.1. *For two strings v and w, if $\ell(v) = \ell(w)$, then $h(\ell(v)) = h(\ell(w))$, $H(\ell(v)) = H(\ell(w))$, and $\mathcal{L}(v) = \mathcal{L}(w)$.*

It should be obvious from looking at the equations for visualizing locality data, Equations 2.6, 2.8, and 2.9, that if the input locality bags are equal then the resulting data is also equal.

We now discuss some equivalence relations on V that define equivalence classes where all the members of the given class have the same locality. An equivalence relation is well defined in set literature as "a binary relation on a set S that is reflexive, symmetric, and transitive" [35, page 256]. For a given equivalence relation ρ, an **equivalence class** of an element $v \in V$ defined by ρ is the set $[v]_\rho = \{w \in V | v \rho w\}$. We use here the definition for equivalence class found in Sudkamp [77, page 14], with similar notation. We define and discuss the following four equivalence

relations: string equality, string shift equivalence, base equivalence with equal order, and equality with respect to locality.

6.1.1 String Equality

First, we formally define what it means for two strings to be equal. Here we use the naïve definition of equality: two strings are equal if and only if they have the same length and the same elements in the same order. Formally, $v = w$ iff $\big[(|v| = |w|) \wedge \forall i(1 \leq i \leq |v|)(v[i] = w[i])\big]$ where $v \in V$, $w \in V$, and i is an integer.

Theorem 6.1. *String equality is an equivalence relation on V.*

Proof. First we note that for any string v, $v = v$ since $|v| = |v|$ and $v[i] = v[i]$ for any element of v. Hence string equality is reflexive.

Second, we show that for strings v and w, if $v = w$, then $w = v$. If $v = w$, then we know that $|v| = |w|$. Due to the symmetry of equality, we know that $|w| = |v|$. Also, since $v = w$, then for all elements of v, $v[i] = w[i]$, which means that $w[i] = v[i]$. Hence $w = v$ and string equality is symmetric.

Third, we show that for strings v, w, and x, if $v = w$ and $w = x$, then $v = x$. If $v = w$ and $w = x$, then $|v| = |w|$ and $|w| = |x|$. Since equality is transitive, we know that $|v| = |x|$. Also, since $v = w$ and $w = x$, we know that for any element of v, $v[i] = w[i]$ and for any element of w, $w[i] = x[i]$. Hence for any element of v, $v[i] = x[i]$. This means that $v = x$ and string equality is transitive.

Since string equality is reflexive, symmetric, and transitive, string equality is an equivalence relation on V. □

Remember that equivalence relations define equivalence classes. Hence $[v]_=$ is the equivalence class of v defined by string equality. Since any equivalence relation ρ is reflexive, we know that $v \in [v]_\rho$ for any type of equivalence class. Thus $v \in [v]_=$.

Property 6.2. *If v and w are strings such that $v = w$, then $\ell(v) = \ell(w)$.*

The locality data function, $\ell(v)$, is deterministic, meaning that the same string always yields the same results. Therefore, two strings that are equivalent yield the same results.

6.1.2 String Shift Equivalence

Another equivalence relation on V is what we term the **string shift**. One string is a shift of another if the two strings are the same length, and each element of the first string is equivalent to the same element of the second string plus a constant. The constant must be an integer, but may be either positive, negative, or zero. We write $v = \mathit{shift}(w)$ to indicate that v is a shift of w. Formally, $v = \mathit{shift}(w)$ iff $\big[(|v| = |w|) \wedge \forall i(1 \leq i \leq |v|)(v[i] = w[i] + c)\big]$ where $v \in V$, $w \in V$, i is an index into either string, and c is a constant integer.

Example 6.1. *Let $v_1 = 2, 7, 5, 10, 5, 2, 8$ as defined in Example 2.1. Let $v_2 = 5, 10, 8, 13, 8, 5, 11$ and $v_3 = 3, 9, 8, 14, 10, 8, 15$.*

Then $v_1 = \mathit{shift}(v_2)$ since $|v_1| = |v_2|$ and $v_1[1] = v_2[1] - 3$, $v_1[2] = v_2[2] - 3$, $v_1[3] = v_2[3] - 3$, $v_1[4] = v_2[4] - 3$, $v_1[5] = v_2[5] - 3$, $v_1[6] = v_2[6] - 3$, and $v_1[7] = v_2[7] - 3$.

$v_1 \neq \mathit{shift}(v_3)$. Even though $|v_1| = |v_3|$, the difference between the elements of the strings is not constant since $v_1[1] = v_3[1] - 1$ and $v_1[2] = v_3[2] - 2$.

We use the term **shift equivalent** to indicate that two strings are string shifts of each other. We write shift equivalence as $\stackrel{s}{=}$. Formally, $(v \stackrel{s}{=} w) \equiv (v = \mathit{shift}(w))$. We now show that the string shift is an equivalence relation on V.

Theorem 6.2. *String shift is an equivalence relation on V.*

Proof. We must show that string shift is reflexive, symmetric, and transitive on V. First we show that string shift is reflexive. Given string v we can easily see that

$|v| = |v|$ and $\forall i(1 \leq i \leq |v|)(v[i] = v[i] + 0)$, hence $v \stackrel{s}{=} v$. Therefore string shift is reflexive on V.

Next, we show symmetry. We are given strings v and w where $v \stackrel{s}{=} w$. To prove symmetry, we must show that $w \stackrel{s}{=} v$. Since $v \stackrel{s}{=} w$, we know that $|v| = |w|$ and $\forall i(1 \leq i \leq |v|)(v[i] = w[i] + c)$. Therefore, $|w| = |v|$, due to the symmetry of equality, and $\forall i(1 \leq i \leq |w|)(w[i] = v[i] - c)$. If c is a constant integer, then $-c$ is also a constant integer. So $w \stackrel{s}{=} v$, and string shift is symmetric on V.

Last, we show that string shift is transitive. We are given strings v, w, and x where $v \stackrel{s}{=} w$ and $w \stackrel{s}{=} x$. To prove it is transitive, we must show that $v \stackrel{s}{=} x$. Since $v \stackrel{s}{=} w$, we know that $|v| = |w|$. Since $w \stackrel{s}{=} x$, we know that $|w| = |x|$. Since equality is transitive, we know that $|v| = |x|$.

We also know that $\forall i(1 \leq i \leq |v|)(v[i] = w[i] + c)$ and $\forall i(1 \leq i \leq |w|)(w[i] = x[i] + d)$ where c and d are constants that may be different. Since $|v| = |w|$, we may now write $\forall i(1 \leq i \leq |v|)(v[i] = x[i] + d + c)$. Since d and c are both constants, their sum is also a constant. Therefore, $v \stackrel{s}{=} x$ and string shift is transitive on V.

Since string shift is reflexive, symmetric, and transitive on V, we know that string shift is an equivalence relation on V. □

Since string shift is an equivalence relation, it also defines an equivalence class for any string v. The notation for the equivalence class of v defined by shift equivalence is $[v]_{\stackrel{s}{=}}$. Therefore, for any string w where $w \in [v]_{\stackrel{s}{=}}$ we know that $w \stackrel{s}{=} v$. We may refer to any equivalence class defined by shift equivalence as a **shift equivalence class**.

We now show that strings that are shift equivalent have the same locality data, and hence all the members of a shift equivalence class have the same locality data.

Theorem 6.3. *If v and w are strings such that $v \stackrel{s}{=} w$ then $\ell(v) = \ell(w)$.*

Proof. Let $k = |v|$. Since $v \stackrel{s}{=} w$, then $k = |w|$ as well. Refer to Equation 2.5 for the definition of locality data. We wish to see if

$$\biguplus_{i=1}^{k} \ell(v[i]) \stackrel{?}{=} \biguplus_{i=1}^{k} \ell(w[i]).$$

Hence we may show that for all i, where $1 \leq i \leq k$, $\ell(v[i]) = \ell(w[i])$.

Given the definition of string shift, we may rewrite string w as $w = v[1]+c, v[2]+c, \cdots, v[k]+c$. The stride between any two elements of v is therefore equal to the stride between the same two elements of w:

$$\begin{aligned} stride(w[a], w[b]) &= w[b] - w[a] \\ &= (v[b] + c) - (v[a] + c) \\ &= v[b] - v[a] + c - c \\ &= v[b] - v[a] \\ &= stride(v[a], v[b]), \end{aligned}$$

where a and b are both valid indices of both v and w.

Similarly, the delay between any two elements of v is equal to the delay between the same two elements of w. We first examine the case where the delay between two elements of w is defined. In this situation:

$$\begin{aligned} delay(w[a], w[b]) &= |\delta(\{w[i] \mid (a \leq i < b)\})| \\ &= |\delta(\{v[i] + c \mid (a \leq i < b)\})|. \end{aligned}$$

Adding a constant to each element of a set does not change the cardinality of the set, so

$$\begin{aligned} |\delta(\{v[i] + c \mid (a \leq i < b)\})| &= |\delta(\{v[i] \mid (a \leq i < b)\})| \\ &= delay(v[a], v[b]). \end{aligned}$$

We now examine the case where the delay between two elements of w is undefined. This occurs when $a \geq b$ or there exists an i such that $a < i < b$ and either $w[a] = w[i]$ or $w[b] = w[i]$. We wish to determine if $delay(w[a], w[b])$ is undefined iff $delay(v[a], v[b])$ is undefined.

If either condition is true, then the delay is undefined. The first condition, that $a \geq b$, gives the same result whether we are referring to string v or string w. Therefore, we must merely prove the second condition is equivalent for both strings. We wish to see if $\exists i(a < i < b)\big[(w[a] = w[i]) \vee (w[b] = w[i])\big]$ is equivalent to $\exists i(a < i < b)\big[(v[a] = v[i]) \vee (v[b] = v[i])\big]$.

We begin with $\exists i(a < i < b)\big[(w[a] = w[i]) \vee (w[b] = w[i])\big]$. Since we know that $w[k] = v[k] + c$ for any valid index k, we may rewrite the condition as $\exists i(a < i < b)\big[(v[a] + c = v[i] + c) \vee (v[b] + c = v[i] + c)\big]$. We can now subtract c from both sides of each equality, resulting in $\exists i(a < i < b)\big[(v[a] = v[i]) \vee (v[b] = v[i])\big]$, as desired. We can use the same argument in the other direction to prove the "only if" portion. Therefore, $delay(w[a], w[b])$ is undefined iff $delay(v[a], v[b])$ is undefined.

We have now covered both cases for the delay. We may now say that for any two elements of w, $delay(w[a], w[b]) = delay(v[a], v[b])$. We previously showed that for any two elements of w, $stride(w[a], w[b]) = stride(v[a], v[b])$. Therefore, we may now write that for any two elements of w, $s/d(w[a], w[b]) = s/d(v[a], v[b])$. Therefore, for all i, where $1 \leq i \leq k$, $\ell(v[i]) = \ell(w[i])$. It follows that $\ell(v) = \ell(w)$. □

Theorem 6.4. *For any shift equivalence class $[v]_{\underline{\underline{s}}}$, if $w \in [v]_{\underline{\underline{s}}}$ then $\ell(v) = \ell(w)$.*

Proof. By definition of a shift equivalence class, if $w \in [v]_{\underline{\underline{s}}}$, then $v \overset{s}{=} w$. By Theorem 6.3, $\ell(v) = \ell(w)$. □

6.1.3 Base Equivalence With Equal Order

When two adjacent elements of a string have the same value, we term the second element an **immediately recurring element**. Formally, $v[i]$ is an immediately recurring element iff $v[i-1] = v[i]$, where $v \in V$ and i is an integer such that $2 \leq i \leq |v|$.

Example 6.2. *Let* $v_1 = 7, 2, 9, 9, 10, 1, 1, 12, 9$. *In this string there are two immediately recurring elements:* $v_1[4]$ *and* $v_1[7]$.

The string $1, 2, 1, 2, 1, 2, 3, 4, 2, 3$ *contains several recurring elements, but none of them are immediately recurring.*

The interesting thing about immediately recurring elements is that changing their location in a string does not change the string's locality data. Changing the number of immediately recurring elements only changes the number of $(0, 1)$ stride/delay relationships in the locality data. In fact, the number of immediately recurring elements in a string is equal to the count of the stride/delay combination $(0, 1)$ in the locality data for that string. We define the **order** of a string to be the number of immediately recurring elements in the string. If we remove all of the immediately recurring elements from a string, the result is what we term the **base** of the string. If two strings have the same base and the same order, then they have the same locality data. For this reason, we wish to define a new equivalence relation on V, **base equivalence with equal order**.

First we formally define some new operations on strings. As just mentioned, we let the order of a string v be the number of immediately recurring elements in v. We formally define $order(v)$ as follows:

$$order(v) = \sum_{k=1}^{|v|} p_3(v, k)$$

where

$$p_3(v,k) = \begin{cases} 1 & \text{if } (k > 1) \wedge (v[k] = v[k-1]), \\ 0 & \text{otherwise.} \end{cases}$$

In the above equation, $p_3(v,k)$ is an intermediary function that returns 1 if $v[k]$ is an immediately recurring element and 0 otherwise.

We can also calculate the order of a substring, designated by $v[i..j]$, where $1 \leq i < j \leq |v|$. When taking the order of a substring, it does not matter if elements of the substring are immediately recurring to elements not in the substring. The substring is considered as a complete string itself. Formally,

$$order(v[i..j]) = \sum_{k=i}^{j} p_4(v,i,k)$$

where

$$p_4(v,i,k) = \begin{cases} 1 & \text{if } (k > i) \wedge (v[k] = v[k-1]), \\ 0 & \text{otherwise.} \end{cases}$$

In the above equation, $p_4(v,i,k)$ is another intermediary function which returns 1 if $v[k]$ is an immediately repeating element and $k > i$. It returns 0 otherwise. We use $p_4(v,i,k)$ rather than $p_3(v,k)$ because requiring $k > i$ means that the first element of the substring is never marked as an immediately recurring element.

Example 6.3. *Let* $v_1 = 7, 2, 9, 9, 10, 1, 1, 12, 9$ *as defined in Example 6.2. Then* $order(v_1) = 2$, $order(v_1[1..4]) = 1$, $order(v_1[1..6]) = 1$, *and* $order(v_1[4..6]) = 0$.

Let $v_2 = 1, 2, 1, 2, 1, 2, 3, 4, 2, 3$. *Then* $order(v_2) = 0$.

Let $v_3 = 4, 4, 4, 4, 4, 4, 4$. *Then* $order(v_3) = 6$ *and* $order(v_3[6..7]) = 1$.

If any element of a string is followed by one or more immediately recurring elements, then we may say that those immediately recurring elements are **associated** with the original element. Note that an element may both be an immediately recurring element and have elements associated with it. Also, an immediately recurring

element may be associated with multiple elements. We write $assoc(v, i)$ to indicate the number of immediately recurring elements associated with $v[i]$:

$$assoc(v, i) = \sum_{k=i+1}^{|v|} p_5(v, i, k)$$

where

$$p_5(v, i, k) = \begin{cases} 1 & \text{if } k = i, \\ 1 & \text{if } \big(k > i\big) \wedge \big(v[k] = v[i]\big) \wedge \big(p_5(v, i, k-1) \neq 0\big), \\ 0 & \text{otherwise.} \end{cases}$$

In the above equation, $p_5(v, i, k)$ is another intermediary function. Remember that we wish to determine what elements of v are associated with $v[i]$. The function $p_5(v, i, k)$ returns 1 when $i = k$ as a base mark. For larger values of k, $p_5(v, i, k)$ returns 1 if $v[i]$ and $v[k]$ are equal and if $p_5(v, i, k-1)$ is also 1. The first requirement means that $v[k]$ is a repeat of the value $v[i]$. The second requirement means that all elements between $v[i]$ and $v[k]$ are also equal to $v[i]$. In all other cases, $p_5(v, i, k)$ returns 0.

Example 6.4. *Let* $v_1 = 7, 2, 9, 9, 10, 1, 1, 12, 9$ *and* $v_3 = 4, 4, 4, 4, 4, 4, 4$ *as defined in Example 6.3. Then* $assoc(v_1, 1) = 0$ *and* $assoc(v_1, 3) = 1$. *Also,* $assoc(v_3, 1) = 6$ *and* $assoc(v_3, 4) = 3$.

Now we are ready to formally define the base of a string. Recall that we obtain the base of a string by removing all the string's immediately recurring elements. We write $base(v)$ to indicate the base of v.

If $w = base(v)$, then $w[i] = v[p_6(v, i)]$ where $|w| = |v| - order(v)$, $1 \leq i \leq |w|$, and $p_6(v, i)$ is defined as follows:

$$p_6(v, i) = \begin{cases} 1 & \text{if } i = 1, \\ p_6(v, i-1) + 1 + assoc\big(v, p_6(v, i-1)\big) & \text{otherwise.} \end{cases}$$

In the above equation, $p_6(v, i)$ is another intermediary function that returns the index of the first instance of the ith unique element in v. It does this by taking the index of the $(i-1)$st unique element of v and then adding 1 and the number of references associated with $v[p_6(v, i-1)]$.

Example 6.5. *Let* $v_1 = 7, 2, 9, 9, 10, 1, 1, 12, 9$, $v_2 = 1, 2, 1, 2, 1, 2, 3, 4, 2, 3$, *and* $v_3 = 4, 4, 4, 4, 4, 4, 4$ *as defined in Example 6.3. Then* $base(v_1) = 7, 2, 9, 10, 1, 12, 9$, $base(v_2) = 1, 2, 1, 2, 1, 2, 3, 4, 2, 3$, *and* $base(v_3) = 4$. *Note that* $base(v_2) = v_2$.

Property 6.3. *For any string* v, $|base(v)| = |v| - order(v)$.

In other words, the length of the base of a string is equal to the length of the original string minus the number of elements removed when creating the base, which is the number of immediately recurring elements in the original string.

Property 6.4. *For any string* v, $order(base(v)) = 0$.

Since the order of a string counts the number of immediately recurring elements and the base of a string removes the immediately recurring elements, then it makes sense that the order of the base of any string is always be zero.

Property 6.5. *For a given string* v, *if* $order(v) = 0$, *then* $v = base(v)$.

If the order of a string is zero, then that string must not contain any immediately recurring elements. Taking the base of a string simply removes the immediately recurring elements. If there are none to remove, then the string is unchanged.

We now show that if two arbitrary strings have the same base, then they have the same number of immediately recurring elements if and only if they have the same length. This allows us to interchange referring to two strings having the same base and length with two strings having the same base and order.

Theorem 6.5. *If* $base(v) = base(w)$, *then* $order(v) = order(w)$ *iff* $|v| = |w|$.

Proof. We know that $|base(v)| = |v| - order(v)$ and $|base(w)| = |w| - order(w)$ from Property 6.3. Since $base(v) = base(w)$, we can write $|v| - order(v) = |w| - order(w)$. Now we first show that if the orders are the same, then the length is the same. Given $order(v) = order(w)$, we may write $|v| - order(v) = |w| - order(v)$. Adding $order(v)$ to both sides we are left with $|v| = |w|$. Therefore, if $base(v) = base(w)$ and $order(v) = order(w)$ we know that $|v| = |w|$.

Now we show that if the lengths are the same, then the order is the same. Given $|v| = |w|$, we may write $|v| - order(v) = |v| - order(w)$. Subtracting $|v|$ from both sides and multiplying both sides by -1, we have $order(v) = order(w)$. So if $base(v) = base(w)$ and $|v| = |w|$ then $order(v) = order(w)$. Therefore, if $base(v) = base(w)$, then $order(v) = order(w)$ if and only if $|v| = |w|$. □

When two strings have the same base, then the locality data is very nearly the same. For this purpose, we define **base equivalence** as follows. Two strings v and w are base equivalent if their bases are equal. We write base equivalent as $\stackrel{b}{=}$. Formally, $(v \stackrel{b}{=} w) \equiv (base(v) = base(w))$.

When the lengths of two strings are the same, and the bases are the same, then the locality data is the same. The only thing that may have changed is where the immediately recurring elements occur in the strings, and floating immediately recurring elements to other locations in the string does not change the locality data. Since they yield the same locality information, for some purposes we may want to treat all such strings as if they are the same. So we define a version of base equivalence that requires the lengths of the strings to be the same.

Recall that the order of a string is the number of immediately recurring elements in the string. Hence, if for string v, $order(v) = n$ then we say string v has order n. Two strings v and w are base equivalent with equal order if they have the same

base and equal orders. We write base equivalent with equal order as $\overset{bo}{=}$. Formally, $(v \overset{bo}{=} w) \equiv (base(v) = base(w) \wedge order(v) = order(w))$.

Example 6.6. *Let* $v_1 = 7, 2, 9, 9, 10, 1, 1, 12, 9$, $v_2 = 1, 2, 1, 2, 1, 2, 3, 4, 2, 3$, *and* $v_3 = 4, 4, 4, 4, 4, 4, 4$ *as defined in Example 6.3.*

Let $v_4 = 4, 4, 4$. *Then* $v_4 \overset{b}{=} v_3$ *since* $base(v_4) = 4$ *and* $base(v_3) = 4$. *But* $v_4 \overset{bo}{\neq} v_3$ *since* $order(v_4) = 2$ *and* $order(v_3) = 6$.

Let $v_5 = 7, 7, 7, 2, 9, 10, 1, 12, 9$. *Then* $v_5 \overset{b}{=} v_1$ *and* $v_5 \overset{bo}{=} v_1$ *since* $base(v_5) = base(v_1) = 7, 2, 9, 10, 1, 12, 9$ *and* $order(v_5) = order(v_1) = 2$.

Theorem 6.6. *Base equivalence is an equivalence relation on* V.

Proof. Two strings, v and w, are base equivalent if and only if $base(v) = base(w)$. To show that base equivalence is an equivalence relation on V, we must show that base equivalence is reflexive, symmetric, and transitive on V.

First, we show base equivalence is reflexive. It should be obvious to see that $v \overset{b}{=} v$ since $base(v) = base(v)$.

Next we show base equivalence is symmetric. Assuming $v \overset{b}{=} w$, show that $w \overset{b}{=} v$. By definition, $base(v) = base(w)$. Since equality is symmetric, we may write $base(w) = base(v)$ and hence $w \overset{b}{=} v$.

Last, we show that base equivalence is transitive. Given $v \overset{b}{=} w$ and $w \overset{b}{=} x$, show that $v \overset{b}{=} x$. By definition, we know that $base(v) = base(w)$ and $base(w) = base(x)$. Since equality is transitive, we can see that $base(v) = base(x)$ and hence $v \overset{b}{=} x$.

Since base equivalence is reflexive, symmetric, and transitive on V, it is an equivalence relation on V. □

Since base equivalence is an equivalence relation on V, it also defines an equivalence class for any string v. The notation for the equivalence class of v defined by base equivalence is $[v]_{\overset{b}{=}}$. We may refer to any equivalence class defined by base equivalence as a **base equivalence class**.

We now show that all strings that are base equivalent have the same locality data when all the $(0, 1)$ stride/delay relationships are removed.

Theorem 6.7. *If v and w are strings such that $v \stackrel{b}{=} w$, then $\ell(v) - \sigma_{s=0 \wedge d=1}(\ell(v)) = \ell(w) - \sigma_{s=0 \wedge d=1}(\ell(w))$.*

Proof. First, we show that for any string v, $\ell(base(v)) = \ell(v) - \sigma_{s=0 \wedge d=1}(\ell(v))$. This takes a number of steps. Recall Equation 2.5,

$$\ell(v) = \biguplus_{i=1}^{|v|} \ell(v[i]).$$

For all the elements of v that are not immediately recurring elements, the locality data of that element in the string is the same as the locality data of that same element in the base. In other words, when computing the locality data of a non-immediately recurring element of a string, removing all the immediately recurring elements from the string does not change that element's locality data. Formally, we must show that, given $u = base(v)$, $\forall v[i](v[i] \neq v[i-1])\big[\ell(v[i]) = \ell(u[k])$ where $k = i - order(v[1..i])\big]$.

Let $v[a]$ be an arbitrary element of v such that $a < i$. We wish to show that removing all the immediately recurring elements between $v[a]$ and $v[i]$ does not change $s/d(v[a], v[i])$. Removing immediately recurring elements does not change whether or not $s/d(v[a], v[i])$ is defined as long as $v[a]$ is not an immediately recurring element. If $s/d(v[a], v[i])$ is undefined, then there is some element between $v[a]$ and $v[i]$ equal to either $v[a]$ and $v[i]$. Since we are only removing immediately recurring elements, this does not change. If $s/d(v[a], v[i])$ is defined, removing elements cannot change that.

Removing any number of elements cannot change the value of $stride(v[a], v[i])$ since that is only dependent on the values of $v[a]$ and $v[i]$ and not their indices. Removing immediately recurring elements also does not change the value of $delay(v[a], v[i])$

when it is defined, since in that case the delay is simply the count of the number of unique elements. Removing immediately recurring elements does not change the unique count.

The only time when removing immediately recurring elements from a string changes $s/d(v[a], v[i])$ is when $v[a]$ is itself an immediately recurring element. We now show that the value of $s/d(v[a], v[i])$ is still preserved. Let $v[b]$ be the element that is associated with $v[a]$. Note that $b < a$, $v[b] = v[a]$, and for any j such that $b < j < a$, $v[b] = v[j]$. Since $b < a$ and $v[b] = v[a]$, $delay(v[b], v[i])$ is undefined. However, after removing $v[a]$ and the other immediately recurring elements, if $delay(v[a], v[i])$ was not undefined, then $delay(v[b], v[i])$ is no longer undefined, instead it is equal to $delay(v[a], v[i])$. Since $v[b] = v[a]$, $stride(v[b], v[i]) = stride(v[a], v[i])$. Thus $s/d(v[b], v[i]) = s/d(v[a], v[i])$ and the stride/delay relationship is not lost.

We now know that removing all the immediately recurring elements of a string does not change an element's locality data. This allows us to write

$$\begin{aligned}
\ell(base(v)) &= \biguplus_{v[i] \neq v[i-1]} \ell(v[i]) \\
&= \biguplus_{i=1}^{|v|} \ell(v[i]) - \biguplus_{v[i]=v[i-1]} \ell(v[i]) \\
&= \ell(v) - \biguplus_{v[i]=v[i-1]} \ell(v[i]).
\end{aligned}$$

Next, we show that $\ell(v[i]) = \{(0, 1)\}$ when $v[i]$ is an immediately recurring element. By definition of immediately recurring element, we know that $v[i] = v[i-1]$. This means that $stride(v[i-1], v[i]) = v[i] - v[i-1] = 0$ and $delay(v[i-1], v[i]) = |\delta(\{v[i-1]\})| = |\{v[i-1]\}| = 1$. Therefore, $(0, 1) \in \ell(v[i])$ when $v[i]$ is an immediately recurring element. Now we must show that $(0, 1)$ is the only element of $\ell(v[i])$. Recall that the delay, and the stride/delay relationship, between two elements is undefined when either element is equal to another element between the

two. For any earlier element of v, $v[i-1]$ is an element between the two, and since $v[i] = v[i-1]$, the delay and the stride/delay relationship is undefined. Therefore $(0,1)$ is the only stride/delay relationship in $\ell(v[i])$, and $\ell(v[i]) = \{(0,1)\}$ when $v[i]$ is an immediately recurring element.

Now we show that if $v[i]$ is not an immediately recurring element then $(0,1) \notin \ell(v[i])$. Recall that the delay is either undefined or the number of unique elements between two given elements, inclusive of the earlier and exclusive of the later. If $v[i]$ is not an immediately recurring element, then $delay(v[i-1], v[i]) = 1$. If $v[i-1]$ is not an immediately recurring element, then $delay(v[i-2], v[i]) = 2$. The delay between an earlier element and $v[i]$ is greater than 1, since there are always at least two unique elements to count, i.e. $v[i-2]$ and $v[i-1]$. If $v[i-1]$ is an immediately recurring element, then $delay(v[i-2], v[i])$ is undefined, since $v[i-2] = v[i-1]$. So the delay is only 1 between $v[i-1]$ and $v[i]$. In this instance, $v[i]$ is not an immediately recurring element, so $v[i-1] \neq v[i]$ and $stride(v[i-1], v[i]) \neq 0$. Therefore $(0,1) \notin \ell(v[i])$.

Then we can write

$$\biguplus_{v[i]=v[i-1]} \ell(v[i]) = \sigma_{s=0 \wedge d=1}(\ell(v))$$

and by replacement

$$\ell(base(v)) = \ell(v) - \sigma_{s=0 \wedge d=1}(\ell(v)).$$

Now we return to proving that if $v \stackrel{b}{=} w$ then $\ell(v) - \sigma_{s=0 \wedge d=1}(\ell(v)) = \ell(w) - \sigma_{s=0 \wedge d=1}(\ell(w))$. Since v and w are base equivalent, we know that

$$base(v) = base(w)$$

and naturally

$$\ell(base(v)) = \ell(base(w)).$$

We just proved that the locality data of the base of a string is equal to the locality data of the string minus the select of all the $(0, 1)$ elements in the string locality data. This allows us to rewrite the above as

$$\ell(v) - \sigma_{s=0 \wedge d=1}(\ell(v)) = \ell(w) - \sigma_{s=0 \wedge d=1}(\ell(w)),$$

which is what we desired to prove. □

We now show that base equivalence with equal order is also an equivalence relation on V.

Theorem 6.8. *Base equivalence with equal order is an equivalence relation on V.*

Proof. Two strings, v and w, are base equivalent with equal order if and only if $base(v) = base(w)$ and $order(v) = order(w)$. To show that base equivalence with equal order is an equivalence relation on V, we must show that it is reflexive, symmetric, and transitive on V.

First, we show base equivalence with equal order is reflexive. It should be obvious to see that $v \overset{bo}{=} v$ since $base(v) = base(v)$ and $order(v) = order(v)$.

Next we show base equivalence with equal order is symmetric. Assuming $v \overset{bo}{=} w$, we show that $w \overset{bo}{=} v$. By definition, $base(v) = base(w)$ and $order(v) = order(w)$. Since equality is symmetric, we may write $base(w) = base(v)$ and $order(w) = order(w)$. Hence $w \overset{bo}{=} v$.

Last, we show that base equivalence with equal order is transitive. Given $v \overset{bo}{=} w$ and $w \overset{bo}{=} x$, we show that $v \overset{bo}{=} x$. By definition, we know that $base(v) = base(w)$ and $base(w) = base(x)$. Since equality is transitive, we can see that $base(v) = base(x)$. By definition, we also know that $order(v) = order(w)$ and $order(w) = order(x)$. Again, since equality is transitive, we know that $order(v) = order(x)$. By showing that both $base(v) = base(x)$ and $order(v) = order(x)$, we know $v \overset{bo}{=} x$.

Since base equivalence with equal order is reflexive, symmetric, and transitive on V, it is an equivalence relation on V. □

Since base equivalence with equal order is an equivalence relation, it also defines an equivalence class for any string v. The notation for the equivalence class of v defined by base equivalence with equal order is $[v]_{\underline{\underline{bo}}}$. We may refer to any equivalence class defined by base equivalence with equal order as a **base and order equivalence class**.

We now show that all strings that are base equivalent with equal order have the same locality data and hence all the members of a base and order equivalence class have the same locality data.

Theorem 6.9. *If v and w are strings such that $v \stackrel{bo}{=} w$ then $\ell(v) = \ell(w)$.*

Proof. First, we show that for any string v,

$$\ell(v) = \biguplus_{i=1}^{order(v)} \{(0,1)\} \uplus \ell(base(v)). \tag{6.1}$$

Recall from the proof for Theorem 6.7 that if $v[i]$ is an immediately recurring element, then $\ell(v[i]) = \{(0,1)\}$. So any time we add an immediately recurring element to a string, its locality data consists entirely of the stride/delay relationship $(0,1)$.

We now show that adding an immediately recurring element does not change the locality data of any of the other elements. We wish to examine the stride/delay relationship between two arbitrary elements of v, namely $v[a]$ and $v[b]$ where $a < b$. If we add an immediately recurring element, $v[i]$, such that $i < a$ or $b < i$, the stride/delay relationship between $v[a]$ and $v[b]$ is not affected at all.

Now let us look closely at the case where $a < i < b$. $stride(v[a], v[b])$ is not affected, since $v[b] - v[a]$ is still the same. If $i \neq a + 1$, then $delay(v[a], v[b])$ is not

affected either, adding an element that is the same as an element already there does not change the number of unique elements. Also, if $delay(v[a], v[b])$ is defined, then we know that neither $v[a]$ nor $v[b]$ is equal to any element in between. Since $v[i]$ is an immediately recurring element, it is equal to an element between $v[a]$ and $v[b]$, meaning $v[a]$ and $v[b]$ are still not equal to any element between them as long as $i \neq a+1$. If $delay(v[a], v[b])$ is undefined, that means that either $v[a]$ or $v[b]$ is equal to an element between them. Adding a new element does not change this.

In the case where $i = a + 1$, if $delay(v[a], v[b])$ was undefined, then it is still undefined. If $delay(v[a], v[b])$ was defined, then we have a change. Since $v[i]$ is an immediately recurring element and $i = a + 1$, we know that $v[a] = v[i]$, meaning that $v[a]$ is now equal to an element between $v[a]$ and $v[b]$ and the delay is now undefined. However, $delay(v[i], v[b])$ gives the same value as $delay(v[a], v[b])$ did. Since $v[a] = v[i]$, $stride(v[i], v[b]) = stride(v[a], v[b])$. So even when there is a change, the overall stride/delay relationships remain the same.

Therefore, adding an immediately recurring element to a string merely adds one $(0, 1)$ stride/delay relationship to the locality data. Adding any number of immediately recurring elements to a string adds that number of $(0, 1)$ stride/delay relationships to the locality data.

Since the strings v and w are base equivalent with equal order, we know that $order(v) = order(w)$. Also, $base(v) = base(w)$ which means that $\ell(base(v)) = \ell(base(w))$. This allows us to replace $order(v)$ with $order(w)$ in Equation 6.1, and

$\ell(base(v))$ with $\ell(base(w))$, with the following result:

$$\begin{aligned}\ell(v) &= \biguplus_{i=1}^{order(v)} \{(0,1)\} \uplus \ell(base(v)) \\ &= \biguplus_{i=1}^{order(w)} \{(0,1)\} \uplus \ell(base(w)) \\ &= \ell(w).\end{aligned}$$

We have now shown that when two strings are base equivalent with equal order, the locality data of one string is equal to the locality data of the other. □

6.1.4 Equal with respect to locality

We now wish to group all the strings that yield the same locality data into the same equivalence class. First we discuss a few more ways that two different strings may yield the same locality data. As has already been shown, if $v \stackrel{s}{=} w$ then $\ell(v) = \ell(w)$ and if $v \stackrel{bo}{=} w$ then $\ell(v) = \ell(w)$. We can also easily see that v and w have the same locality data if $v \stackrel{s}{=} x \stackrel{bo}{=} w$ even if $v \neq w$. Note that $v \stackrel{s}{\neq} w$ and $v \stackrel{bo}{\neq} w$ and yet v and w have the same locality data.

For this reason, we define a **chain** of equalities. We say that v chains w over $\stackrel{s}{=}$ and $\stackrel{bo}{=}$ if there exists some finite number of strings, $x_1 \ldots x_n$, such that $v \diamond x_1 \diamond x_2 \cdots x_n \diamond w$ where $\diamond$ may be either $\stackrel{s}{=}$ or $\stackrel{bo}{=}$. We can now say that if v chains w over $\stackrel{s}{=}$ and $\stackrel{bo}{=}$ then $\ell(v) = \ell(w)$.

There are other ways in which two different strings may have the same locality data. We now give one example which we term **reverse d1 equivalence**. We later show that reverse d1 equivalence is not an equivalence relation on V, since it is neither reflexive nor transitive. Before giving a precise definition of reverse d1 equivalence, we define a new function on a string.

The **delay = 1 string**, or **d1 string**, is a string that contains, in order, all the strides that occur at a delay of 1 in a given string. We write $d1(v)$ to indicate the *delay* $= 1$ string of v. Notice that the length of the d1 string is always one less than the length of the given string, similar to Theorem 2.8. Formally, given a string v, $w = d1(v)$ when $|w| = |v| - 1$ and $w[i] = v[i+1] - v[i]$ where $1 \leq i \leq |w|$.

Example 6.7. *Let* $v_1 = 7, 2, 9, 9, 10, 1, 1, 12, 9$, $v_2 = 1, 2, 1, 2, 1, 2, 3, 4, 2, 3$, *and* $v_3 = 4, 4, 4, 4, 4, 4, 4$ *as defined in Example 6.3.*

Then $d1(v_1) = -5, 7, 0, 1, -9, 0, -11, -3$, $d1(v_2) = 1, -1, 1, -1, 1, 1, 1, -2, 1$, *and* $d1(v_3) = 0, 0, 0, 0, 0, 0$.

We can also use the d1 string to compute the stride between any two elements of the original string.

Property 6.6. *For a string* v, *if* $u = d1(v)$, *then*

$$stride(v[a], v[b]) = \sum_{i=a}^{b-1} u[i].$$

Since the d1 string records the difference between immediately successive elements of the original string, v, *we can determine the stride between non-successive elements of* v *by summing all the intermediate differences.*

Notice that saying that two strings have equal d1 strings is equivalent to saying that the two strings are shift equivalent. We formalize this as a property.

Property 6.7. *For two strings* v *and* w, $d1(v) = d1(w)$ *iff* $v \stackrel{s}{=} w$.

As was shown in the proof for Theorem 6.3, when two strings are shift equivalent, the stride between any two elements of one string is the same as the stride between the same two elements of the other string.

Two strings v and w have equivalent locality data if the d1 string for v is equal to the reverse of the d1 string of w. Recall the reverse function of a string, defined in Section 2.1. Example 6.8 demonstrates this for two specific strings.

Example 6.8. *Let* $v_1 = 1, 4, 8, 6, 2, 5, 3, 10, 7, 9$ *and* $v_2 = 2, 4, 1, 8, 6, 9, 5, 3, 7, 10$. *We may therefore calculate that* $d1(v_1) = 3, 4, -2, -4, 3, -2, 7, -3, 2$, $d1(v_2) = 2, -3, 7, -2, 3, -4, -2, 4, 3$, *and* $rev(d1(v_2)) = 3, 4, -2, -4, 3, -2, 7, -3, 2$. *It is easy to see that* $d1(v_1) = rev(d1(v_2))$.

If we calculate the locality data for v_1 *and* v_2, *we get equal results:*

$$\begin{aligned}\ell(v_1) = \; & \{(-4,1),(-3,1),(-2,1),(-2,1),(2,1),(3,1),(3,1),\\ & (4,1),(7,1),(-6,2),(-1,2),(-1,2),(1,2),(2,2),(4,2),\\ & (5,2),(7,2),(-3,3),(-3,3),(-2,3),(2,3),(5,3),(6,3),\\ & (8,3),(-5,4),(1,4),(1,4),(4,4),(4,4),(5,4),(-1,5),\\ & (1,5),(2,5),(4,5),(7,5),(-1,6),(2,6),(3,6),(6,6),\\ & (1,7),(3,7),(9,7),(5,8),(6,8),(8,9)\} \qquad = \ell(v_2).\end{aligned}$$

We later show that this is true for any two strings where the d1 string of one is equal to the reverse of the d1 string of the other. For this reason we name this property reverse d1 equivalence. Two strings are reverse d1 equivalent if and only if the d1 string for one string is equal to the reverse of the d1 string for the second string. We write reverse d1 equivalence as $\overset{r}{=}$. Formally, $(v \overset{r}{=} w) \equiv (d1(v) = rev(d1(w)))$.

Theorem 6.10. *Reverse d1 equivalence is symmetric but not reflexive or transitive.*

Proof. Two strings, v and w, are reverse d1 equivalent if and only if $d1(v) = rev(d1(w))$.

First, we show that reverse d1 equivalence is symmetric. If $v \overset{r}{=} w$, then we know that $d1(v) = rev(d1(w))$. We know take the reverse of both sides of the equation, yielding $rev(d1(v)) = rev(rev(d1(w)))$. By Theorem 2.1, $rev(rev(d1(w))) = d1(w)$. We now know that $rev(d1(v)) = d1(w)$. By the reflexitivity of equality, $d1(w) = rev(d1(v))$ and $w \overset{r}{=} v$. Hence reverse d1 equivalence is symmetric.

Now we show that reverse d1 equivalence is not reflexive. If $v \overset{r}{=} v$, then $d1(v) = rev(d1(v))$. It is easy to see that this is not true unless the d1 string of v is its own reverse. Hence $v \overset{r}{\neq} v$ for some $v \in V$ and reverse d1 equivalence is not reflexive.

Last, we show that reverse d1 equivalence is not transitive. Given $v \overset{r}{=} w$ and $w \overset{r}{=} x$ we wish to show that $v \overset{r}{=} x$ is not necessarily true. Due to the symmetry of reverse d1 equivalence, we can write that $w \overset{r}{=} v$ and hence $rev(d1(v)) = d1(w)$. Since $w \overset{r}{=} x$, we write $d1(w) = rev(d1(x))$. Since equality is transitive, $rev(d1(v)) = rev(d1(x))$. Taking the reverse of both sides and using Theorem 2.1 we obtain $d1(v) = d1(x)$. Now, unless the d1 string of x is its own reverse, $d1(v) \neq rev(d1(x))$. Hence $v \overset{r}{\neq} x$ and reverse d1 equivalence is not transitive. □

Theorem 6.11. *Reverse d1 equivalence is not an equivalence relation on V.*

Proof. To be an equivalence relation on V, reverse d1 equivalence must be reflexive, symmetric, and transitive. In Theorem 6.10 we saw that reverse d1 equivalence is neither reflexive nor transitive. Hence it cannot be an equivalence relation. □

Theorem 6.12. *If $v \overset{r}{=} w$ then $\ell(v) = \ell(w)$.*

Proof. First, we remind ourselves that since $v \overset{r}{=} w$, we know that $d1(v) = rev(d1(w))$. Now, we prove this theorem using induction on the length of v.

We first examine the base case, where $|v| = 2$. Let us write that $v = v[1], v[2]$ and $w = w[1], w[2]$. Then $d1(v) = v[2] - v[1]$ and $d1(w) = w[2] - w[1]$. Since $d1(v) = rev(d1(w))$, we know that $v[2] - v[1] = w[2] - w[1]$. Then we can write $\ell(v) = \{(v[2] - v[1], 1)\} = \{(w[2] - w[1], 1)\} = \ell(w)$.

Next, we assume that when $|v| = k$, $\ell(v) = \ell(w)$.

Lastly, we prove that when $|v| = k + 1$, $\ell(v) = \ell(w)$. Notation is very important here. Let v_k and w_k be the strings that are length k, where we already know $\ell(v_k) = \ell(w_k)$. Let v_{k+1} and w_{k+1} be the strings of length $k + 1$. The two sets of

strings relate as follows:

$$\begin{aligned} v_{k+1} &= v_{k+1}[1], \quad v_{k+1}[2], \quad \cdots, \quad v_{k+1}[k], \quad v_{k+1}[k+1] \\ &= v_k[1], \quad v_k[2], \quad \cdots, \quad v_k[k], \quad v_{k+1}[k+1] \\ w_{k+1} &= w_{k+1}[1], \quad w_{k+1}[2], \quad w_{k+1}[3], \quad \cdots, \quad w_{k+1}[k+1] \\ &= w_{k+1}[1], \quad w_k[1], \quad w_k[2], \quad \cdots, \quad w_k[k]. \end{aligned}$$

Notice that the added elements that change the strings from length k to length $k+1$ come in different places. In string v the added element is at the end and in string w it is at the beginning. This is because of the reverse equality requirement. We know that $d1(v_k) = rev(d1(w_k))$. Adding an element to the end of v_k adds the element $v_k[k] - v_k[k-1]$ to the end of $d1(v_k)$, which means $v_k[k] - v_k[k-1]$ must be added to the beginning of $d1(w_k)$, since the d1 strings are reverses of each other. Hence we get a new element on the beginnning of the string w.

Using this notation, we know that $\ell(v_k) = \ell(w_k)$ and we wish to prove that $\ell(v_{k+1}) = \ell(w_{k+1})$. Recall from the definition of locality data that

$$\ell(v_{k+1}) = \biguplus_{i=1}^{k+1} \ell(v_{k+1}[i]).$$

We can rewrite this as

$$\ell(v_{k+1}) = \ell(v_k) \uplus \ell(v_{k+1}[k+1]).$$

We can write the locality data for w_{k+1} as

$$\ell(w_{k+1}) = \biguplus_{i=1}^{k} \{s/d(w_{k+1}[1], w_k[i])\} \uplus \ell(w_k).$$

Since we know that $\ell(v_k) = \ell(w_k)$, we need only show

$$\ell(v_{k+1}[k+1]) \stackrel{?}{=} \biguplus_{i=1}^{k} \{s/d(w_{k+1}[1], w_k[i])\}$$

to prove that $\ell(v_{k+1}) = \ell(w_{k+1})$. To show that the two sides are equal, we prove that each side is a subset of the other side. First, we show that the left side is a subset of the right side.

To do this, we pick an arbitrary element of v_{k+1} that is earlier than $v_{k+1}[k+1]$. Let $v_{k+1}[a]$ be such an element. Note that $1 \leq a < k+1$. For any such a where $s/d(v_{k+1}[a], v_{k+1}[k+1])$ is defined, we know that $s/d(v_{k+1}[a], v_{k+1}[k+1]) \in \ell(v_{k+1}[k+1])$ and must prove that $s/d(v_{k+1}[a], v_{k+1}[k+1]) \in \biguplus_{i=1}^{k}\{s/d(w_{k+1}[1], w_k[i])\}$.

First, we examine the stride portion of the stride/delay relationship, using Property 6.6:

$$\begin{aligned} stride(v_{k+1}[a], v_{k+1}[k+1]) &= \sum_{i=a}^{k} d1(v_{k+1})[i] \\ &= \sum_{i=1}^{k+1-a} d1(w_{k+1})[i] \\ &= stride(w_{k+1}[1], w_{k+1}[k+1-a]) \\ &= stride(w_{k+1}[1], w_k[k-a]). \end{aligned}$$

Next, we examine the delay portion. The number of unique references between $v_{k+1}[a]$ and $v_{k+1}[k]$ can be determined by examining the d1 string between $d1(v_{k+1}[a])$ and $d1(v_{k+1}[k-1])$. One may determine the number of repeated elements by noting the number of times you can sum any number of consecutive d1 string elements with the result of zero. The same answer is reached if the d1 string is reversed, so we can instead examine the elements between $d1(w_{k+1}[1])$ and

$d1(w_{k+1}[k-a])$ and get the same result. Hence,

$$\begin{aligned} delay(v_{k+1}[a], v_{k+1}[k+1]) &= |\delta(\{v_{k+1}[a] \cdots v_{k+1}[k]\})| \\ &= |\delta(\{w_{k+1}[1] \cdots w_{k+1}[k-a+1]\})| \\ &= |\delta(\{w_{k+1}[1] \cdots w_k[k-a]\})| \\ &= delay(w_{k+1}[1], w_k[k-a]). \end{aligned}$$

Therefore, we know that $s/d(v_{k+1}[a], v_{k+1}[k+1]) = s/d(w_{k+1}[1], w_k[k-a])$, which is an element of $\biguplus_{i=1}^{k}\{s/d(w_{k+1}[1], w_k[i])\}$. So, for any stride/delay relationship in $\ell(v_{k+1}[k+1])$ we know the same stride/delay relationship exists in $\biguplus_{i=1}^{k}\{s/d(w_{k+1}[1], w_k[i])\}$. We have just shown that

$$\ell(v_{k+1}[k+1]) \subseteq \biguplus_{i=1}^{k}\{s/d(w_{k+1}[1], w_k[i])\}.$$

We now show that the right side is a subset of the left side. We pick an arbitrary element of $\biguplus_{i=1}^{k}\{s/d(w_{k+1}[1], w_k[i]])\}$. Let $i = a$ and assume that $s/d(w_{k+1}[1], w_k[a])$ is defined. If it is not defined, then it is not an element of $\biguplus_{i=1}^{k}\{s/d(w_{k+1}[1], w_k[i]])\}$ and we do not need to consider it.

Again, we examine the stride first, using Property 6.6:

$$\begin{aligned} stride(w_{k+1}[1], w_k[a]) &= \sum_{i=1}^{a-1} d1(w_{k+1})[i] \\ &= \sum_{k-a+2}^{k} d1(v_{k+1})[i] \\ &= stride(v_{k+1}[k-a+2], v_{k+1}[k+1]). \end{aligned}$$

Now we examine the delay:

$$\begin{aligned} delay(w_{k+1}[1], w_k[a]) &= |\delta(\{w_{k+1}[1] \cdots w_k[a-1]\})| \\ &= |\delta(\{v_{k+1}[k-a+2] \cdots v_{k+1}[k]\})| \\ &= delay(v_{k+1}[k-a+2], v_{k+1}[k+1]). \end{aligned}$$

Therefore, we know that $s/d(w_{k+1}[1], w_k[a]) = s/d(v_{k+1}[k-a+2], v_{k+1}[k+1])$, which is an element of $\ell(v_{k+1}[k+1])$. So, for any stride/delay relationship in $\biguplus_{i=1}^{k}\{s/d(w_{k+1}[1], w_k[i])\}$ we know the same stride/delay relationship exists in $\ell(v_{k+1}[k+1])$. We have just shown that

$$\biguplus_{i=1}^{k}\{s/d(w_{k+1}[1], w_k[i])\} \subseteq \ell(v_{k+1}[k+1]).$$

And we may now write that

$$\ell(v_{k+1}[k+1]) = \biguplus_{i=1}^{k}\{s/d(w_{k+1}[1], w_k[i])\}.$$

Which is what we needed to show that $\ell(v_{k+1}) = \ell(w_{k+1})$. Therefore, by induction on the length of the string, if $v \stackrel{r}{=} w$ then $\ell(v) = \ell(w)$. □

We can also notice that if the reverse of the d1 string is equal to the d1 string, then two strings being reverse d1 equivalent is the same as the two strings being string shift equivalent.

Property 6.8. *For any two strings v and w, if $rev(d1(v)) = d1(v)$ and $v \stackrel{r}{=} w$, then $v \stackrel{s}{=} w$.*

Since $v \stackrel{r}{=} w$, we know that $d1(v) = rev(d1(w))$. We can replace $d1(v)$ with $rev(d1(v))$, since they are equal. We then take the reverse of both sides and use Theorem 2.1 to remove the double reverses. This leaves us with $d1(v) = d1(w)$. From Property 6.7 we now know that $v \stackrel{s}{=} w$.

We must now add reverse d1 equivalence to our list of equivalences over which one string may chain another. Specifically, we now say that if v chains w over $\stackrel{s}{=}$, $\stackrel{bo}{=}$, or $\stackrel{r}{=}$, then $\ell(v) = \ell(w)$. However, there still exist other ways by which $\ell(v) = \ell(w)$. We now give one example.

Example 6.9. *Let* $v_1 = 3, 8, 2, 8, 3$. *Let* $v_2 = 2, 8, 3, 8, 2$. *Let* $v_3 = 8, 2, 7, 2, 8$. *When we compute the locality data for each string, they are equivalent:*

$$\begin{aligned}\ell(v_1) &= \{(-6,1),(-5,1),(5,1),(6,1),(-1,2),(0,2),(1,2),(0,3)\}\\ &= \ell(v_2)\\ &= \ell(v_3).\end{aligned}$$

Notice in this example that $v_1 \overset{s}{\neq} v_2$, $v_1 \overset{s}{\neq} v_3$, $v_1 \overset{bo}{\neq} v_2$, $v_1 \overset{bo}{\neq} v_3$, $v_1 \overset{r}{\neq} v_2$, $v_1 \overset{r}{\neq} v_3$, and v_1 does not chain either v_2 or v_3 over $\overset{s}{=}$, $\overset{bo}{=}$, and $\overset{r}{=}$. Therefore, there must exist at least one other way where different strings may have the same locality. We believe that the list of ways various strings may have the same locality grows as the length of the strings increases, so we do not attempt to list all these ways in this dissertation.

When two strings have the same locality, whether by one of the equivalences we have discussed in this chapter, or one chains the other over all known equivalences, or by some way not discussed here, we say that the strings are **equal with respect to locality**. We write $\overset{l}{=}$ to indicate equal with respect to locality. Formally, $v \overset{l}{=} w$ iff $\ell(v) = \ell(w)$.

Theorem 6.13. *Equality with respect to locality is an equivalence relation on* V.

Proof. We must show that equality with respect to locality is reflexive, symmetric, and transitive on V. First we show it is reflexive. Given string v, we can easily see that $v \overset{l}{=} v$ since $\ell(v) = \ell(v)$. Therefore equality with respect to locality is reflexive on V.

Next, we show symmetry. Given two strings, v and w, we must show that if $v \overset{l}{=} w$ then $w \overset{l}{=} v$. If $v \overset{l}{=} w$, then $\ell(v) = \ell(w)$. Due to the symmetry of equality, we know that $\ell(w) = \ell(v)$ and $w \overset{l}{=} v$. Hence equality with respect to locality is symmetric on V.

Last, we show equality with respect to locality is transitive. Given three strings, v, w, and x, we must show that if $v \overset{l}{=} w$ and $w \overset{l}{=} x$, then $v \overset{l}{=} x$. Since $v \overset{l}{=} w$ and

$w \overset{l}{=} x$, we may write that $\ell(v) = \ell(w)$ and $\ell(w) = \ell(x)$. Since equality is transitive, we can see that $\ell(v) = \ell(x)$ and therefore $v \overset{l}{=} x$. Hence equality with respect to locality is transitive on V.

Since equality with respect to locality is reflexive, symmetric, and transitive on V, then it is an equivalence relation on V. □

Since equality with respect to locality is an equivalence relation, it also defines an equivalence class for any string v. The notation for the equivalence class of v defined by equality with respect to locality is $[v]_{\overset{l}{=}}$. We may refer to any equivalence class defined by equality with respect to locality as a **locality equivalence class**.

Theorem 6.14. *If $\ell(v) = \ell(w)$, then $|v| = |w|$.*

Proof. If $\ell(v) = \ell(w)$, then

$$\sigma_{d=1}(\ell(v)) = \sigma_{d=1}(\ell(w)),$$

and

$$\pi_s\big[\sigma_{d=1}(\ell(v))\big] = \pi_s\big[\sigma_{d=1}(\ell(w))\big].$$

Choosing the stride portion of all the stride/delay relationships where $d = 1$ from a locality bag is equivalent to putting all the elements of the $d1$ string in a bag, i.e. $\pi_s\big[\sigma_{d=1}(\ell(v))\big] = \{d1(v)\}$. Therefore,

$$\{d1(v)\} = \{d1(w)\}.$$

If the bags are equivalent, then the number of elements in each bag is equivalent:

$$\sum_{i=-\infty}^{\infty} \#\big(i, \{d1(v)\}\big) = \sum_{i=-\infty}^{\infty} \#\big(i, \{d1(w)\}\big).$$

Counting the number of elements in a bag that consists of all the elements of a string is equivalent to obtaining the length of the string, i.e. $\sum_{i=-\infty}^{\infty} \#\big(i, \{v\}\big) = |v|$.

Therefore,

$$|d1(v)| = |d1(w)|.$$

From the definition of the $d1$ string, we know that $|d1(v)| = |v| - 1$. So,

$$|v| - 1 = |w| - 1$$

and

$$|v| = |w|.$$

Therefore, if $\ell(v) = \ell(w)$, then $|v| = |w|$. □

Corollary 6.1. *If $|v| \neq |w|$, then $\ell(v) \neq \ell(w)$.*

Proof. This follows from the simple logic result that if $A \rightarrow B$ and $\sim B$, then $\sim A$. If we let A be equivalent to the statement that $\ell(v) = \ell(w)$ and B be equivalent to the statement that $|v| = |w|$, then $A \rightarrow B$ is the logical representation of Theorem 6.14. For this corollary, we assume that $|v| \neq |w|$, or $\sim B$. We can now conclude $\sim A$, or $\ell(v) \neq \ell(w)$. □

6.1.5 How the various kinds of equalities relate

All these different types of equivalence classes relate to each other as shown in Figure 6.1. Relating the equivalence classes like this helps us to recognize how far reaching our information about a particular string is. For example, if a string is base equivalent with equal order to another string, do we know anything about whether or not the two strings are also equal, shift equivalent, or locality equivalent? We now put into theorems and prove some of the relationships shown in Figure 6.1.

We already saw in Section 6.1.4 that there may be a string, w, that has the same locality as string v without being either equal or string shift equivalent or base equivalent with equal order. We now show that any string, w, that is either string

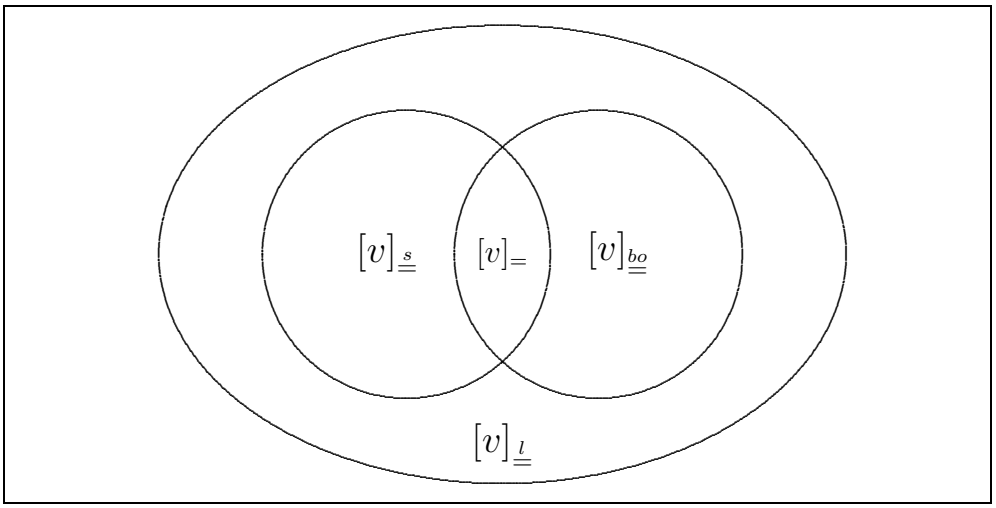


Figure 6.1: This figure demonstrates how the various equivalence classes relate.

shift equivalent to a string v or base equivalent with equal order to a string v is also locality equivalent to v.

Theorem 6.15. *For any string v, $[v]_{\underline{\underline{s}}} \subseteq [v]_{\underline{\underline{l}}}$ and $[v]_{\underline{\underline{bo}}} \subseteq [v]_{\underline{\underline{l}}}$.*

Proof. To show that $[v]_{\underline{\underline{s}}} \subseteq [v]_{\underline{\underline{l}}}$, we must show that for any string w, if $w \in [v]_{\underline{\underline{s}}}$ then $w \in [v]_{\underline{\underline{l}}}$. Let us assume that $w \in [v]_{\underline{\underline{s}}}$. This means that $v \stackrel{s}{=} w$, and, from Theorem 6.3, $\ell(v) = \ell(w)$. If $\ell(v) = \ell(w)$, then we know that $w \in [v]_{\underline{\underline{s}}}$. Hence $[v]_{\underline{\underline{s}}} \subseteq [v]_{\underline{\underline{l}}}$.

Similarly, to show that $[v]_{\underline{\underline{bo}}} \subseteq [v]_{\underline{\underline{l}}}$, we must show that for any string w, if $w \in [v]_{\underline{\underline{bo}}}$ then $w \in [v]_{\underline{\underline{l}}}$. Again, we assume that $w \in [v]_{\underline{\underline{bo}}}$. This means that $v \stackrel{bo}{=} w$, and, from Theorem 6.9, $\ell(v) = \ell(w)$. If $\ell(v) = \ell(w)$, then we know that $w \in [v]_{\underline{\underline{bo}}}$. Hence $[v]_{\underline{\underline{bo}}} \subseteq [v]_{\underline{\underline{l}}}$. □

Now we show that if a string, w, is both string shift equivalent to string v and base equivalent with equal order to string v, then w and v must be equal.

Theorem 6.16. *For strings v and w, $w \in [v]_{\underline{\underline{s}}}$ and $w \in [v]_{\underline{\underline{bo}}}$ if and only if $w = v$.*

Proof. First, we show that if $v = w$, then $v \stackrel{s}{=} w$ and $v \stackrel{bo}{=} w$. If $v = w$, then $|v| = |w|$ and $v[i] = w[i] + 0$. Since the lengths of v and w are equal, and each element of v is equal to the same element of w plus a constant, we know that $v \stackrel{s}{=} w$. If $v = w$, we know that $base(v) = base(w)$ and $order(v) = order(w)$, which is all that is required to show that $v \stackrel{bo}{=} w$. Therefore, if $v = w$, then $v \stackrel{s}{=} w$ and $v \stackrel{bo}{=} w$. We may also write that if $v = w$, $w \in [v]_{\stackrel{s}{=}}$ and $w \in [v]_{\stackrel{bo}{=}}$.

Now we show that if $v \stackrel{s}{=} w$ and $v \stackrel{bo}{=} w$ then $v = w$. To do this, we must show that $|v| = |w|$ and $\forall i(1 \leq i \leq |v|)(v[i] = w[i])$. Since $v \stackrel{s}{=} w$, we know that $|v| = |w|$. We also know that $\forall i(1 \leq i \leq |v|)(v[i] = w[i] + c)$. If we can show that $c = 0$, then we know that $v = w$.

Since we know how each element of v and w relate, we also know how each element of the bases of v and w relate. Let $u = base(v)$ and $x = base(w)$. Then $\forall j(1 \leq j \leq |u|)(u[j] = x[j] + c)$. Note that the value of c has not changed. Since $v \stackrel{bo}{=} w$, we know that $|u| = |x|$. We also know that $\forall j(1 \leq j \leq |u|)(u[j] = x[j])$. Therefore, $c = 0$. We now know that $\forall i(1 \leq i \leq |v|)(v[i] = w[i])$ and hence $v = w$. So if $w \in [v]_{\stackrel{s}{=}}$ and $w \in [v]_{\stackrel{bo}{=}}$ then $v = w$. □

We have just examined a number of ways by which various strings may have equivalent locality data. We now determine whether two strings may be equivalent with respect to locality, but not equivalent with respect to miss rate.

6.2 Matching Locality Data and Cache Performance

Equation 5.5 showed that the input string and cache configuration are sufficient to determine miss rate. We now show whether or not the locality data (or the locality

surface), the length of the input string, and the cache configuration are sufficient to determine miss rate. Essentially, we wish to replace the actual values in the string with the locality of the string. In other words, does the function $Miss(B, a, C)$ exist, where $B = \ell(v)$, $a = |v|$, and C is the cache configuration. Also, does the function $Miss(T, a, C)$ exist, where $T = \mathcal{L}(v)$, $a = |v|$, and C is the cache configuration.

Note that the locality surface contains less information than the locality data. Transforming from $\ell(v)$ to $\mathcal{L}(v)$ is a lossy transformation. As the binning occurs, we lose information about the distribution of stride/delay relationships within a given bin. Therefore, if $Miss(B, a, C)$ does not exist for a given class of caches, then $Miss(T, a, C)$ also does not exist. We first look at Case One caches, the Cases Two and Four together, and lastly Case Three caches.

6.2.1 Case One

Fully associative caches, where the cache line sizes equals the trace granularity, are the most basic caches. They also tend to be the caches that require the most time to simulate. (When simulating a cache, each reference in the trace must be compared with each entry in the appropriate set. Fully associative caches have the largest set size and therefore the longest search time for each trace entry.) We now show that simulation is unnecessary for 100% accuracy if the locality surface, or binned locality histogram, is available.

Theorem 6.17. *If a cache is fully associative and the input trace granularity matches the cache line size, then the miss rate of the trace in the cache can be determined entirely from the locality data (or the locality surface), the length of the input trace, and the cache configuration.*

Proof. For a traditional cache, the misses may be divided into three types: compul-

sory misses, capacity misses, and conflict misses [43]. This allows us to write

$$\sum_{i=1}^{|v|} miss(C, v, i, g) = \textit{compulsory} + \textit{capacity} + \textit{conflict}. \tag{6.2}$$

Compulsory misses occur when a reference has never been seen before. It is equal to the number of unique cache lines referenced by the trace. When the granularity of the trace equals the cache line size, the number of compulsory misses equals the number of unique references in the trace. This can be determined from either the locality data or the locality surface.

For any given string element $v[a]$, we can determine if the value has been seen earlier in the string from the locality data for $v[a]$. If $(0, d) \in \ell(v[a])$ for some delay d, then $v[a] = v[b]$ where $b < a$, or the value $v[a]$ has been seen earlier in the string. If $(0, d) \notin \ell(v[a])$ for some delay d, then $v[a]$ is the first instance of that value in the string. To count the number of elements that are the first instance of that value, we count the number of elements in the string where $\sim \exists d\big[(0, d) \in \ell(v[a])\big]$, or count the number of elements where $\exists d\big[(0, d) \in \ell(v[a])\big]$ and subtract from $|v|$. We can do this using either the locality data or the locality histogram:

$$\begin{aligned} \textit{compulsory} &= |v| - \sum_{d=1}^{\infty} \#\big((0, d), \ell(v)\big) \\ &= |v| - \sum_{d=1}^{\infty} h(\ell(v), 0, d). \end{aligned} \tag{6.3}$$

When we convert from the histogram to the binned histogram, nothing is changed in the stride direction where stride is zero. In the delay direction, some of the values are summed together to make delay bins, but the smallest delay value is still one. Since we are summing to infinity, we do not need to worry about the maximum value. We may therefore write

$$\textit{compulsory} = |v| - \sum_{b=1}^{\infty} H(\ell(v), 0, b). \tag{6.4}$$

When we convert from the binned histogram to the surface, each bin is divided by $|v| - 1$ where stride equals zero:

$$\begin{aligned} \textit{compulsory} &= |v| - \sum_{b=1}^{\infty} \big[(|v| - 1)\mathcal{S}(\ell(v), v, 0, b)\big] \\ &= |v| - (|v| - 1) \sum_{b=1}^{\infty} \mathcal{L}(v, 0, b). \end{aligned} \tag{6.5}$$

As can be easily seen from Equations 6.3 and 6.5, the number of compulsory misses for a fully associative cache where the line size and granularity match can be determined not only from the locality data for the input trace, but also from the locality surface for the input trace.

Capacity misses occur when an element of the input trace is a repeated value, but the delay between the two values is larger than the number of lines in the cache. This can be determined from the locality data by counting the occurrences of the stride/delay combination $(0, d)$ where d is greater than the number of lines in the cache. Specifically:

$$\begin{aligned} \textit{capacity} &= \sum_{d=(C_s/C_l)+1}^{\infty} \#\big((0, d), \ell(v)\big) \\ &= \sum_{d=(C_s/C_1)+1}^{\infty} h(\ell(v), 0, d). \end{aligned} \tag{6.6}$$

Since we are again focused entirely on the temporal axis of the locality histogram, we may perform the same operations for capacity misses that we performed for compulsory misses:

$$\begin{aligned} \textit{capacity} &= \sum_{b=\log_2(C_s/C_l)+2}^{\infty} H(\ell(v), 0, b) \\ &= \sum_{b=\log_2(C_s/C_l)+2}^{\infty} \big[(|v| - 1)\mathcal{S}(\ell(v), v, 0, b)\big] \\ &= (|v| - 1) \sum_{b=\log_2(C_s/C_l)+2}^{\infty} \mathcal{L}(v, 0, b). \end{aligned} \tag{6.7}$$

Again, Equations 6.6 and 6.7 show that the number of capacity misses can be determined not only directly from the locality data, but also from the locality surface for the input trace.

By definition, there are no conflict misses in a fully associative cache. We may therefore write $conflict = 0$.

We have shown that we can determine the number of each type of misses possible for a conventional cache using either the locality data or the locality surface. We can then use Equation 6.2 to compute the total number of misses for the input trace in a fully associative cache where the granularity matches the line size. To compute the miss rate, we merely divide the number of misses by the length of the input trace. Therefore, for a Case One cache, we can compute the miss rate given the locality data or locality surface of the input trace and the length of the input trace. There exists equations for $Miss()$ such that $Miss(v, C) = Miss(B, a, C) = Miss(T, a, C)$ where v is the input string, C is the cache configuration, $B = \ell(v)$, $a = |v|$, and $T = \mathcal{L}(v)$. □

6.2.2 Cases Two and Four

Cases Two and Four caches are both situations where the cache line size and granularity do not match. The difference is that Case Two caches are fully associative and Case Four caches are not. However, we now show that, regardless of the associativity, the locality data is insufficient to determine the miss rate when the cache line size does not match the trace granularity.

Theorem 6.18. *If the input trace granularity does not match the line size of the desired cache, then the locality data of the input trace, the length of the input string, and the cache configuration are insufficient to determine the miss rate.*

Proof. We prove this by contradition. Assume the function $Miss(B, a, C)$ exists for caches where $C_l \neq g$. Recall that $B = \ell(v)$, $a = |v|$, and C is the cache configuration. This means that given the locality data, string length, and cache configuration, we should be able to determine the miss rate.

Let $v_1 = 1, 2, 3, 4$. Let $v_2 = 2, 3, 4, 5$. Note that $v_1 \stackrel{s}{=} v_2$ and therefore $\ell(v_1) = \ell(v_2)$ from Theorem 6.3. We compute $\ell(v_1) = \{(1, 1),\ (1, 1),\ (1, 1),\ (2, 2),\ (2, 2),\ (3, 3)\} = \ell(v_2)$. We choose a granularity of 8 bytes, i.e. $g = 8$ bytes, for both strings.

Now, let us pick a cache configuration. Let $C_l = 16$ bytes, $C_s = 16$ Kbytes, and $C_a = 1024$. Therefore, C is a fully associative cache. Now we compute the miss rate for v_1 and v_2 in C using $Miss(v, C)$ from Equation 5.5. For both v_1 and v_2, we compute $h = \log_2(C_l/g) = \log_2(2) = 1$. We calculate that $zoom(v_1, 1) = 0, 1, 1, 2$ and $zoom(v_2, 1) = 1, 1, 2, 2$. $Miss(v_1, C) = 3/4 = 75\%$. $Miss(v_2, C) = 2/4 = 50\%$.

Now we return to the function we assumed existed, i.e. $Miss(B, a, C)$. Let $a = 4$ and $B = \ell(v_1) = \ell(v_2)$. Then $Miss(B, a, C) = 0.75$ sometimes and $Miss(B, a, C) = 0.5$ sometimes. This is a contradiction. Therefore, the function $Miss(B, a, C)$ cannot exist when $C_l \neq g$. In other words, the locality data, string length, and cache configuration are insufficient to determine miss rate when the line size does not equal the granularity. □

The basic reason for this result can be seen in Equation 5.4. When the line size and granularity do not match, we need to use the function $zoom(v, h)$. Recall that computing the result of $zoom(v, h)$ requires the actual value of each element of v. The locality data only contains the relative values of each element of v, not the actual values. When $v \stackrel{s}{=} w$, we know that $\ell(v) = \ell(w)$, but $zoom(v, h)$ may not be equal to $zoom(w, h)$, as seen in the proof for Theorem 6.18. If $zoom(v, h) \neq zoom(w, h)$, then we cannot guarantee that the miss rate of v is equal to the miss rate of w.

Although the locality data is insufficient to precisely predict cache performance, the spatial locality on the locality surface is helpful in Case Two situations. We already saw, in Chapter 4, that the information off the temporal axis guides qualitative assessments of optimal line size. In Chapter 7 we show how we can predict, with some error, the number of compulsory misses when the cache line size is larger than the trace granularity.

6.2.3 Case Three

We now look at caches that are set associative and where the cache line size equals the trace granularity. By definition, the capacity misses in a set associative cache are equal to the capacity misses in a fully associative cache that has the same cache and line size. The conflict misses in a set associatve cache are equal to the total number of misses in that cache minus the compulsory misses and the capacity misses [43, page 423].

It periodically occurs, however, that a capacity miss in a fully associative cache becomes a hit in a set associative cache with the same size. Because of this, it is difficult to determine whether a given trace element is a capacity or conflict miss in a set associative cache. For this reason, we group capacity and conflict misses together and only differentiate between compulsory misses and capacity/conflict misses.

Theorem 6.19. *If a cache is not fully associative, but the line size of the cache matches the granularity of the trace, then the locality data of each element of the trace is sufficient to predict the trace miss rate in the given cache.*

Proof. Let C be the given cache configuration and v be the input trace. Since we know that the cache is not fully associative and the line size matches the granularity, we may write that $C_a \neq C_s/C_l$ and $C_l = g$. We now show that we can compute

$miss(C, \ell(v[i]), g)$ for any i such that $1 \leq i \leq |v|$. In other words, the locality data for the trace element is sufficient to determine if that element is a hit or miss in the cache.

We can easily determine if $v[i]$ is a compulsory miss by detecting if there is a stride/delay relationship in the locality data for $v[i]$ where the stride is zero. Specifically, if $\sim \exists d[(0, d) \in \ell(v[i])]$ then $v[i]$ is a compulsory miss.

Given that $v[i]$ is not a compulsory miss, we now wish to determine if it is either a capacity or conflict miss. We do this by counting how many unique elements have occurred in the same cache set and comparing it with the number of lines in the cache set. We know from the cache configuration that there are C_a lines in each cache set of the cache. If there were less than C_a unique accesses to the same cache set since $v[i]$ was last referenced, then $v[i]$ is a hit. If there were C_a or more unique accesses, then $v[i]$ is a miss.

We now need to determine the number of unique accesses to the same cache set since $v[i]$ was last referenced. We can get this information from $\ell(v[i])$. Let $v[j]$ be the most recent element where $v[j] = v[i]$. For any element $v[k]$ such that $k < j$, we know that $s/d(v[k], v[i]) \notin \ell(v[i])$ since $v[j]$ is between $v[k]$ and $v[i]$ and $v[j] = v[i]$ (see Equation 2.4). Therefore, $\ell(v[i])$ does not contain any stride/delay relationships with elements of v earlier than the last time $v[i]$ was seen. We also know, from Theorem 2.4, that $\ell(v[i])$ only contains unique strides, and therefore does not reference earlier duplicate accesses.

We can determine which stride/delay relationships in $\ell(v[i])$ are with elements that use the same cache set as $v[i]$ by examining the stride portion. Let $s/d(v[j], v[i])$ be a stride/delay relationship in $\ell(v[i])$. If $stride(v[j], v[i])$ is a multiple of the number of cache sets in the cache C, then $v[j]$ is assigned to the same cache set as $v[i]$. This should be easy to see when we recall from Chapter 5 how the cache

set for a particular element is computed. Let t be the cache set $v[i]$ is assigned to. Then $t = v[i] \bmod n$ where $n = C_s/(C_aC_l)$ or the number of cache sets in the cache. Let $s = stride(v[j], v[i]) = v[i] - v[j]$. If s is a multiple of n, then $v[j] \bmod n = (v[i] - s) \bmod n = v[i] \bmod n$.

Therefore, to determine the number of unique accesses to the same cache set since $v[i]$ was last referenced, we need only remove the stride/delay relationships where the stride is not a multiple of the number of sets, and count the number of relationships remaining. Specifically, $v[i]$ is a capacity or conflict miss if $\left|\sigma_{(s \bmod n=0)}\big(\ell(v[i])\big)\right| > C_a$, where $n = C_s/(C_aC_l)$. To summarize, $v[i]$ is a miss if either $\sim \exists d[(0, d) \in \ell(v[i])]$ or $\left|\sigma_{(s \bmod n=0)}\big(\ell(v[i])\big)\right| > C_a$, where $n = C_s/(C_aC_l)$, otherwise $v[i]$ is a hit.

Since we can determine if each element of v is a hit or miss in a set associative cache using the locality data of that element, we can sum up the number of misses and divide by the length of v to determine the miss rate of v in C using the locality data of each element of v. □

We have just shown that, given the cache configuration and the locality data for a given trace element, we can determine if the element is a hit or miss in a set associative cache where the cache line size equals the trace granularity. However, when the locality data for all the elements are bagged together it is difficult to determine which stride/delay relationships are associated with which elements.

Our interest is primarily with stride/delay relationships where the stride is either 0 or n, where n is the number of cache sets. The $stride = 0$ relationships are preserved when the locality histogram is binned. However, if $n > 2$, all the $stride = n$ relationships are binned with other strides, and impossible to accurately separate. Even when $n = 2$, the binning in the delay direction still removes much of the necessary detail. We therefore conclude that the method used to determine hits

and misses in Theorem 6.19 is no longer possible using either the binned locality histogram or the locality surface. In the next chapter, we discuss another method for estimating the miss rate in Case Three caches.

6.3 Summary

In this chapter, we have shown how two different strings may have the same locality data, and hence the same locality surface, for various reasons, and grouped these reasons into equivalence classes. These equivalences can be applied in a number of ways. For example, Theorem 6.7 allows us to conveniently reduce a trace size. We may store the base of a given trace along with the trace's immediately repeating reference count without losing any locality information. For many of our traces, this is a quick removal of about 30% of the trace. Many trace compression methods take advantage of immediately repeating elements, however they generally preserve the specific location of each one [51, 60]. We now know that it is unnecessary to preserve the location, since immediately repeating elements affect any cache in the same manner no matter where they are located.

We have also proved that two strings that are equivalent with respect to locality have the same miss rate in fully associative caches where the cache line size matches the traces' granularity. We then examined how two strings that are equivalent with respect to locality may have different miss rates in other types of caches. This limits our ability to use the locality surface to predict miss rate for specific categories of caches. Next, we demonstrate the accuracy of our predictions by attempting to predict cache performance for several workloads with various cache configurations.

Chapter 7

Quantitative Cache Performance Prediction Using Locality Surfaces

As just discussed in Chapter 6, we are limited in our ability to predict the results of cache simulation using the locality surface. In this chapter, we validate the conclusions drawn and examine some approximating methods. We use the same six traces we used in Chapter 4. We reproduce Table 4.1 here as Table 7.1 for convenience. Recall from Chapter 4 that these traces were selected as representative of a variety of localities.

We first examine fully associative caches where the line size equals the trace's granularity (Case One). Next we examine fully associative caches where the line size does not match the trace granularity (Case Two), and then discuss set associative caches (Case Three). Lastly we review the results of other researchers, compare them with our work, and examine the difficiencies of their work.

workload	suite	type	total refs	uniq refs	description
applu	FP	D	46,261,474	1,524,041	Parabolic/Elliptic Partial Differential Equations
crafty	INT	I	50,020,348	30,338	Game Playing: Chess
galgel	FP	D	37,070,561	1,255,136	Computation Fluid Dynamics
perlbmk.diffmail	INT	I	54,083,478	34,648	PERL Programming Language
swim	FP	D	42,031,084	7,988,204	Shallow Water Modeling
twolf	INT	I	50,191,887	21,988	Place and Route Simulator

Table 7.1: Description of the traces used in this chapter. All of these traces were taken under the Linux operating system.

7.1 Case One

As presented in Chapter 5, a Case One situation occurs when the cache is fully associative and the cache line size equals the input string's granularity. The proof for Theorem 6.17 tells us that in this situation we can determine the miss rate of the trace from the locality surface or the binned locality histogram, the length of the trace, and the cache configuration.

When outputting the results from our locality program, we routinely output both the binned locality histogram and the locality surface. The values on the binned histogram are output as unsigned longs and the values on the locality surface are output as floats with six significant digits. We use the latter to make the visual locality surface, but the former has more precision and is therefore more useful for calculations such as needed here.

Using the binned locality histogram for all six traces with cache sizes from 8 bytes (the smallest cache possible with 8-byte lines) to 128 Mbytes, we achieve 100%

accuracy, as proved in Theorem 6.17. Using the locality surface, rather than the binned locality histogram, we have very small errors (on the order of 0.05%) due to round off.

7.2 Case Two

As presented in Chapter 5, a Case Two situation occurs when the cache is fully associative and the cache line size does not equal the input string's granularity. We know from Theorem 6.18 that the locality data is insufficient to precisely determine the miss rate of the input trace. We can, however, use the locality surface to make some approximations. Since this is a fully associative cache, we wish to predict the compulsory misses and the capacity misses. The conflict misses are zero.

7.2.1 Compulsory Misses

First we discuss the compulsory misses. We can use the locality data and some probability to make some guesses as to the number of compulsory misses for a given trace in a given cache when the trace granularity and cache line size do not match. When the line size is less than the granularity, the trace itself contains no information about performance with the smaller line size. If the trace does not have the information, the locality cannot either. Therefore, we only examine cases where the line size is greater than the granularity.

We first examine the situation when the cache line size is twice the size of the granularity. If we knew how many unique references in the trace had a $stride = 1$ relationship with another unique reference, then we could guess that half of those would be in the same line as the other reference, and reduce our compulsory miss count accordingly.

workload	$uniq_8$	$str1$	$\widehat{uniq}_{16}$	$uniq_{16}$	error
applu	1,524,041	1,204,681	921,700	888,859	3.69%
crafty	30,338	24,689	17,993	17,683	1.76%
galgel	1,255,136	487,704	1,011,284	985,057	2.66%
perlbmk.diffmail	34,648	27,682	20,807	20,274	2.63%
swim	7,988,204	4,781,027	5,597,690	5,577,900	0.35%
twolf	21,988	15,290	14,343	13,713	4.59%

Table 7.2: The data and calculation results for $\widehat{uniq}_{16}$ given the number of unique references that are stride 1 from another unique reference.

We let $str1$ represent the count of how many unique references have a $stride = 1$ relationship with another unique reference. We may compute $str1$ from the trace by listing all the unique references in the given trace (at the specified granularity), sorting the list, and counting how many references are $stride = 1$ from the previous reference. We let $uniq_8$ represent the number of unique references that exist in a given trace with a granularity of 8 bytes, $uniq_{16}$ be the number of unique references in the trace with a granularity of 16 bytes, etc. Further, we let $\widehat{uniq}_{16}$ be our estimate of the number of unique references at a granularity of 16 bytes, etc. We then use the following equation to calculate $\widehat{uniq}_{16}$:

$$\widehat{uniq}_{16} = uniq_8 - \tfrac{1}{2}str1. \tag{7.1}$$

Table 7.2 shows the results, using Equation 7.1, for all six traces, and the errors. Note that the number of unique references at a given granularity is equivalent to the number of compulsory misses at that line size.

These errors are quite good. The problem is that we computed $str1$ from the trace, not the locality surface. The locality data tells us how often $stride = 1$ occurs relative to the *total* number of references, not the *unique* number of references as required. We must find a way to approximate $str1$ from the locality surface of the trace.

If we sum along the $(1, d)$ slice of the binned histogram, we get the total number of references in the trace that have a $stride = 1$ relationship with some other reference in the trace at some delay. If, before summing, we divide each bin by the total number of references in the trace, then each bin represents the percentage of references that have a $stride = 1$ relationship with some other reference at the given delay. By summing these divided bins we get the percentage of references in the trace that have a $stride = 1$ relationship with some other reference at some delay. We remind the reader that this is total references, not unique references.

Recall that we divided each bin by the total number of references. However, we know that any reference that has a $stride = 0$ relationship at some delay cannot have a $stride = 1$ relationship at a larger delay. We also know that any reference that has a $stride = 0$ relationship is not the first instance of that value, and hence is not unique. Therefore, when dividing each bin, we wish to divide by the number of references that do not have $stride = 0$ relationships at a smaller delay rather than dividing by the total number of references.

When we divide this way and sum the results, we get an approximation of the percentage of unique references in the trace that have a $stride = 1$ relationship with some other unique reference at some delay. If we multiply this percentage by the number of unique references, we get an approximation for $str1$, as desired. We let $\widehat{str1}$ represent this approximation:

$$\widehat{str1} = \sum_{d_1=1}^{\infty} \frac{H(\ell(v), 1, d_1)}{|v| - \sum_{d_2=1}^{d_1} H(\ell(v), 0, d_2)} \tag{7.2}$$

Table 7.3 compares the actual $str1$ with its approximation calculated using Equation 7.2 and shows the error. We now use $\widehat{str1}$ instead of $str1$ in Equation 7.1 to compute $\widehat{uniq}_{16}$. Table 7.4 shows these results for all six traces, along with the error.

We can use a similar technique for predicting $uniq_{32}$ and $uniq_{64}$, or the number of

workload	$\widehat{str1}$	$str1$	error
applu	1,049,329	1,204,681	-12.90%
crafty	22,859	24,689	-7.41%
galgel	681,458	487,704	39.75%
perlbmk.diffmail	26,783	27,682	-3.25%
swim	3,238,640	4,781,027	-32.26%
twolf	19,006	15,290	24.30%

Table 7.3: Both $str1$ and $\widehat{str1}$ are listed here, along with the error. $\widehat{str1}$ is calculated using Equation 7.2.

workload	$uniq_8$	$\widehat{str1}$	$\widehat{uniq}_{16}$	$uniq_{16}$	error
applu	1,524,041	1,049,329	999,376	888,859	12.43%
crafty	30,338	22,859	18,908	17,683	6.93%
galgel	1,255,136	681,458	914,407	985,057	-7.17%
perlbmk.diffmail	34,648	26,783	21,256	20,274	4.84%
swim	7,988,204	3,213,000	6,381,704	5,577,900	14.41%
twolf	21,988	19,006	12,485	13,713	-8.96%

Table 7.4: Computing $\widehat{uniq}_{16}$ using $\widehat{str1}$ calculated from the binned locality histogram using Equation 7.2.

compulsory misses for cache line sizes of 32 byte and 64 bytes, respectively. In these cases, we not only need the number of $stride = 1$ relationships among the unique references, we also need the $stride = 2$ through $stride = 7$ relationships. Our results would be even more accurate if we knew the $stride = 2$ through $stride = 7$ relationships that did not span a smaller stride relationship. For example, if the sorted list of unique references included $4, 5, 6$ we would technically have one $stride = 2$ relationship. However, the probability of 4 and 6 being in the same line is already covered by the two instances of $stride = 1$ relationships and therefore the $stride = 2$ relationship should not be counted.

Let $str2$, $str3$, $str4$, etc. be the frequency of stride 2, 3, 4, etc. occuring between unique references without counting strides that are covered by smaller strides. We calculated $strk$ (where k represents the desired stride value) directly from the trace by first creating a sorted list of the unique references and then counting the number of immediately successive references with a stride k. Using this data, we can estimate $uniq_{32}$ and $uniq_{64}$ as follows:

$$\widehat{uniq}_{32} = uniq_8 - \tfrac{3}{4}str1 - \tfrac{1}{2}str2 - \tfrac{1}{4}str3, \tag{7.3}$$

$$\widehat{uniq}_{64} = uniq_8 - \tfrac{7}{8}str1 - \tfrac{3}{4}str2 - \tfrac{5}{8}str3 - \tfrac{1}{2}str4 - \tfrac{3}{8}str5 - \tfrac{1}{4}str6 - \tfrac{1}{8}str7, \tag{7.4}$$

where $\widehat{uniq}_{32}$ and $\widehat{uniq}_{64}$ are the estimates of $uniq_{32}$ and $uniq_{64}$, respectively.

Table 7.5 shows the results calculated using Equations 7.3 and 7.4 where $str1$ through $str7$ were calculated from the trace. Again, the results are quite promising, if we are able to adequately calculate $str2$ through $str7$ from the binned histogram rather than the trace.

Unfortunately, approximating $str2$ through $str7$ from the binned locality histogram is more difficult than approximating $str1$. In addition to converting from total references to unique references, we must also subtract off the references that we

workload	$\widehat{uniq}_{32}$	$uniq_{32}$	error	$\widehat{uniq}_{64}$	$uniq_{64}$	error
applu	598,243	557,382	7.33%	410,211	388,508	5.59%
crafty	10,787	10,718	0.65%	6,468	6,393	1.18%
galgel	781,998	739,463	5.75%	542,442	519,490	4.42%
perlbmk.diffmail	12,607	12,463	1.16%	7,644	7,524	1.60%
swim	4,195,654	4,156,728	0.94%	2,338,565	2,315,298	1.00%
twolf	9,486	9,420	0.70%	5,926	5,833	1.60%

Table 7.5: Calculations for $\widehat{uniq}_{32}$ and $\widehat{uniq}_{64}$ (if $str1$ through $str7$ were available) using Equations 7.3 and 7.4.

already covered by smaller strides. We cannot just subtract the values obtained for the smaller strides, since not all of them contribute to a larger stride. For example, the string $4, 5, 67, 68, 70, 71, 72$ has four instances of stride 1 and two instances of stride 2. There is no way to determine from the count of strides how many of the smaller strides contibute to the larger stride count. Another issue is the binning on the locality surface and binned histogram. For strides greater than 2, multiple strides are binned together and we must guess how many are of each type.

We have investigated a number of methods for estimating the desired figures despite these constraints, however none have proved more accurate for most of our workloads than not using the values at all. The simplest method is to merely use $\widehat{str1}$, which we have already computed. This method is about as accurate as any we have investigated that use $\widehat{str2}$ through $\widehat{str7}$. The equations are as follows:

$$\widehat{uniq}_{32} = uniq_8 - \tfrac{3}{4}str1, \tag{7.5}$$

$$\widehat{uniq}_{64} = uniq_8 - \tfrac{7}{8}str1. \tag{7.6}$$

The results and errors for $\widehat{uniq}_{32}$ and $\widehat{uniq}_{64}$ using Equations 7.5 and 7.6 are shown in Table 7.6. When we compare with the results for $\widehat{uniq}_{16}$ from Table 7.4, we can see that in general the error increases as the line size increases. We believe Equations 7.5 and 7.6 work relatively well because the values for $str2$ through $str7$

workload	$\widehat{uniq}_{32}$	$uniq_{32}$	error	$\widehat{uniq}_{64}$	$uniq_{64}$	error
applu	737,044	557,382	32.23%	605,878	388,508	55.95%
crafty	13,193	10,718	23.10%	10,336	6,393	61.68%
galgel	744,043	739,463	0.62%	658,860	519,490	26.83%
perlbmk.diffmail	14,561	12,463	16.83%	11,213	7,524	49.03%
swim	5,559,224	4,156,728	33.74%	5,154,394	2,315,298	122.62%
twolf	7,733	9,420	-17.90%	5,358	5,833	-8.15%

Table 7.6: Results and errors for $\widehat{uniq}_{32}$ and $\widehat{uniq}_{64}$ using Equations 7.5 and 7.6 and the estimate for $str1$.

are generally small. When $str2$ through $str7$ approach zero, Equations 7.3 and 7.4 become Equations 7.5 and 7.6.

In summary, we can estimate the compulsory misses when the line size does not match the granularity of the locality surface. When the line size is twice the granularity, the errors are reasonable (less than ±15%), however, they can become much larger as the line size increases. Most of the error comes from the estimate of $str1$ through $str7$ from the locality surface. Perhaps a method that also uses the negative stride information would prove more accurate.

7.2.2 Capacity Misses

Predicting the capacity misses for a Case Two situation is more difficult than predicting compulsory misses. If we had the temporal axis for the locality surface where the granularity did match, we would have a Case One situation and 100% accuracy. Perhaps there is a way to use the spatial locality information from a locality surface to estimate the temporal axis for larger granularities.

We computed the locality surfaces for both the data trace of *applu* and the instruction trace of *twolf* with granularities of 16 bytes, 32 bytes, and 64 bytes so we can determine how accurate our estimates are. We picked these two workloads

because they have very different locality surfaces. *applu* has poor locality and is a floating point data trace. *twolf* has fairly good locality and is an integer instruction trace. If we can approximate the values on the temporal axis for the larger granularity surfaces, then we may predict the miss rate for caches with those line sizes.

We can predict the first couple of entries on the temporal axis fairly accurately. When multiplying the granularity by two, we know that half the $stride = 1$ and $stride = -1$ entries where $delay = 1$ are probably in the same line as the previous reference. We use a similar philosophy for estimating how many are in the same line when the line size is multiplied by four or eight.

Again, we are using the binned histogram, with the following notation. We let $H_8(a, b)$ be the bin labeled a and b on the histogram with 8-byte granularity. We let $H_{16}(a, b)$, $H_{32}(a, b)$, and $H_{64}(a, b)$ be the 16-byte, 32-byte, and 64-byte (respectively) granularity histogram bins labeled a and b. We let $\widehat{H}_{16}(a, b)$, $\widehat{H}_{32}(a, b)$, and $\widehat{H}_{64}(a, b)$ be the estimates for the 16-byte, 32-byte, and 64-byte (respectively) granularity histogram bins labeled a and b.

Using this notation, we use the following equations for estimating the delay 1 entries on the temporal axis:

$$\widehat{H}_{16}(0,1) = H_8(0,1) + \tfrac{1}{2}\big[H_8(1,1) + H_8(-1,1)\big], \tag{7.7}$$

$$\begin{aligned}\widehat{H}_{32}(0,1) =& H_8(0,1) + \tfrac{3}{4}\big[H_8(1,1) + H_8(-1,1)\big] + \tfrac{1}{2}\big[H_8(2,1) + H_8(-2,1)\big] \\ &+ \tfrac{1}{8}\big[H_8(3,1) + H_8(-3,1)\big],\end{aligned} \tag{7.8}$$

$$\begin{aligned}\widehat{H}_{64}(0,1) =& H_8(0,1) + \tfrac{7}{8}\big[H_8(1,1) + H_8(-1,1)\big] + \tfrac{3}{4}\big[H_8(2,1) + H_8(-2,1)\big] \\ &+ \tfrac{9}{16}\big[H_8(3,1) + H_8(-3,1)\big] + \tfrac{3}{16}\big[H_8(4,1) + H_8(-4,1)\big].\end{aligned} \tag{7.9}$$

Notice that when dealing with bins that contain multiple strides we assume

	applu		*twolf*	
	delay 1	delay 2	delay 1	delay 2
$\widehat{H}_{16}(0,d)$	7,382,719	4,431,380	27,603,743	3,141,872
$H_{16}(0,d)$	7,268,095	3,857,957	28,656,416	4,076,924
error	1.58%	14.86%	-3.67%	-22.94%
$\widehat{H}_{32}(0,d)$	10,279,541	6,086,303	32,714,425	9,522,786
$H_{32}(0,d)$	9,038,482	4,802,816	33,455,514	4,745,447
error	13.73%	26.72%	-2.22%	100.67%
$\widehat{H}_{64}(0,d)$	12,200,856	7,552,402	35,803,129	13,653,779
$H_{64}(0,d)$	10,486,305	5,512,830	37,013,995	5,073,421
error	16.35%	37.00%	-3.27%	169.12%

Table 7.7: Estimating the temporal axis for larger granularity temporal axes for delays 1 and 2 using Equations 7.7 – 7.9.

that each stride in the bin is equally represented with the other strides in the bin. Table 7.7 shows the results for delays 1 and 2 using Equations 7.7 – 7.9.

The errors are very good where $delay = 1$, but are much larger for delay 2. The errors increase even further at larger delays. This occurs because there are references in the 8-byte granularity trace that create a stride/delay relationship of $(1, 1)$, but create a $(0, 1)$ in a larger granularity. In the smaller granularity, the reference has relationships with larger strides. However, the larger granularity does not have any relationships at larger strides. As the delay gets larger on the temporal axis, more and more of these relationships are ones that would simply not be calculated at larger granularities and our errors become unreasonably large. Therefore, this method for estimating the temporal axis at larger granularities is not practical for delays larger than 2 or 3, depending on the trace.

We can use our estimates of compulsory misses to estimate the maximum delay that has data for larger granularities. We know that there cannot be a delay larger than the number of unique references. Therefore, we take the estimate of the number of unique references and round up to the next largest power of two for our estimate

	applu	*twolf*
$\widehat{uniq}_{16}$	999,376	12,485
estimated max delay	1 M	16 K
real max delay	1 M	16 K
$\widehat{uniq}_{32}$	737,044	7,733
estimated max delay	1 M	16 K
real max delay	1 M	16 K
$\widehat{uniq}_{64}$	605,878	5,358
estimated max delay	1 M	8 K
real max delay	512 K	8 K

Table 7.8: Here we use the estimate of the unique references at larger granularities to predict the largest delay that has data on the temporal axis. Only one of the six cases is inaccurate.

of the maximum delay with data at a given granularity. Table 7.8 shows these results. In the six test cases, we were only off once.

We can also use our estimates of compulsory misses to estimate the total number of values that should appear on the temporal axis. Recall Equation 6.4, which we redisplay here as Equation 7.10:

$$compulsory = |v| - \sum_{b=1}^{\infty} H(\ell(v), 0, b). \tag{7.10}$$

Recall also that the compulsory misses are equivalent to the number of unique references. Therefore, if we know the length of the trace and have an estimate for the number of unique references, we can estimate the sum along the temporal axis on the binned locality histogram.

So, when attempting to predict the temporal axis at larger granularities, we have the first few points fairly accurately, we know at what delay the values become zero, and we know approximately how many values should be on the temporal axis. We have attempted a number of methods for predicting the other values on the temporal axis, but none that work well for both of the workloads selected. Perhaps if more workloads were examined, better equations or heuristics could be discovered.

granularity	*applu*	*twolf*
8 bytes	2d 11:30:38	0:45:54
16 bytes	1d 05:31:35	0:19:41
32 bytes	13:16:46	0:08:48
64 bytes	09:21:38	0:04:11

Table 7.9: The time to compute the locality surfaces for the data trace of *applu* and the instruction trace of *twolf* at various granularities. The time is shown in days, hours:minutes:seconds.

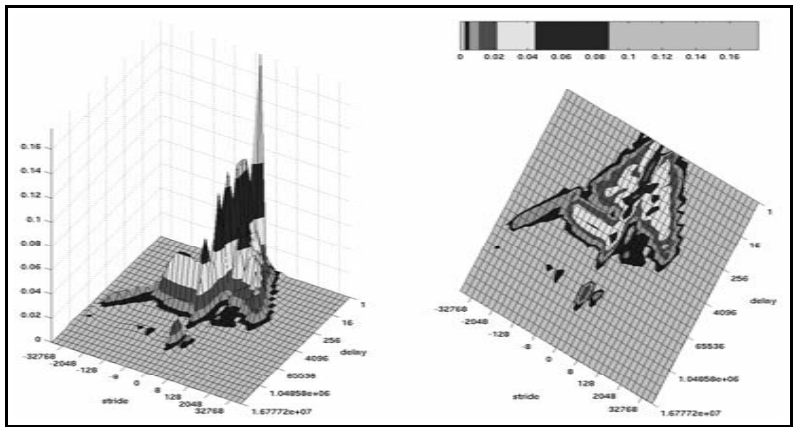
7.2.3 Recomputing the Locality

Rather than using the smaller granularity locality surface to predict the results at larger granularities, we can convert the original trace to a larger granularity and recompute the locality surface. Essentially, we convert a Case Two situation into a Case One situation, where we have 100% accuracy. Figure 7.1 shows the locality surface at all four computed granularities for the data trace of *applu*. Figure 7.2 shows the locality surface for all four computed granularities for the instruction trace of *twolf*.

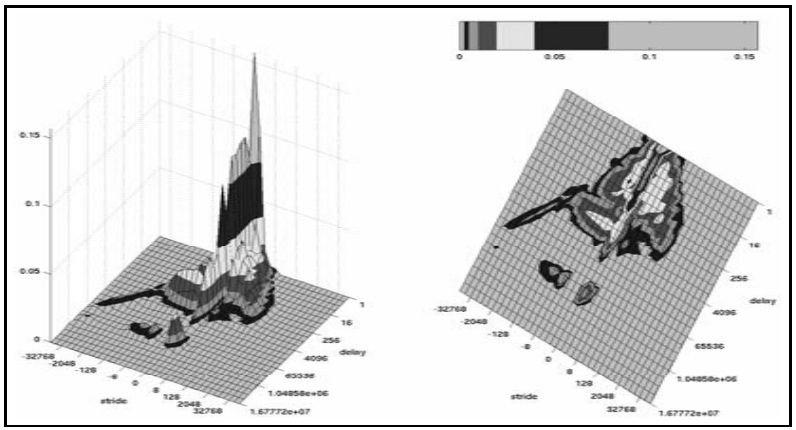
The advantage of computing the surface with a larger granularity is 100% accurate cache simulation predictions for that particular granularity and faster locality computation. Table 7.9 shows the times necessary to run the locality program for our two traces at various granularities. These timing runs were all performed on a 3.20 GHz Pentium 4 machine with 512 Kbytes of L2 cache.

As the granularity increases, the locality also improves, meaning that our stack based algorithm runs faster. We can see significant improvement for both workloads in the time to compute the locality as the granularity increases. If a researcher has no interest in results at a smaller granularity, it is advantageous to the compute time to use the larger granularity.

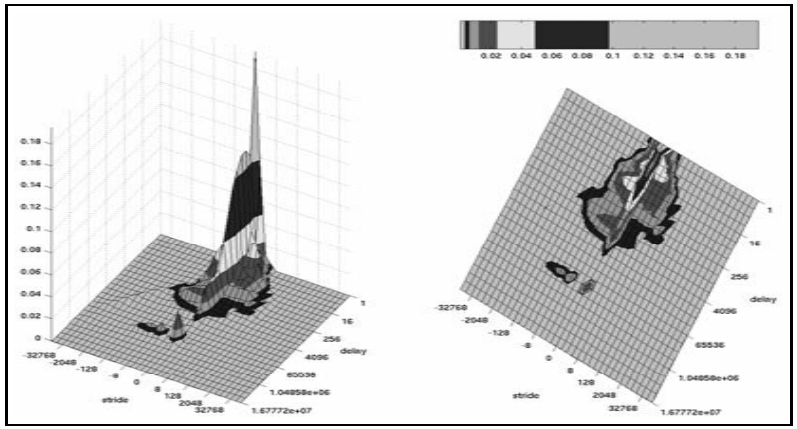
The disadvantage of computing the locality at a larger granularity is the loss of



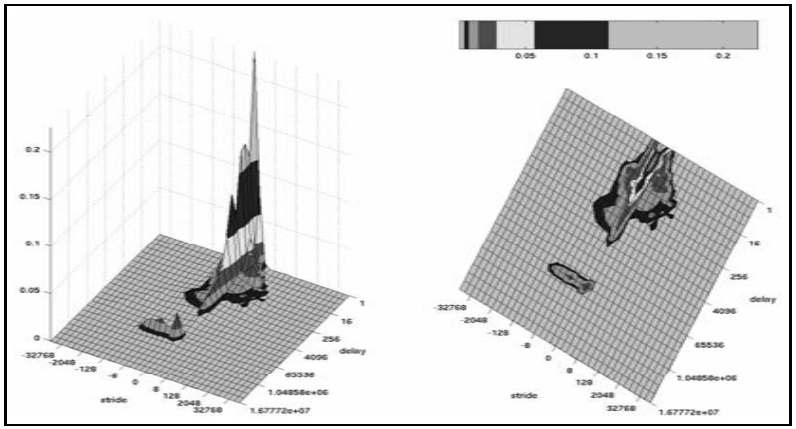
(a) 8-byte granularity



(b) 16-byte granularity

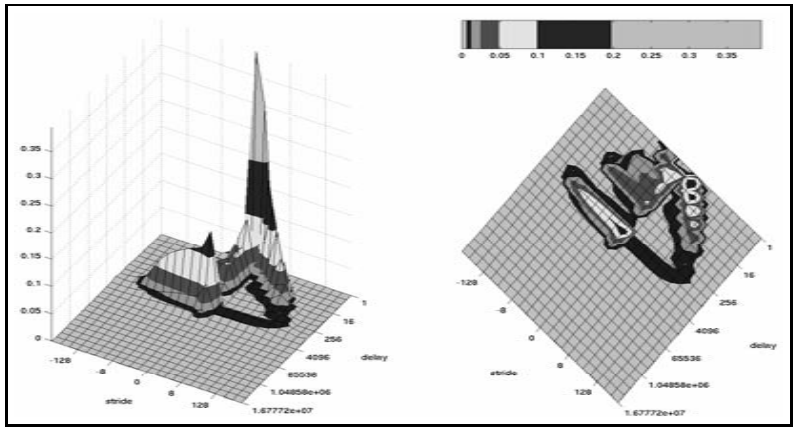


(c) 32-byte granularity

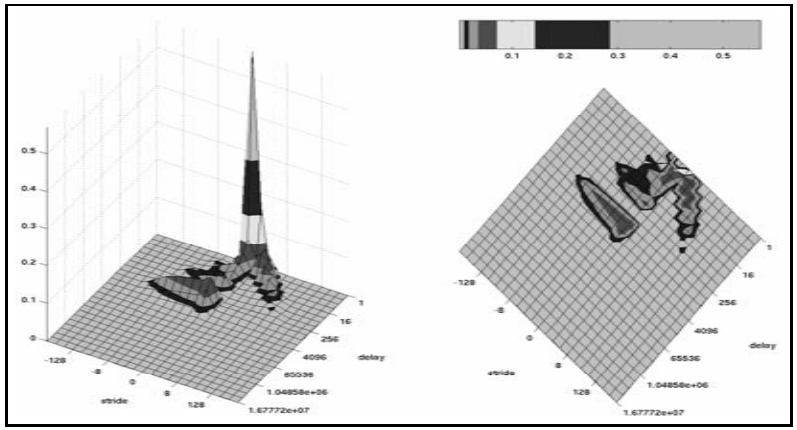


(d) 64-byte granularity

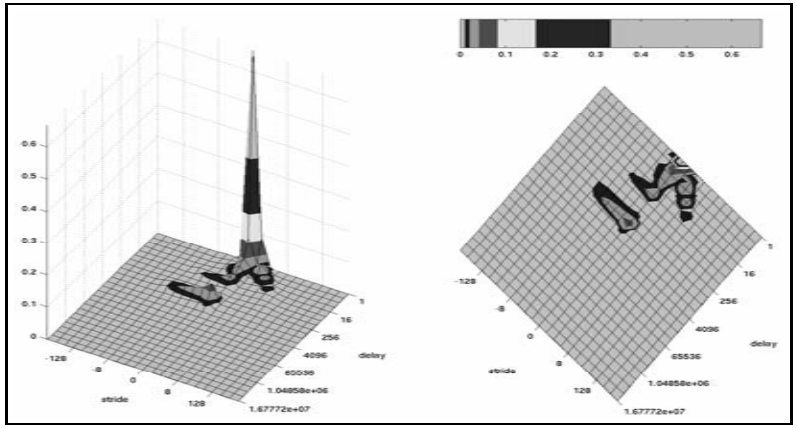
Figure 7.1: The locality surfaces for the data trace of *applu* with various granularities.



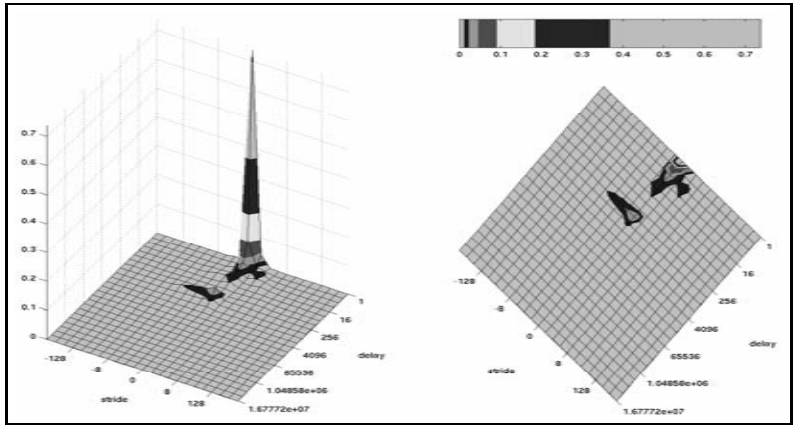
(a) 8-byte granularity



(b) 16-byte granularity



(c) 32-byte granularity



(d) 64-byte granularity

Figure 7.2: The locality surfaces for the instruction trace of *twolf* with various granularities.

information. We lose information about smaller granularities and we lose our ability to perform much of the workload characterization discussed in Chapter 3. It can be easily seen in both Figure 7.1 and 7.2 that features disappear as the granularity increases. Ding and Zhong point out similar advantages to keeping the granularity smaller, even at the expense of more accurate cache simulation predictions [30].

In addition, all our techniques for estimating cache results at different granularities involve increasing the granularity, not decreasing it. We can qualitatively predict cache performance for larger granularities using the techniques of Chapter 4, or quantitatively predict some aspects of cache simulation using the techniques described in this section. However, there is no way to predict performance at smaller granularities; the information is lost.

We therefore recommend computing the locality surface at the smallest granularity of interest. If accuracy is required at larger granularities, recompute the locality for each of the granularities, keeping in mind that the compute time at the larger granularity is not as intensive as the compute time for the smaller granularity.

7.3 Case Three

As presented in Chapter 5, a Case Three situation occurs when the cache is set associative (i.e. not fully associative) and the cache line size equals the input string's granularity. In Section 6.2.3 we described what information is needed to determine if a given reference is a hit or miss in a set associative cache. We also discussed why that data is unobtainable from the locality surface or the binned locality histogram.

We can, however, make some estimates. Recall from Theorem 6.17 that we can determine the compulsory misses from the binned locality histogram, using Equation 7.10. Note that the number of compulsory misses does not change as we

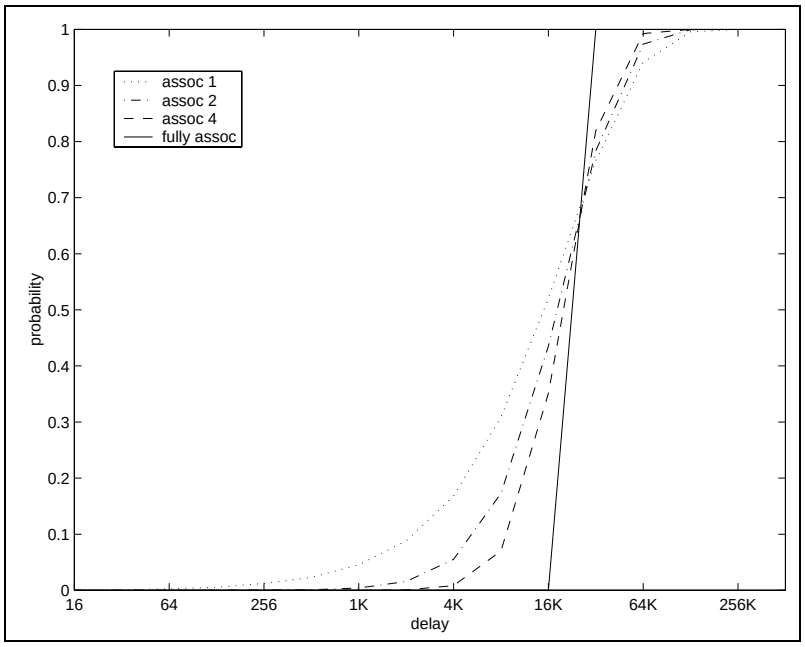


Figure 7.3: The temporal axes from several 128 Kbyte cache characterization surfaces with varying associativities.

adjust the cache associativity; the compulsory misses only change if we change the line size. Therefore, we may use Equation 7.10 to compute the compulsory misses in Case Three.

To determine the capacity and conflict misses, we use the temporal axis from the appropriate cache characterization surface. Recall from Chapter 5 that the associativity changes the shape of the temporal axis curve on the cache characterization surface. Figure 7.3 shows the temporal axis for several 128 Kbyte caches with varying associativities.

Notice that the temporal axis for the fully associative cache moves directly from 0 to 100%. The portions under the set associative curves that fall to the left of the fully associative curve represent misses in the set associative caches that would have been hits in the fully associative cache. The portions over the set associative curves that fall to the right of the fully associative curve represent hits in the set

associative cache that would have been misses in the fully associative cache. It is interesting that the curves all cross at about 70%, and the volume under the curves to the left of the fully associative line is greater than the volume over the curves to the right. This matches with the well known fact that set associative caches have more misses than fully associative caches.

We use these curves in Figure 7.3 to estimate the capacity and conflict misses of the set associative caches. Recall that we calculated the capacity misses for fully associative caches by summing along the temporal axis from the size of the cache to infinity (Equation 6.7). This is equivalent to multiplying the temporal axis of the locality histogram (or surface) with the temporal axis of the fully associative cache characterization surface for the correct cache size and summing the result.

We use this same technique for the set associative caches. For example, to estimate the number of misses in the 128 Kbyte, 2-way set associative cache for the data trace of *applu*, we multiply the temporal axis of the binned locality histogram for *applu* with the temporal axis of the appropriate cache characterization surface and sum the result. If we let $\widehat{capcon}$ represent our estimate of the number of capacity plus conflict misses, the following equation specifies this process:

$$\widehat{capcon} = \sum_{b=1}^{\infty} \big[H(\ell(v), 0, b) * \mathcal{C}(C, 0, b) \big], \tag{7.11}$$

where v is the desired trace and C is the desired cache configuration. Combining Equation 7.11 with Equation 7.10, we get:

$$\begin{aligned} \widehat{misses} &= |v| - \sum_{b=1}^{\infty} H(\ell(v), 0, b) + \sum_{b=1}^{\infty} \big[H(\ell(v), 0, b) * \mathcal{C}(C, 0, b) \big] \\ &= |v| + \sum_{b=1}^{\infty} \big[H(\ell(v), 0, b) * \big(\mathcal{C}(C, 0, b) - 1 \big) \big], \end{aligned} \tag{7.12}$$

where $\widehat{misses}$ is our estimate of the number of misses for trace v in cache C.

workload		$C_a = 1$	$C_a = 2$	$C_a = 4$	$C_a = 8$
applu	$\widehat{misses}$	4,883,919	4,181,017	3,968,537	3,903,054
	$misses$	8,612,874	7,670,076	6,306,170	5,418,684
	error	-43.30%	-45.49%	-37.07%	-27.97%
crafty	$\widehat{misses}$	2,556,467	793,541	243,458	109,406
	$misses$	2,752,372	261,022	119,685	80,303
	error	-7.12%	204.01%	103.42%	36.24%
galgel	$\widehat{misses}$	4,508,783	4,308,332	4,259,317	4,238,313
	$misses$	8,858,121	7,898,721	6,057,235	5,222,165
	error	-49.10%	-45.56%	-29.68%	-18.84%
perlbmk.diffmail	$\widehat{misses}$	758,162	263,960	129,608	93,155
	$misses$	855,076	441,268	324,779	147,151
	error	-11.33%	-40.18%	-60.09%	-36.69%
swim	$\widehat{misses}$	15,115,953	14,196,168	13,667,975	13,349,677
	$misses$	18,886,397	18,214,693	16,449,426	15,718,587
	error	-19.96%	-22.06%	-16.91%	-15.07%
twolf	$\widehat{misses}$	813,824	166,721	44,522	28,867
	$misses$	859,166	445,543	195,992	96,489
	error	-5.28%	-62.58%	-77.28%	-70.08%

Table 7.10: Results using Equation 7.12 for all six traces used in this chapter with 128 Kbyte caches of various associativities.

Table 7.10 shows the results for $\widehat{misses}$ using Equation 7.12, the real number of misses, and the error for all six workloads used in this chapter with 128 Kbyte caches of various associativities. The errors are acceptable, but not excellent. Note that all but two of the errors are negative, indicating that our estimates tend to be high. We guess this is because our estimate is based on the purely random data used to create the cache characterization surface, while real traces are not random. Perhaps this could be used for better results in the future.

7.4 Case Four

As presented in Chapter 5, a Case Four situation occurs when the cache is set associative and the cache line size does not equal the input string's granularity. We do not specifically investigate this situation, due to the lack of reliable results for Case Two caches. Should a reasonable method be discovered for Case Two caches, it would be valuable to combine the technique with the best Case Three equations found and apply the results to Cache Four situations.

7.5 Previous Work

We now briefly examine the results of other researchers attempting to predict cache miss rates so we can better evaluate our results. Many of these other researchers do not specifically mention the error between their predictions and the actual miss rates. Instead they display graphs where readers can visually see the separation between the lines representing the predicted miss rates versus the lines reprsenting the actual miss rates. To ease comparison between our results and these other researchers' results, we have calculated a few errors from these graphs by selecting

a few points, approximating their values, and then calculating the error. We hope that this method does not in any way misrepresent anyone's results. We first discuss researchers who used locality to predict miss rates, than other methods, and finally compare our research with theirs.

7.5.1 Using Locality to Predict Miss Rate

Several research groups have spent time examining temporal locality by calculating the number of unique addresses between repeated references to the same memory address. To differentiate from researchers who use other methods for evaluating temporal locality, many have termed this **reuse distance** [11, 16, 30]. The two-dimensional reuse distance graphs are equivalent with the temporal axis from our lcoality surface, except some researchers visualize the data differently.

Most reuse distance researchers have noticed, as we did in Section 7.1, that temporal locality can predict Case One cache miss rates with 100% accuracy [16, 88]. These researchers generally do not examine line sizes that are different than their chosen granularity. When addresssing set associative caches, they typically claim that accurate fully associative results are sufficient for one of two reasons.

Some point out that since temporal locality can accurately predict the capacity misses, it can also pinpoint the cache sizes around which more detailed simulation should be done [88]. Others reference Smith's paper [72] that uses the fully associative cache results as inputs to a single, complex equation to predict the set associative results [19, 45]. Hill and Smith claim that the relative error, using this equation, is usually less than 5% and rarely greater than 10% [45].

Since others have adequately examined how the results of the fully associative cache can predict the results of set associative caches, we have not replicated this

work. We simply note that since we also can predict Case One cache performance results with 100% accuracy, we may also use these same methods for Case Three caches.

It is not troubling to note that temporal locality researchers typically do not predict miss rates for caches with varying line sizes, since changing the line size is more related to spatial locality. However, most spatial locality researchers apply their locality metrics to other areas such as prefetching [48, 53, 87] and redistributing data in memory [82, 90] rather than finding the optimal cache line size.

7.5.2 Using Non-locality Methods to Predict Miss Rate

Other methods for predicting cache miss rate results include cache miss equations, analytical models, sampling, and synthetic traces. We briefly discuss an example of each.

Cache miss equations are detailed, mathematical representations of loops and other memory accessing patterns in the code of a program [37, 65]. Researchers using these equations have reported excellent miss rate prediction errors, specifically, their errors are less than 0.4% [37]. Cache miss equations have been used to determine optimal line size at arbitrary associativies. In other words, they are powerful enough to predict results for all four of our cache cases. However, the information necessary for the equations is obtained at the compiler level and is therefore not useful to replace trace driven simulation and cannot be compared with our work.

Analytical models involve extracting a few parameters from a given trace and using them as inputs to the model's equations that calculate the miss rate. A couple of examples include Singh et al. [70] and Agarwal et al. [5]. We here discuss the first example.

Singh et al. created an analytical model that extracts four parameters from a given trace [70]. Using these four parameters, they predict miss rate with "high accuracy" in large fully associative caches. They apply their model to caches with various cache size and line sizes and show their results using a series of graphs. For large caches, the errors approach zero as the cache size and line size increase. However, some of the relative errors are as large as 46%. For small caches, the errors do not appear to be directly related to the cache size. The smallest errors appear at medium line sizes and are close to zero. We calculated their maximum error for small, fully associative caches at about 190%. Singh et al. did not present results for varying associativities.

Sampling is typically used to dramatically reduce the time necessary for trace driven simulation by selecting a small portion of a trace to represent the entire trace. A number of sampling methods have been examined over the years. One recent example is Berg and Hagersten's work [15] that used sampling to estimate miss rate for fully associative caches with random replacement. Since line size is not mentioned, we assume that the granularity and line size matched.

Berg and Hagersten presented their results using a series of graphs and did not specifically mention any errors. We calculated that their maximum relative error was around 85% for Case One caches. A problem with Berg and Hagersten's work is that they used random replacement caches, rather than the more common LRU caches [9]. Obviously our results for Case One caches compares favorably with theirs.

Lastly, we discuss synthetic traces. When using synthetic traces to predict miss rate, the object is to represent the trace as a small number of parameters which take much less space to store than the entire trace. (This is discussed in more detail in the next chapter.) Thiebaut et al. represented each of seven traces using merely two parameters [79]. They then created synthetic traces, ran them through cache

simulators, and compared the miss rate with the actual miss rate.

Again, the paper did not specifically mention errors in most cases, so we have approximated the errors from the graphs in the paper. Thiebaut et al. first investigated the results of all seven traces they used in a range of fully associative caches up to 4 Kbytes in size. Since line size is not mentioned, we assume that the granularity matches the line size. Some of the larger errors are well over 300% for these Case One caches.

When investigating set associative caches, Thiebaut et al. picked one of the traces (which appears to have the smallest maximum relative error, around 29%) and simulated the synthetic and original versions of the trace on caches with associativity from 1 to 128. The paper claims that the maximum relative error found was 25%.

7.5.3 Comparing Our Work

In general, it is difficult to directly compare our work with the results presented by other researchers. Some of the older papers use traces much shorter than ours, and older benchmarks [45, 70, 72, 79]. As mentioned in Chapter 1, we believe our traces to be among the most accurate available. For a fair comparison, all cache prediction methods should be performed on the same set of traces.

In addition, completely evaluating a given method involves cache simulations and method predictions for a large number of traces and cache configurations. Even if performing all the simulations is within reason, it is difficult to present all the results in a single paper. Therefore, most reseachers limit either the number of caches or the number of traces used. For example, in their synthetic traces paper, Thiebaut et al. only use one trace on their range of set associative caches [79]. In contrast, we chose to use only one cache size.

When reducing the number of traces to a reasonable level, most researchers do not mention how they selected the traces used. We, however, used our locality surface to select traces with a variety of locality features and sizes to make sure our method applies to all varieties of traces.

In general, our miss rate predictions are just as good, if not better than the results presented by other researchers. Only the reuse distance researchers were able to also obtain 100% accuracy for Case One caches. Few researchers even attempted to predict results for Case Two caches. For those that did, their errors were in the same range with the errors we obtained for Case Two compulsory misses [70]. The researchers who evaluated Case Three caches reported errors in the same range as our Case Three errors [79].

We also point out that the goal for many of these researchers was not necessarily 100% accuracy. For example, the point of sampling is to reduce the trace size and simulation time while keeping accuracy within reason [15]. The goal of the analytical modeling and synthetic trace papers mentioned was to reduce the trace to two or four scalar parameters [5, 70, 79]. Our locality surface is admittedly time consuming to compute and uses many more than four parameters. However, the few kilobytes necessary to store the locality surface and binned locality histogram is still significantly better than the gigabytes necessary for today's traces. We more completely address the issue of time to compute our locality surface in Chapter 9.

7.6 Summary

We have found that, using the temporal axis of our locality surface, we can predict cache miss rates with 100% accuracy for fully associative caches where the cache line size and trace granularity match. In addition to using the locality surface

visually to qualitatively determine optimal line size, we can quantitatively predict the compulsory misses, with some error, when the cache line size is larger than the trace granularity. Another option is to recompute the locality surface at the desired larger granularity for 100% accuracy. Finally, we can use the cache characterization surface and the locality surface to quantitatively predict, with some error, the miss rate for set associative caches.

In general, it appears that other researchers who have results much better than ours are estimating the miss rate at a different point than we do [37]. A number of researchers have discovered, as we did, that the miss rate of Case One caches can be accurately predicted using the temporal locality. However few of these researchers have attempted to expand the results to include Case Two or Three caches. For those who have, our errors are in the same range as theirs. We next use the locality surface as a method of evaluating a number of synthetic traces presented by other researchers.

Chapter 8

Using Locality Surfaces to Evaluate Synthetic Traces

We have spent considerable time discussing how we can use the locality surface to predict cache performance. We now investigate another use of the locality surface: evaluating the accuracy of synthetic trace models.

One of the most popular methods for evaluating systems is trace-driven simulation. Originally, synthetic traces were used in simulations because real traces didn't exist. Now, as mentioned in Section 3.2, real and accurate traces are obtainable. The length of these real traces is only limited by the time necessary to obtain the trace and the storage limitations. As caches get larger, however, the storage required for even one sufficiently long trace increases significantly [17]. One solution is a return to synthetic traces.

Synthetic traces have small storage requirements. For a given model, one can extract the necessary parameters from a real trace. Storing only these parameters, one can later produce an arbitrarily long trace. Synthetic traces can also be created for systems not yet developed, and controlled to "stress" a system in ways that

workloads have not yet reached. In this situation, one must only specify the parameters necessary for the given model to produce a long trace. Using this method, it is much easier to create a synthetic trace than to collect a real trace by taking costly time to instrument and trace the system.

Unfortunately, synthetic traces have a large drawback—even when the parameters are extracted from a real trace, the synthetic trace tends to be inaccurate. When using traces for simulation purposes, one would want the simulation results of a synthetic trace to be the same as the real trace the model parameters came from. In this chapter, we evaluate the accuracy of synthetic traces by comparing the synthetic trace's locality surface with the original trace's locality surface.

In addition to all its previous uses, the locality surface is a good measure for evaluating how well a synthetic trace duplicates the locality of the original trace. If the locality surfaces match, the synthetic trace is accurate in terms of locality. If the surfaces do not match, the synthetic trace is not accurate. We here show that none of the synthetic models evaluated reproduce the original trace's locality. It would be ideal to be able to use the locality surface itself to generate a synthetic trace; however, we do not attempt that at this time.

We first describe the six models we discuss, and then compare the locality surface for each synthetic trace with the original trace locality surface. We then compare the cache results for each synthetic trace with the original trace cache results. This confirms the conclusions reached when comparing the locality surfaces, and further validate the argument that the locality surface represents cache results. Lastly, we roughly describe a potential algorithm for creating synthetic traces from the locality surface.

In [39], Grimsrud used his original locality surfaces to evaluate several synthetic trace models. However, his surface had many weaknesses that kept him from specif-

ically determing what portions of each model were accurate and what portions were not. Grimsrud's surface was not very effective at predicting cache performance, leaving some question as to its ability to determine whether a given model is effective at replicating cache results.

8.1 Previous Models

We selected six synthetic trace generation models to examine. The necessary parameters for each model can either be extracted from a real trace or independently determined by a researcher. These models are all fairly simple, and have been around for at least ten years.

8.1.1 Independent Reference Model

First is the **Independent Reference Model** (IRM) [8, 14]. From a real trace, we determine the frequency of any given memory address occuring. We use these frequencies as probabilities for generating a sequence of references having the same distribution.

8.1.2 Stack Model

Next is the **Stack Model** (SM) from [13]. For this model, we maintain a LRU stack of references already seen and record the frequency of accessing a reference x deep in the stack or a reference not yet in the stack. In implementing this model, when we were required to create a new reference not in the stack, we generated a random, uniformly distributed, reference.

8.1.3 Partial Markov Reference Model

The **Partial Markov Reference Model** (PM) [5] assumes that all references are either sequential or random. The model has two states, one that creates random references and one that generates sequential references. From a real trace, we determine the frequency of staying in each state versus switching states. Using these statistics, we can switch states and generate references. When creating random references, we used a uniform distribution between the minimum and maximum values found in the real trace.

8.1.4 Distance Model

Next is the **Distance Model** (D) [75]. Again we take a real trace and determine the frequency of any stride occuring between succesive references in memory. In calculating references we generate a stride from the stored probabilities, add the stride to the previous reference and thereby produce the next reference.

8.1.5 Distance-Strings Model

The **Distance-Strings Model** (DS) [75, 34] also uses the frequencies of the strides from a real trace. However, instead of using the strides between successive references, the Distance-Strings Model uses the strides between successive bursts of sequential references. It also uses the frequencies of the lengths of sequential bursts. When creating references, we use the stored probabilities to generate a stride and a burst length. We add the stride to the first reference of the last sequential burst and then produce x sequential references, where x is the randomly chosen length.

8.1.6 Random Walk Model

The **Random Walk Model** (RW) [79] simulates a random walk "with references governed by a hyperbolic probability law." (This is the synthetic generation model that was briefly discussed in Chapter 7.) Two parameters are extracted from a real trace. The authors claim that the two parameters correspond to the working set size and the locality of the real trace. These parameters are input into a program that simulates the random walk through memory, outputting the memory address at each step. To extract the necessary two parameters from the original trace, the **footprint** of the trace must be calculated. The footprint is a graph showing the number of unique references seen versus the total number of references processed.

8.2 Traces Used

We selected the workload *twolf*, from the SPEC CINT 2000 benchmark suite, to use as a base for modeling each of the six models mentioned. Again, we separated instruction fetches from data reads and writes, effectively giving us two traces to model. For simplicity, in this chapter we are only using the first 10 million instructions and the first 10 million data reads and writes from the *twolf* trace. The first 10 million instructions of *twolf* have $18,407$ unique references; the first 10 million data reads and writes of *twolf* have $207,852$ unique references. Figure 8.1 shows the first 10 million references from the instruction trace of *twolf*. Figure 8.2 shows the first 10 million references from the data trace of *twolf*.

We selected *twolf* because its two traces demonstrate several of the features discussed in Chapter 3. The locality surfaces for both the instructions and the data of *twolf* show significant temporal locality. The locality surface for the instruction trace also shows a significant sequential ridge and some strong loops. The locality

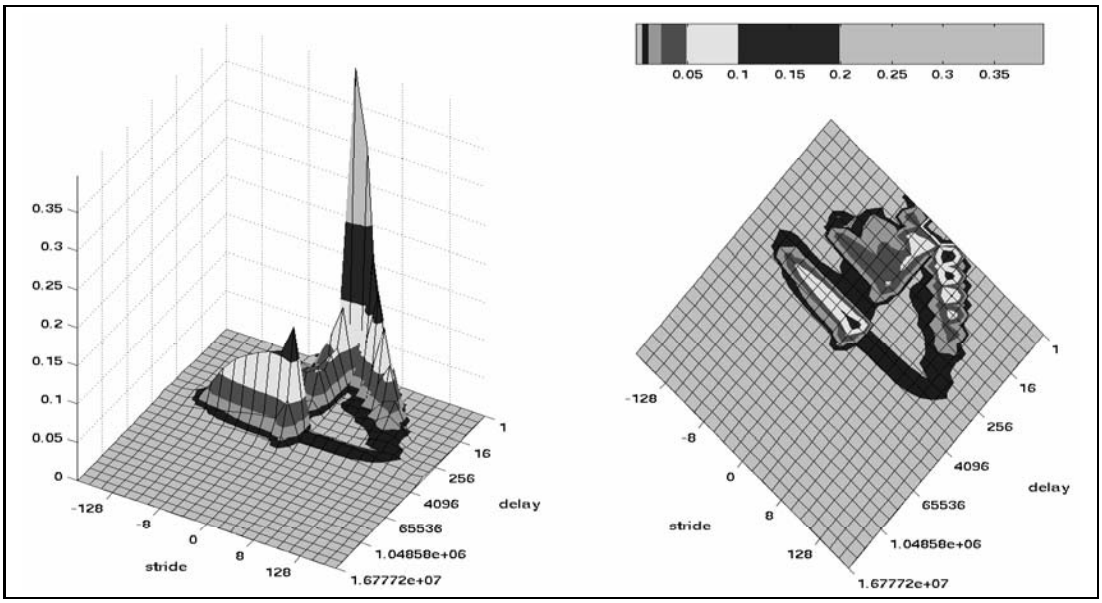


Figure 8.1: Locality surface for the instruction trace of *twolf*.

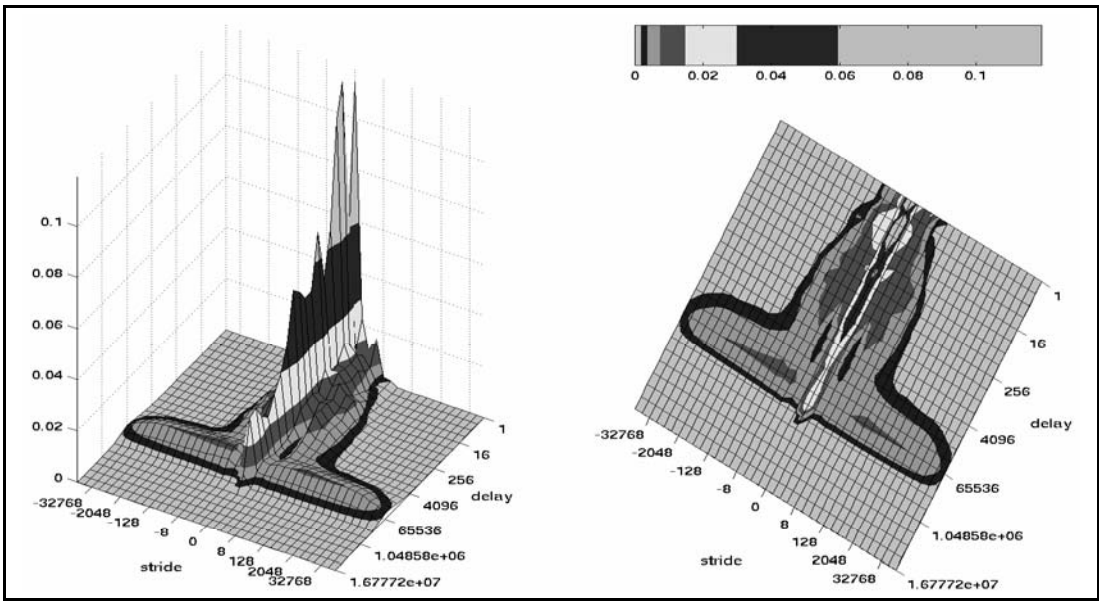


Figure 8.2: Locality surface for the data trace of *twolf*.

surface for the data trace of *twolf* consists of mainly temporal locality, with some random features around 64 Kwords.

The effective working set sizes and and effective memory ranges for the two traces is significantly different. For the instruction trace, the effective working set size is 4 Kwords and the effective memory range is 128 words. For the data trace, the effective working set size is 256 Kwords and the effective memory range is 128 Kwords. These differences allow us to determine if a particular synthetic model is effective across a range of effective working set sizes and effective memory ranges.

Using the appropriate statistics gained from these two traces, we generated a stream of references 10 million long using each of the six models. The locality surfaces for each of these twelve synthetic traces are shown in the next section and should be compared wtih Figures 8.1 and 8.2. The closer the surfaces are to each other and the closer the cache results are, the better the model.

8.3 Comparing Locality Surfaces

We now compare the locality surface for the traces generated by each of the models with the locality surface for the original traces from which we extracted model statistics. If a model is accurate, the locality surface for the model looks the same as the locality surface for the real trace.

8.3.1 Independent Reference Model

First we examine the Independent Reference Model. The locality surface from the trace IRM generated for the *twolf* instructions is in Figure 8.3, and the locality surface for the data is in Figure 8.4. Compare these surfaces to Figures 8.1 and 8.2.

First we note that Figure 8.3 has neither the looping structures nor the sequential

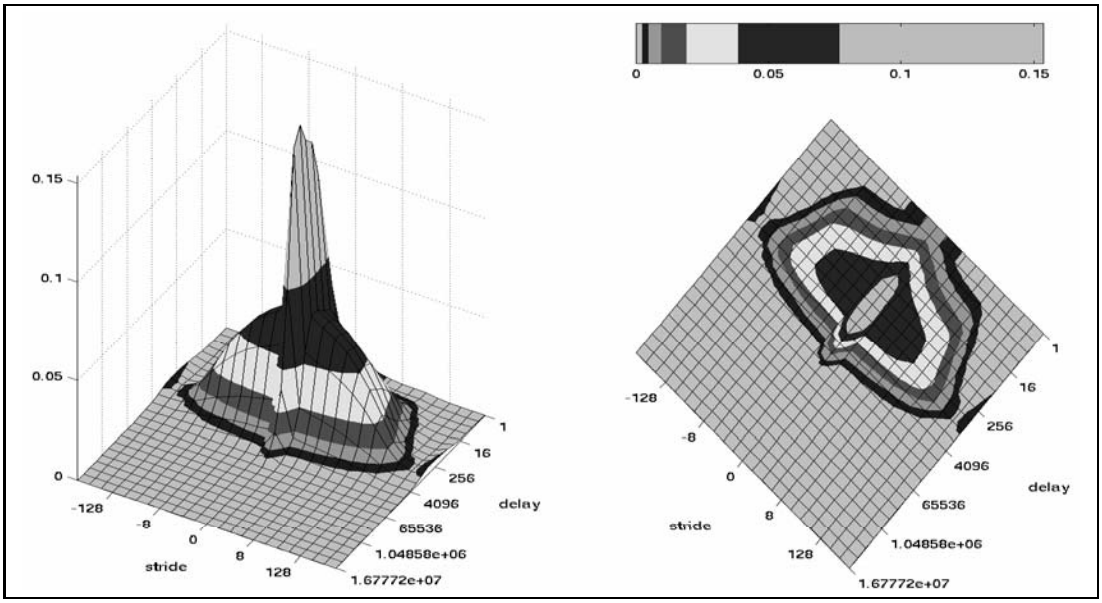


Figure 8.3: Locality surface for the references generated by the Independent Reference Model for the instruction trace of *twolf*. Compare with Figure 8.1.

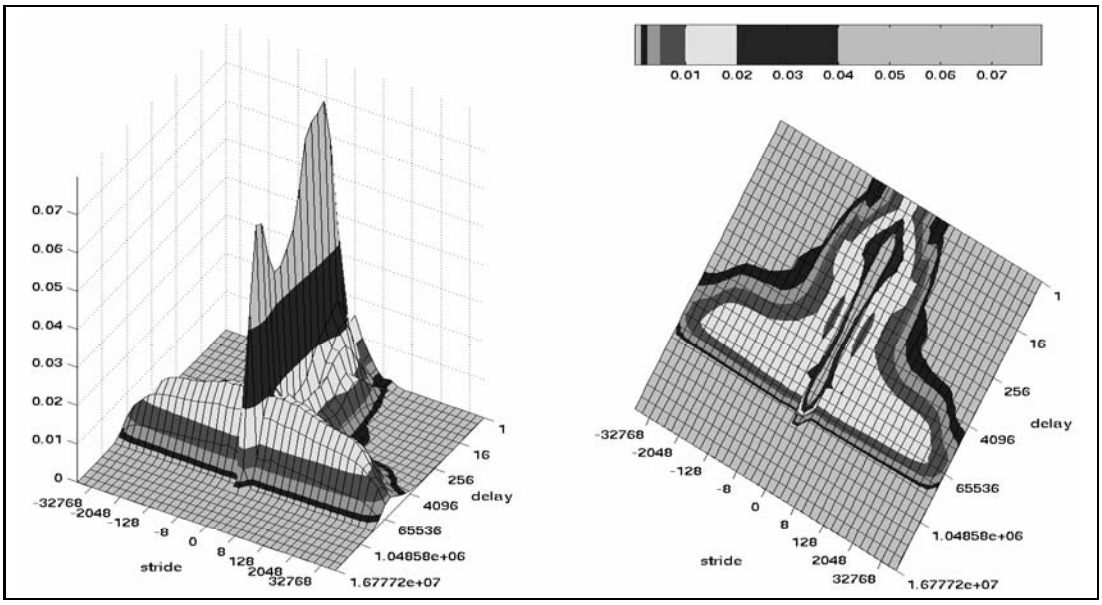


Figure 8.4: Locality surface for the references generated by the Independent Reference Model for the data trace of *twolf*. Compare with Figure 8.2.

ridge prominantly displayed in Figure 8.1. Figure 8.4 does look quite similar to Figure 8.2 in shape. But we see a significant difference when examining the top of the temporal ridge–the shape and overall heights are different. This difference is more evident if we directly compare the $stride = 0$ axis of the locality surfaces on a separate 2-D graph. Figure 8.5 does this for the instruction fetches of the real trace and the associated IRM generated trace; Figure 8.6 does this for the data reads and writes. It is easy to see from Figures 8.5 and 8.6 the differences along the $stride = 0$ axis.

The biggest advantage of the IRM is that it creates an effective working set size similar in size to the real trace. For the real instruction trace, the effective working set size is 4 Kwords and for the IRM instruction trace, the effective working set size is 16 Kwords. For both the real and the IRM data trace, the effective working set size is 128 Kwords. The effective memory range is further off for the instruction trace. For the real trace, the effective memory range is 128 words, but for the IRM instruction trace, it is 2 Kwords. For both the real and the IRM data trace, the effective memory range is 128 Kwords.

It appears that the IRM is better at approximating the effective working set size and effective memory range for larger values than for smaller. The only locality feature that the IRM appears able to preserve is a random hump. Even for traces with large working set sizes and memory ranges, the IRM does not preserve even such basic locality features as the temporal spike.

8.3.2 Stack Model

Next is the Stack Model. The locality surface this model generated for the instruction trace of *twolf* is in Figure 8.7; the locality surface for the data trace of *twolf* is

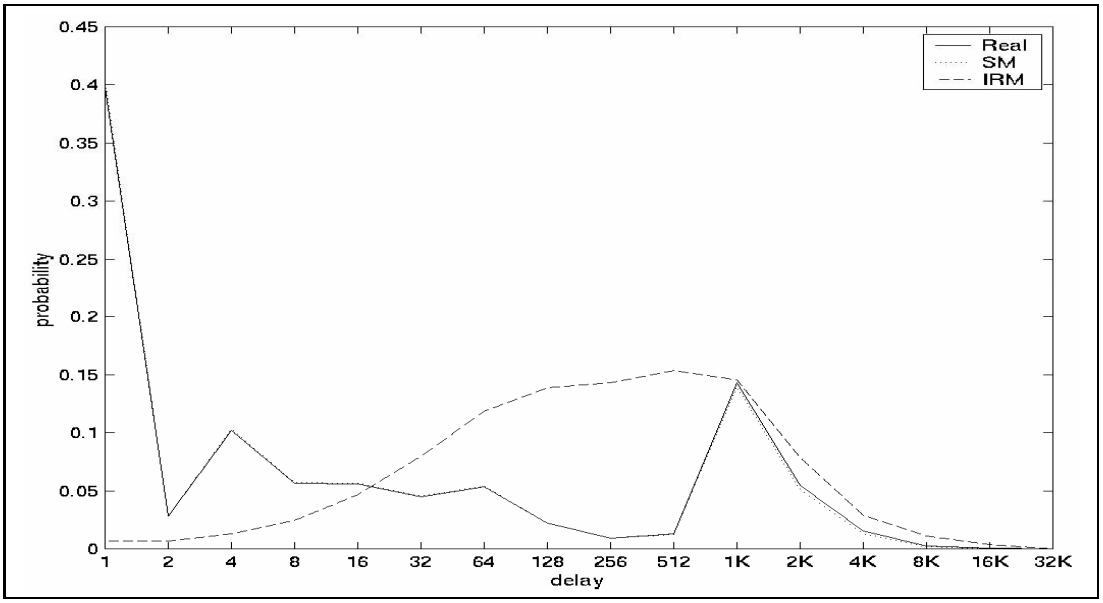


Figure 8.5: Duplicates of the $stride = 0$ axes of the locality surfaces for the original instruction trace of *twolf*, the Independent Reference Model, and the Stack Model. Notice how the lines for the original trace and the Stack Model are very close.

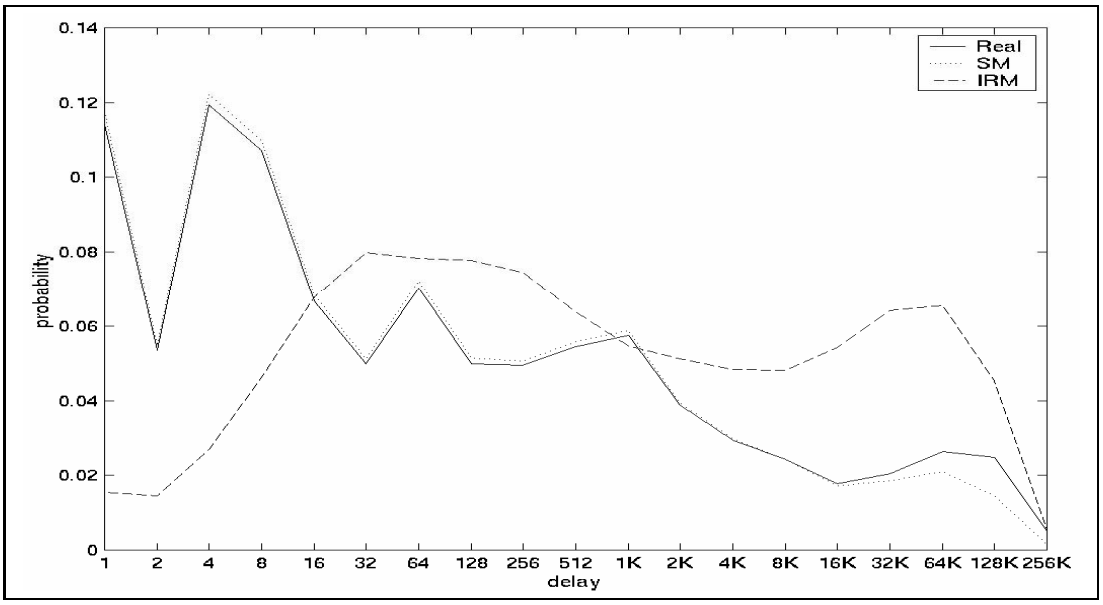


Figure 8.6: Duplicates of the $stride = 0$ axes of the locality surfaces for the original data trace of *twolf*, the Independent Reference Model, and the Stack Model. Notice how the lines for the original trace and the Stack Model are quite close.

in Figure 8.8. Compare with Figures 8.1 and 8.2.

This model appears to be extremely accurate along the $stride = 0$ axis. Figure 8.5 shows the $stride = 0$ axis from the instructions of the real trace and the Stack Model. The lines for the real trace and the stack model are almost identical. Figure 8.6 shows the $stride = 0$ axis from the data reads and writes of the real trace and the Stack Model. The lines for the real trace and the Stack Model are again quite close.

While both the height and shape of the temporal ridge appears very close for the stack model, there are no other features at all on the Stack Model's locality surfaces. The effective memory range for both Stack Model traces is zero. We do not see the sequentiality or looping of the *twolf* instruction fetches, nor the random lump at a delay of 64 Kwords of the *twolf* data. While the temporal locality is extremely accurate, we are missing the spatial locality. Notice, however, that the effective working set sizes match with those of the real trace.

8.3.3 Partial Markov Model

Next we look at the Partial Markov Model. The locality surface this model generated for the instructions of *twolf* is in Figure 8.9, while Figure 8.10 holds the locality surface for the data. Again, compare these surfaces to Figures 8.1 and 8.2.

The Partial Markov Model attempts to generate sequential references similar to the sequential portions of the real trace. However, comparing Figures 8.9 and 8.1, we see that the sequential ridge generated by the model is nothing like the ridge seen for the real trace. Most sequential runs in the synthetic trace are less than 4 references long. The real trace of the data of *twolf* does not contain any sequential ridge at all, yet the Partial Markov Model has the beginnings of such a ridge. It

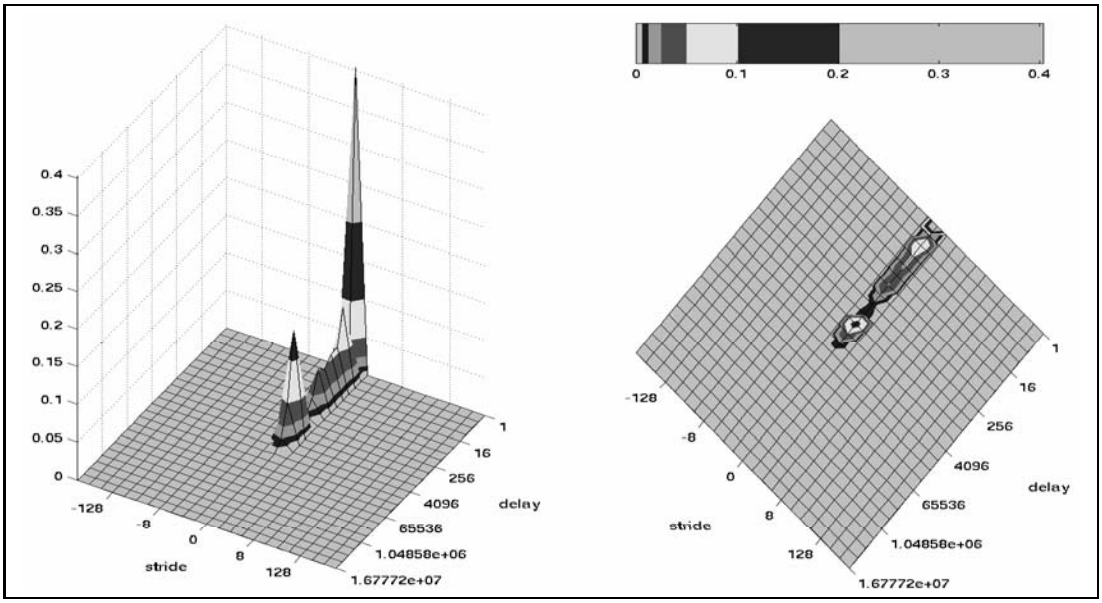


Figure 8.7: Locality surface for the references generated by the Stack Model for the instruction trace of *twolf*. Compare with Figure 8.1.

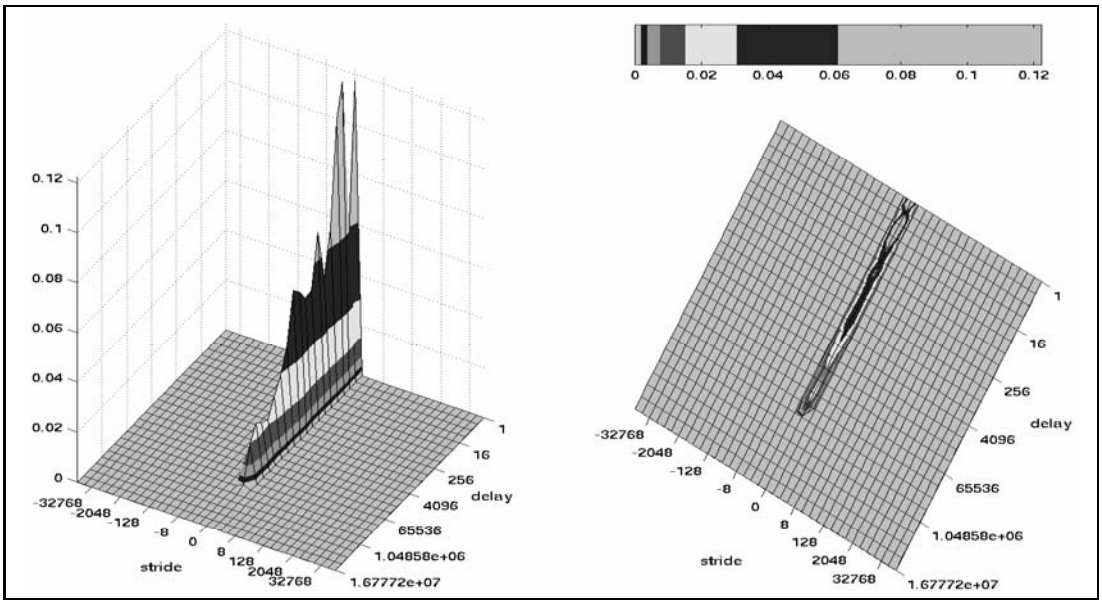


Figure 8.8: Locality surface for the references generated by the Stack Model for the data trace of *twolf*. Compare with Figure 8.2.

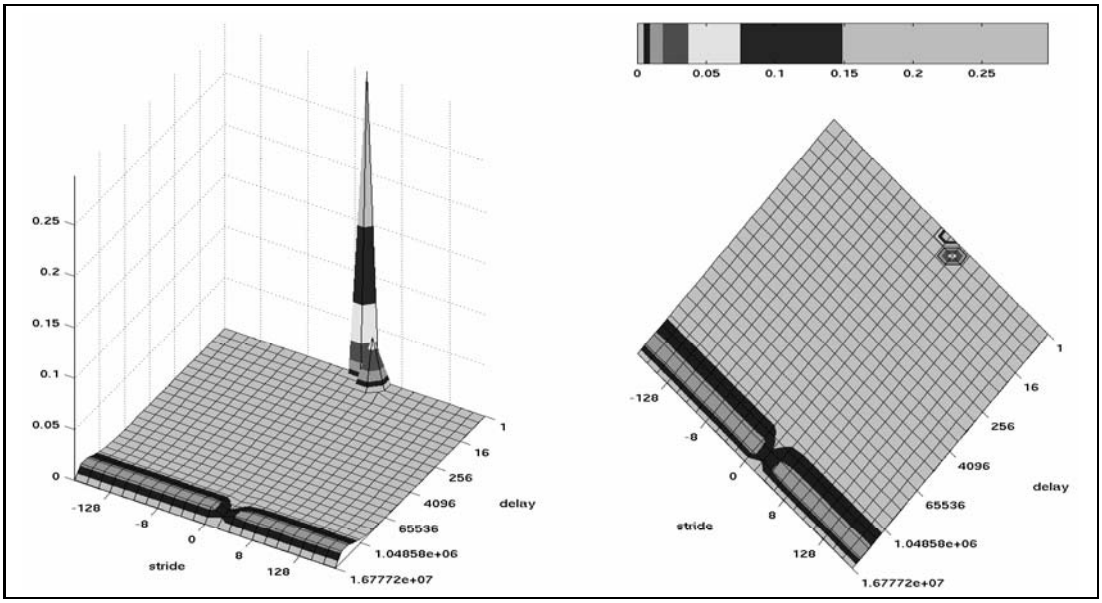


Figure 8.9: Locality surface for the references generated by the Partial Markov Model for the instruction trace of *twolf*. Compare with Figure 8.1.

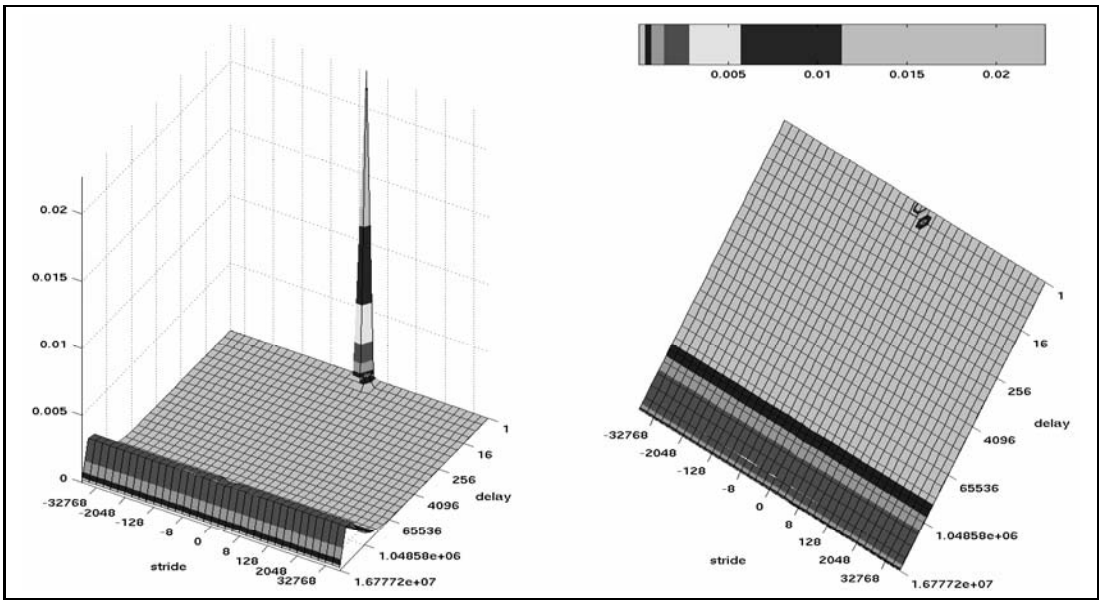


Figure 8.10: Locality surface for the references generated by the Partial Markov Model for the data trace of *twolf*. Compare with Figure 8.2.

also has some random references at a much larger delay than exhibited by the real trace, meaning that the effective working set size of the model is much larger than the target effective working set size of the real trace.

Since the effective working set sizes for the Partial Markov Model traces are nearly equivalent and so much larger than for the real traces (8 Mwords for the instruction trace and 16 Mwords for the data trace), we guess that the effective working set size is more closely related to the parameters of the random number generator used than the original trace. The effective memory range does differ between the two Partial Markov traces (128 Kwords for the instruction trace and 1 Mwords for the data trace). We guess that this is relative to how many random references are generated by the model, i.e. the length of time spent in the random state of the model versus the sequential state.

In general, the Partial Markov Model does not even come close to representing the real trace in any respect. The only feature from the locality surface of the real trace that even begins to appear on the Partial Markov Model locality surface is the sequential ridge, and that is woefully inadequate. Even the temporal spike is non-existant on either Partial Markov Model locality surface.

8.3.4 Distance Model

The locality surface of the trace generated by the Distance Model for the *twolf* instructions is in Figure 8.11, and the locality surface for the data of *twolf* is in Figure 8.12. Compare these figures to Figures 8.1 and 8.2.

Immediately we notice the lack of effective working set size in the traces generated by the Distance Model. For both instruction fetches and data reads and writes the

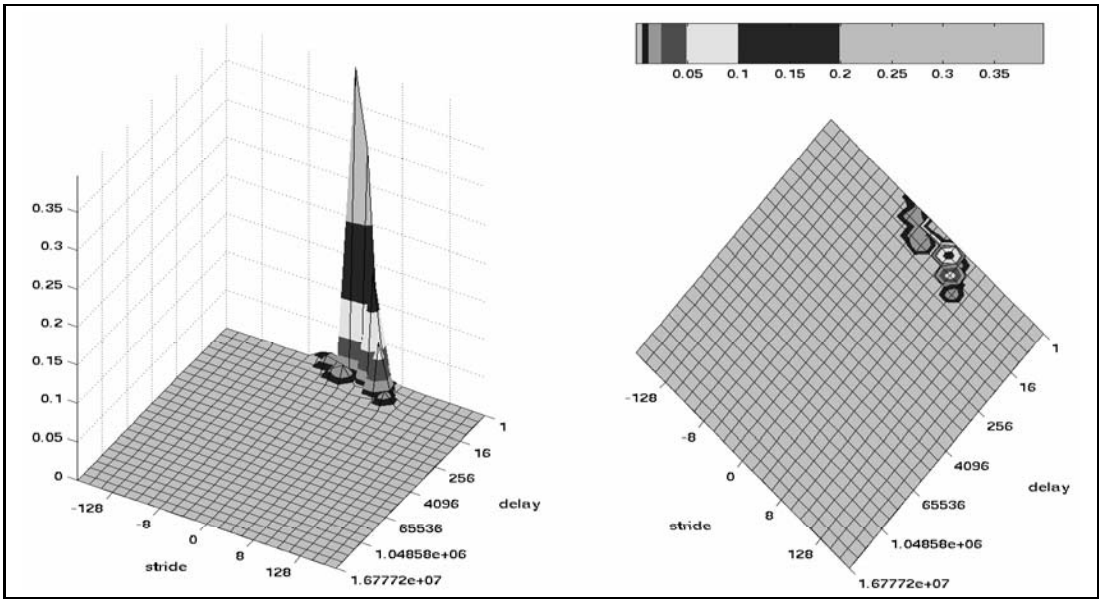


Figure 8.11: Locality surface for the references generated by the Distance Model for the instruction trace of *twolf*. Compare with Figure 8.1.

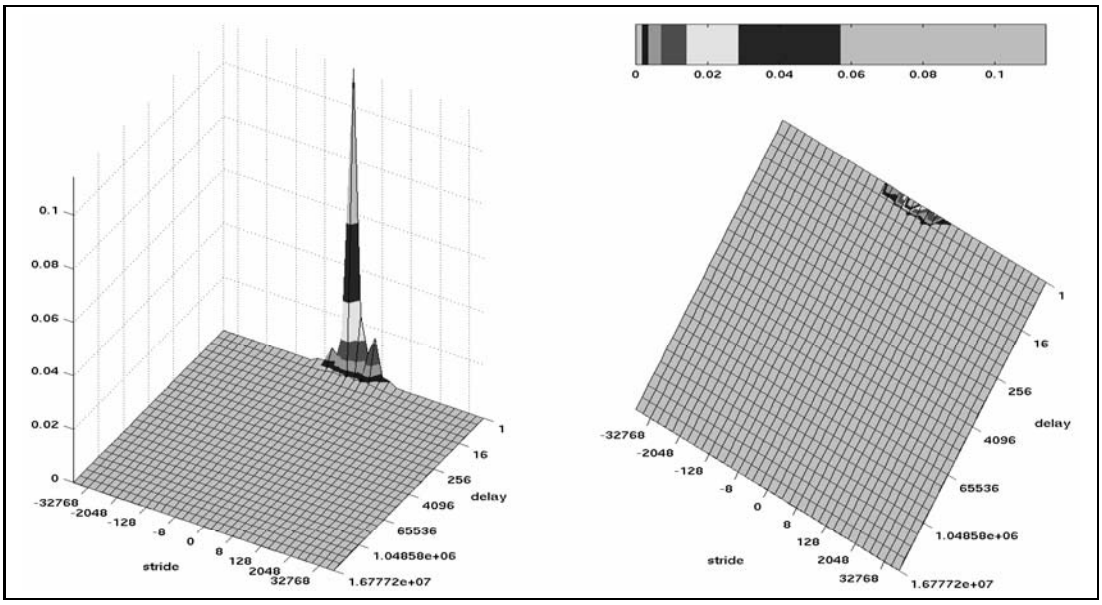


Figure 8.12: Locality surface for the references generated by the Distance Model for the data trace of *twolf*. Compare with Figure 8.2.

maximum delay shown on the locality surface is less than 8 words long, a very small effective working set size.

Since the Distance Model focuses entirely on the strides where the delay equals one, it is interesting to examine its accuracy along this axis alone. Figures 8.13 and 8.14 show the curves from the $delay = 1$ axis of the locality surfaces for several traces. Figure 8.13 shows the instruction fetches traces and Figure 8.14 shows the data reads and writes traces. Notice that the curves for the real traces and the Distance Model traces are indistinguishable in both figures. This demonstrates the accuracy of the Distance Model along the $delay = 1$ axis.

However, the effective memory range is still significantly off. For the instruction trace, the Distance Model has an effective memory range of 8 words, rather than the 128 words of the real trace. For the data trace, the Distance Model has an effective memory range of 16 words, rather than the 128 Kwords of the real trace. In summary, the Distance Model is useful for recreating the $delay = 1$ relationships, but little else.

8.3.5 Distance-Strings Model

The locality surface generated by the Distance-Strings Model for the *twolf* instructions is in Figure 8.15, and the locality surface for the data of *twolf* is in Figure 8.16. Compare these figures to Figures 8.1 and 8.2.

This model attempts to retain all the advantages of the Distance Model and add appropriate sequentiality. Comparing the locality surfaces for the real trace fetches and the model fetches, the Distance-Strings model appears to have failed in both respects. While the shape along the $delay = 1$ axis is still similar between the two instruction trace locality surfaces, it is nowhere near as clear, and the heights are

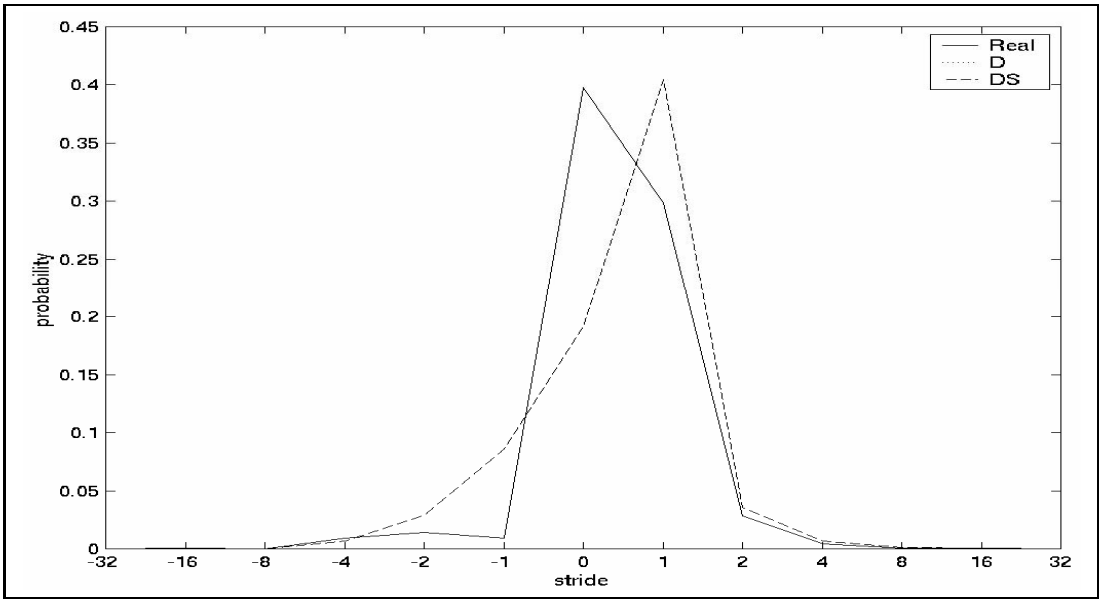


Figure 8.13: Duplicates of the $delay = 1$ axes of the locality surfaces for the original instruction trace of *twolf*, the Distance Model, and the Distance-Strings Model. Notice how the lines for the original trace and the Distance Model are the same.

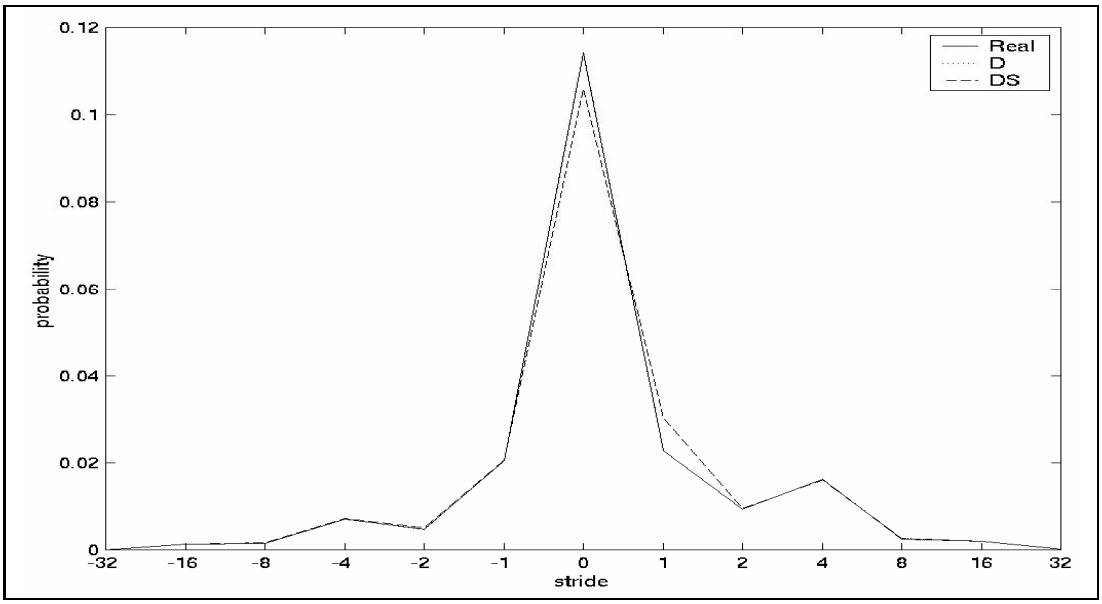


Figure 8.14: Duplicates of the $delay = 1$ axes of the locality surfaces for the original data trace of *twolf*, the Distance Model, and the Distance-Strings Model. Notice how the lines for the original trace and the Distance Model are the same.

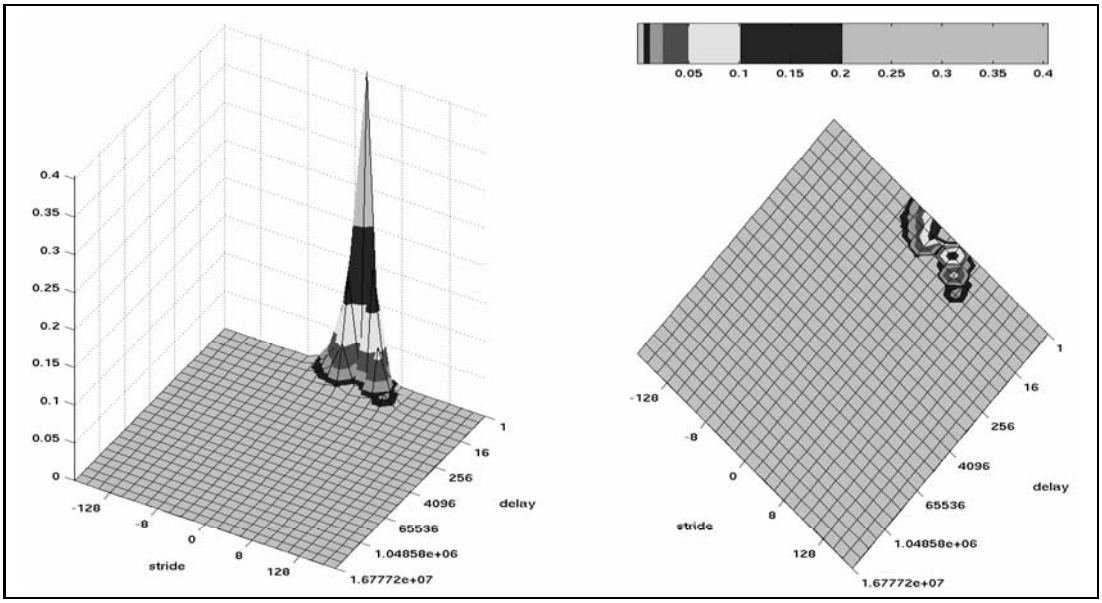


Figure 8.15: Locality surface for the references generated by the Distance-Strings Model for the instruction trace of *twolf*. Compare with Figure 8.1.

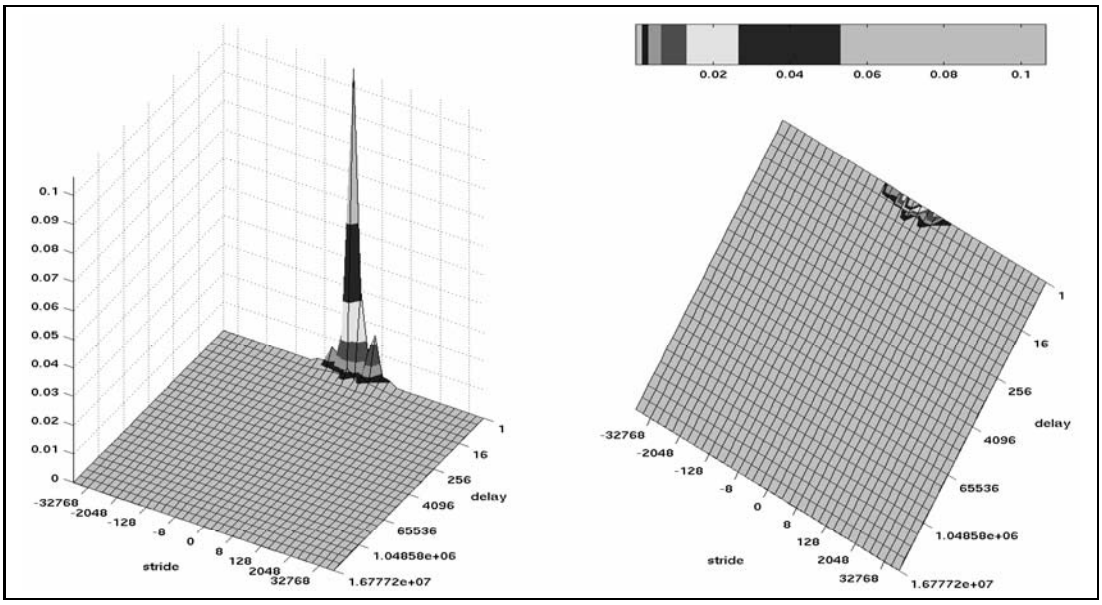


Figure 8.16: Locality surface for the references generated by the Distance-Strings Model for the data trace of *twolf*. Compare with Figure 8.2.

now significantly off. This can be more easily seen by comparing the $delay = 1$ axis of the locality surfaces, shown in Figure 8.13.

Figure 8.14 shows the $delay = 1$ axis of the data locality surfaces. Here the Distance-Strings Model is still quite close to the line for the real trace, probably because the real trace has no significant sequentiality. Since sequentiality is the only difference between the Distance Model and the Distance-Strings Model, the locality surfaces in Figures 8.16 and 8.12 look almost the same.

As with the Distance Model, the effective memory range is significantly off. Overall, we say the Distance-Strings Model loses some of the accuracy the Distance Model had, without any benefit.

8.3.6 Random Walk Model

The locality surface of the trace generated by the Random Walk Model for the instruction trace of *twolf* is in Figure 8.17, and the locality surface for the data trace of *twolf* is in Figure 8.18. Compare these figures with Figures 8.1 and 8.2.

The Random Walk Model appears to create a tall slice along the temporal axis and a random hump. For the data trace of *twolf*, it also appears to have created a few delayed sequential references (see Section 3.1.5). However, there is almost no data along the $delay = 1$ axis and thus no temporal spike. The random hump for the instruction Random Walk trace appears at the same delay as the loop structure in the original instruction trace. However, converting a loop into random references cannot be advantageous. The random hump seen on the original data trace is at about 64 Kwords. Unfortunately, there is little information at 64 Kwords on the Random Walk data trace.

The effective working set size for the Random Walk Model is one order larger

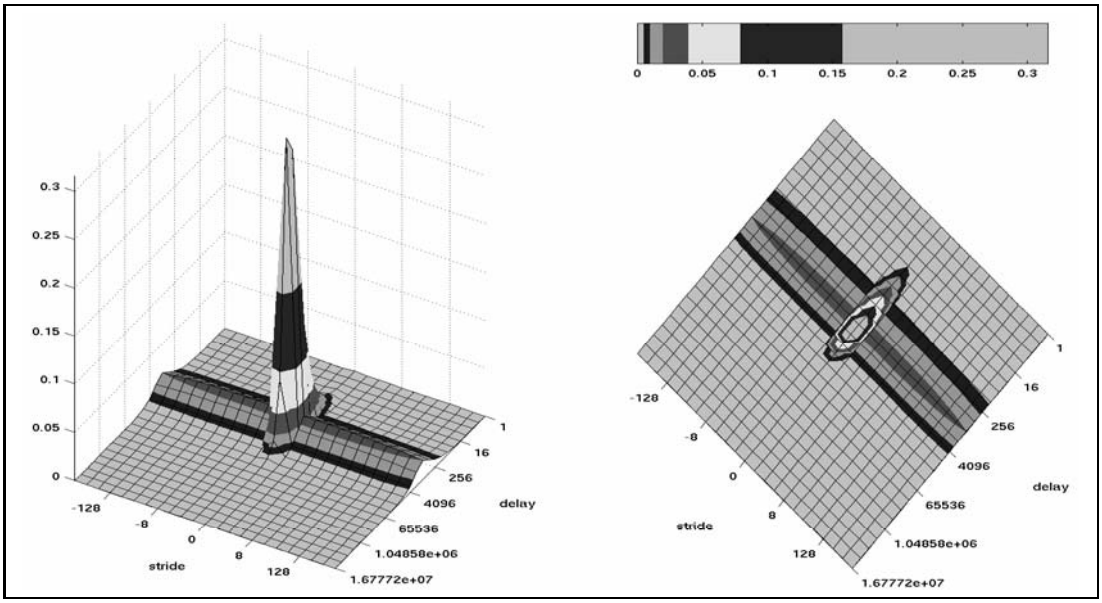


Figure 8.17: Locality surface for the references generated by the Random Walk Model for the instruction trace of *twolf*. Compare with Figure 8.1.

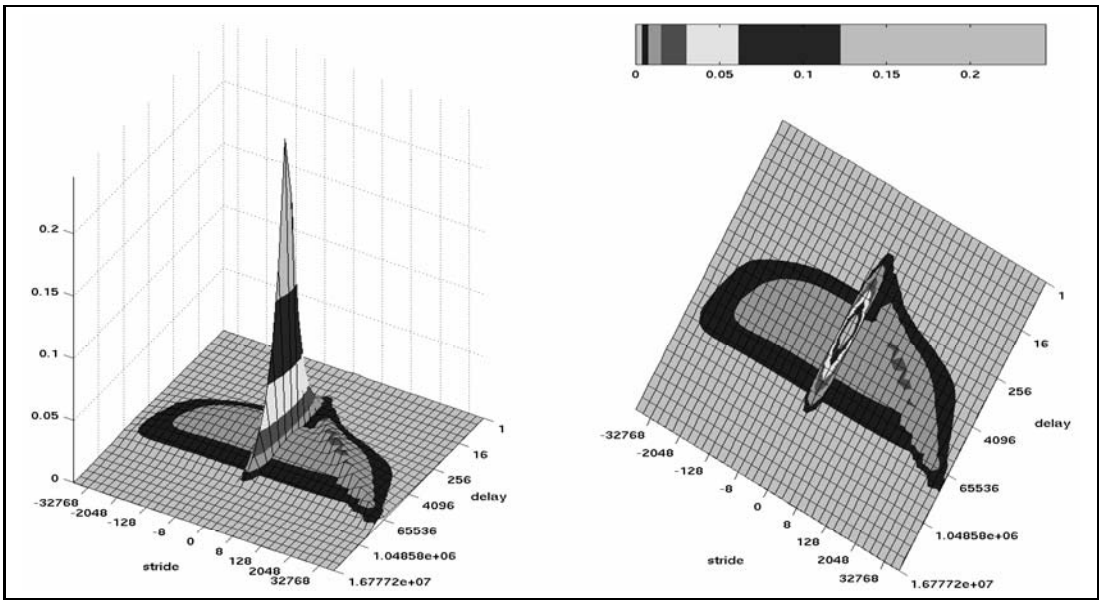


Figure 8.18: Locality surface for the references generated by the Random Walk Model for the data trace of *twolf*. Compare with Figure 8.2.

than the original effective working set size for the instruction trace and one order smaller for the data trace. It appears that the Random Walk Model does much better than either the Distance or Distance-Strings Models at aproximating the effective working set size of the original trace. The effective memory range is the same for the Random Walk data trace as the original *twolf* data trace. However, the Random Walk instruction trace is significantly different than the original instruction trace. Figure 8.17 does not show it, but the effective memory range is 8 Kwords. The original effective memory range is 128 words.

The Random Walk Model appears to approximate the effective working set size reasonably, but does not adequately represent either the temporal axis or the $delay = 1$ slice of the locality surface. It also does not replicate loops or the sequential ridge. We speculate that one of its two parameters controls the effective working set size and the other controls the magnitude of the delayed sequential ridge.

8.4 Comparing Cache Simulation Results

We now examine the cache simulation results for the real traces with each trace generated by one of the synthetic models. This further validates our argument that the locality surface and cache simulation results are related. We selected cache configurations that are typical for L1 caches in many systems today. Specifically, we simulated caches with sizes 32 Kbytes and 64 Kbytes, associativities from 1-way to 8-way, and all with 32-byte lines.

The cache simulation results for the original *twolf* traces and all six models are in Figures 8.19 and 8.20. Tables 8.1 and 8.2 show the percent error of each of the six models versus the original trace. We first notice that the errors are very large in most cases. The results from the Distance Model and the Distance-Strings Model

are very similar to each other, especially for the data where the real trace had no sequential features. The Partial Markov Model has the worst miss rate, probably because it relies quite heavily on random references.

The Independent Reference Model, the Stack Model, and the Random Walk Model are the only models that give us much variability between the different cache configurations. That is probably because these were the models that created remotely accurate effective working set sizes. Yet the percent errors are still quite large.

The only real trace and model combination that has any reasonable error for all the cache simulation results is for the data of *twolf* and the Stack Model. If we remember from the locality surfaces in Figures 8.2 and 8.8, the only real features of the real trace were the temporal locality. As we saw earlier, the Stack Model reproduces the temporal locality very well and only falls short in dealing with spatial locality. For traces such as the data of *twolf*, where there is no significant spatial locality, the Stack Model would perform quite well.

This also further validates the conclusions from Chapters 6 and 7, that temporal locality is sufficient for fully associative caches where $C_l = g$. However, if the spatial locality does not match, there are errors for Cases Two through Four. These errors should be minimized for traces with little spatial locality. Recall that the Stack Model recreated the temporal locality almost exactly and only failed in the matter of spatial locality. Now, we see that for the trace with little spatial locality (i.e. the data trace of *twolf*), the Stack Model gave reasonable approximations of the Case Four cache results for the real trace. However, the trace with significant spatial locality (i.e. the instruction trace of *twolf*) was not approximated well by the Stack Model.

Interestingly enough, the errors for the Random Walk Model are very different

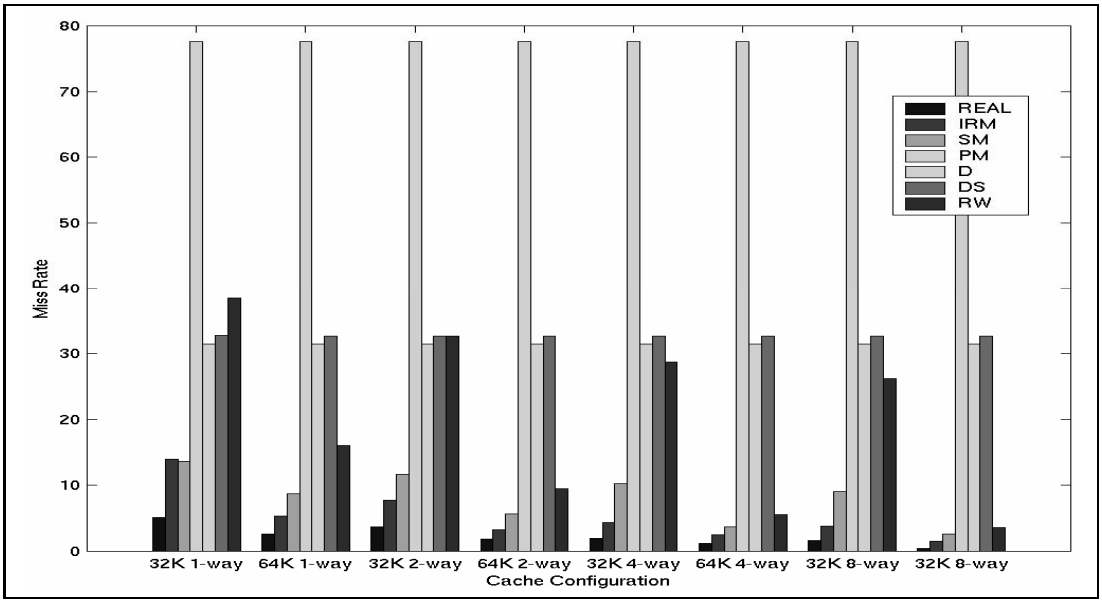


Figure 8.19: Cache simulation results for the original instruction trace of *twolf* and the references generated by the six studied models. The left-most bar in each group indicates the real trace results, the next bar the IRM trace results, etc.

cache configuration	IRM	SM	PM	D	DS	RW
32K, 1-way	172%	167%	1,413%	513%	537%	648%
64K, 1-way	101%	226%	2,807%	1,077%	1,123%	503%
32K, 2-way	106%	210%	1,967%	738%	771%	851%
64K, 2-way	75%	204%	4,051%	1,580%	1,645%	409%
32K, 4-way	126%	428%	3,881%	1,511%	1,574%	1,373%
64K, 4-way	109%	207%	6,315%	2,496%	2,597%	361%
32K, 8-way	138%	469%	4,751%	1,863%	1,939%	1,537%
64K, 8-way	206%	429%	15,741%	6,310%	6,559%	627%

Table 8.1: Errors for the cache simulation results of the original instruction trace of *twolf* versus the cache simulation results of the traces generated by each of the six studied models. The error is calculated by taking the difference between the model cache results and the real trace cache results and dividing the result by the real trace cache results.

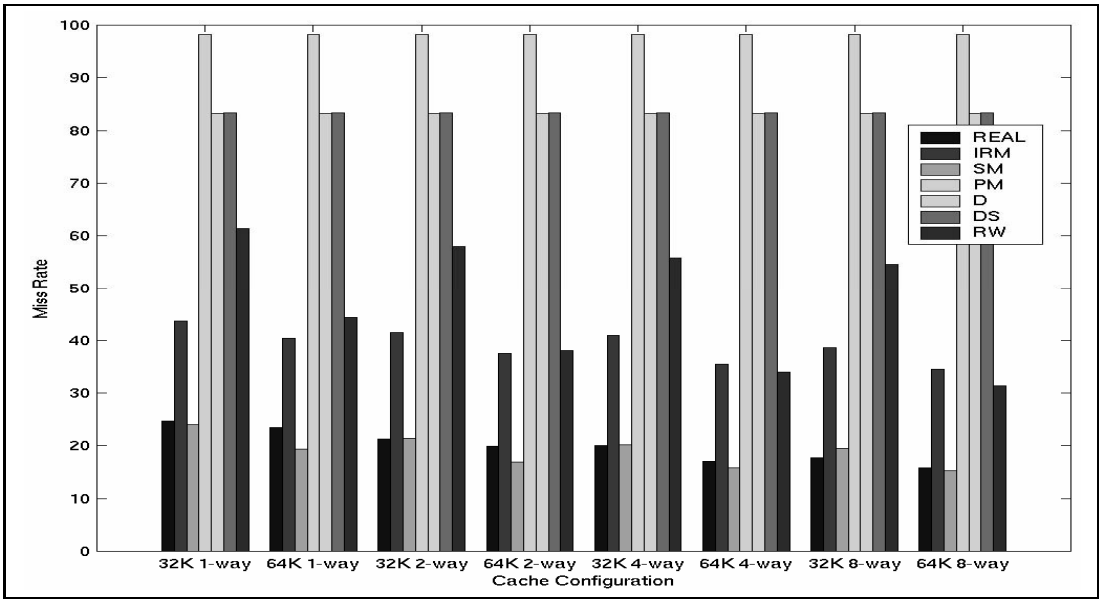


Figure 8.20: Cache simulation results for the original data trace of *twolf* and the references generated by the six studied models. The left-most bar in each group indicates the real trace results, the next bar the IRM trace results, etc.

cache configuration	IRM	SM	PM	D	DS	RW
32K, 1-way	76.4%	-3.0%	297%	236%	237%	147.9%
64K, 1-way	71.7%	-17.4%	318%	254%	254%	88.4%
32K, 2-way	95.3%	0.7%	363%	278%	293%	172.9%
64K, 2-way	88.3%	-15.2%	393%	317%	318%	90.8%
32K, 4-way	99.5%	0.6%	391%	316%	317%	178.7%
64K, 4-way	107.6%	-7.7%	476%	388%	389%	98.7%
32K, 8-way	117.9%	9.8%	454%	370%	370%	207.4%
64K, 8-way	118.5%	-3.5%	523%	428%	428%	99.2%

Table 8.2: Percent errors for the cache simulation results of the original data trace of *twolf* versus the cache simulation results of the traces generated by each of the six studied models. The error is calculated by taking the difference between the model cache results and the real trace cache results and dividing the result by the real trace cache results.

from the errors reported by Thiebaut et al. in their paper [79]. Recall from Section 7.5.2 that their reported maximum error for set associative caches was 25%. We acknowledge that the difference in errors may be due to an error in our implementation of the Random Walk Model. However, we also notice that the single trace used by Thiebaut et al. for set associative simulations was only 433,152 references long [79]. This is significantly shorter than our 10,000,000 long traces. We think it more likely that the difference in errors is attributable to the difference in traces; perhaps the Random Walk Model has much smaller errors for shorter traces or the particular trace chosen.

8.5 Synthetic Traces from the Locality Surface

It would be nice if we could develop a synthetic trace creation mechanism based on the locality surface itself. If we could create a trace from a locality surface such that the synthetic trace had the same locality surface, the synthetic trace would have the same cache simulation results as the original trace. We could then store the locality surface rather than the large trace for significant space savings. We could also create traces of the future by creating locality surfaces that have the features we believe will be on the locality surfaces of the next generation traces.

We have spent considerable time investigating probabilistic methods for using the locality surface to create a synthetic trace. We have found that we can either duplicate the temporal locality (as the Stack Model did) or the $delay = 1$ locality (as the Distance Model did). The trick is to combine these two.

We believe the way to do this is to take advantage of the fact that temporal locality does not involve the relative values of the elements in the trace, it only considers the delay until each value is repeated. The $delay = 1$ locality slice does

not consider when a value may be repeated, it only considers the difference between the given value and the next value.

We assume that, in addition to the locality surface, we also know the length of the desired trace. We first use the $\mathcal{L}(0,1)$ entry on the locality surface to determine how many immediately repeating elements there are in the trace and subtract this from the length of the trace. We therefore create the base of the desired trace. This simplifies many calculations. We know from Theorem 6.7 that the immediately repeating elements may be added in at any place in the trace when we are done. We let *len* represent the length of the trace base.

We know from the locality surface how many unique elements are in the trace (see Equation 6.5) and call this *uniq*. We create *uniq* variables and rearrange them in a list of length *len* until the temporal locality is correct. There may be many arrangements that match the temporal locality.

We then take each arrangement and assign values to the variables such that the resulting trace has $delay = 1$ locality that matches the desired $delay = 1$ slice. Note that we must not allow any of the variables to be equal to any of the other variables or the temporal locality is no longer correct. Note also that for a given arrangement of *uniq* variables in a *len* long trace there may be no assignments of values such that the result matches the $delay = 1$ data.

To reduce the time for this last step, we can extract the range between the largest and smallest values in the trace from the locality data. We do this by finding the largest absolute value stride in the locality data. The largest and smallest values in the trace have a stride/delay relationship between them at some point in the trace. This stride/delay relationship contains the largest absolute value stride. Using this range, we can limit the number of values to attempt to assign each variable.

We can also reduce the number of values that need to be tried by deciding that

we only need one string from each shift equivalence class. If we require the largest or the smallest value to be a constant, then we reduce the number of values to be tried and also reduce the number of shift equivalent strings found. For example, we may require that the smallest value in the final string is 1.

At this point we should have a trace with length *len* that has temporal locality and $delay = 1$ locality that matches the temporal and $delay = 1$ locality on the original locality surface. The maximum absolute value stride should also match. However, this is not sufficient to determine that all the locality data matches.

Example 8.1. *Let* $v = 3, 5, 4, 1, 4, 2, 4, 1, 4, 1$ *and* $w = 3, 5, 2, 5, 2, 5, 3, 5, 4, 1$. *Note that* $\sigma_{s=0}(B)$ *represents the temporal locality in bag* B *and* $\sigma_{d=1}(B)$ *represents the* $delay = 1$ *locality in bag* B.

For v *and* w *above,* $\sigma_{s=0}(\ell(v)) = \sigma_{s=0}(\ell(w))$ *and* $\sigma_{d=1}(\ell(v)) = \sigma_{d=1}(\ell(w))$. *Also, the minimum value in each string is 1 and the maximum value is 5.*

Therefore, v *and* w *have matching temporal locality, matching* $delay = 1$ *locality, and equivalent maximum absolute value strides. However,* $\ell(v) \neq \ell(w)$ *since* $(-3, 3) \in \ell(v)$ *and* $(-3, 3) \notin \ell(w)$. *Also,* $(2, 3) \in \ell(w)$ *and* $(2, 3) \notin \ell(v)$.

As seen in Example 8.1, just because a string has the same temporal locality and $delay = 1$ locality and has the same maximum absolute value stride does not mean that the rest of the locality data on the string matches. Note also that in Example 8.1 the two stride/delay relationships that are out of place would not be assigned to the same bin when the binned histogram is created, so the mismatch would also be present on the locality surface.

Therefore, all strings at this point should have the locality created for them and compared with the desired locality surface. Obviously for many locality surfaces this entire process would be extremely time consuming. However, since it is possible, it merely remains to find methods to shorten the time necessary for each step. The final result would be a synthetic trace that has the desired locality features and also accurate cache simulation results.

8.6 Summary

None of the models examined in this chapter were very accurate, as exhibited by both the locality surfaces and the cache simulation results. However, the locality surface also gave us information as to which models adequately reproduced what they intended. In that respect, IRM, the Distance Model, and the Stack Model all did well. The Distance-Strings Model lost the spatial locality accuracy of the Distance Model without gaining any benefit. The Partial Markov Model relied too heavily on random references and still did not produce a reasonable sequential ridge. It especially failed to model the data of *twolf*, which did not have a significant sequential ridge anyway. The Random Walk Model certainly reduced the original trace to two parameters, as desired, however the errors in cache simulation are unreasonable large.

We conclude that if the real trace has minimal spatial locality, it may be approximated fairly well by the stack model. In all other cases, none of these models are adequate. Unfortunately, as seen in Chapter 3, most workloads have significant spatial locality.

We also described a rough algorithm for using the locality surface itself to generate a synthetic trace. Due to the time involved, the algorithm is impractical at this point, however it does give a starting place for researchers interested in using the locality surface to create synthetic traces. We next discuss methods for improving the time to compute the locality surface using a parallel algorithm. Such an improvement in the time to compute may help make creating a synthetic trace from the locality surface more feasible.

Chapter 9

Speeding Up the Locality Program

One of the biggest disadvantages of using the locality surface is the time necessary to compute it. As mentioned in several places in previous chapters, computing the locality data for some inputs takes an unreasonable amount of time. Some of the traces used in this dissertation take many days, depending on the trace's locality. Traces with "good" temporal locality, i.e. most references are repeats of recently used references, are computed quite quickly. For example, the locality data for the instruction trace of *mcf* took less than 3 minutes to compute on a 2.2 GHz machine. The locality surface for the instructions of *mcf* is shown in Figure 9.1. The instructions of *mcf* have 57,174,298 references and only 16,170 unique references.

Workloads with poor locality, such as the data of *wupwise*, take weeks to compute. The data of *wupwise* has 51,477,244 references and 8,404,245 unique references. A 2.2 GHz machine took over 47 days to compute the locality surface for the data of *wupwise*, shown in Figure 9.2. This was the longest it took for any of the SPEC traces we analyzed. (All our locality surfaces are listed in Appendix B.) However, the randomly generated traces used in Section 5.3 would have taken even longer to compute using the sequential algorithm, since it is much longer (500 million refer-

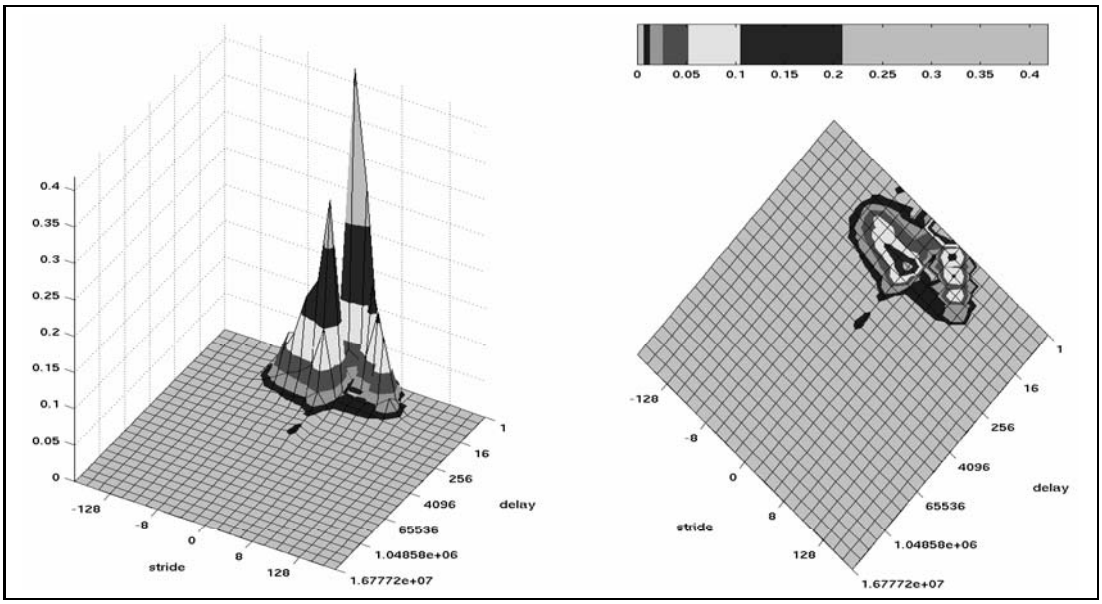


Figure 9.1: The locality surface for the instruction trace of *mcf*. It took less than 3 minutes to compute the data for this surface.

ences) with more unique references (7.9 million unique references). Clearly, a faster algorithm is necessary to make the locality techniques presented in this dissertation feasable and other associated surfaces practical to compute.

9.1 Sequential Algorithm

The naïve method for calculating the locality surface uses an LRU stack based algorithm as shown in the pseudo-code in Figure 9.3. The stack is ordered such that the most recently seen reference is always on the top. Each reference is compared with each element of the stack, beginning at the top of the stack. For each member of the stack, the stride and delay are calculated and recorded. If the current reference is discovered in the stack, the stack traversal finishes, no more strides and delays are computed, and the reference is removed from its position in the stack and reinserted at the top.

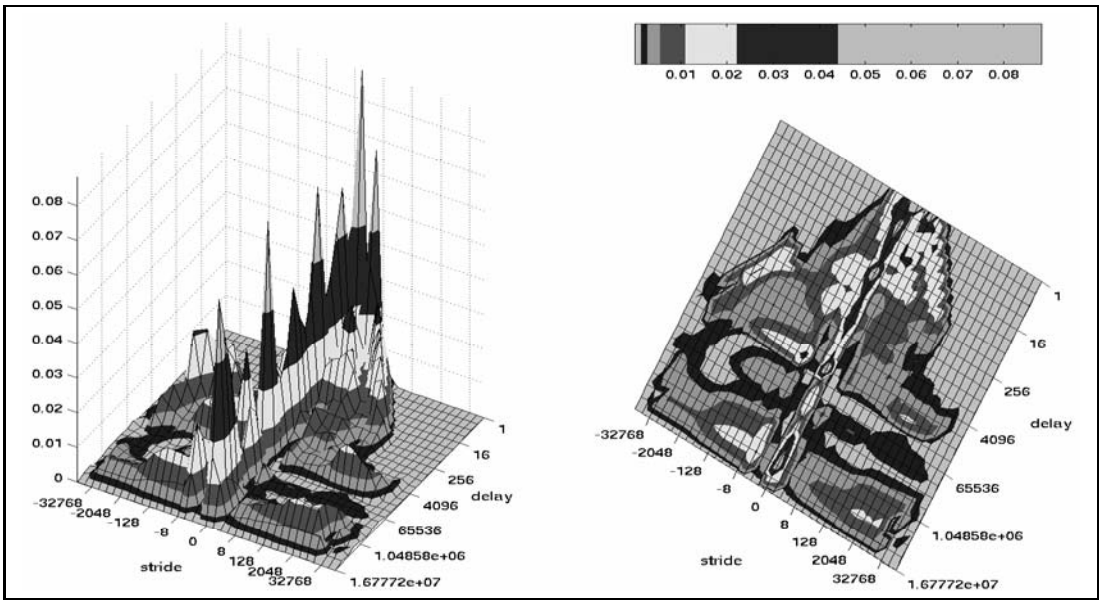


Figure 9.2: The locality surface for the data trace of *wupwise*. It took over 47 days to compute the data for this surface.

Calculating the locality surface in this manner is a sequential, $O(mn)$ algorithm, where m is the number of unique references and n is the total number of references. We may also write it as $O(na)$ where a is the average stack depth of the elements. For traces with great locality, a is very small. For traces with poor locality, a approaches m and the overall time is excessive. For random traces, not only does a approach m, but m aproaches n making the sequential algorithm essentially $O(n^2)$.

Other researchers have made efforts to speed up stack based algorithms [11, 78, 80]. However, none of these methods help compute the locality surface since the aim of such previous algorithms is merely to compute the stack depth of each reference. The other algorithms create tricks for skipping the stack traversal step of the naïve algorithm. Since the locality surface requires a calculation at each depth of the stack before the current reference is discovered, we cannot shortcut the stack traversal.

The only way to speedup the time to compute the locality surface is to split the

```
void main()
{
    ulong addr = GetReference();
    AddToStack(addr);
    int uniques = 1;
    int delay, stride;
    while (MoreReferences())
    {
        addr = GetReference();
        for (int i = 1; i <= uniques; i++)
        {
            stride = addr - stack[i];
            delay = i;
            MarkLocalityEvent(stride,delay);
            if (stride == 0)
            {
                RemoveFromStack(i);
                uniques--;
                break;
            }
        }
        AddToStack(addr);
        uniques++;
    }
}
```

Figure 9.3: Pseudo-code listing for the sequential locality algorithm. The function `GetReference()` returns the next references from the input trace. The function `AddToStack(addr)` pushes `addr` on the top of the stack. The variable `uniques` keeps a count of the number of unique references seen, equivalent to the current depth of the stack. The function `MoreReferences()` returns true if there are more references to be read from the trace. The function `MarkLocalityEvents(stride,delay)` records the input stride/delay relationship in either a histogram or binned histogram, as desired. The function `RemoveFromStack(delay)` removes the value at depth `delay` from the stack.

stack among several processors. The trick is to figure out a method to split the stack evenly such that each portion of the stack may be traversed simultaneously and such that updating the stack is simple and fast.

9.2 The Parallel Algorithm

The obvious way to split the stack among p processors is to put the top $1/p$ references in the stack on the first processor, the next $1/p$ references in the stack on the next processor, etc. However, this method does not meet our requirements. When the references are split among the processors according to location in the stack, it is difficult to process all the sections of the stack at the same time. If, for example, a reference was found halfway through the stack, then half the processors would need to throw away the results they calculated.

More importantly, stack update would be tricky and time consuming. If, for example, a reference was not in the stack, it would be added to the processor that has the top section of the stack. If a reference was found in the stack, it would be removed from that processor and reinserted on the processor containing the top section of the stack. Basically, every reference processed would add another reference to the processor containing the top of the stack. These added references would cause an increasing unbalance of the processors and the stack would need to be periodically resectioned and redistributed among the processors. This rebalancing would need to occur frequently, and remove any benefit of using a parallel algorithm.

We need an algorithm that allows us to split up the stack arbitrarily among the processors, i.e. an algorithm that does not require the processors to store the stack in order. We do this by requiring each processor to store each of its assigned

elements with the depth that element would be if an LRU stack were maintained. Processing a new reference now takes two steps, or phases.

In Phase I, each processor reads the reference from the file and determines if the reference is assigned to itself. If the reference belongs to that processor, the processor looks up the stack depth of that reference and broadcasts the depth to the other processors. If the reference has never been seen before, and hence no depth is stored, then a depth of infinity is broadcast. See Figure 9.4.

In Phase II, every processor now knows both the reference value and the depth. All the processors then simultaneously look at each reference it has stored. If the depth of the stored reference is less than the depth of the new reference, the stride and delay are computed and recorded and the stored depth is increased by one. When the processor that has the new reference assigned to it comes across that reference, the stride and delay are computed and stored and the stored depth is assigned a value of 1. If the new reference has never been seen before, then the processor that has it assigned adds the reference to its list with a depth of 1.

At the end of Phase II, the stack may be recreated by examining the references stored by each processor and their depths. See Figure 9.5. Note that we have achieved our goal. The stack elements may be stored on any processor, the section of the stack on a given processor may be processed at the same time as the sections on the other processors, and updating the stack is simple and fast.

9.2.1 Load Balancing

We wish to have the seen references evenly balanced across all the processors. However, the time to determine which processor has that reference assigned should be short, since it is done by every processor for every reference. As discussed above, we

```
void main() {
    int myrank, numprocs, depth, stride, delay, myuniques = 0;
    GetRankAndSize(&myrank,&numprocs);
    int shift = FindBestShift(numprocs);
    ulong addr = GetReference();
    int proc = (addr >> shift) & (numprocs - 1);
    if (proc == myrank) {
        myrefs[myuniques] = addr;
        mydepths[myuniques++] = 1;
    }
    while (MoreReferences()) {
        addr = GetReference();
        proc = (addr >> shift) & (numprocs - 1);
        if (proc == myrank) {
            depth = FindDepth(addr); // Phase I
            SendBroadcast(depth);
            for (int i = 0; i < myuniques; i++) { // Phase II
                if (mydepths[i] < depth) {
                    stride = addr - myrefs[i];
                    delay = mydepth[i]++;
                    MarkLocalityEvent(stride,delay);
                }
                if (mydepths[i] == depth) {
                    MarkLocalityEvent(0,depth);
                    mydepth[i] = 1;
                }
            }
            if (depth == INFINITY) {
                myrefs[myuniques] = addr;
                mydepths[myuniques++] = 1;
            }
        } else {
            depth = ReceiveBroadcast(); // Phase I
            for (int i = 0; i < myuniques; i++) { // Phase II
                if (mydepths[i] < depth) {
                    stride = addr - myrefs[i];
                    delay = mydepth[i]++;
                    MarkLocalityEvent(stride,delay);
                }
            }
        }
    }
    SumLocalityTables();
}
```

Figure 9.4: Pseudo-code listing for the basic parallel locality algorithm. The function `GetRankAndSize` retrieves the rank for this processor and the total number of processors. The array `myrefs` contains all the references assigned to this processor that have been seen. The array `mydepth` contains the stack depth of each reference.

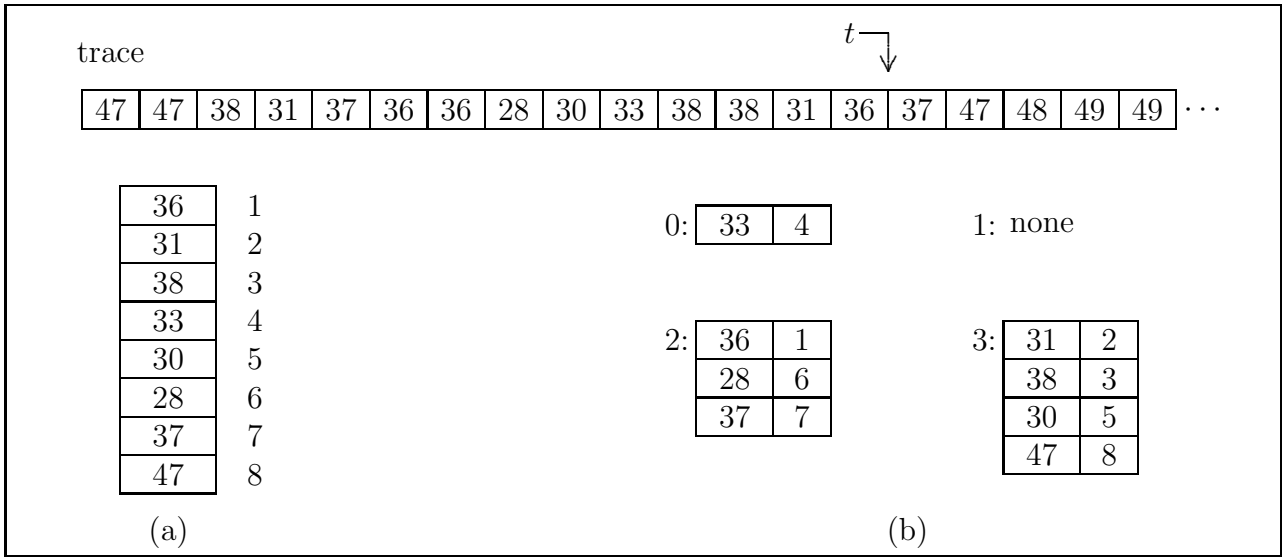


Figure 9.5: Time t indicates just after the reference valued 36 is processed, but before the reference valued 37 is read. Part (a) shows the LRU stack at time t during the sequential algorithm. Note that only the value of the reference is stored; the depth of the stack can be calculated as the stack is traversed. Part (b) shows the stack at time t, split between 4 processors, in the parallel algorithm. Note that both the value of the reference and its depth in the LRU stack are stored. In fact, the LRU stack can be recreated from the information stored in Part (b).

cannot assign references to processors by location in the stack, since that changes constantly. We decide to assign processors by performing a mod of the value of the reference with the number of processors. We have determined that right shifting the reference value one or more times changes the load balance.

For our input traces, it takes less than one or two minutes to read through the entire trace and determine the optimal number of bits to shift to give the best load balance. For some traces, the time saved is only ten to thirty minutes. However, for other traces the parallel program ran over twice as fast with the better load balancing. For example, the data trace of *wupwise* improved from 61+ hours with zero shift to 28+ hours with the optimal shift, a savings of almost 33 hours.

When determining the best shift, we must remember that the processors only store each reference once, so we want to balance the *unique* references rather than the *total references*. For example, assume we are using two processors and our trace consists of the numbers 1, 2, 1, 4, 1, 6, 1, 8, 1, 10, 1. If we load balance based on the total number of references we decide to shift zero bits. But this assigns one value, 1, to the first processor and five values to the second. When processing the last value in the trace, the first processor only has one reference to compare it with during Phase II, while the second processor is comparing five references. If we instead load balance based on the unique references, then we decide to shift one bit. This assigns three values to the first processor and three values to the second. The work of Phase II is more evenly balanced.

We now look at each phase in more detail. We see how various implementation choices affect the time involved for each phase, and discuss possible improvements.

9.2.2 Phase I

For most of Phase I, only one processor is doing any work. It is looking up the stack depth of the current reference. If we store the references and depths as a linked list, sorted by depth, then fewer references must be visited when the depth is small. If the locality is good, this is more likely to happen. However, if the locality is good, then the sequential version of the locality program performs well and the parallel version is not needed. If the parallel algorithm is being used, we assume that the locality is poor. Storing the references and depths in an unsorted array may be faster, since arrays are accessed faster than linked lists [59, page 96].

Other ways to improve the speed-up of Phase I are to store the references in sorted structures, where the worst-case lookup is O(log n) or O(1) rather than O(n). Examples include AVL trees [23] and hash tables [46]. However, such structures make it more difficult for Phase II to operate easily. One option is to use the tree or hash table and have each entry point to the reference and depth stored in the array or linked list.

Another idea that may improve the speed of Phase I is to overlap the two phases together. If we know the next reference as well as the current one, then while traversing the array or linked list during Phase II we may record the new depth of the next reference. This also avoids the problem of one processor doing work while the others wait.

9.2.3 Phase II

As with Phase I, this phase may be faster with a linked list sorted by depth. In Phase II we only wish to calculate strides and delays and update depths when the current depth is less than the broadcast depth. If a linked list is used, then once

the stored depth is greater than the broadcast depth we may discontinue traversing the list. Since we are usually just adding 1 to each depth in the list, there is usually no need to reorder the list. The only exception occurs when the current reference is discovered in the list. In this case, the reference must be removed from the list and re-inserted at the top with a depth of one.

Again, if the locality of the input trace is good, then we may save significant amounts of time by using a linked list and only traversing some of the stored references. However, the poor locality traces that are more in need of speed-up may often traverse most of the list, meaning that the periodic updating of the list and the slower nature of linked lists may not be worth it.

Another way to optimize Phase II is to notice that when the depth of the current reference is one, the current reference is an immediately recurring element. In this situation, the only stride/delay combination that needs to be recorded is a stride of zero and delay of one (see Theorem 6.7). In addition, the stack does not need to be changed. So whenever the broadcast depth is 1, then the processor with the reference assigned to it records a stride/delay of (0,1) and all the other processors do nothing. No stack traversal is needed.

Combining this optimization with the overlap optimization of Phase I is somewhat tricky but can be done. When the next reference is the same as the current one, then we know the next reference has a depth of one. The assigned processor records the stride/delay of (0,1) and all the processors then read another reference and use it as the next reference. The complexity of making this work may remove some of the benefits of overlapping the two phases.

workload	type	total	unique	i.r.e.	weight
wupwise	D	57,464,980	8,588,924	2,341,977	473,447
swim	D	42,031,084	7,988,204	2,298,995	317,388
**lucas*	D	38,488,490	2,712,775	3,650,248	94,508
perlbmk	D	37,370,781	1,840,857	3,422,644	62,494
**mcf*	D	34,938,679	1,386,300	1,665,496	46,127
bzip2	D	36,776,336	766,146	4,626,989	24,631
**gcc*	D	40,662,494	265,933	4,365,358	9,653
**gcc*	I	51,067,057	77,404	19,954,787	2,408
wupwise	I	34,931,332	79,684	14,127,065	1,658
lucas	I	55,170,861	33,077	20,600,033	1,144
perlbmk	I	54,061,602	33,803	21,887,304	1,088
swim	I	50,749,935	27,389	17,904,937	900
mcf	I	57,174,298	16,170	23,960,306	537
**bzip2*	I	54,240,485	15,421	19,960,745	529

Table 9.1: The traces that we used in this chapter. The workloads were chosen to give a variety of weights and to represent the distribution of weights among all our SPEC traces. The starred workloads were used with all four versions of the parallel program with 2, 4, 8, 16, 32, and 64 processors.

9.3 The Experimental Platform

The traces used in this chapter are shown in Table 9.1 along with whether they are the instructions or the data, the total number of references in the trace, the number of unique references in the trace, the number if immediately repeating elements in the trace, and the **weight** of the trace. All of our traces are from the Spec 2000 suite. Four are SPEC CINT and three are SPEC CFP.

The weight of the trace is a metric we have defined that is intended to be relative to the time to compute the locality of the trace using a stack based algorithm. The equation for the weight is

$$weight = \frac{(total - ire) * unique + ire}{1,000,000,000}, \tag{9.1}$$

where *total* is the total number of references in the trace, *ire* is the number of immediately repeating elements, and *unique* is the number of unique references

in the trace. Note that the weight is different depending on the granularity the trace is processed at. All weights shown in this dissertation were computed with a granularity of 8 bytes.

We developed this weight equation based on the big O notation for the stack based locality algorithm when we know *total*, *unique*, and *ire*. At its simplest level, the stack based algorithm is $O(total^2)$. When we also know *unique* for the given trace, the algorithm becomes $O(total * unique)$.

We discovered, however, that this does not sufficiently represent our traces. A few traces have many more immediately repeating elements than the other traces and hence are much faster to compute than other traces that have the same value for $total * unique$. Since *ire* is almost as simple to calculate as *total*, we make use of the knowledge that all immediately repeating elements only access the stack once, and require no reordering of the stack. Therefore the time to process an immediately repeating element is constant and should not be multiplied by *unique*. The final order is $O\big((total - ire) * unique + ire\big)$.

We divided this equation by 1 billion to reduce the values of all our traces to a reasonable range. The final result is Equation 9.1, which we decided to name the weight of the trace. Smaller weights indicate better locality, and faster stack based computaton. We hope to see a trend that as the weight increases, the value of using the parallel algorithm increases. Hopefully we can pick a cut-off point where traces with smaller weights perform better in the sequential, stack based program and traces with larger weights perfrom better with the parallel version.

Our parallel cluster is comprised of 32 dual processor Opteron systems running at 2.2 GHz. Each processor has 1 Mbyte of L2 cache. Each system has 4 Gbytes of physical memory and is connected to the others via switched Gigabit Ethernet. We used MPI when programming for this machine.

We tested four versions of our parallel locality program using the above machine and the starred traces in Table 9.1. All four versions use the depth one shortcut. Version *AR* stores the references in unsorted arrays with no other optimizations, Version *LL* stores the references in linked lists with no other optimizations, Version *ARol* stores the references in unsorted arrays and uses overlap, Version *LLol* stores the references in linked lists and uses overlap. We now present our results.

9.4 Results

We first ran the starred traces in Table 9.1 through the stack based, sequential locality program as well as all four versions of the parallel program using 2, 4, 8, 16, 32, and 64 processors. After selecting the best parallel version, we ran each of the traces in Table 9.1 through the sequential locality program and the parallel locality program using 2, 4, 8, 16, 32, and 64 processors. We did not run our parallel locality program on one processor; whenever we refer to the one processor results, we are referring to the sequential version of the locality program.

We ran all versions of the program on the same machine so timing comparisons are fair. When running the parallel version of the program, we always used both processors on each node used. When running the sequential version on one processor, we made sure that the other processor on the same node was unused. This may have given a slight advantage to the sequential version of the program, meaning that our speedups may be pessimistic.

9.4.1 Comparing Versions

Figure 9.6 shows the time to compute the locality data of the data trace of *lucas* across the range of number of processors for all four versions of the parallel locality

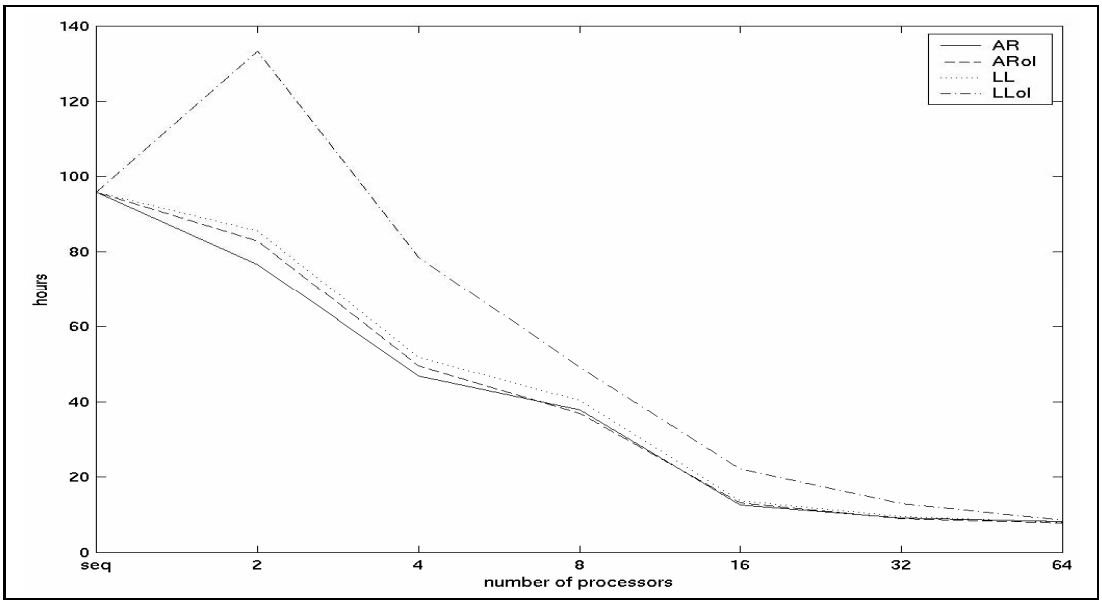


Figure 9.6: A comparison of the time (in hours) to compute the locality data for the data trace of *lucas* using all four versions of the parallel locality program.

program. Figure 9.7 shows the same thing for the instruction trace of *gcc* with the *scilab* input. These two graphs are representative of the results we found for the starred traces in Table 9.1. Figure 9.6 represents the traces with larger weights and worse locality. Figure 9.7 represents the traces with smaller weights and better locality.

First, let us examine how the four versions of the parallel locality program compare for traces with worse locality, represented by Figure 9.6. The version that performs best over our range of number of processors is AR, i.e. array without the overlap. Next best is ARol, i.e. array with the overlap. Next is LL, i.e. linked list without the overlap. All three of these versions are quite close together. Significantly worse is LLol, i.e. linked list with overlap.

We can see that the advantage of using a linked list (i.e. not having to traverse all the references stored on a particular processor) does not outweigh the disadvantage

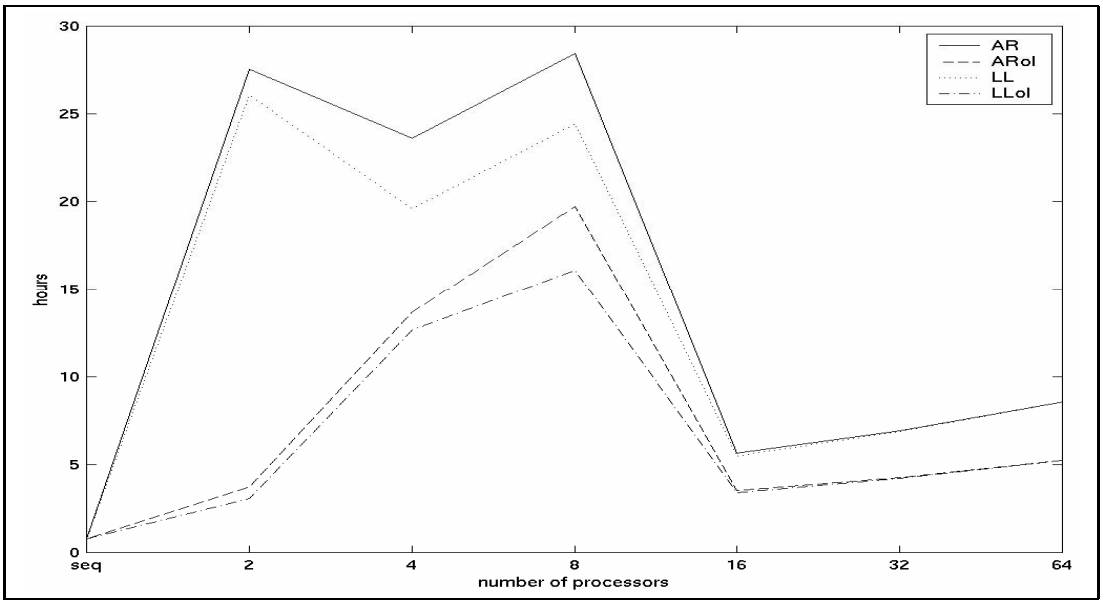


Figure 9.7: A comparison of the time (in hours) to compute the locality data for the instructions of *gcc* with the *scilab* input using all four versions of the parallel locality program.

that a linked list takes longer to traverse than an array. The linked list version with overlap performs significantly worse because the overlap lessens the advantage of the linked list. When traversing a list looking for two items simultaneously, one is more likely to traverse deeper in the list.

It is interesting to note that adding overlap to the array version does not improve the results. Without overlap, the assigned processor first does Phase I by searching through its array for the reference and retrieving its depth. If the reference has already been seen, then only part of the array is traversed, with a compare at each point. With overlap, this compare is done by the assigned processor for the next reference at the same time as the stride/delay relationships for the current reference are computed. In this case, the entire array is traversed, with a Phase I compare at each point. We speculate that this is why overlap takes slightly longer than without overlap. The advantage of traversing only part of the array an extra time for the

compare outweighs the advantage of piggy-backing the compare on another traversal of the entire array.

The timing results for other large weight, poor locality, traces are similar. On average, the AR version of the program performs best, even though sometimes not overwhelmingly better. We therefore conclude that for poor locality traces, the best version of the parallel locality program is AR.

It is perhaps not surprising to see that the versions of the program perform differently in relation to each other for traces with better locality, as seen in Figure 9.7. In this case, the best version of the program is LLol, with ARol second best. The worst performer is AR, with LL only slightly better.

Again, these results make sense. For traces with good locality, the disadvantages of the overlap method are minimized. In addition, the advantage of the linked list is maximized, since good locality means less of the list needs to be traversed each time. Therefore, we conclude that for traces with good locality LLol is perferred.

As mentioned before, we desire to focus on optimizing our parallel locality program for traces with poor locality. Therefore, we use the AR version of the parallel locality program from now on.

9.4.2 Speedups

After picking the AR version of our program, we recorded the time to compute the locality data across the range of number of processors for each of the traces in Table 9.1. Table 9.6 shows the timing results for all these runs. We now show graphs for the results of six of the traces. We use the sequential version of the locality program as our basis. The goal is to significantly improve that time.

First we look at several traces with large weights and poor locality. Figure 9.8

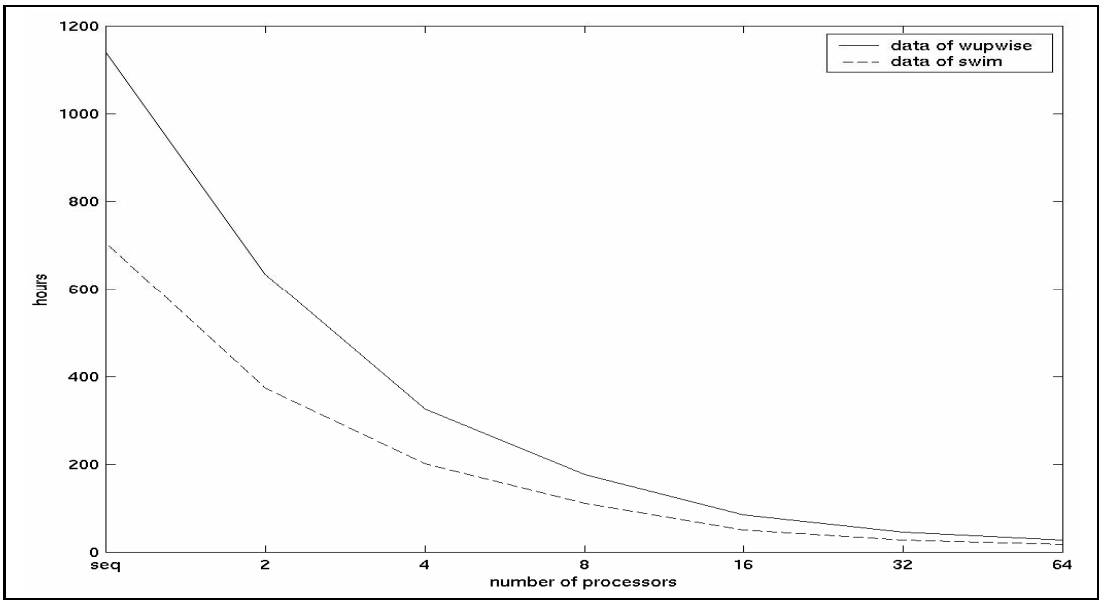


Figure 9.8: Run times on various numbers of processors for the data of *wupwise* and the data of *swim* using the AR version of the parallel locality program.

shows the results for the data trace of *wupwise* and the data trace of *swim*. Figure 9.9 shows the results for the data trace of *lucas* and the data trace trace of *mcf*. Notice that the maximum number of hours on the y-axis is different for each graph. We show the data of *wupwise* because it had the worst time on the sequential version of all our SPEC traces. We show the data of *mcf* because it had the worst time on the sequential version of all our integer traces.

Let us look at the results for the data trace of *wupwise* first. Using the sequential version, it took over 1141 hours (47+ days) to calculate the locality data. Switching to the AR version of the parallel program, using just two processors, it only takes 633 hours (26+ days), a speedup of 1.8 times. We see similar speedups as we increase the number of processors. Using 64 processors, it only takes 28 hours to compute the locality data, an overall speedup of 40.7 times.

It should be no surprise to see similar speedups for the data trace of *swim*. It

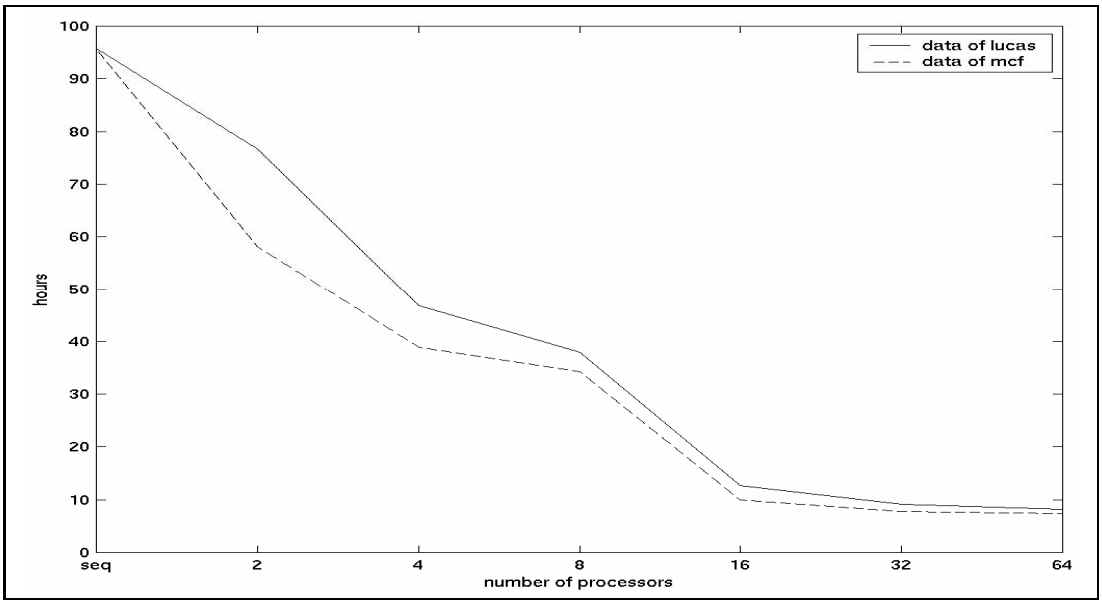


Figure 9.9: Run times on various numbers of processors for the data of *lucas* and the data of *mcf* using the AR version of the parallel locality program.

took almost 706 hours (29+ days) to calculate the locality data using the sequential version. Using two processors, the time was already reduced to almost 374 hours (15+ days), a speedup of 1.9 times. Using all 64 processors, it only took 17.6 hours to compute the locality data for the data trace of *swim*, an overall speedup of 40.2.

The results for the data traces of *lucas* and *mcf* in Figure 9.9 are not as smooth as the results in Figure 9.8. We believe this is because the traces in Figure 9.9 have significantly fewer unique references than the traces in Figure 9.8 and therefore the load is less likely to be equitably distributed. Regardless, the overall speedup is still significant.

It took almost 96 hours (4 days) to compute the locality data using the sequential algorithm for both the data trace of *lucas* and the data trace of *mcf*. Switching to the AR parallel locality program, using just two processors, we see improvements, but not as dramatic as seen in Figure 9.8. Specifically, it took over 76 hours (3+

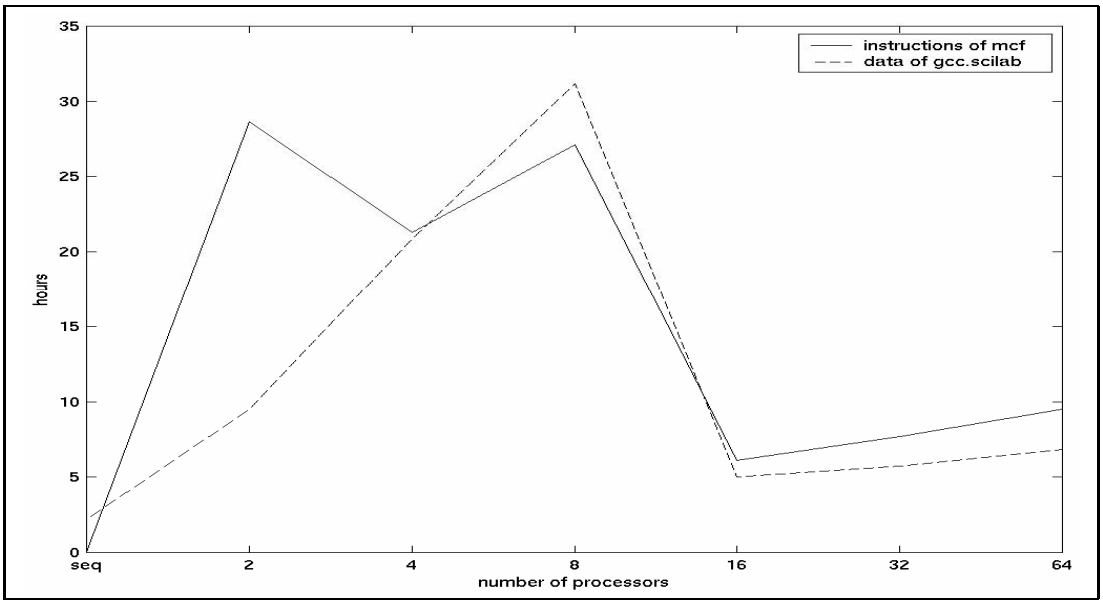


Figure 9.10: Run times on various numbers of processors for the instructions of *mcf* and the data of *gcc* with the *scilab* input using the AR version of the parallel locality program.

days) to compute the locality data for the data trace of *lucas* and over 58 hours (2+ days) for the data trace of *mcf*. The speedups were 1.2 and 1.6 times respectively.

The parallel program appears to reach a knee in its curve when 16 processors are used for both the traces in Figure 9.9. Using 16 processors, it took 12.7 hours to compute the locality data for the data trace of *lucas* for an overall speedup of 7.6 and 10 hours to compute the locality data for the data trace of *mcf* for an overall speedup of 9.6. The overall speedups using 64 processors for *lucas* and *mcf* are only 11.6 and 13.0, respectively. For these traces, there is little further performance improvement when moving from 16 to 32 to 64 processors.

Now we look at workloads where the weight is small and the locality is quite good. Figure 9.10 shows the results for the instruction trace of *mcf* and the data trace of *gcc.scilab* using the AR version of the parallel locality program. Again, when comparing with Figures 9.8 and 9.9, note the difference in scale.

The results for the instruction trace of *mcf* are representative of all the instruction traces in Table 9.1. The sequential locality program took less than one hour to compute the locality data for each of the seven instruction traces listed. The time to compute using the parallel program took hours at best. For example, it took 2.9 minutes to compute the locality data for the instruction trace of *mcf* using the sequential program. The best results obtained using the parallel program was when using 16 processors, which took 6.13 hours, a slowdown of 127 times.

The results for the data trace of *gcc.scilab* are a good example of how the parallel program responds to data traces that have mid-ranged weights, i.e. good locality for a data trace. It takes 2.19 hours to compute the locality data for the data trace of *gcc.scilab* using the sequential program. This is not as quick as the 3 minute time for the instruction trace of *mcf*, however using the parallel algorithm does not improve on this time for any number of processors.

We conclude that for traces where the weight is larger than 10,000, or where it takes longer than about 5 hours to compute the locality data using the sequential program on our 2.2 GHz processor, it is advantageous to use the parallel program. There may be little further improvement in speedup after 16 processors, however. For traces where the weight is larger than about 100,000, and it takes longer than a week to compute the locality data using the sequential program on our 2.2 GHz processor, we expect to see significant speedups as the number of processors is increased to at least 64. Obviously, more work could be done to find more precise values for drawing these lines. In addition, it would be interesting to determine why the knee of the curve is at 16 processors. Is the knee location related to our particular parallel machine, or the number of unique references in the trace, or some other factor?

9.5 Cache Characterization Surfaces

Recall in Chapter 5 that we discussed creating cache characterization surfaces using random data and briefly mentioned the time involved. The random trace we actually used was created using the Laplacian distribution with $\lambda = 600,000$ and was 500 million references long. Computing just the locality for this random trace using our parallel algorithm took over 14 days on 64 processors. We did not compute the locality data using fewer processors, or the sequential algorithm, for obvious reasons.

When creating cache characterization surfaces, we modified the locality program to incorporate the cache simulations necessary. In addition, we computed multiple cache characterization surfaces simultaneously. When doing this with the parallel locality algorithm, we assigned a cache simulator to each processor, rather than writing a parallel cache simulator. Our particular machine is made up of 32 nodes, with two processors on each node. Since we created less than 32 cache characterization surfaces at any time, we assigned cache simulators to only the odd numbered nodes. This spread out the calculations more evenly among the nodes, allowing each node to maximize its processor and memory sharing. The basic parallel multiple cache characterization surface algorithm is shown in Figure 9.11.

We can see a general trend that as the weight increases and locality diminishes the overall speedups increase and the advantage of moving to more processors continues with larger numbers of processors. We can use this to speculate how long it would take to process the random trace used in Section 5.3 using the sequential algorithm. As mentioned previously, there were 500 million references, of which 7,931,968 were unique and 192 were immediately repeating elements for a weight of 3,965,982. Obviously, this weight is much greater than for either the data trace of *wupwise* or *swim*. Therefore, we use the overall speedup for the data trace of *wupwise* as a

```
void main() {
    .
    .
    int misses[numcaches];
    int cachenum = MyCacheAssignment(myrank);
    .
    .
    ulong addr = GetReference();
    if (cachenum >= 0) {
        SubmitToCache(addr);
    }
    proc = (addr >> shift) & (numprocs - 1);
    .
    .
    while (MoreReference()) {
        addr = GetReference();
        ClearMisses(misses);
        if (cachenum >= 0) {
            misses[cachenum] = SubmitToCache(addr);
        }
        AllReduce(misses);
        proc = (addr >> shift) & (numprocs - 1);
        .
        .
    }
    SumMissTables();
    SumLocalityTables();
}
```

Figure 9.11: Pseudo-code listing for the basic parallel multiple cache characterization surface algorithm. See Figure 9.4 for the $\cdots$ portions of the algorithm. The variable `numcaches` holds the number of caches for which cache characterization surfaces are being created. The array `misses` stores whether or not each reference is a miss in each cache. The function `MyCacheAssignment(myrank)` returns the index of the cache assigned to this processor, or -1 if no cache is assigned. The function `SubmitToCache(addr)` returns 1 if `addr` is a miss in the assigned cache and 0 otherwise. The function `ClearMisses(misses)` assigns a 0 to every entry in `misses`. The function `AllReduce(misses)` adds the values from all the processors and distributes the results back to all the processors. In this situation, it effectively informs each processor which caches have misses. The function `MarkLocalityEvent`, although not shown, involves marking locality events in miss tables for which the current reference missed in the given cache.

best-case scenario.

It took 14 days, 10.2 hours to compute the locality for the random trace from Section 5.3 on all 64 processors. Assuming an overall speedup of 40.7, we estimate that it would take 587 days, or 1.6 years, to compute just the locality data for the random trace. This does not take into consideration the time to do the cache simulation calculations necessary to create cache characterization surfaces. Recall that it took over 7 weeks to compute 24 cache characterization surfaces in parallel on all 64 processors, 3.5 times as long as it took to do the locality alone.

In addition, this is a best-case scenario estimate. We speculate that the overall speedup is actually much better, and the sequential time much longer, since the random trace weight is significantly larger than for the data trace of *wupwise*. We therefore conclude that the cache characterization surfaces used in Section 5.3 would not have been computed if we did not have a parallel version of the locality program available.

9.6 Summary

In this chapter, we have presented a new parallel algorithm that signficantly decreases the time necessary to compute the locality data for many traces. We can use either the weight or the time to compute using the sequential algorithm on our 2.2 GHz processor to determine when the parallel algorithm would improve run times versus destoying run times. Specifically, if the weight is greater than 10,000, or if the time to compute using the sequential program on our 2.2 GHz processor is greater than 5 hours, then it is probably advantageous to use the parallel version of the program on up to 16 processors. If the weight is greater than 100,000 or the time to compute using the sequential program on our 2.2 GHz processor is greater

than 1 week, then it is probably advantageous to use the parallel program on as many processors as you can.

We further pointed out that without this parallel algorithm it would have taken years to compute the cache characterization surfaces shown in Section 5.3 and used in Chapter 7. Future work could be done to investigate other ways to further optimize the time for the parallel program. In addition, more work could be done to determine more specifically the point at which it would be better to use the parallel program.

workload	type	weight	sequential	2 procs	4 procs	8 procs	16 procs	32 procs	64 procs
wupwise	D	473,447	1141.611	633.19	326.40	177.30	85.41	45.96	28.04
swim	D	317,388	705.898	373.69	201.44	111.43	51.21	27.86	17.57
lucas	D	94,508	95.812	76.67	46.84	37.90	12.68	9.16	8.24
perlbmk	D	62,494	45.182	48.88	35.50	33.83	9.19	7.49	7.32
mcf	D	46,127	95.688	34.15	38.93	34.29	9.96	7.73	7.38
bzip2	D	24,631	6.396	17.66	22.05	20.07	5.22	5.54	6.22
gcc	D	9,653	2.189	9.51	20.85	31.19	4.99	5.71	6.81
gcc	I	2,408	0.772	27.55	23.63	28.45	5.64	6.92	8.54
wupwise	I	1,658	0.278	19.23	13.98	19.30	3.78	4.71	5.87
lucas	I	1,144	0.085	25.57	22.90	28.56	5.86	7.37	9.32
perlbmk	I	1,088	0.290	15.95	24.54	27.66	5.82	7.26	9.02
swim	I	900	0.087	26.21	19.11	24.26	5.50	6.83	8.48
mcf	I	537	0.048	28.64	21.29	27.14	6.13	7.68	9.50
bzip2	I	529	0.113	21.00	20.05	24.59	5.94	7.28	9.00

Table 9.2: Table reporting actual times, in hours, to compute the locality for all the traces used in this chapter on a variety of number of processors using the AR version of the parallel program.

Chapter 10

Conclusion and Future Work

In this dissertation we have reintroduced our improved locality surface. Our surface is a significant improvement over Grimsrud's original locality surface because it is better tailored to LRU cache performance and has a strong mathematical description. In Chapter 2 we described our definition of locality using the mathematics of bags and sets. We also wrote the equations that allow us to view the locality data as the locality surface. We now discuss the various ways we used the locality surface throughout this dissertation and then discuss how our locality principles may be applied to caches.

Throughout the dissertation, we have shown a number of applications for the new locality surface. In Chapter 3 we explained the underlying patterns that create several common features on the locality surface. We then characterized a number of real workloads from the SPEC C2000 benchmark suite based on these features. The locality surfaces for all the traces from the SPEC C2000 benchmark suite are found in Appendix B. In Chapter 4 we qualitatively predicted cache performance using the locality surface. We later performed some limited quantitative predictions in Chapter 7. We also used the locality surface, in Chapter 8, to evaluate the accuracy

of a number of previous synthetic trace models.

In addition to describing traces in terms of locality, we described caches in terms of locality. In Chapter 5 we first used mathematics to describe when a given trace reference is a hit or a miss in four different types of caches. We then created the miss surface, the miss rate surface, and the cache characterization surface. This last surface helps us understand how caches and locality relate and gives us a better picture of why the qualitative and quantitative cache predictions work as they do. In Chapter 6 we mathematically proved a number of ways that two different strings may have the same locality. For some caches, this allowed us to prove why the locality data predicts cache performance with 100% accuracy. For other caches, we proved why the locality data is insufficient to quantitatively predict cache performance.

Finally, we discussed the time necessary to compute the locality data in Chapter 9 and presented a new, parallel algorithm. The parallel version of the locality program significantly improved the time to compute the locality data and made the creation of cache characterization surfaces possible.

We believe the locality surface, as here presented, to be a significant contribution to the study of caches and workload behavior. In fact, the locality surface could be used to examine any level of the memory hierarchy that uses a LRU replacement. We acknowledge that there are other locality metrics that are essentially subsets of our locality surface, such as reuse distance [16, 88]. There may be times when a researcher desires to focus exclusively on only one aspect of locality. For example, when a cache line size is unchangeable, there may be no need to study spatial locality. Or when focusing on prefetching, there may be little interest in temporal locality. However, the locality surface is available for whenever the overall picture is desired.

Depending on the circumstances, a subset of the locality surface may be of more use for a specific application than the entire surface. For example, Chilimbi and

Hirzel wrote a dynamic prefetching tool that involved computing some aspects of spatial locality on the fly [20]. Obviously the locality surface would involve too much processing for it to be useful in such a tool. However, it would be informative to use the locality surface to evaluate the original memory reference locality versus the memory reference locality with the tool running. Such a study would help researchers understand how the tool works, how it changes the locality, and if it adversely impacts other aspects of locality.

We can now use locality to describe both workloads and caches, visually and independently. We can also predict cache performance for a given workload and cache. Finally, the locality surface is a powerful metric for evaluating any methodology that involves locality. We now discuss some directions for future work.

10.1 Future Work

A number of areas for future work were discussed throughout the dissertation. As previously mentioned, further optimizations may be possible for the parallel algorithm. There is room for improvement of the Case Two cache simulation prediction method in Chapter 7. There may also be methods that use the results of Theorem 6.19 to better predict Case Three cache results. As mentioned in Section 7.5, it would be interesting to perform a study of all cache simulation prediction methods using the same traces.

A more detailed workload characterization study than done in Chapter 3 is always possible, especially as new and improved benchmark suites are adopted. In addition, we mentioned in Chapter 3 that it would be useful to determine exactly what patterns in a trace cause various other features on the locality surface.

As mentioned in Chapter 5, it would be interesting to create cache characteri-

zation surfaces for other types of caches, such as column-associative [6] and skewed associative [68]. If a new synthetic trace generation method is proposed, it would be valuable to evaluate it as we did for other synthetic trace models in Chapter 8. It may even be possible to use the locality surface itself to generate synthetic traces. Our general algorithm, suggested in Chapter 8, may prove useful, or there may be another method. It may help to figure how the shape of a loop structure relates to the loop contents, as discussed in Chapter 3.

The locality surface could be used to evaluate various solutions in a number of areas such as tracing methods [83], compilers [54], prefetching [87], and operating systems [22]. For example, what causes the jut discussed in Section 3.1.6? Is it a feature of the compiler, or the operating system? We already used the locality surface to see how the locality changes for the same workload in different operating systems. Other researchers have suggested numerous methods for improving the locality of workloads [26, 36]. How do they affect the overall locality? Perhaps the spatial locality is improved, as measured by the researchers, but the temporal locality worsens.

Other locality researchers have noticed varying phases of locality throughout the life of a benchmark [25, 29, 69]. The locality surface may be useful for determing locality phases in a workload. Different phases may have very different locality features. Comparing the locality surfaces created in each phase may help understand how the phases of a workload affect performance.

Other researchers have used various forms of locality to characterize locality in the world wide web [12] and to create synthetic web proxy cache traces [61]. The locality surface may add additional insights into any area, such as these, where previous locality metrics have proved useful in the past. In addition, it may be useful to apply the locality surface to such areas as file systems, disk caching, and

network traffic.

The purpose of the locality surface is to unify temporal and spatial locality. However, there are other types of locality that may be of interest, such as multiprocessor locality. Agarwal and Gupta [4] and Johnson [52] have both suggested multiprocessor locality metrics. Perhaps there is a way to combine one of these metrics with the locality surface to provide better understanding of locality with yet another dimension.

In short, we believe there are numerous possible applications for the locality surface. Some areas extend work begun in this dissertation; other areas apply the locality surface in a completely new way. In any area where there is temporal and spatial locality, our locality surface provides a useful and accurate visual representation of the locality.

Appendix A

Trace Details

This appendix contains the trace details for all the SPEC CPU 2000 traces from the BYU trace library [1]. For each trace we list the workload name, the input file (if necessary), whether the workload is from the Integer or Floating Point sub-suite, whether the trace is an instruction trace or data trace, which operating system the trace was taken under (Linux, Windows NT, or Windows 2000), the number of references in the trace, the number of unique references in the trace, and a description of the workload. In this appendix the traces are listed in alphabetical order by workload.

workload	input	suite	type	OS	total	unique	description
ammp		FP	I	L	54,110,139	38,803	Computational Chemistry
				NT	54,439,604	63,108	
				2k	53,891,043	63,277	
ammp		FP	D	L	36,931,208	872,932	Computational Chemistry
				NT	38,247,667	294,750	
				2k	37,200,232	256,603	
applu		FP	I	L	47,149,788	29,212	Parabolic/Elliptic Partial Differential Equations
				NT	58,372,015	47,895	
				2k	55,905,014	102,310	
applu		FP	D	L	46,261,474	1,524,041	Parabolic/Elliptic Partial Differential Equations
				NT	33,520,025	896,819	
				2k	34,799,526	1,398,494	
apsi		FP	I	L	46,328,675	44,585	Meteorology: Pollutant Distribution
				NT	47,758,143	76,545	
				2k	49,285,073	82,669	
apsi		FP	D	L	45,049,441	1,891,123	Meteorology: Pollutant Distribution
				NT	43,989,817	1,842,030	
				2k	42,048,040	1,677,317	
art		FP	I	L	54,174,676	31,616	Image Recognition / Neural Networks
				NT	54,953,387	49,929	
				2k	55,101,665	62,842	
art		FP	D	L	36,731,239	829,413	Image Recognition / Neural Networks
				NT	37,215,208	707,587	
				2k	35,663,815	782,518	
bzip	*g7*	INT	I	L	54,692,291	13,652	Compression
				NT	57,163,419	47,644	
				2k	57,035,030	38,050	

workload	input	suite	type	OS	total	unique	description
bzip	*g7*	INT	D	L	34,797,524	194,560	Compression
				NT	33,362,067	380,062	
				2k	33,692,972	266,911	
bzip	*g9*	INT	I	L	54,328,616	12,065	Compression
				NT	57,261,444	20,252	
				2k	57,228,352	30,799	
bzip	*g9*	INT	D	L	34,986,924	197,783	Compression
				NT	33,475,441	258,483	
				2k	33,121,679	263,571	
bzip	*p7*	INT	I	L	54,934,099	12,882	Compression
				NT	57,036,806	32,346	
				2k	57,203,371	36,746	
bzip	*p7*	INT	D	L	34,682,234	207,637	Compression
				NT	33,504,682	529,882	
				2k	33,376,219	282,698	
bzip	*p9*	INT	I	L	54,464,929	12,531	Compression
				NT	57,421,280	22,012	
				2k	57,189,236	28,210	
bzip	*p9*	INT	D	L	34,851,024	206,136	Compression
				NT	33,044,045	268,291	
				2k	33,403,255	263,229	
bzip	*s7*	INT	I	L	54,240,485	15,421	Compression
				NT	57,115,923	35,567	
				2k	55,950,719	84,807	
bzip	*s7*	INT	D	L	35,075,864	765,288	Compression
				NT	33,387,007	368,984	
				2k	34,415,506	415,114	

workload	input	suite	type	OS	total	unique	description
bzip	*s9*	INT	I	L	54,617,083	13,354	Compression
				NT	57,043,696	24,318	
				2k	57,516,624	31,105	
bzip	*s9*	INT	D	L	34,847,007	195,971	Compression
				NT	33,070,833	271,923	
				2k	33,165,965	266,720	
crafty		INT	I	L	50,020,348	30,338	Game Playing: Chess
				NT	56,719,324	70,458	
				2k	57,310,437	143,196	
crafty		INT	D	L	41,984,531	209,783	Game Playing: Chess
				NT	33,880,040	376,469	
				2k	32,679,125	41,1496	
eon	*cook*	INT	I	L	48,100,350	29,999	Computer Visualization
				NT	48,334,172	107,674	
				2k	49,080,139	93,327	
eon	*cook*	INT	D	L	41,085,829	183,524	Computer Visualization
				NT	42,113,950	645,576	
				2k	41,383,193	324,466	
eon	*kajiya*	INT	I	L	48,518,645	28,220	Computer Visualization
				NT	47,558,117	90,371	
				2k	48,328,083	62,011	
eon	*kajiya*	INT	D	L	40,792,089	70,798	Computer Visualization
				NT	42,921,271	435,957	
				2k	41,922,762	155,186	
eon	*rushmeier*	INT	I	L	48,298,304	28,451	Computer Visualization
				NT	47,822,625	48,211	
				2k	48,934,723	51,327	

workload	input	suite	type	OS	total	unique	description
eon	*rushmeier*	INT	D	L	42,559,400	74,821	Computer Visualization
				NT	42,668,076	129,109	
				2k	41,370,463	134,926	
equake		FP	I	L	52,044,863	33,454	Seismic Wave Propagation Simulation
				NT	54,834,756	42,752	
				2k	53,919,336	45,261	
equake		FP	D	L	39,217,825	931,276	Seismic Wave Propagation Simulation
				NT	37,119,678	710,110	
				2k	36,476,868	653,057	
facerec		FP	I	L	52,220,409	31,009	Image Processing: Face Recognition
				NT	51,968,858	53,539	
				2k	53,075,767	63,073	
facerec		FP	D	L	38,875,071	832,050	Image Processing: Face Recognition
				NT	39,744,298	660,967	
				2k	38,161,823	654,788	
fma3d		FP	I	L	55,003,830	36,674	Finite-element Crash Simulation
				NT	55,815,847	67,928	
				2k	56,278,892	93,482	
fma3d		FP	D	L	36,103,933	886,284	Finite-element Crash Simulation
				NT	35,516,833	236,101	
				2k	35,075,417	266,182	
galgel		FP	I	L	55,581,541	98,670	Computational Fluid Dynamics
				NT	57,102,958	46,894	
				2k	55,885,798	71,442	
galgel		FP	D	L	37,070,561	1,255,136	Computational Fluid Dynamics
				NT	34,551,638	219,845	
				2k	34,141,440	268,778	

workload	input	suite	type	OS	total	unique	description
gap		INT	I	L	57,785,246	24,749	Group Theory, Interpreter
				NT	57,162,970	51,269	
				2k	56,764,700	65,462	
gap		INT	D	L	31,714,213	923,381	Group Theory, Interpreter
				NT	33,386,414	1,093,040	
				2k	33,242,917	991,604	
gcc	*166*	INT	I	L	52,326,567	66,312	C Programming Language Compiler
				NT	55,292,634	116,096	
				2k	55,138,518	165,735	
gcc	*166*	INT	D	L	37,437,695	594,109	C Programming Language Compiler
				NT	35,475,013	791,696	
				2k	35,900,408	749,043	
gcc	*200*	INT	I	L	51,927,541	82,148	C Programming Language Compiler
				NT	54,910,021	132,755	
				2k	56,280,796	163,170	
gcc	*200*	INT	D	L	37,928,447	308,257	C Programming Language Compiler
				NT	35,700,888	592,878	
				2k	34,776,185	514,472	
gcc	*expr*	INT	I	L	51,080,978	87,479	C Programming Language Compiler
				NT	54,409,170	133,173	
				2k	55,773,523	181,518	
gcc	*expr*	INT	D	L	38,525,253	343,258	C Programming Language Compiler
				NT	36,374,683	528,816	
				2k	35,487,393	529,068	
gcc	*integ*	INT	I	L	51,853,273	86,076	C Programming Language Compiler
				NT	55,362,847	149,913	
				2k	56,370,127	166,250	

workload	input	suite	type	OS	total	unique	description
gcc	*integ*	INT	D	L	37,809,982	283,785	C Programming Language Compiler
				NT	35,418,273	508,906	
				2k	34,806,677	499,236	
gcc	*scilab*	INT	I	L	51,067,057	77,404	C Programming Language Compiler
				NT	54,310,833	130,187	
				2k	55,353,106	152,097	
gcc	*scilab*	INT	D	L	40,662,494	265,933	C Programming Language Compiler
				NT	36,231,831	469,809	
				2k	35,759,696	442,530	
gzip	*graphic*	INT	I	L	53,182,990	16,158	Compression
				NT	57,740,299	28,859	
				2k	58,643,949	48,624	
gzip	*graphic*	INT	D	L	38,939,980	210,952	Compression
				NT	33,312,091	615,246	
				2k	32,482,459	376,181	
gzip	*log*	INT	I	L	58,889,532	15,954	Compression
				NT	61,289,408	30,348	
				2k	62,397,552	41,156	
gzip	*log*	INT	D	L	33,229,137	348,452	Compression
				NT	32,501,419	686,829	
				2k	31,262,262	530,226	
gzip	*program*	INT	I	L	54,666,119	14,773	Compression
				NT	58,773,335	26,218	
				2k	59,598,864	58,438	
gzip	*program*	INT	D	L	37,279,953	230,374	Compression
				NT	32,266,912	444,425	
				2k	31,704,715	410,063	

workload	input	suite	type	OS	total	unique	description
gzip	*random*	INT	I	L	53,169,428	14,600	Compression
				NT	58,530,581	46,915	
				2k	59,110,732	45,370	
gzip	*random*	INT	D	L	38,850,318	226,258	Compression
				NT	32,559,692	522,397	
				2k	31,828,600	435,771	
gzip	*source*	INT	I	L	54,496,031	20,448	Compression
				NT	59,161,078	29,290	
				2k	59,111,271	86,664	
gzip	*source*	INT	D	L	37,405,566	1,046,032	Compression
				NT	31,863,296	522,016	
				2k	31,950,755	558,298	
lucas		FP	I	L	55,170,861	33,077	Number Theory / Primality Testing
				NT	51,969,949	43,100	
				2k	53,600,277	53,536	
lucas		FP	D	L	38,488,490	2,712,775	Number Theory / Primality Testing
				NT	39,645,156	2,473,766	
				2k	37,693,515	2,507,487	
mcf		INT	I	L	57,174,298	16,170	Combinatorial Optimization
				NT	57,454,830	36,004	
				2k	56,785,344	103,422	
mcf		INT	D	L	33,151,617	1,385,873	Combinatorial Optimization
				NT	33,007,536	1,700,815	
				2k	33,315,357	1,413,240	
mesa		FP	I	L	52,720,098	62,555	3-D Graphics Library
				NT	52,772,340	51,609	
				2k	52,590,461	93,010	

workload	input	suite	type	OS	total	unique	description
mesa		FP	D	L	38,044,541	1,601,370	3-D Graphics Library
				NT	39,012,417	1,144,253	
				2k	37,612,889	1,288,752	
mgrid		FP	I	L	54,059,465	74,843	Multi-grid Solver: 3D Potential Field
				NT	50,006,445	54,919	
				2k	51,326,288	50,638	
mgrid		FP	D	L	37,351,118	5,598,803	Multi-grid Solver: 3D Potential Field
				NT	41,755,272	5,570,840	
				2k	39,023,447	5,090,300	
parser		INT	I	L	52,386,657	24,577	Word Processing
				NT	53,267,547	54,023	
				2k	53,331,009	61,794	
parser		INT	D	L	37,342,215	555,962	Word Processing
				NT	37,126,856	883,206	
				2k	36,692,403	754,238	
perlbmk	*diffmail*	INT	I	L	54,083,478	34,648	PERL Programming Language
				NT	51,700,137	74,055	
				2k	52,069,217	80,692	
perlbmk	*diffmail*	INT	D	L	35,496,885	1,875,818	PERL Programming Language
				NT	38,707,764	2,384,882	
				2k	38,193,865	2,345,393	
perlbmk	*makerand*	INT	I	L	52,443,672	20,671	PERL Programming Language
				NT	50,920,100	56,390	
				2k	51,364,794	63,642	
perlbmk	*makerand*	INT	D	L	37,210,340	1,663,964	PERL Programming Language
				NT	39,605,675	2,019,494	
				2k	38,965,905	1,962,578	

workload	input	suite	type	OS	total	unique	description
perlbmk	*perfect*	INT	I	L	51,230,726	34,191	PERL Programming Language
				NT	51,911,857	62,228	
				2k	52,582,143	67,731	
perlbmk	*perfect*	INT	D	L	38,424,060	97,950	PERL Programming Language
				NT	38,469,188	271,664	
				2k	37,713,204	191,588	
perlbmk	*splitmail*	INT	I	L	54,061,602	33,803	PERL Programming Language
				NT	51,629,893	61,207	
				2k	52,263,141	67,949	
perlbmk	*splitmail*	INT	D	L	35,421,140	1,840,653	PERL Programming Language
				NT	38,912,111	2,312,410	
				2k	38,161,405	2,206,257	
sixtrack		FP	I	L	54,108,893	32,353	High Energy Nuclear Physics Accelerator Design
				NT	52,874,057	59,219	
				2k	52,657,642	79,652	
sixtrack		FP	D	L	37,130,406	784,656	High Energy Nuclear Physics Accelerator Design
				NT	39,257,889	456,331	
				2k	38,489,357	473,085	
swim		FP	I	L	50,749,935	27,389	Shallow Water Modeling
				NT	53,856,645	52,375	
				2k	54,208,799	42,249	
swim		FP	D	L	42,031,084	7,988,204	Shallow Water Modeling
				NT	40,635,716	6,939,791	
				2k	38,038,018	5,674,829	
twolf		INT	I	L	50,191,887	21,988	Place and Route Simulator
				NT	55,570,914	45,100	
				2k	54,416,626	107,710	

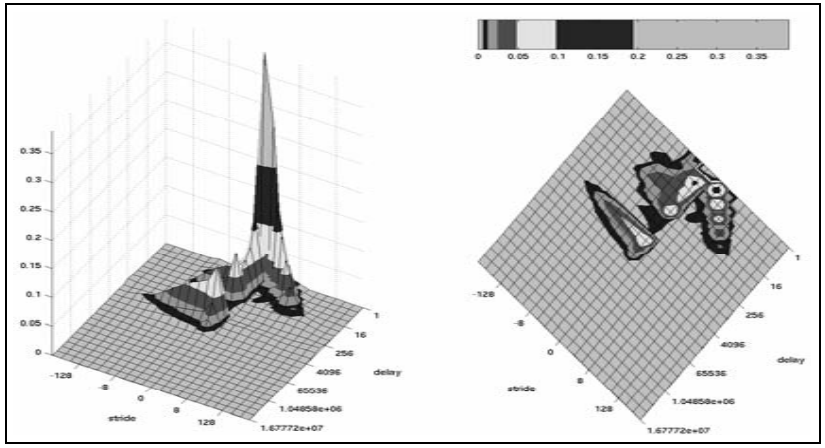
workload	input	suite	type	OS	total	unique	description
twolf		INT	D	L	39,611,264	332,193	Place and Route Simulator
				NT	34,760,706	677,984	
				2k	35,924,405	720,157	
vortex	*one*	INT	I	L	46,981,702	42,786	Object-oriented Database
				NT	46,654,977	83,431	
				2k	47,839,598	137,684	
vortex	*one*	INT	D	L	42,578,602	1,108,348	Object-oriented Database
				NT	43,766,489	2,001,326	
				2k	42,503,602	2,057,004	
vortex	*three*	INT	I	L	46,799,080	39,218	Object-oriented Database
				NT	45,753,225	81,955	
				2k	47,663,631	87,136	
vortex	*three*	INT	D	L	42,862,138	848,432	Object-oriented Database
				NT	44,743,960	2,014,335	
				2k	42,813,966	1,904,491	
vortex	*two*	INT	I	L	46,692,130	39,420	Object-oriented Database
				NT	46,753,160	87,012	
				2k	47,925,284	90,117	
vortex	*two*	INT	D	L	42,882,191	835,303	Object-oriented Database
				NT	43,622,475	1,887,022	
				2k	42,611,646	1,801,499	
vpr	*place*	INT	I	L	50,443,903	20,493	FPGA Circuit Placement and Routing
				NT	53,924,388	40,117	
				2k	54,238,720	96,574	
vpr	*place*	INT	D	L	39,367,649	343,223	FPGA Circuit Placement and Routing
				NT	36,982,683	565,457	
				2k	36,905,248	580,306	

workload	input	suite	type	OS	total	unique	description
vpr	*route*	INT	I	L	51,471,889	39,543	FPGA Circuit Placement and Routing
				NT	56,190,947	40,157	
				2k	57,364,826	63,990	
vpr	*route*	INT	D	L	38,269,566	376,602	FPGA Circuit Placement and Routing
				NT	34,426,469	634,162	
				2k	33,748,586	580,783	
wupwise		FP	I	L	34,931,332	79,684	Physics / Quantum Chromodynamics
				NT	54,807,649	43,292	
				2k	54,098,822	119,619	
wupwise		FP	D	L	57,464,980	8,588,924	Physics / Quantum Chromodynamics
				NT	36,672,267	936,409	
				2k	36,316,042	874,529	

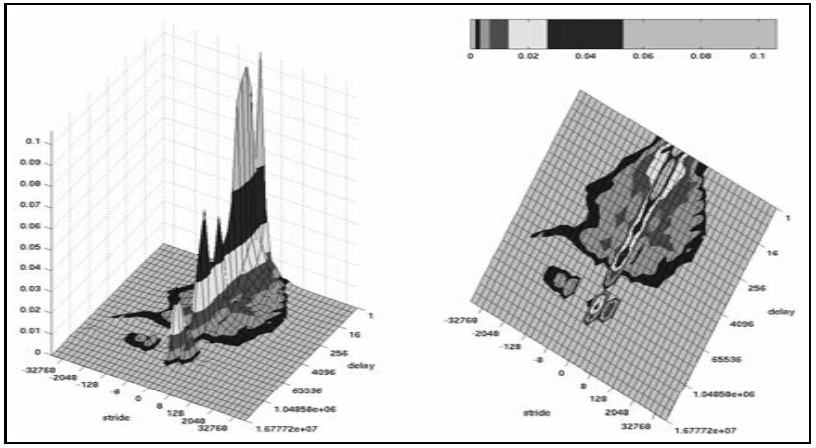
Appendix B

Locality Surfaces

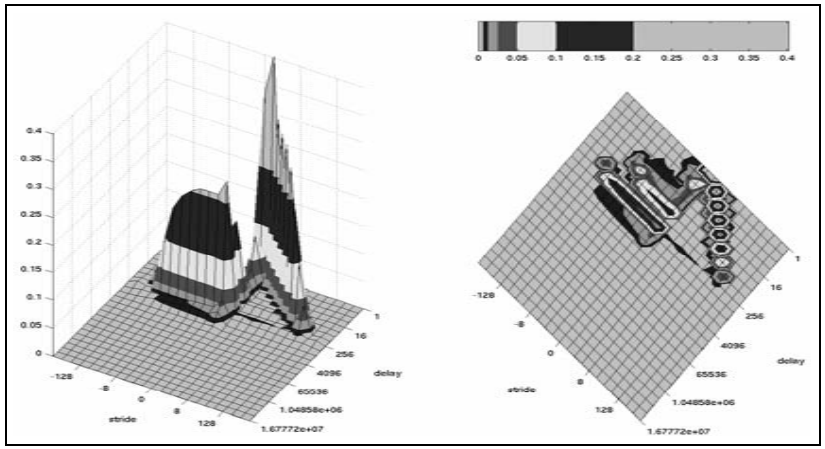
This appendix contains the locality surfaces for all the SPEC CPU 2000 traces from the BYU trace library [1]. All the traces taken under Linux are listed first, then the traces taken under Windows NT, and lastly the traces taken under Windows 2000. Within each operating system, the traces are listed in alphabetical order by workload name. For traces with multiple inputs, the input used for the particular trace is listed after the workload name with a dot between. For example, the trace of *eon* with the *rushmeier* input is labeled *eon.rushmeier*.



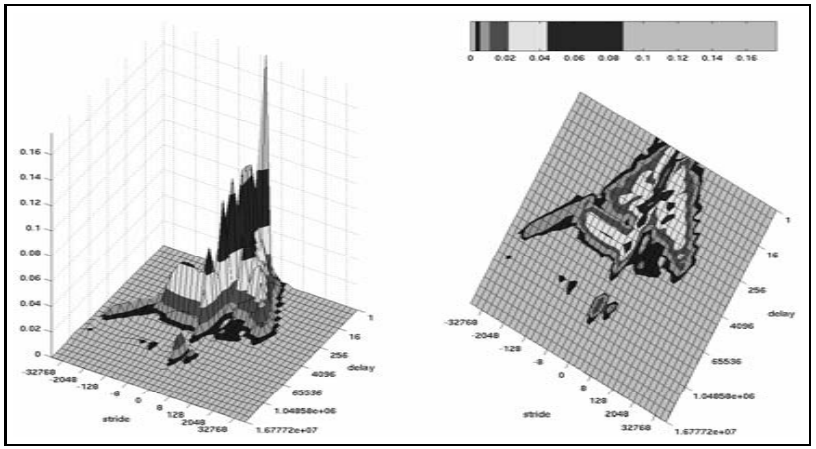
Instruction trace of *ammp* under Linux.



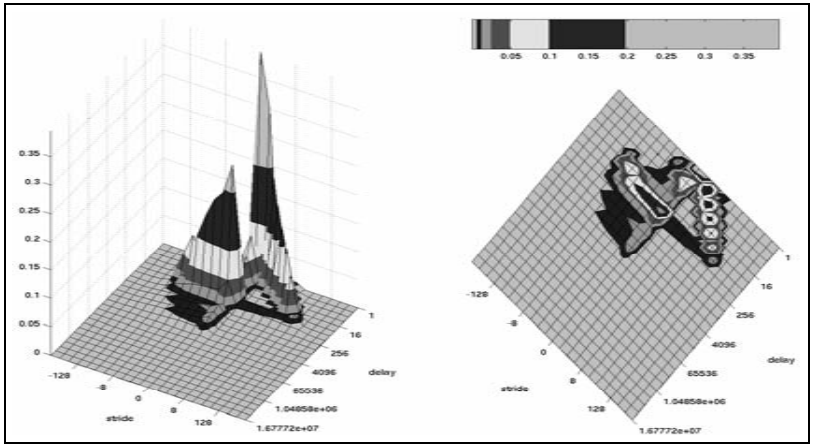
Data trace of *ammp* under Linux.



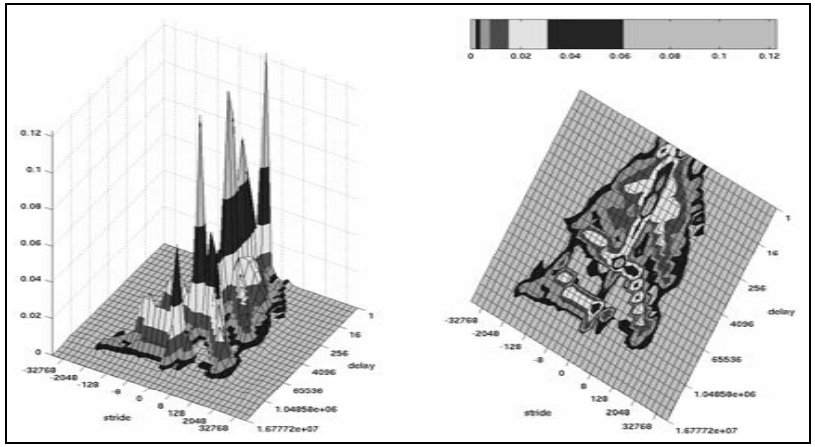
Instruction trace of *applu* under Linux.



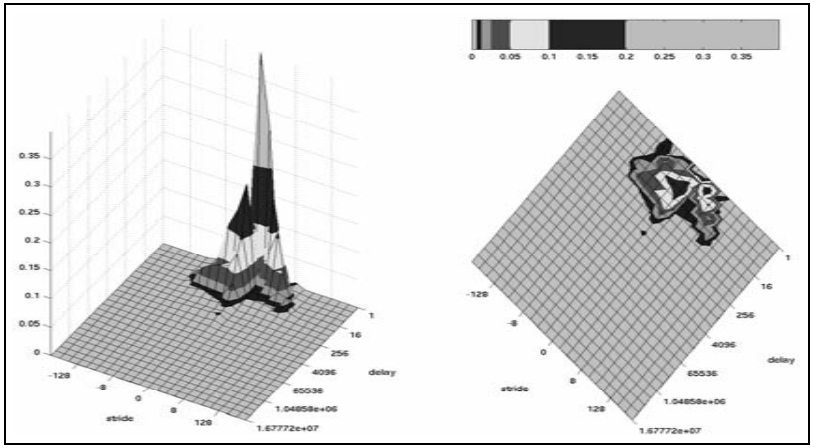
Data trace of *applu* under Linux.



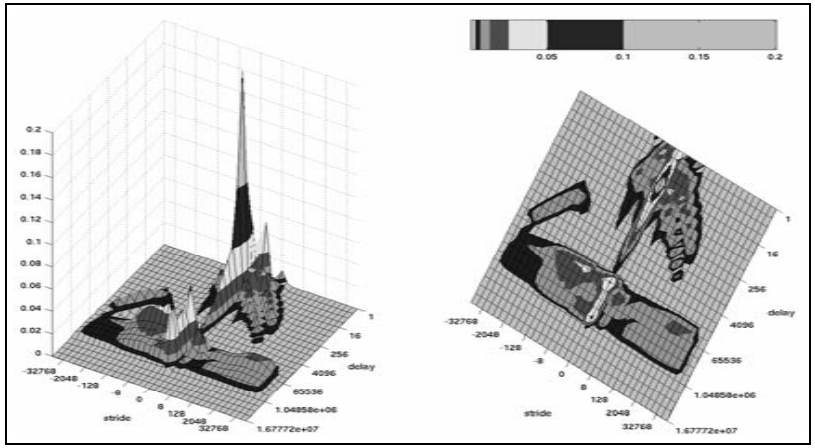
Instruction trace of *apsi* under Linux.



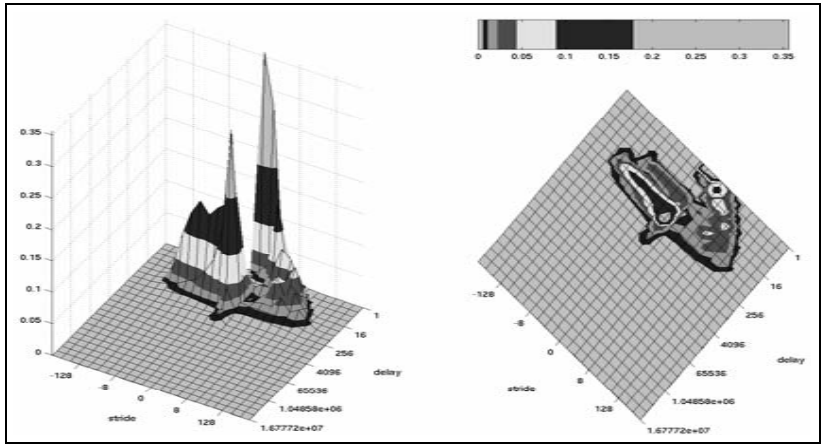
Data trace of *apsi* under Linux.



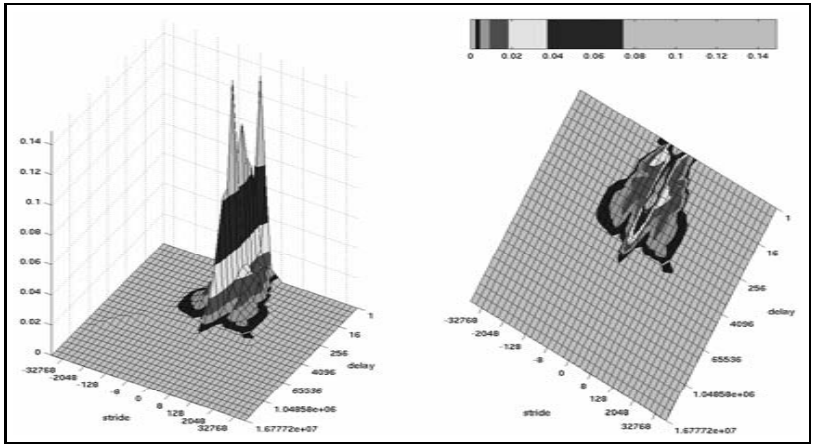
Instruction trace of *art* under Linux.



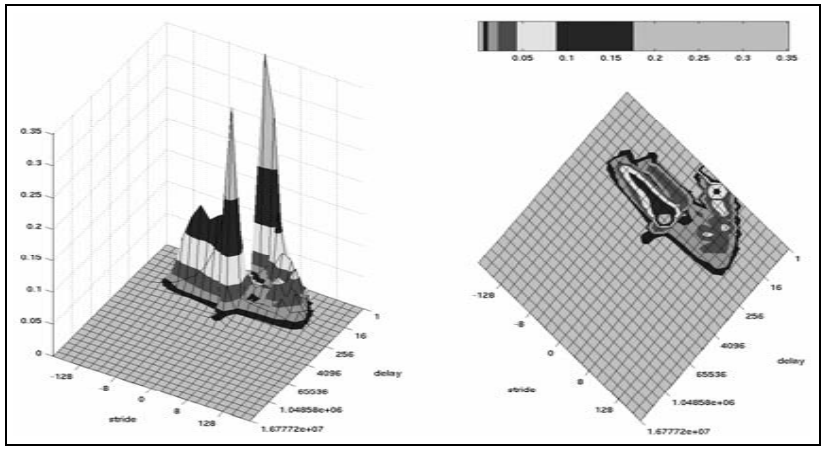
Data trace of *art* under Linux.



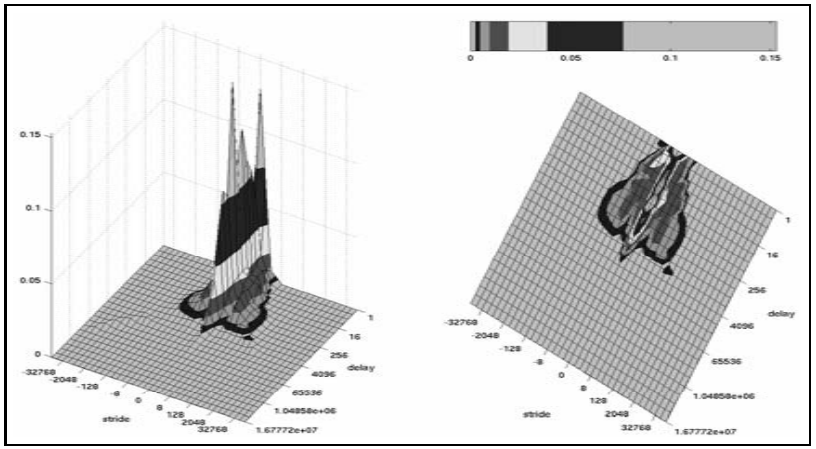
Instruction trace of *bzip2.g7* under Linux.



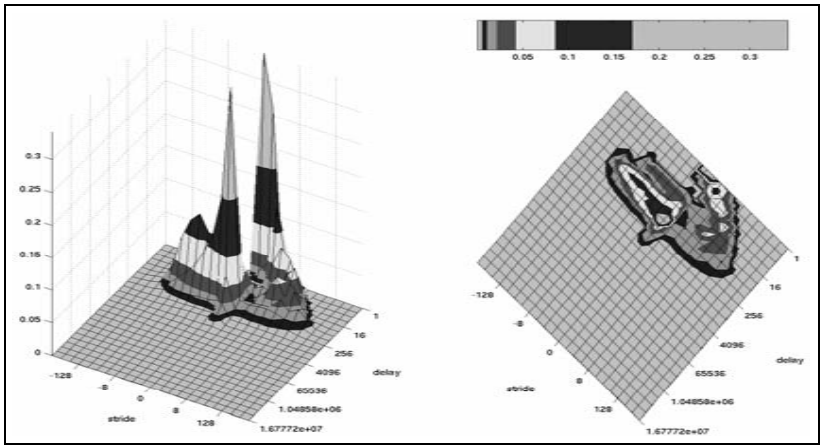
Data trace of *bzip2.g7* under Linux.



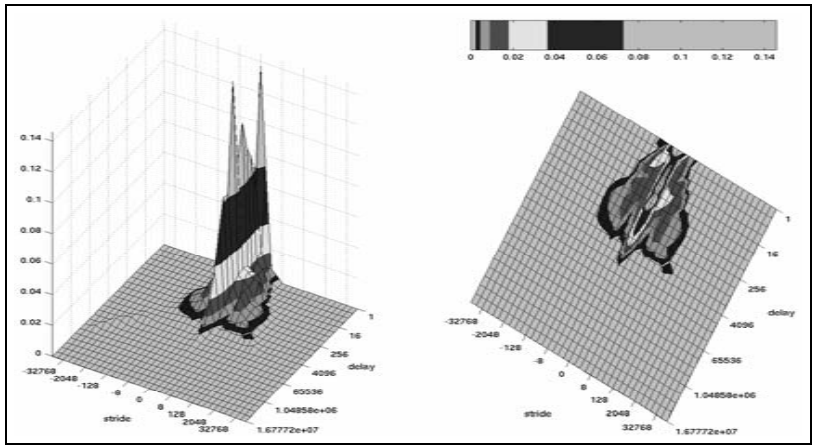
Instruction trace of *bzip2.g9* under Linux.



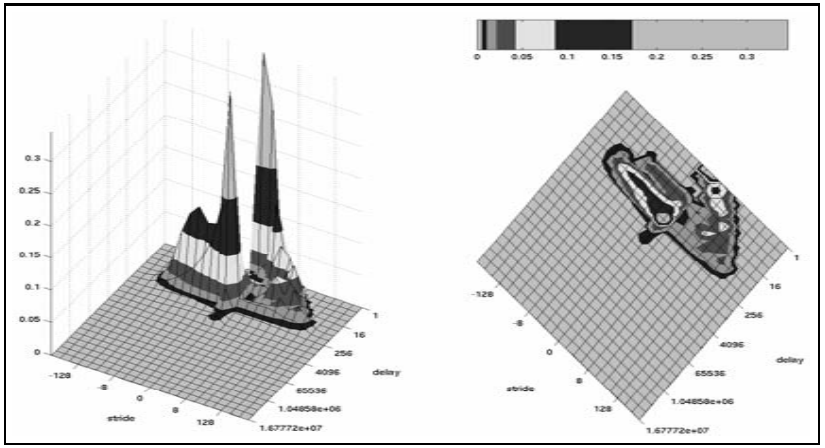
Data trace of *bzip2.g9* under Linux.



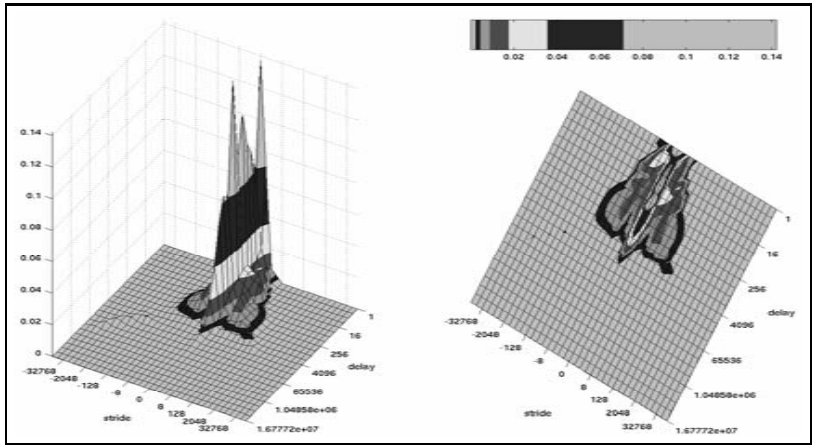
Instruction trace of *bzip2.p7* under Linux.



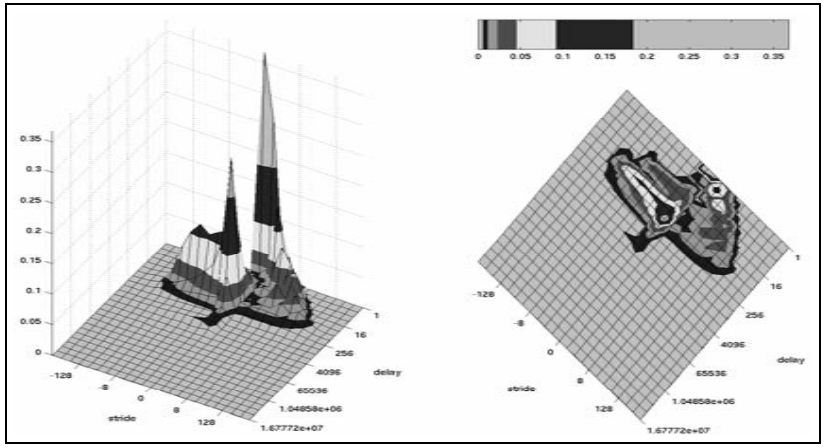
Data trace of *bzip2.p7* under Linux.



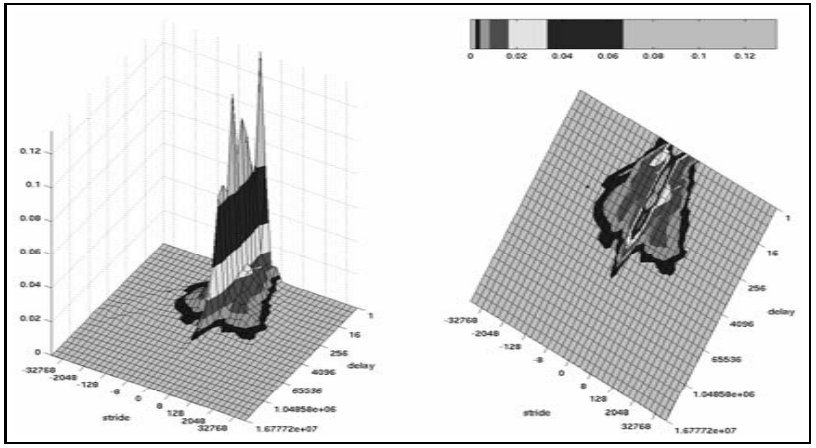
Instruction trace of *bzip2.p9* under Linux.



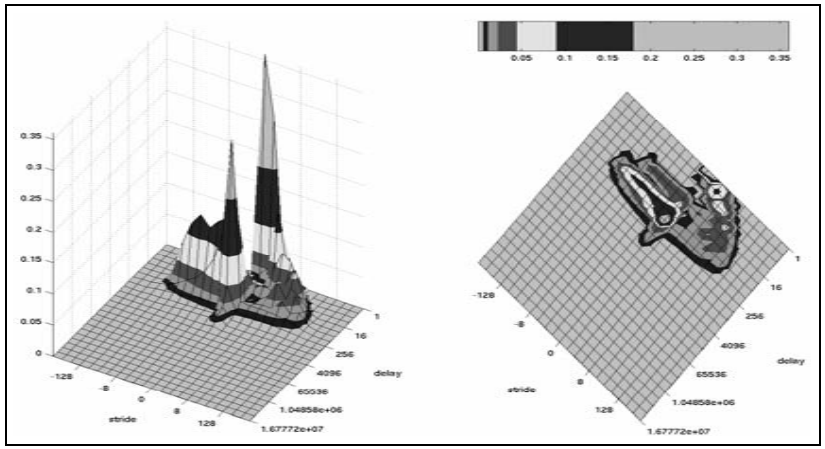
Data trace of *bzip2.p9* under Linux.



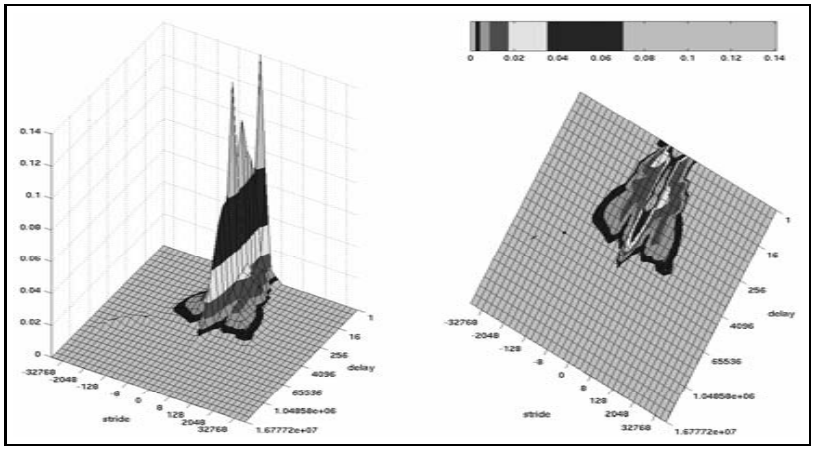
Instruction trace of *bzip2.s7* under Linux.



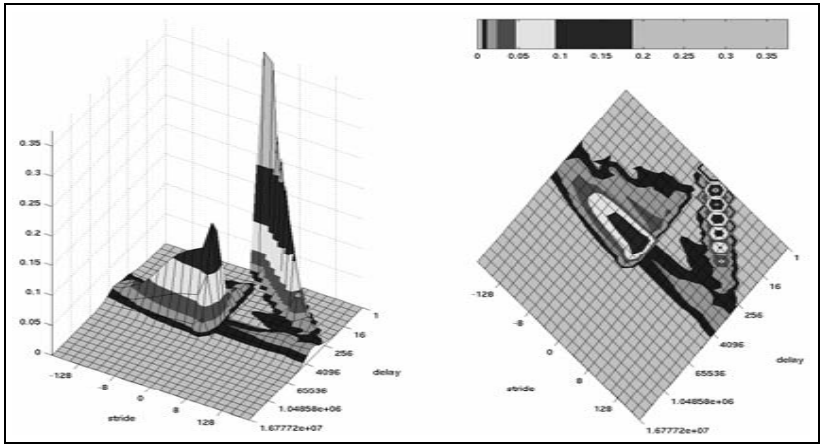
Data trace of *bzip2.s7* under Linux.



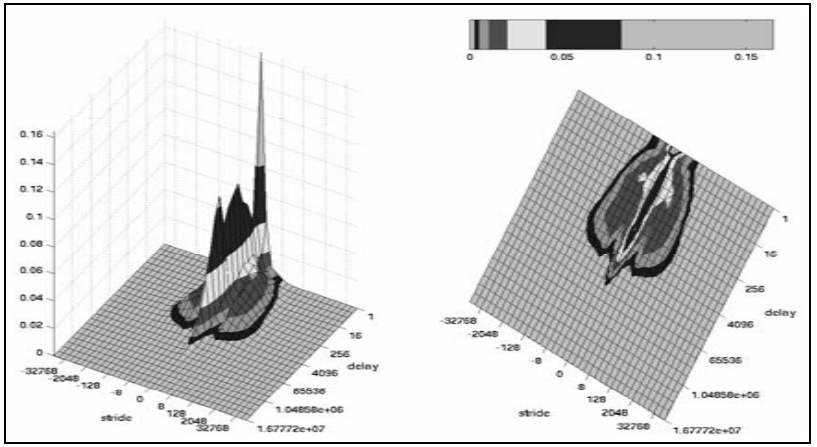
Instruction trace of *bzip2.s9* under Linux.



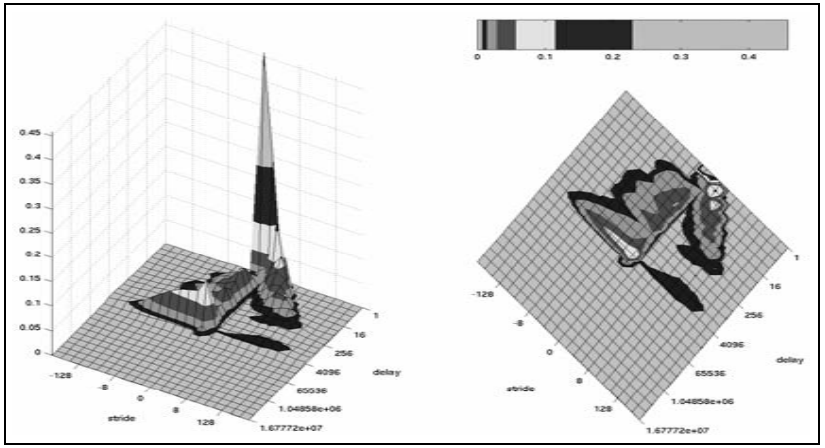
Data trace of *bzip2.s9* under Linux.



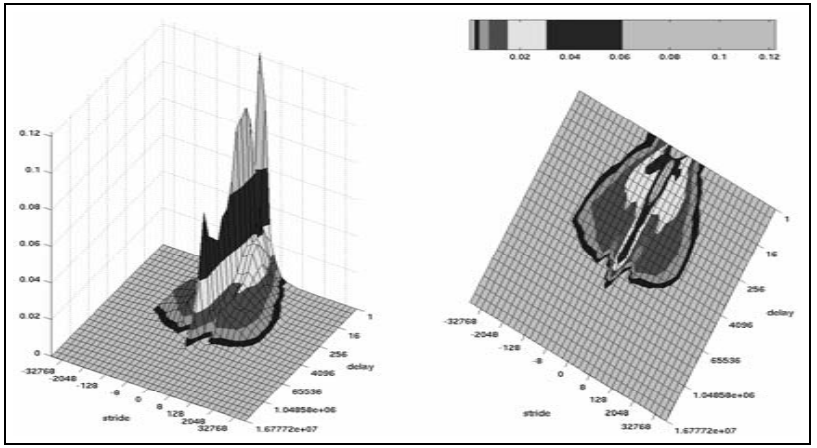
Instruction trace of *crafty* under Linux.



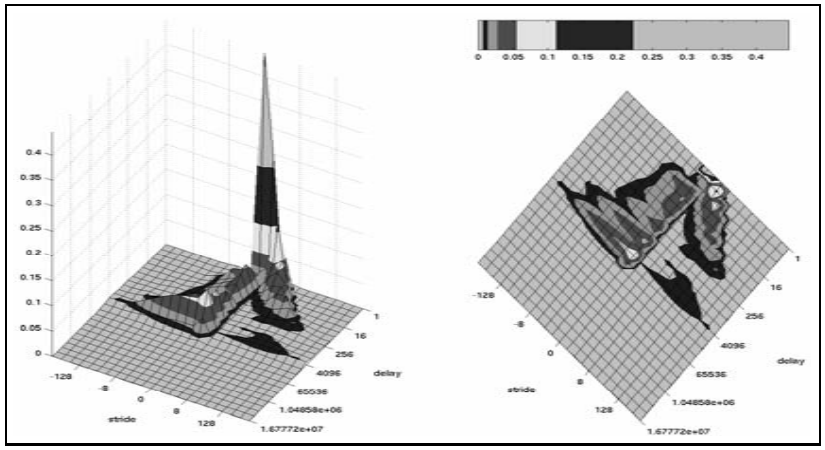
Data trace of *crafty* under Linux.



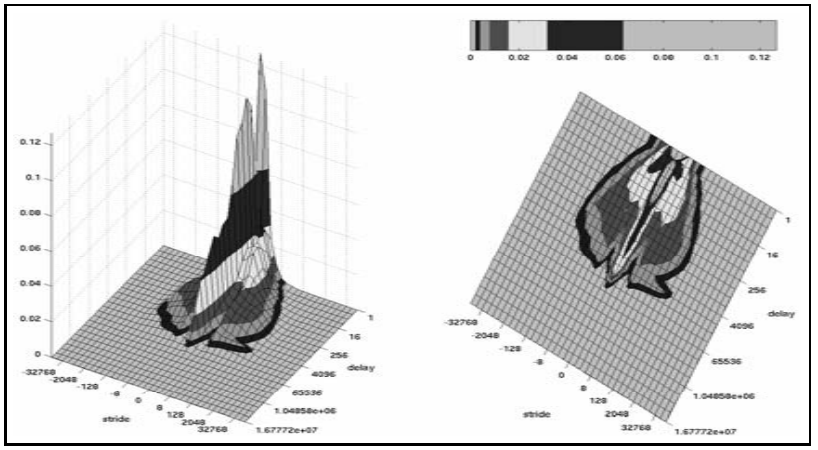
Instruction trace of *eon.cook* under Linux.



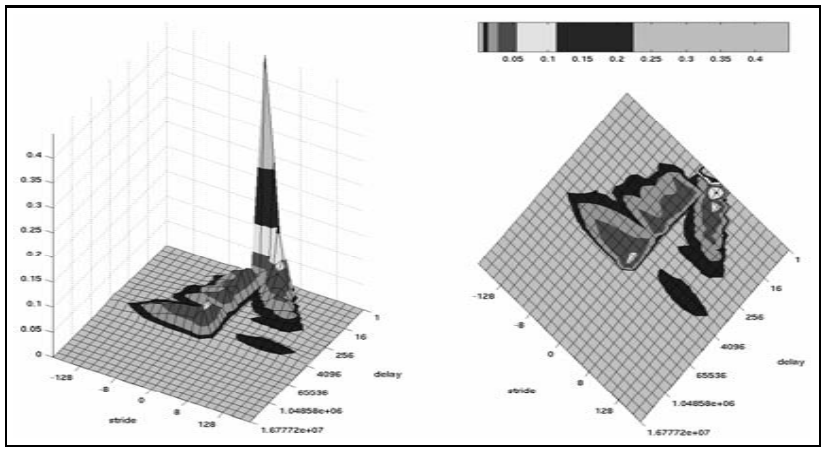
Data trace of *eon.cook* under Linux.



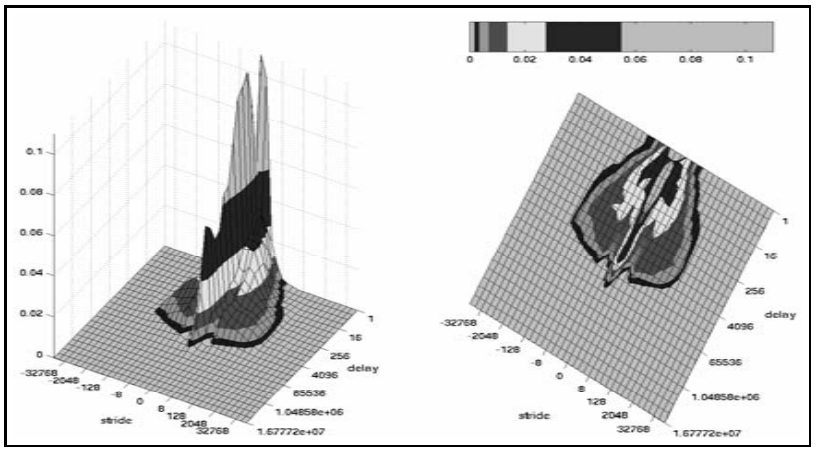
Instruction trace of *eon.kajiya* under Linux.



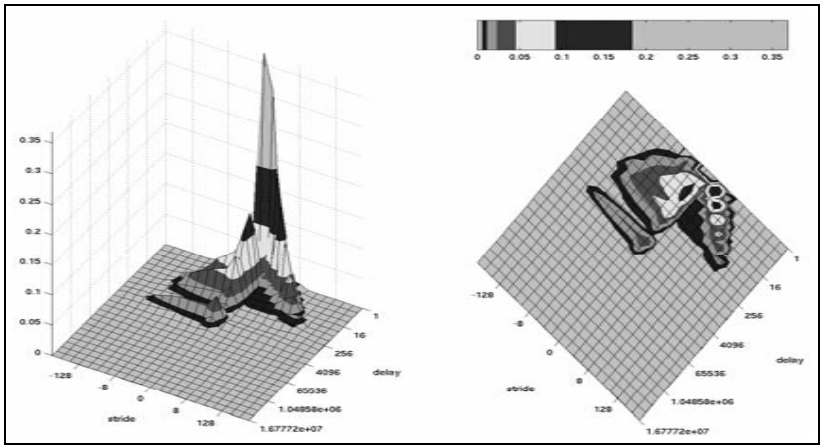
Data trace of *eon.kajiya* under Linux.



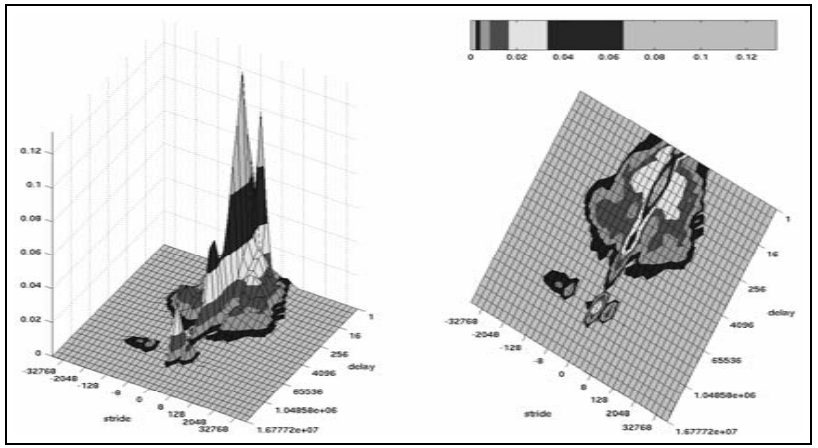
Instruction trace of *eon.rushmeier* under Linux.



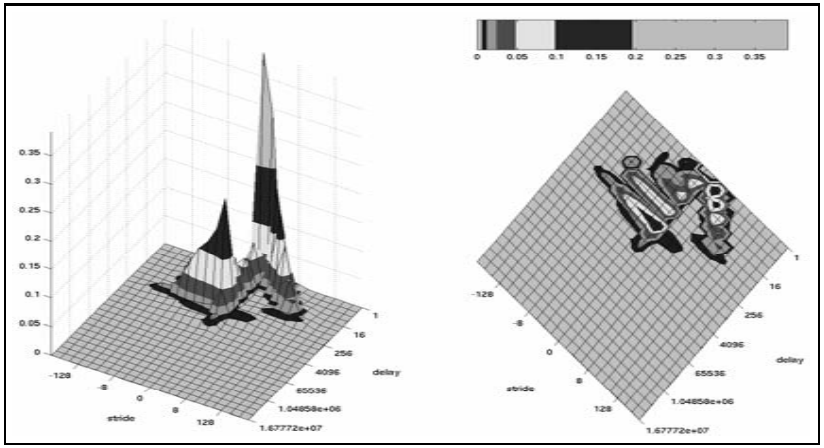
Data trace of *eon.rushmeier* under Linux.



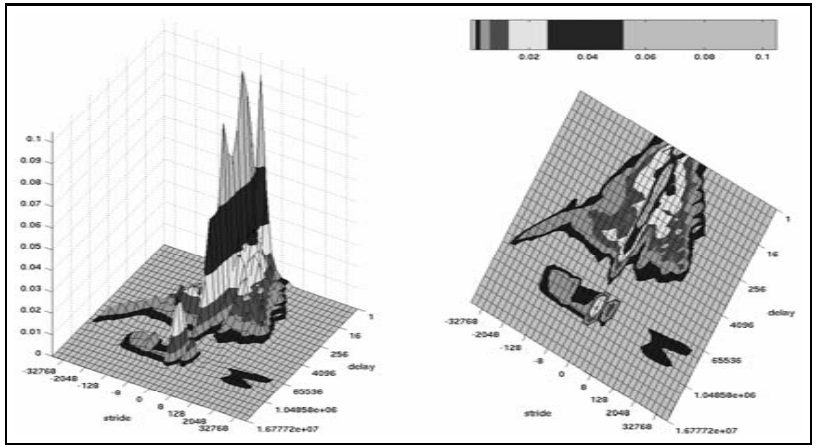
Instruction trace of *equake* under Linux.



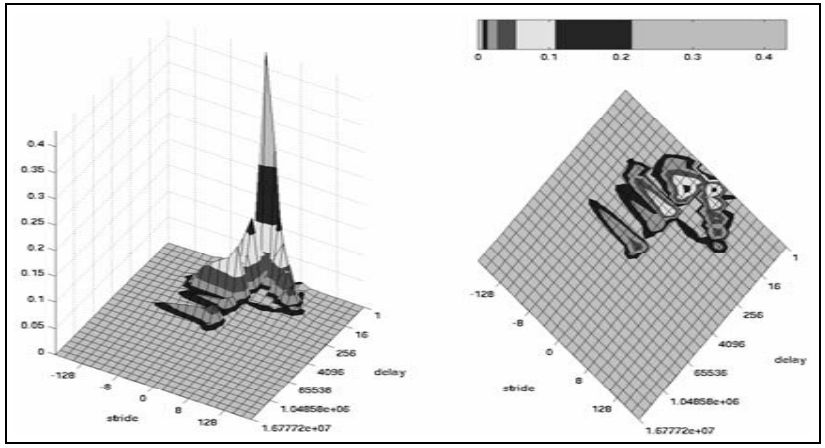
Data trace of *equake* under Linux.



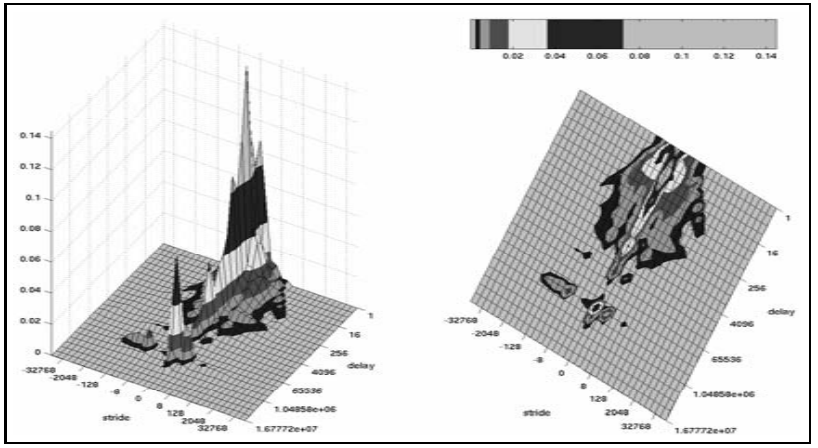
Instruction trace of *facerec* under Linux.



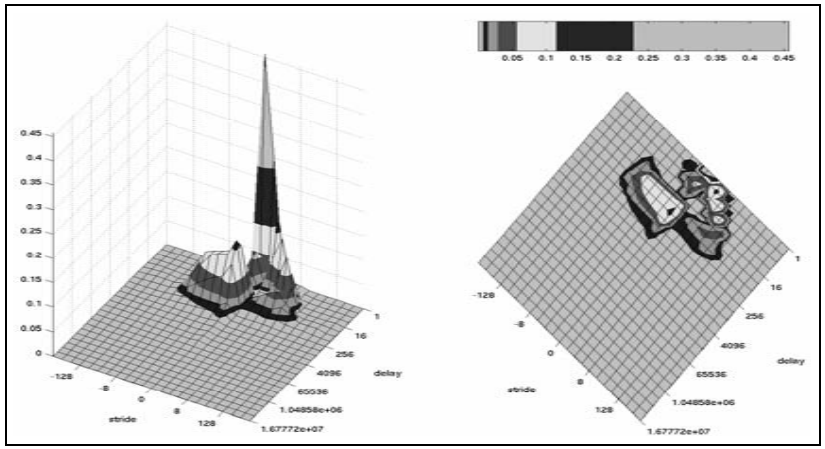
Data trace of *facerec* under Linux.



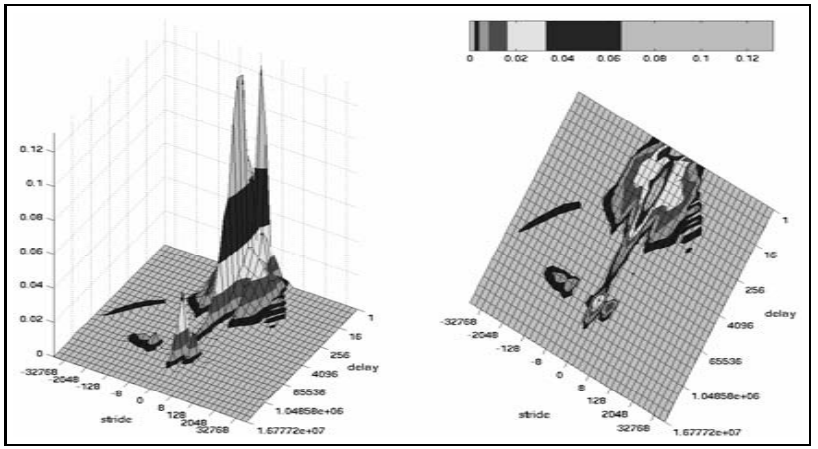
Instruction trace of *fma3d* under Linux.



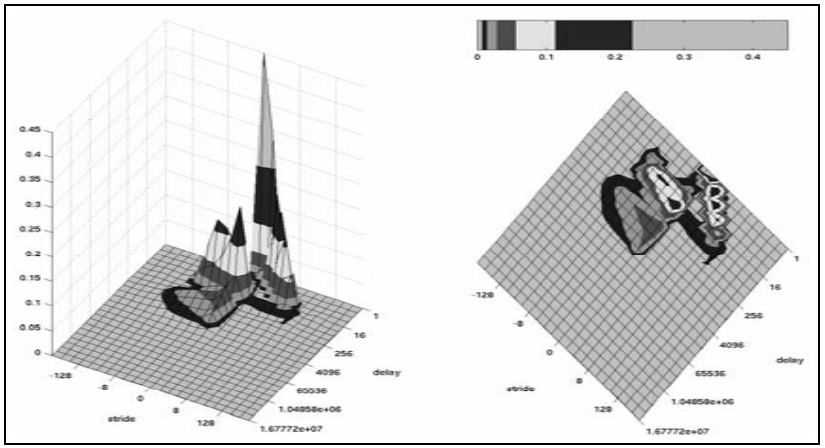
Data trace of *fma3d* under Linux.



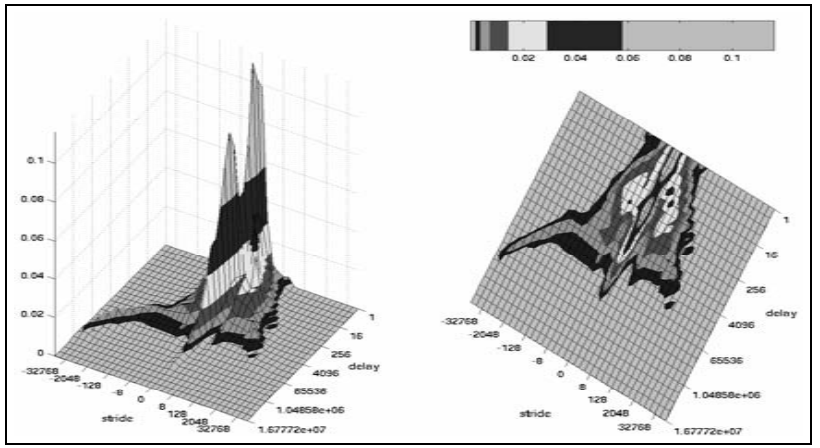
Instruction trace of *galgel* under Linux.



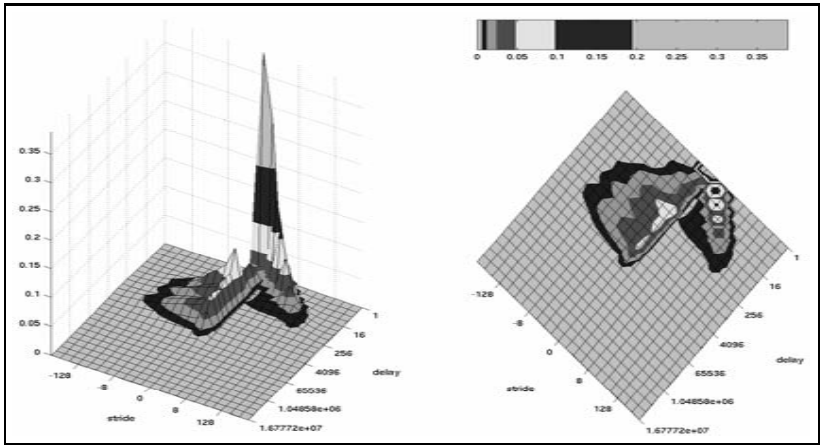
Data trace of *galgel* under Linux.



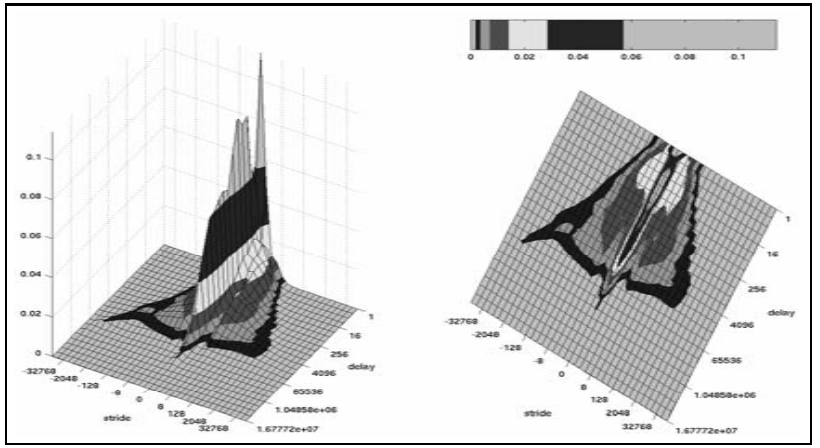
Instruction trace of *gap* under Linux.



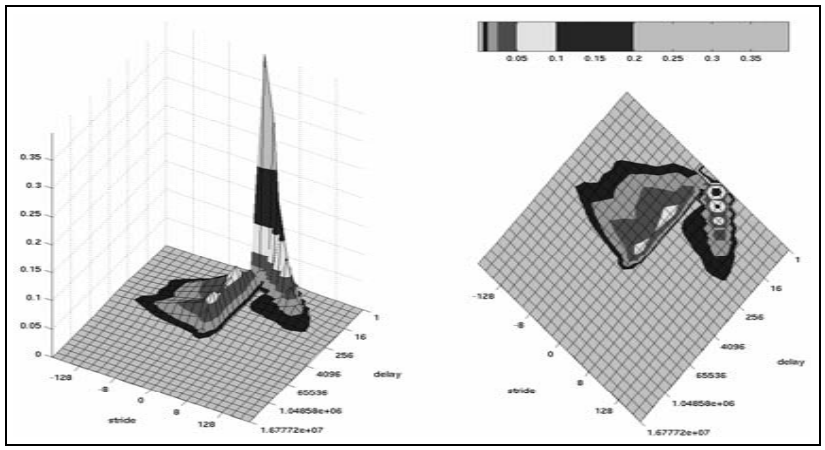
Data trace of *gap* under Linux.



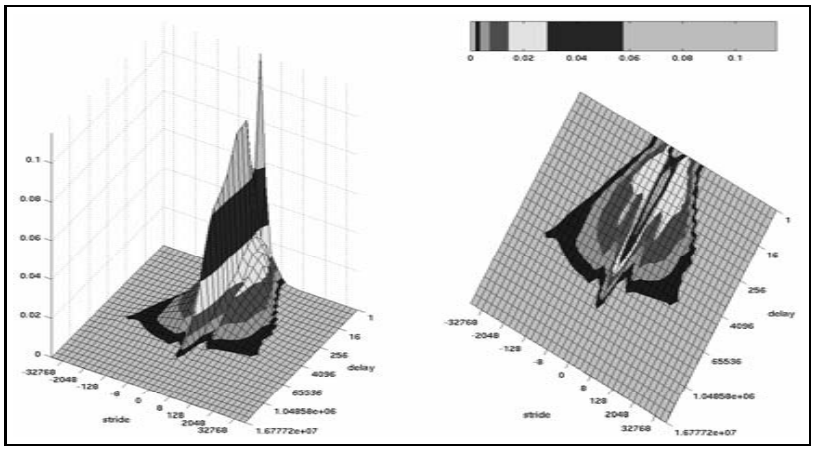
Instruction trace of *gcc.166* under Linux.



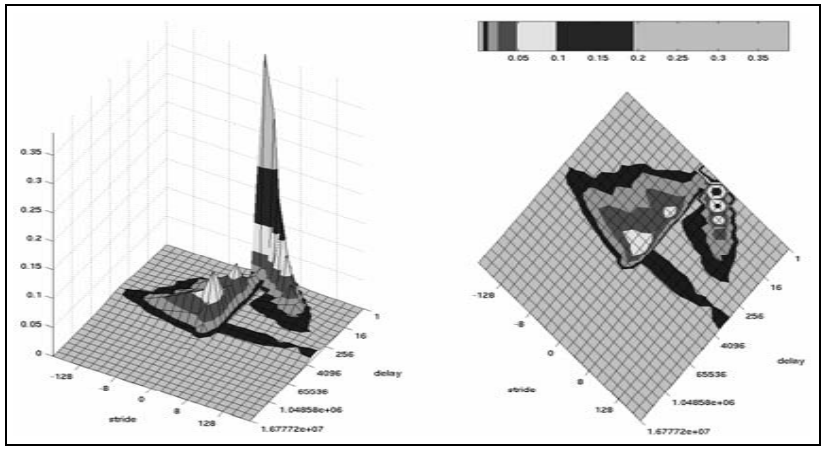
Data trace of *gcc.166* under Linux.



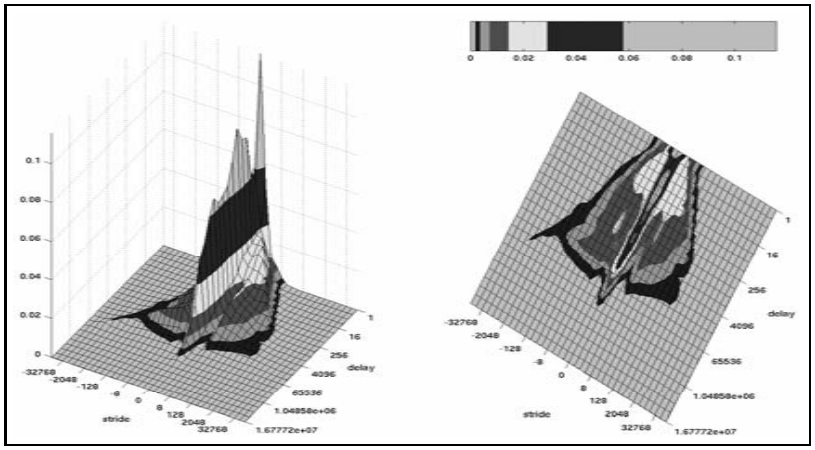
Instruction trace of *gcc.200* under Linux.



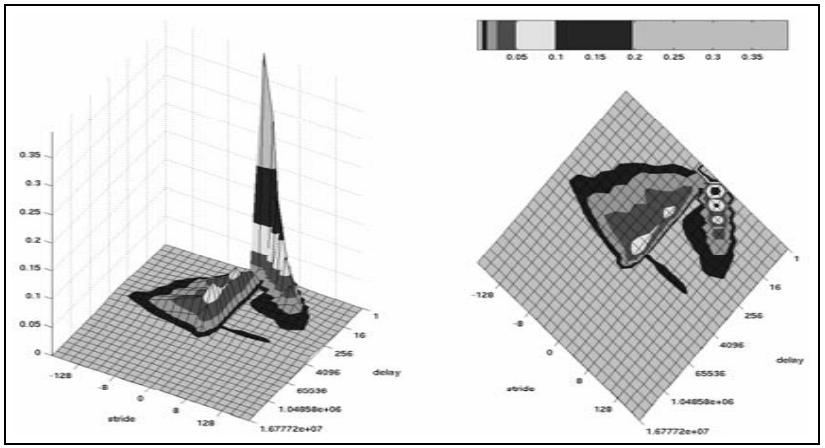
Data trace of *gcc.200* under Linux.



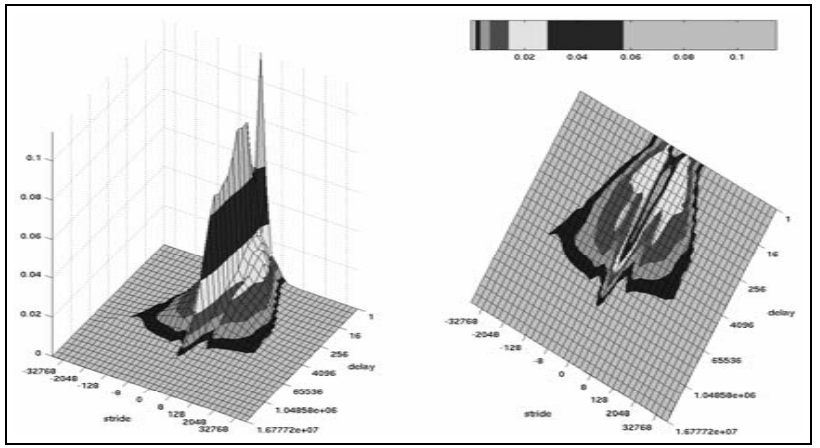
Instruction trace of *gcc.expr* under Linux.



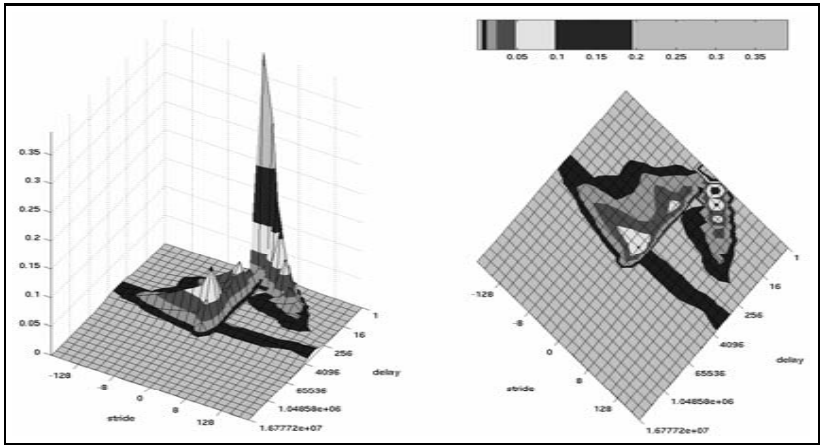
Data trace of *gcc.expr* under Linux.



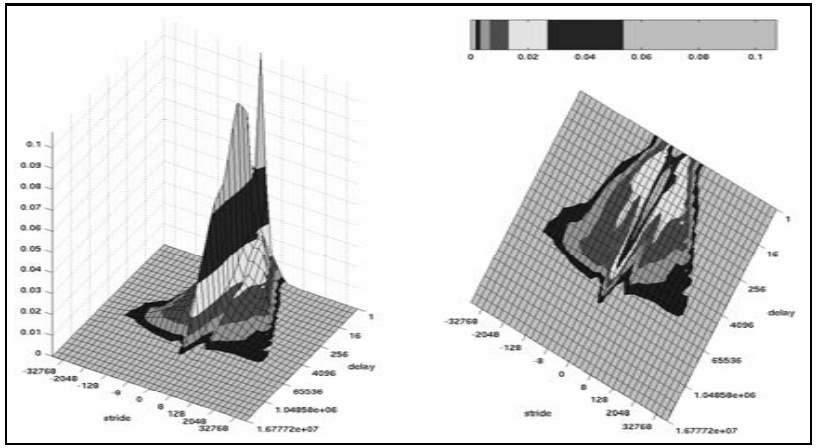
Instruction trace of *gcc.integ* under Linux.



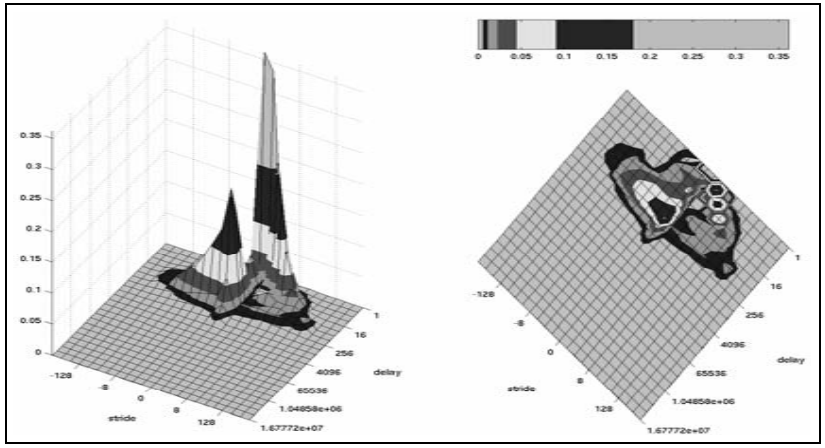
Data trace of *gcc.integ* under Linux.



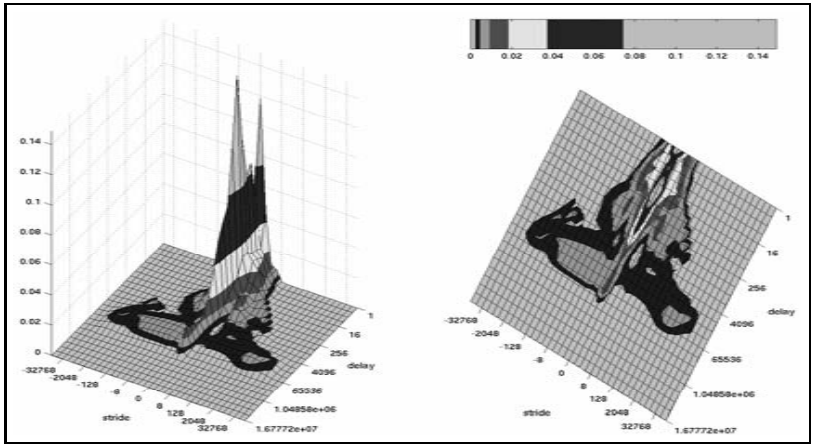
Instruction trace of *gcc.scilab* under Linux.



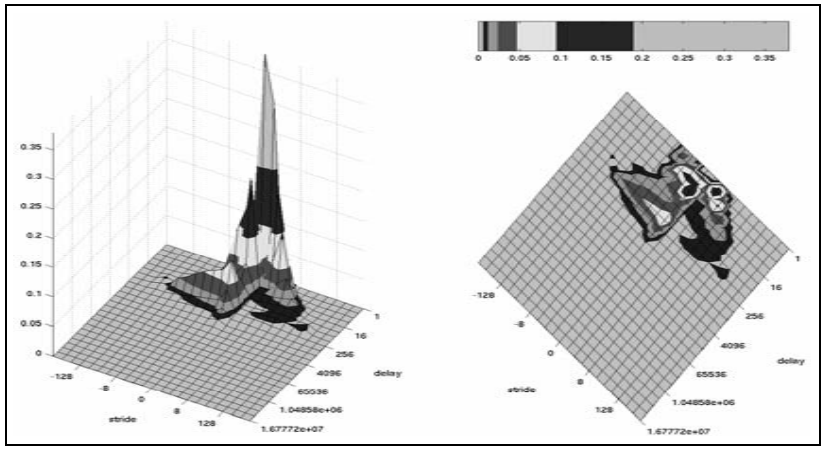
Data trace of *gcc.scilab* under Linux.



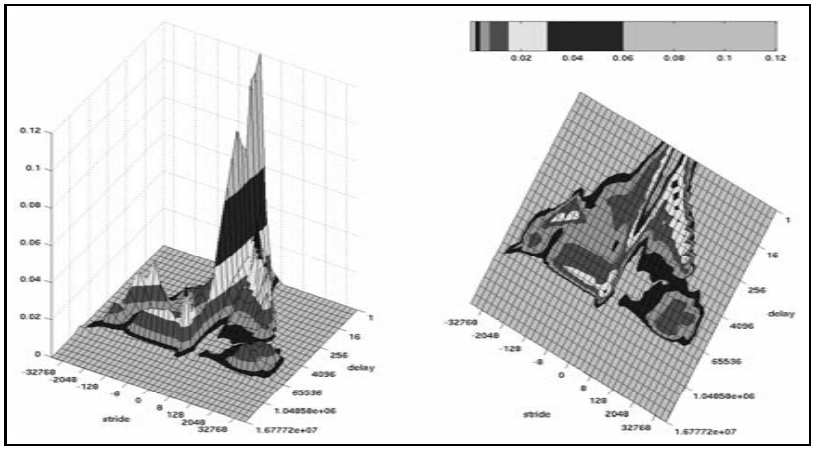
Instruction trace of *gzip.graphic* under Linux.



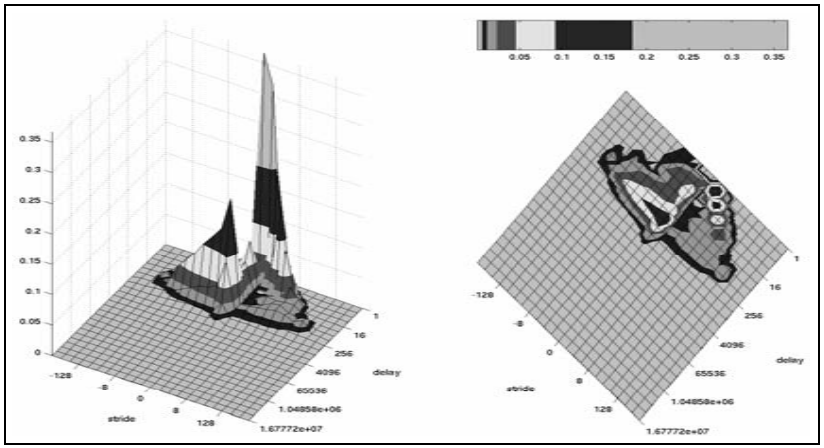
Data trace of *gzip.graphic* under Linux.



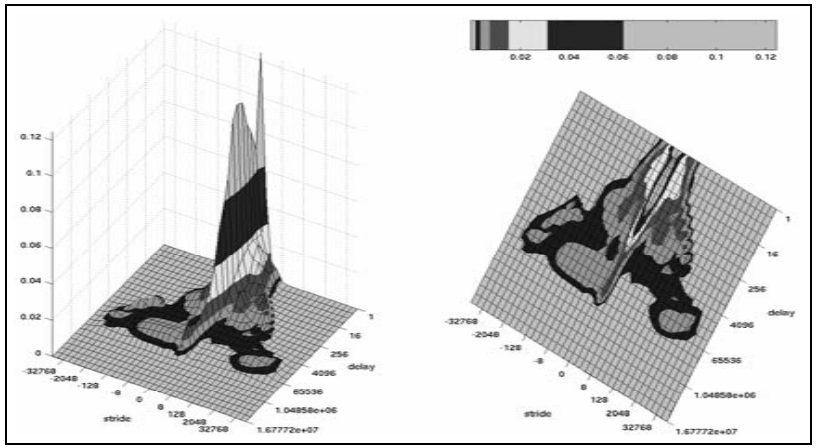
Instruction trace of *gzip.log* under Linux.



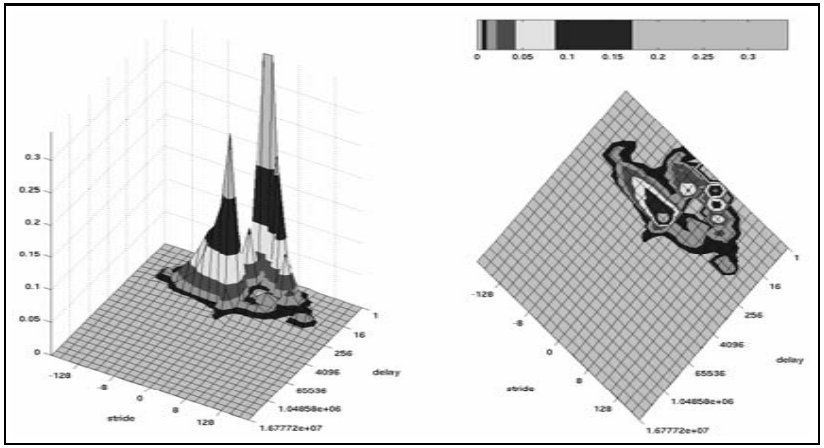
Data trace of *gzip.log* under Linux.



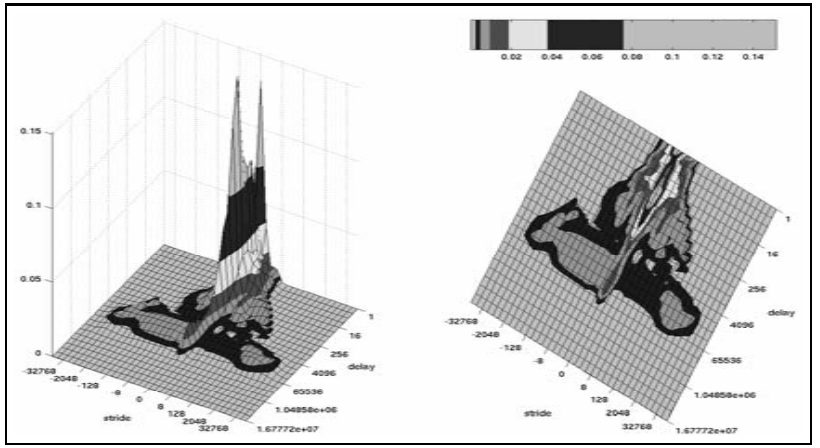
Instruction trace of *gzip.program* under Linux.



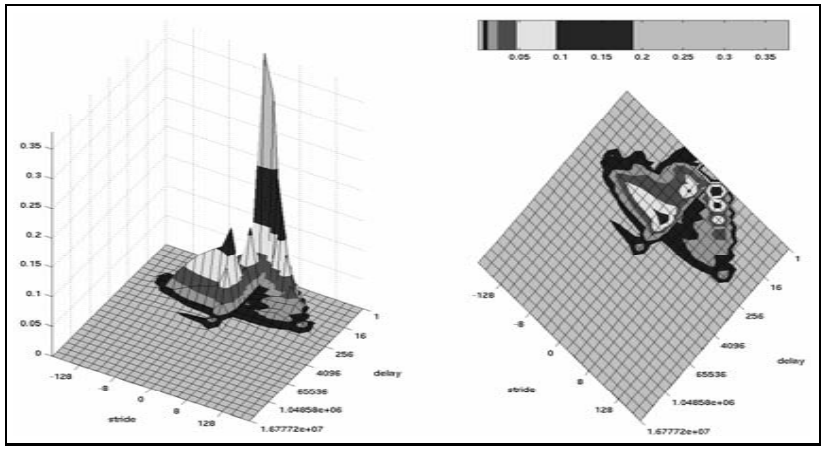
Data trace of *gzip.program* under Linux.



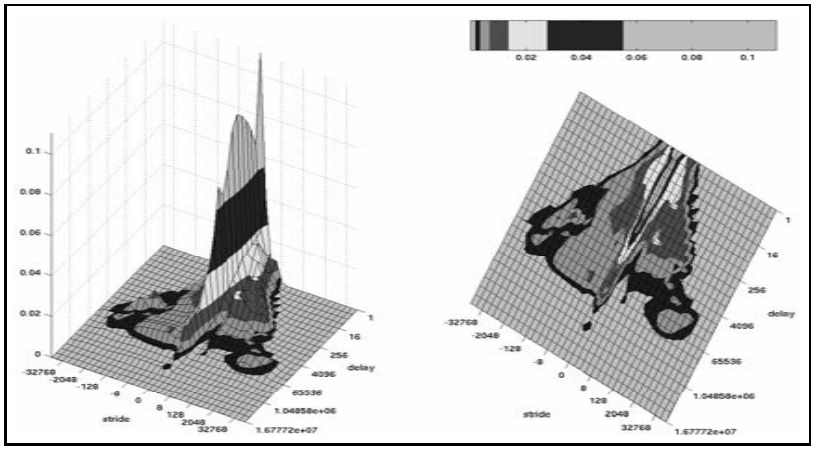
Instruction trace of *gzip.random* under Linux.



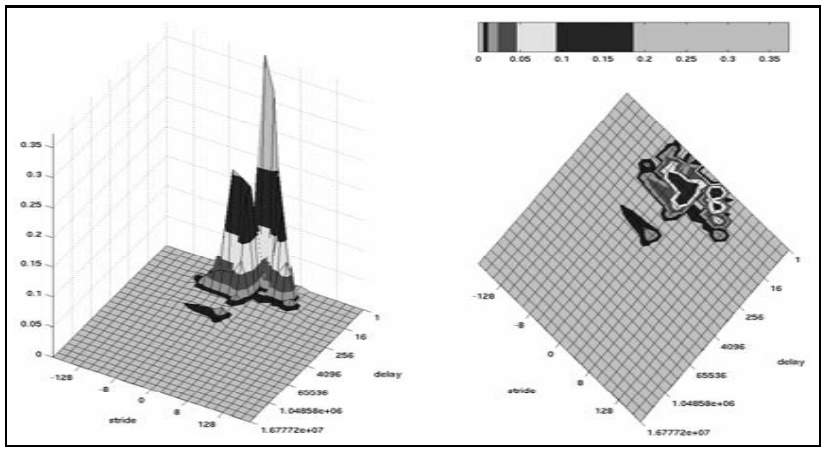
Data trace of *gzip.random* under Linux.



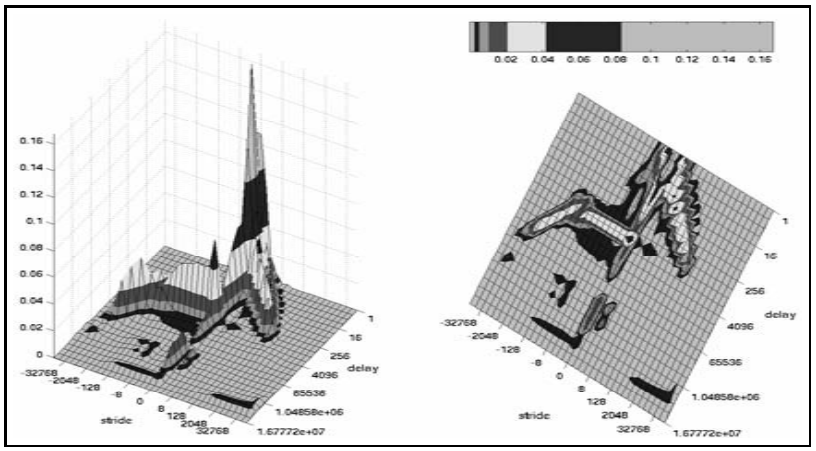
Instruction trace of *gzip.source* under Linux.



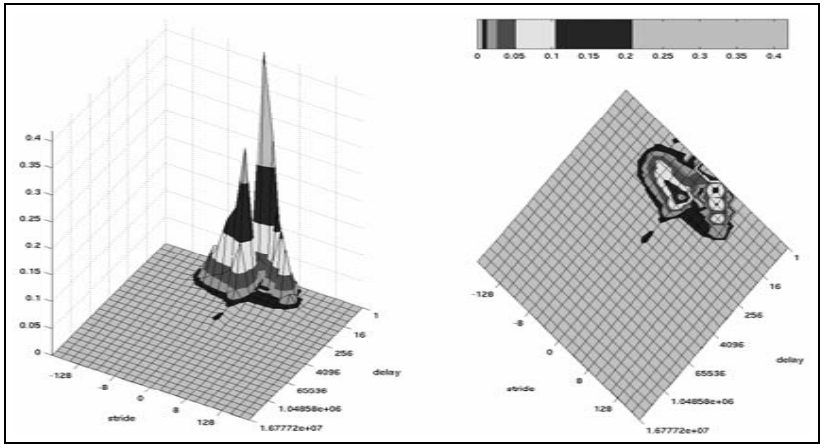
Data trace of *gzip.source* under Linux.



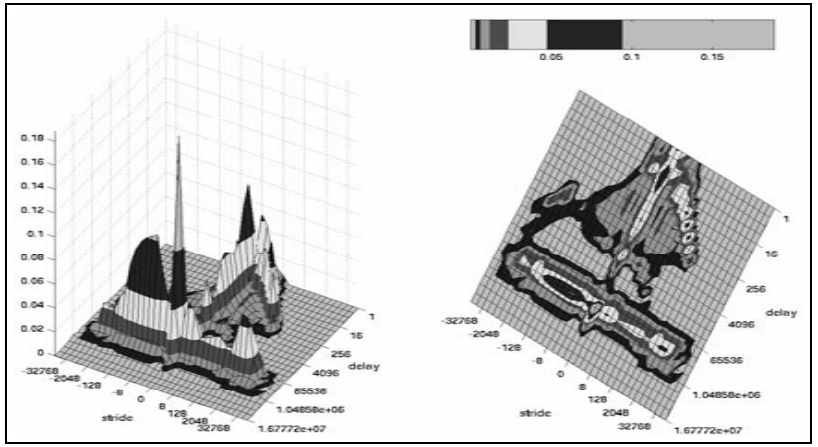
Instruction trace of *lucas* under Linux.



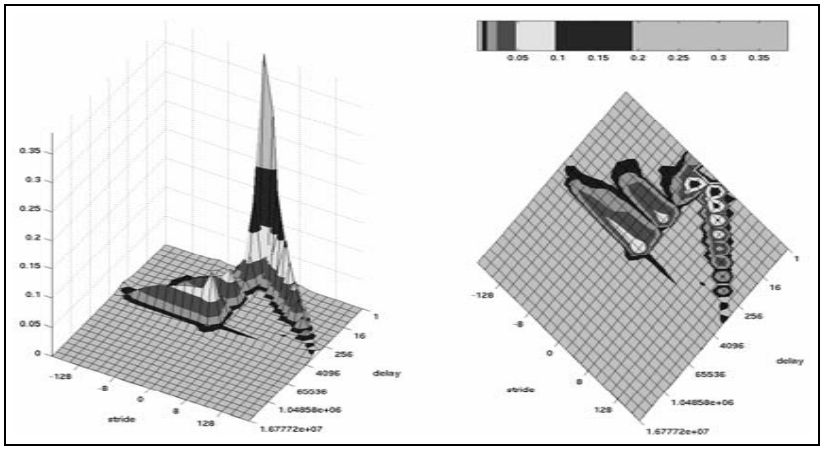
Data trace of *lucas* under Linux.



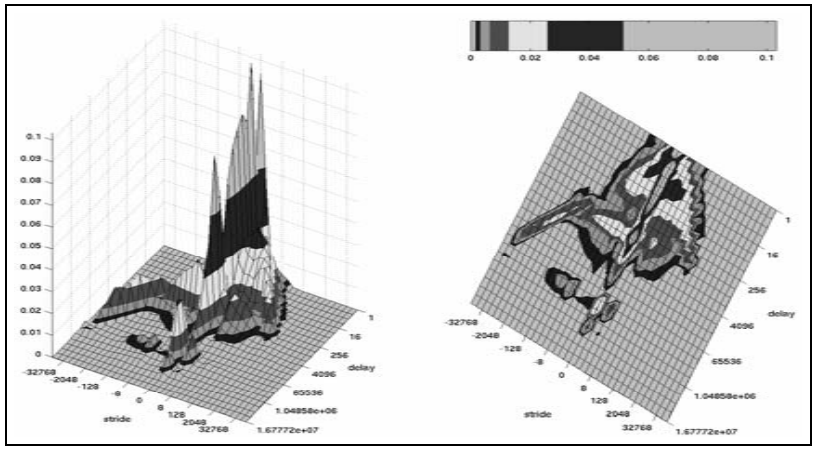
Instruction trace of *mcf* under Linux.



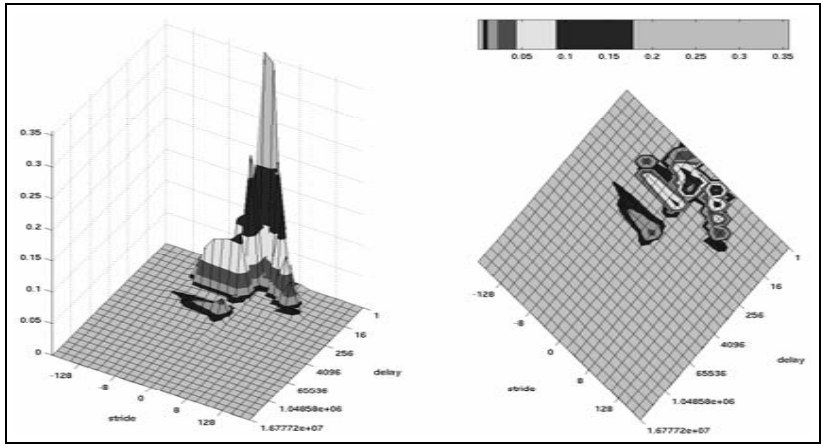
Data trace of *mcf* under Linux.



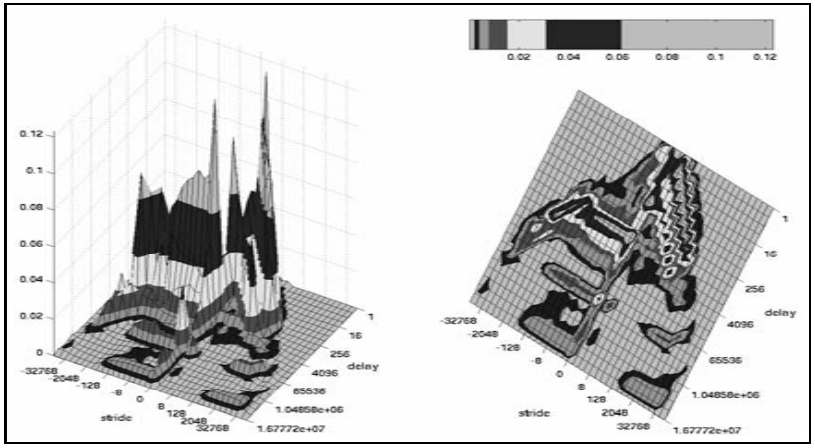
Instruction trace of *mesa* under Linux.



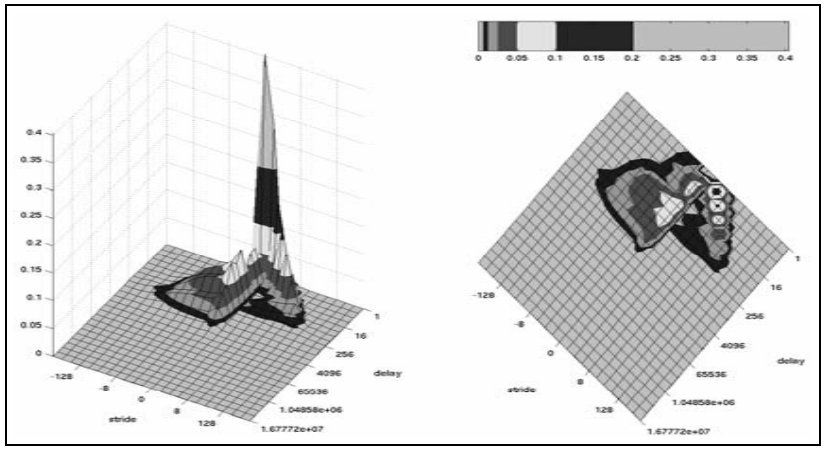
Data trace of *mesa* under Linux.



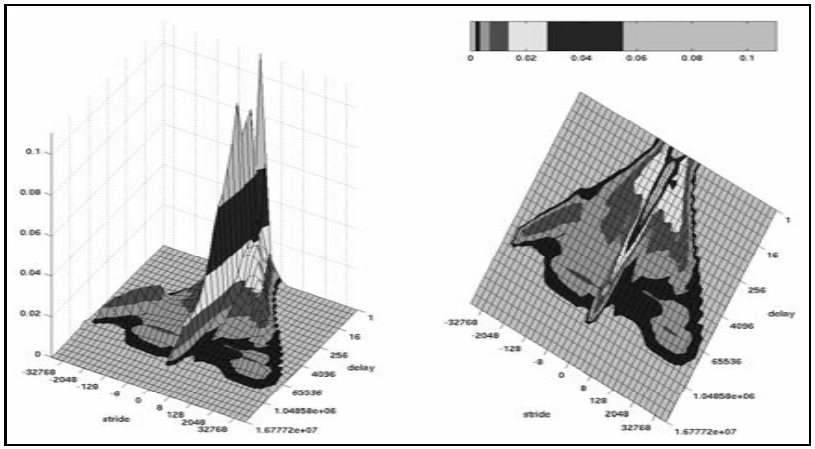
Instruction trace of *mgrid* under Linux.



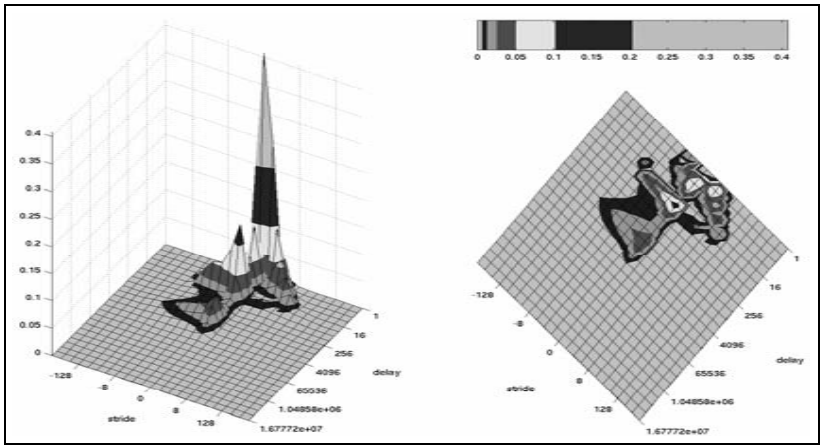
Data trace of *mgrid* under Linux.



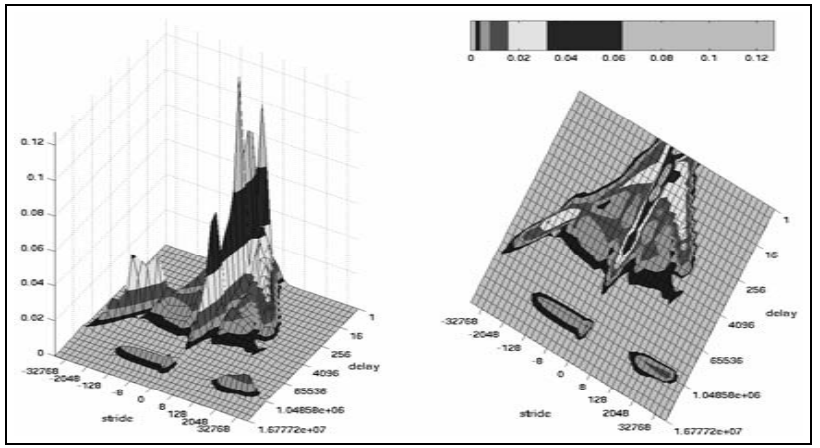
Instruction trace of *parser* under Linux.



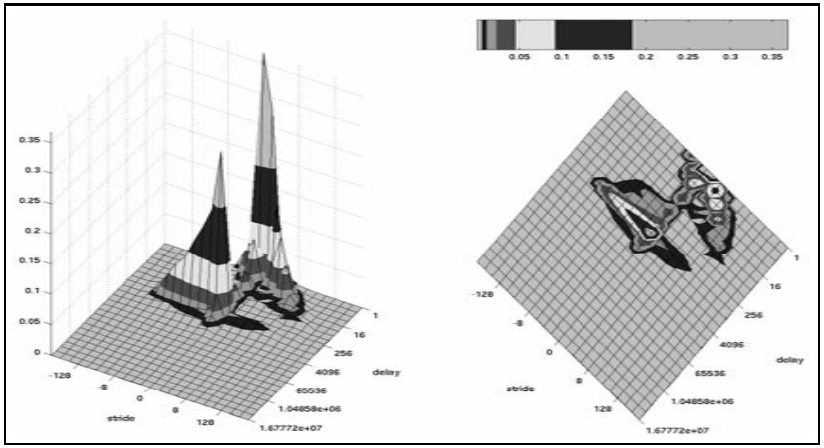
Data trace of *parser* under Linux.



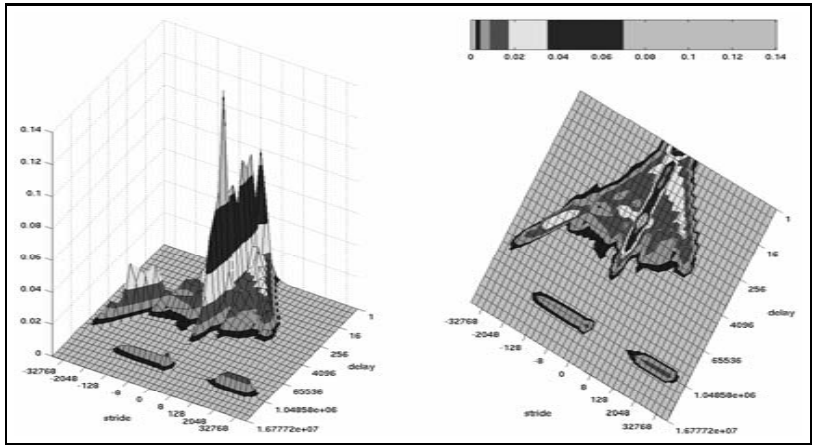
Instruction trace of *perlbmk.diffmail* under Linux.



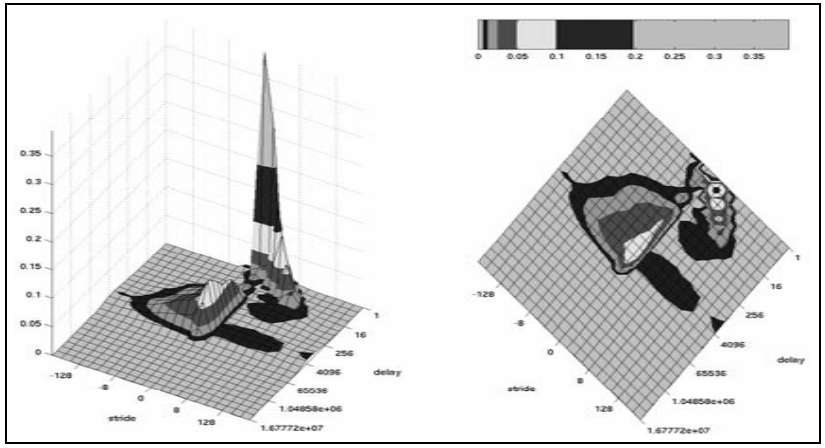
Data trace of *perlbmk.diffmail* under Linux.



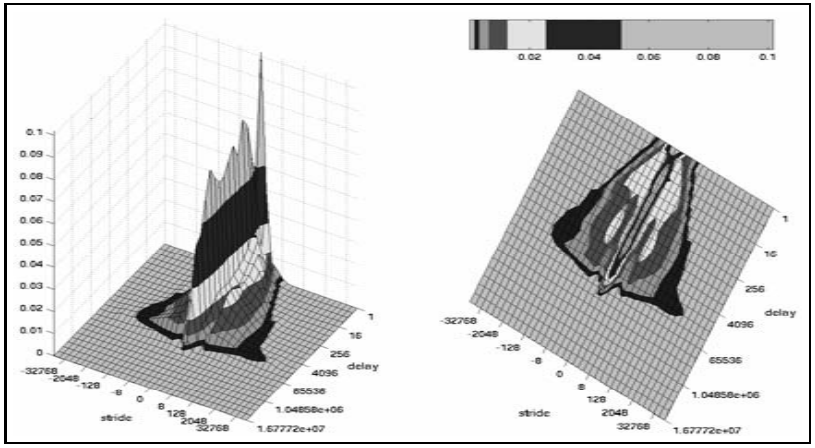
Instruction trace of *perlbmk.makerand* under Linux.



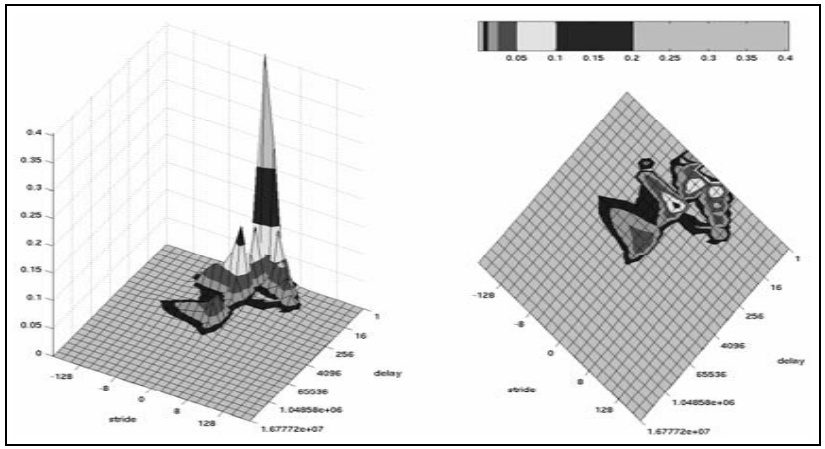
Data trace of *perlbmk.makerand* under Linux.



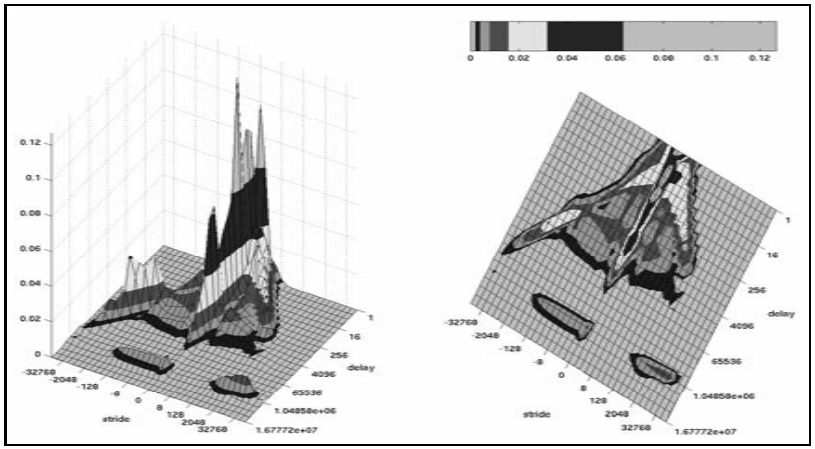
Instruction trace of *perlbmk.perfect* under Linux.



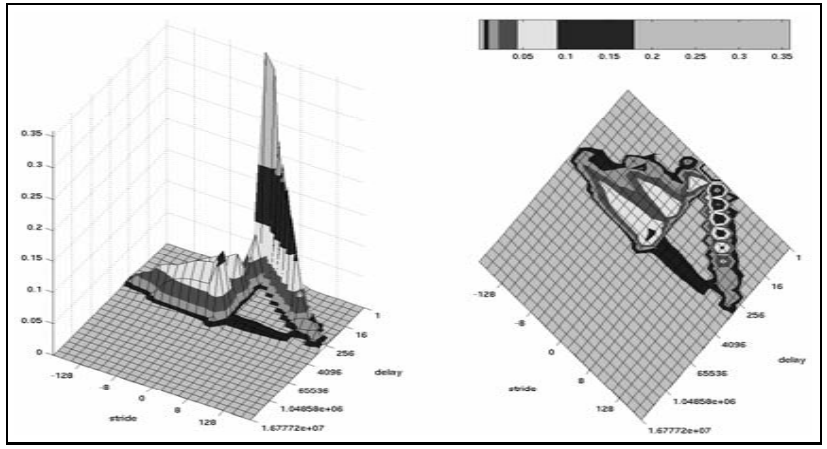
Data trace of *perlbmk.perfect* under Linux.



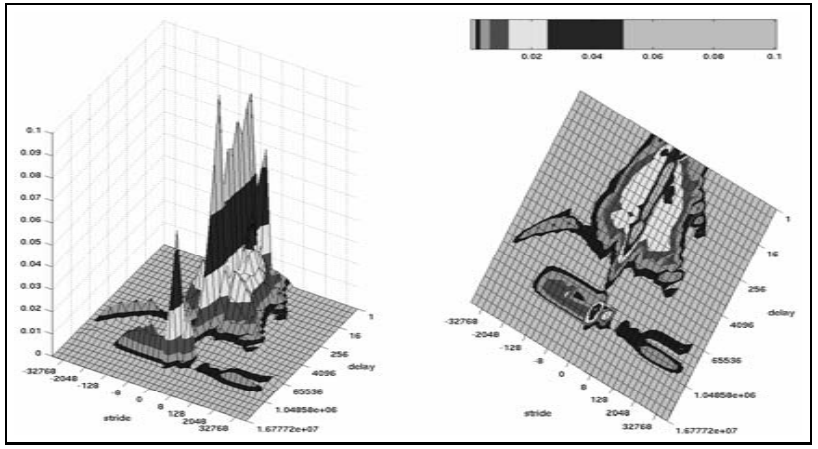
Instruction trace of *perlbmk.splitmail* under Linux.



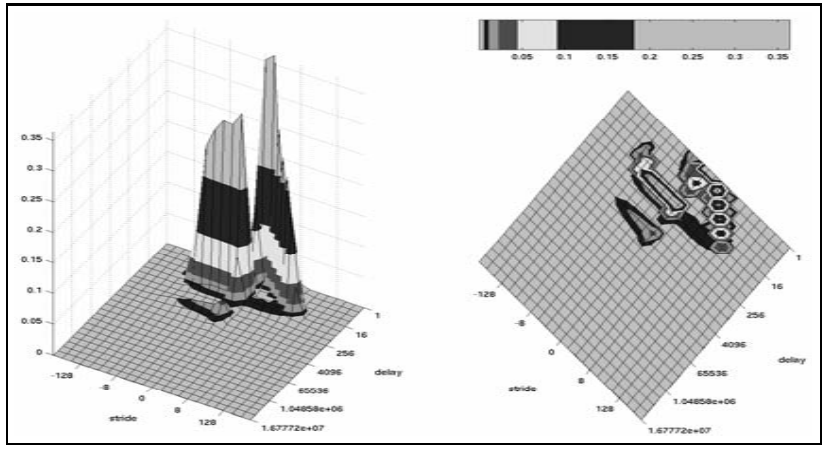
Data trace of *perlbmk.splitmail* under Linux.



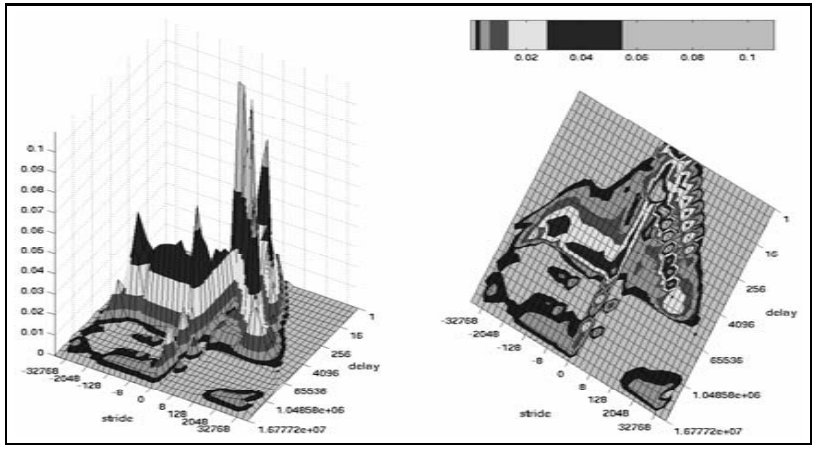
Instruction trace of *sixtrack* under Linux.



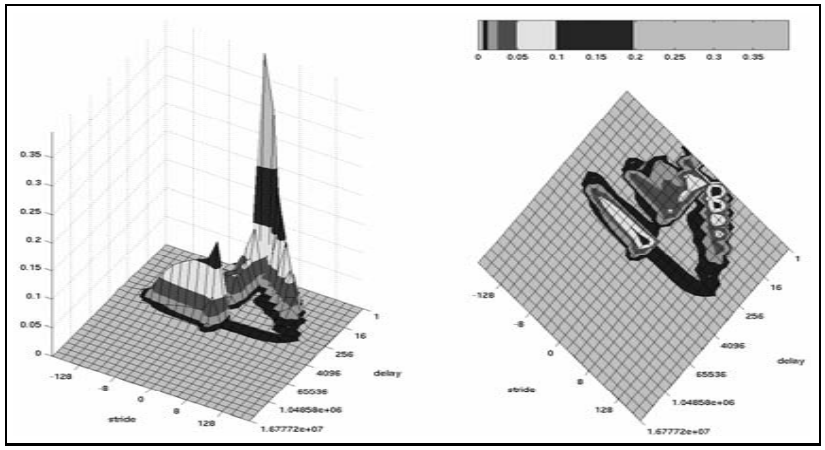
Data trace of *sixtrack* under Linux.



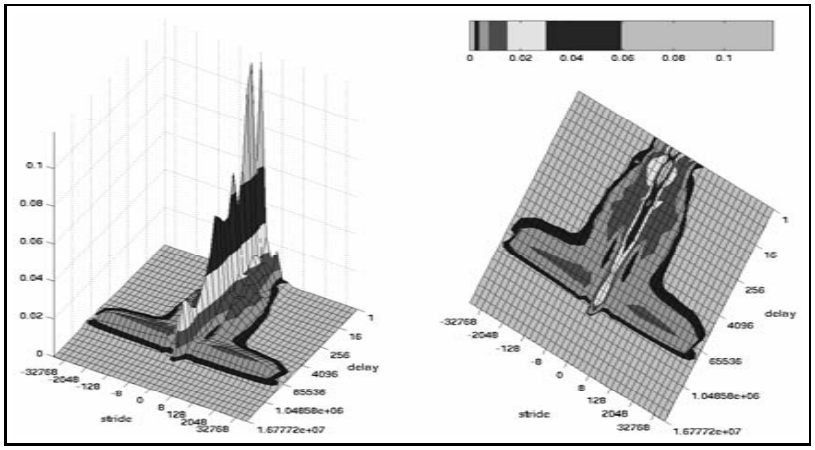
Instruction trace of *swim* under Linux.



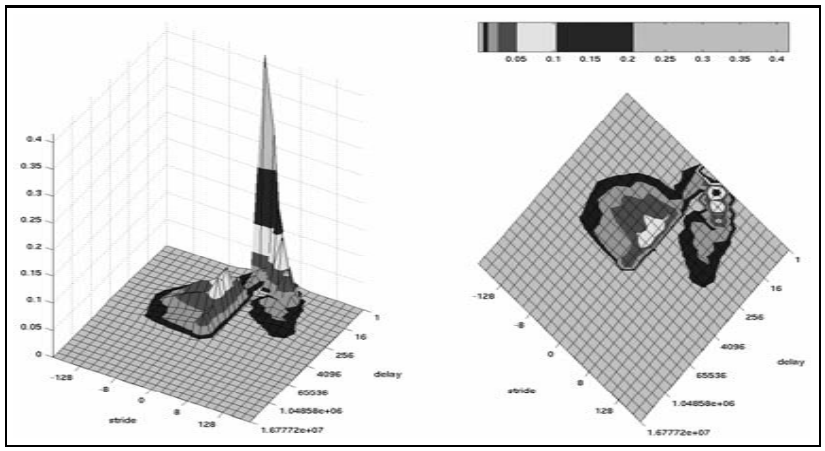
Data trace of *swim* under Linux.



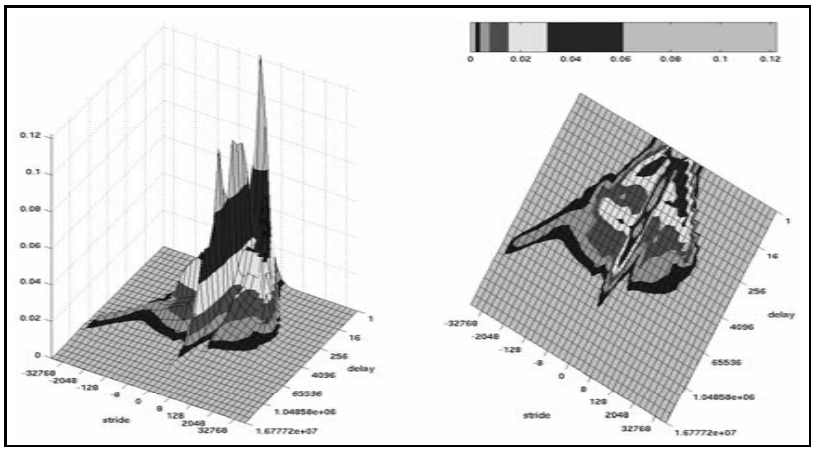
Instruction trace of *twolf* under Linux.



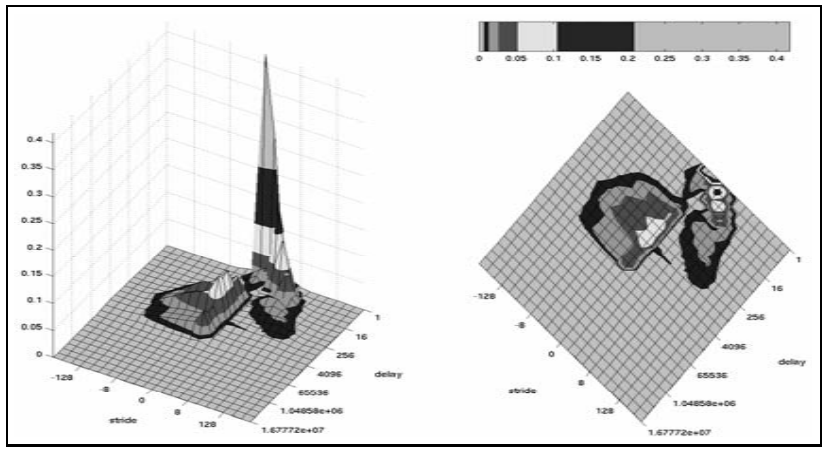
Data trace of *twolf* under Linux.



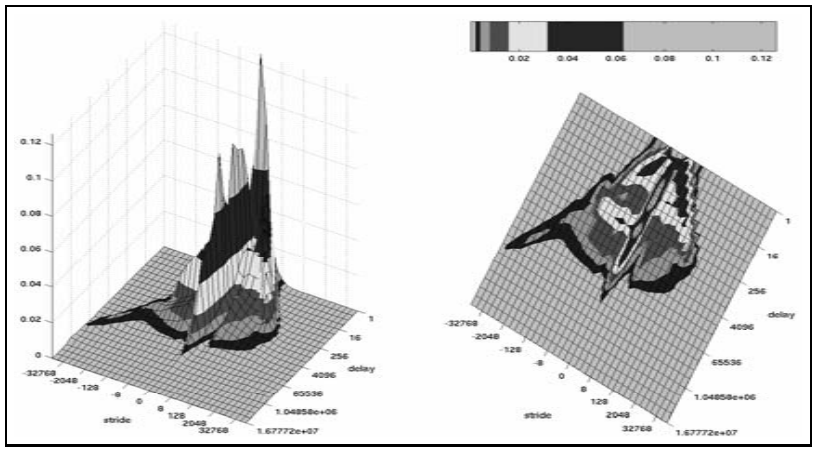
Instruction trace of *vortex.one* under Linux.



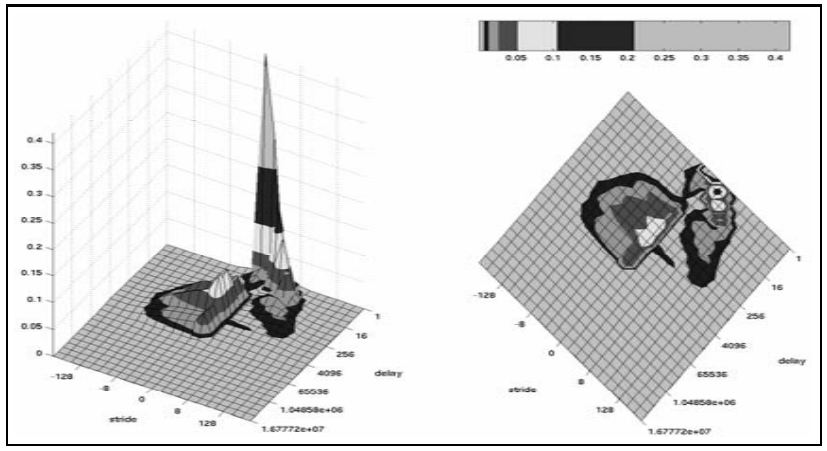
Data trace of *vortex.one* under Linux.



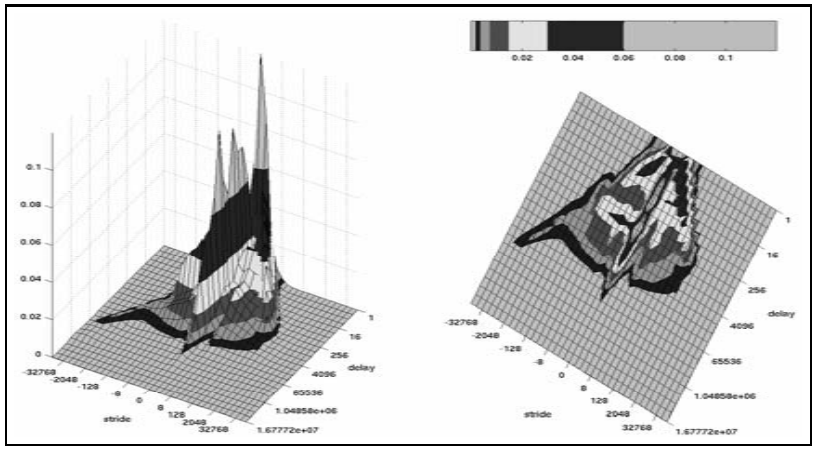
Instruction trace of *vortex.three* under Linux.



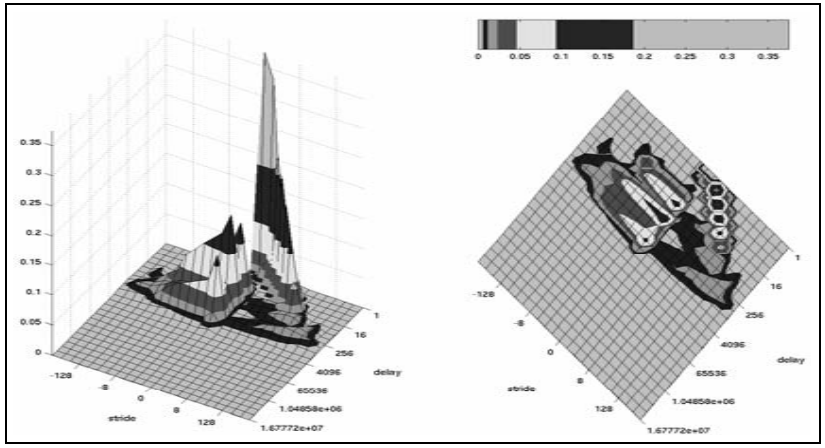
Data trace of *vortex.three* under Linux.



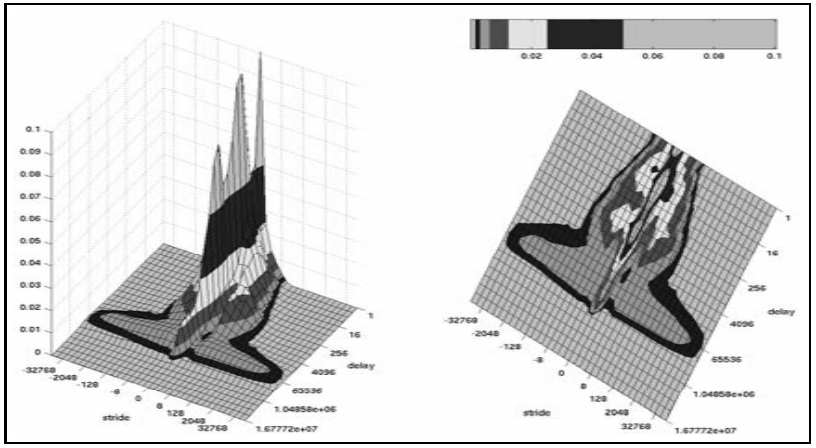
Instruction trace of *vortex.two* under Linux.



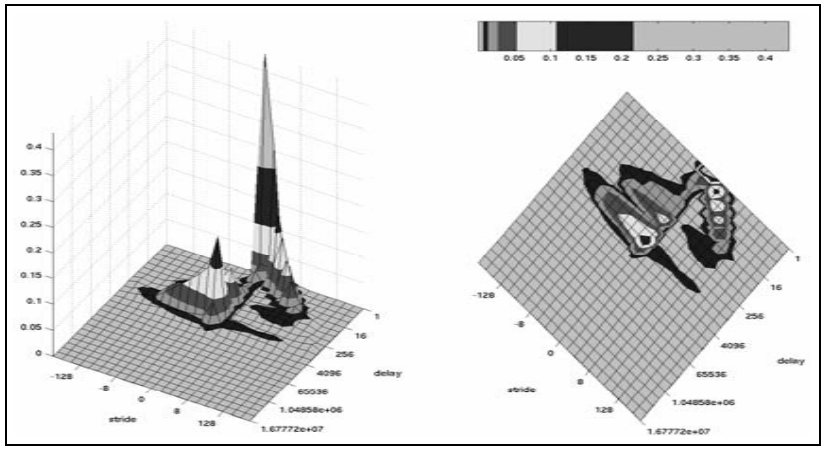
Data trace of *vortex.two* under Linux.



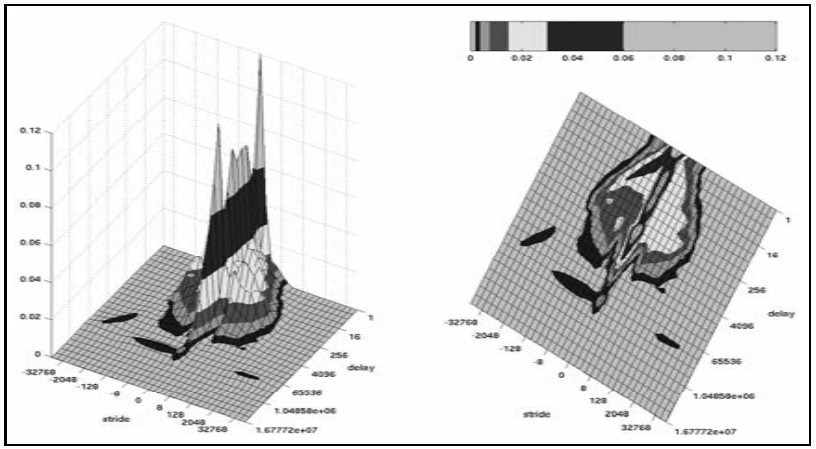
Instruction trace of *vpr.place* under Linux.



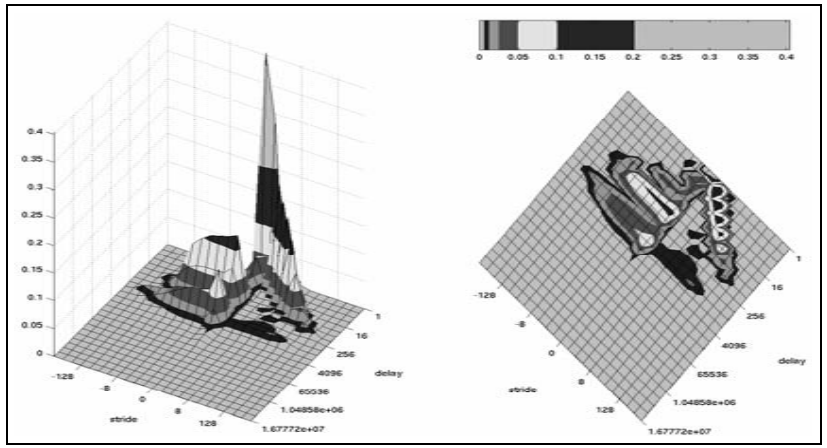
Data trace of *vpr.place* under Linux.



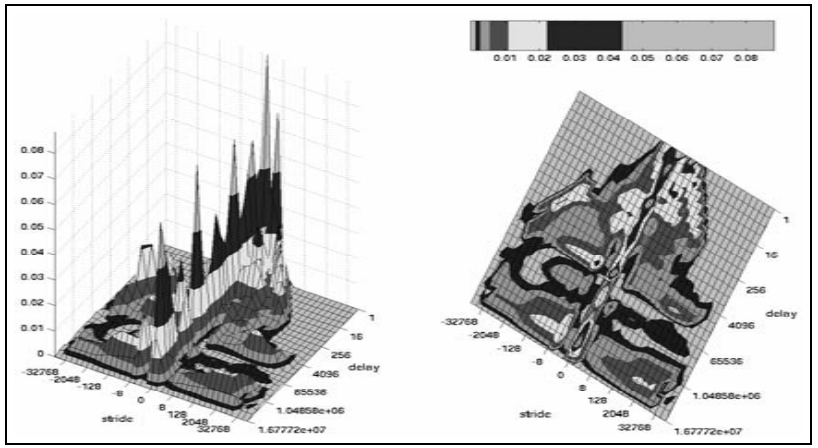
Instruction trace of *vpr.route* under Linux.



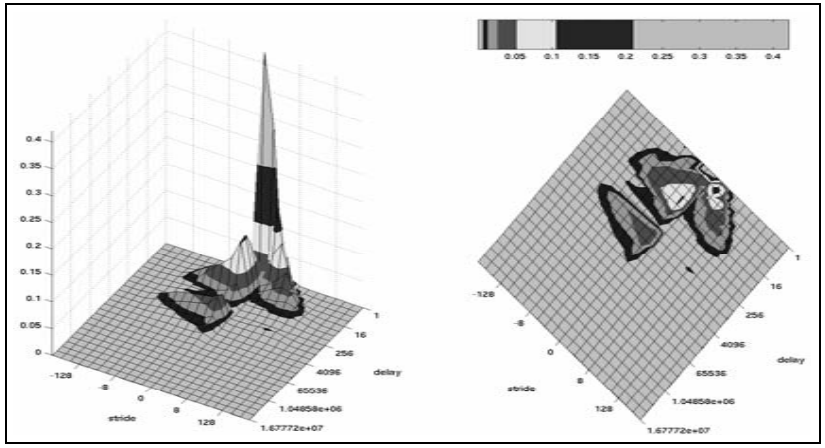
Data trace of *vpr.route* under Linux.



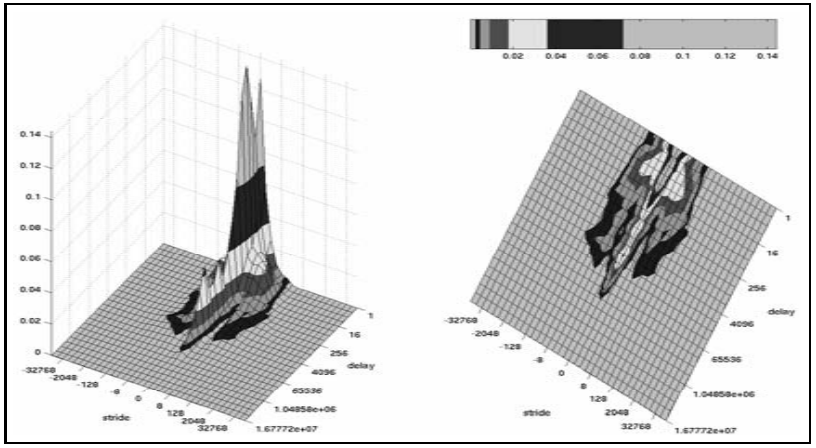
Instruction trace of *wupwise* under Linux.



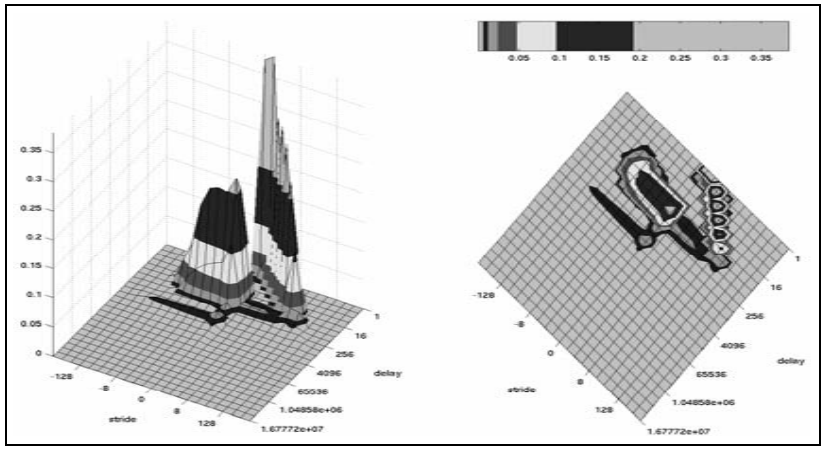
Data trace of *wupwise* under Linux.



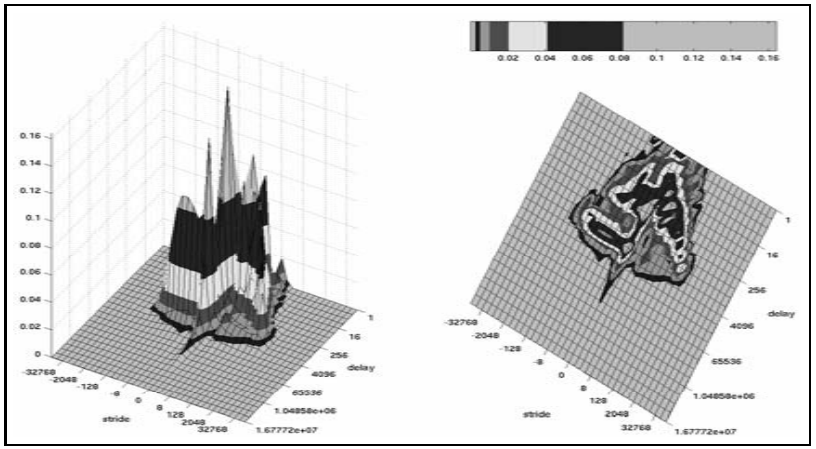
Instruction trace of *ammp* under Windows NT.



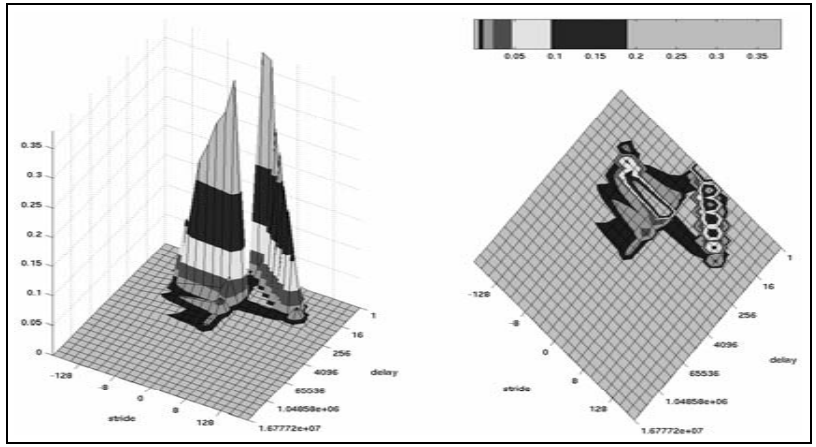
Data trace of *ammp* under Windows NT.



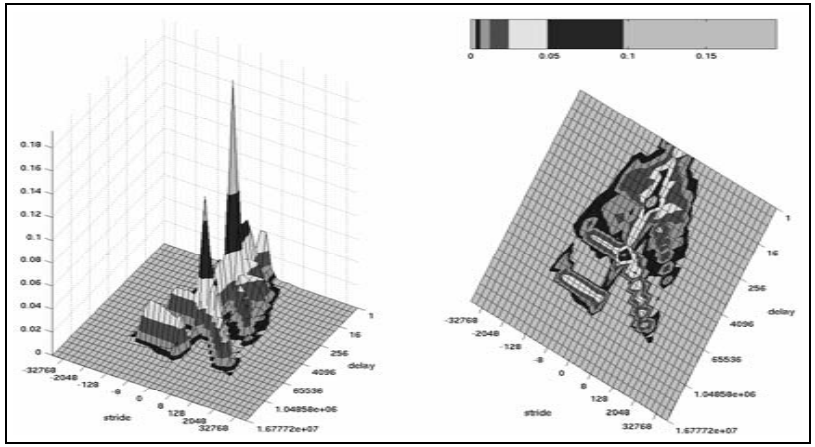
Instruction trace of *applu* under Windows NT.



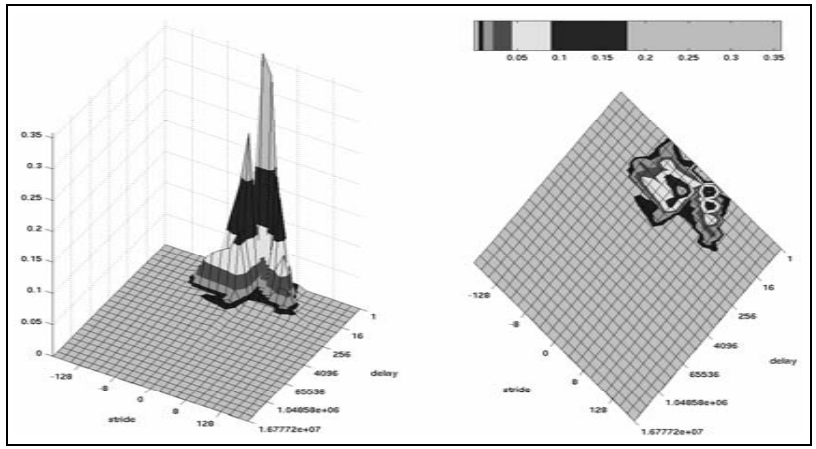
Data trace of *applu* under Windows NT.



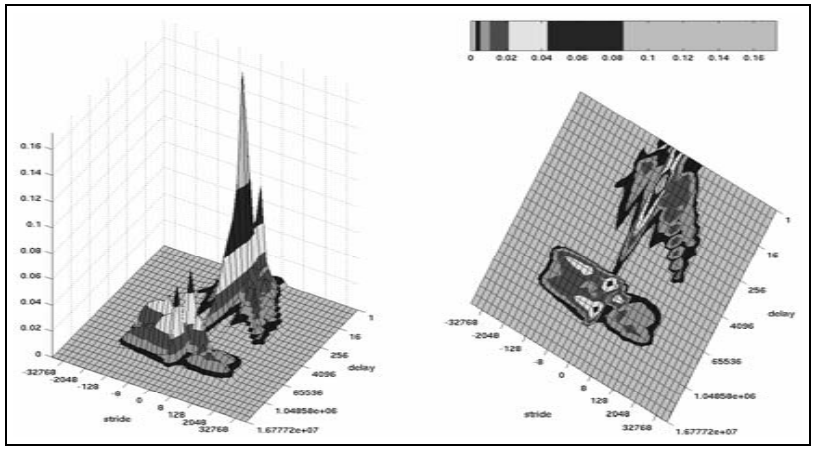
Instruction trace of *apsi* under Windows NT.



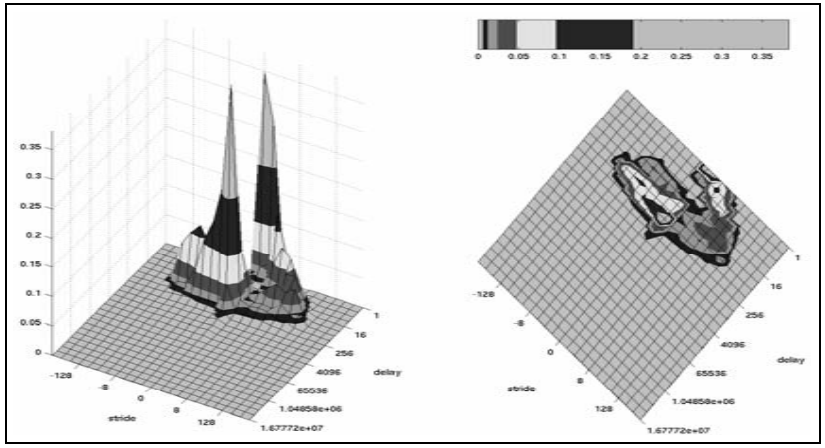
Data trace of *apsi* under Windows NT.



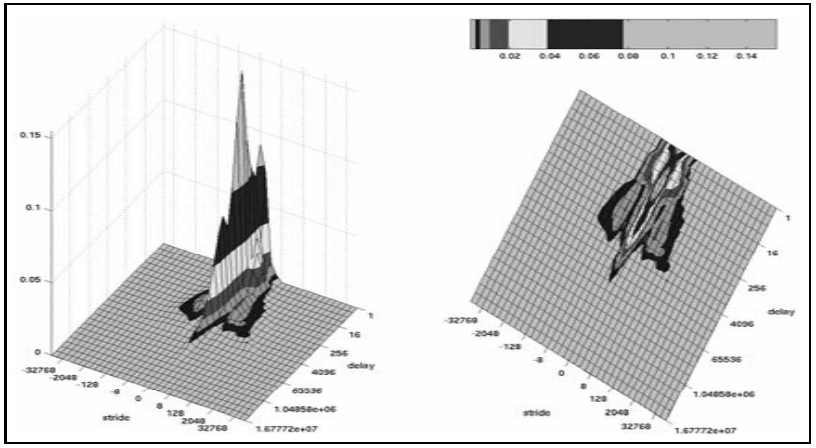
Instruction trace of *art* under Windows NT.



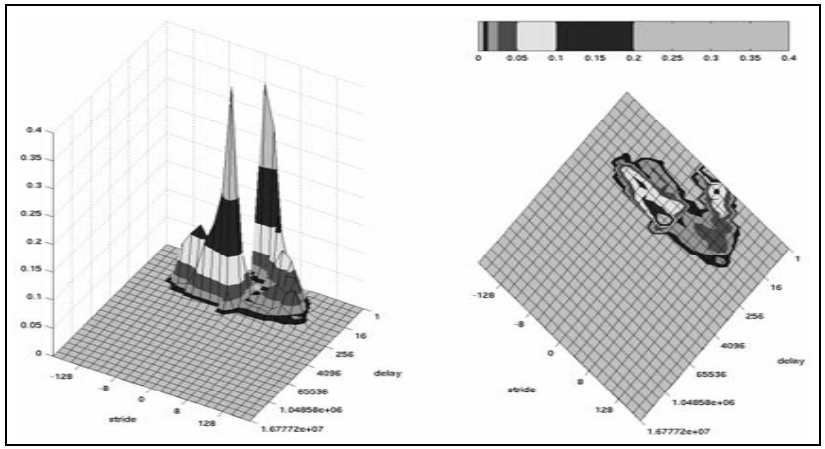
Data trace of *art* under Windows NT.



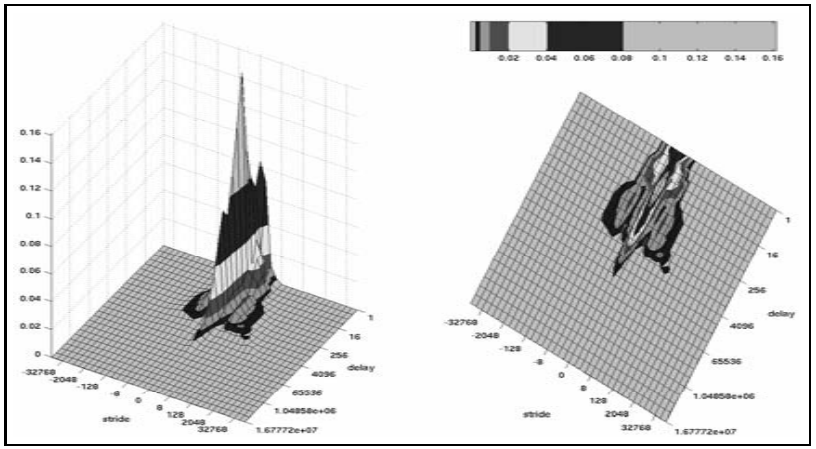
Instruction trace of *bzip2.g7* under Windows NT.



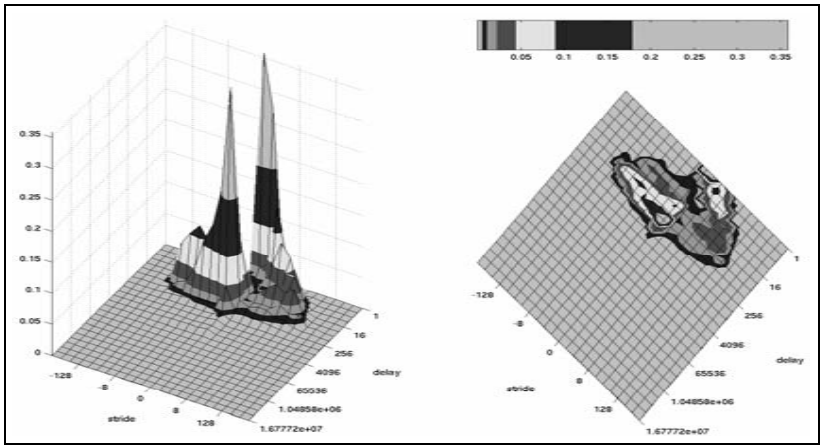
Data trace of *bzip2.g7* under Windows NT.



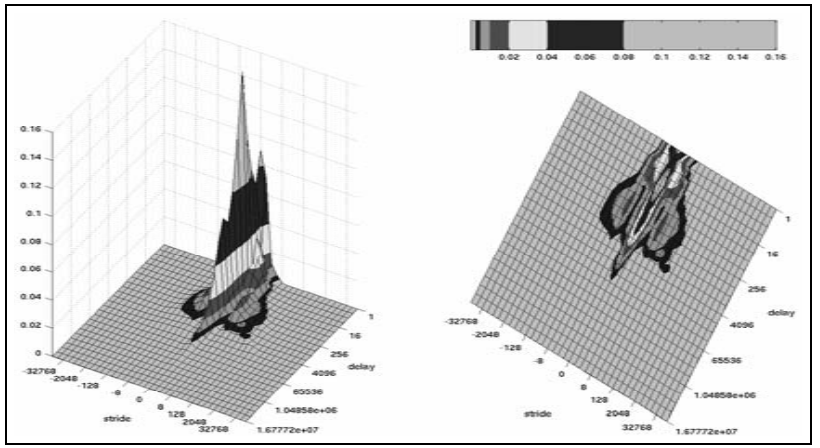
Instruction trace of *bzip2.g9* under Windows NT.



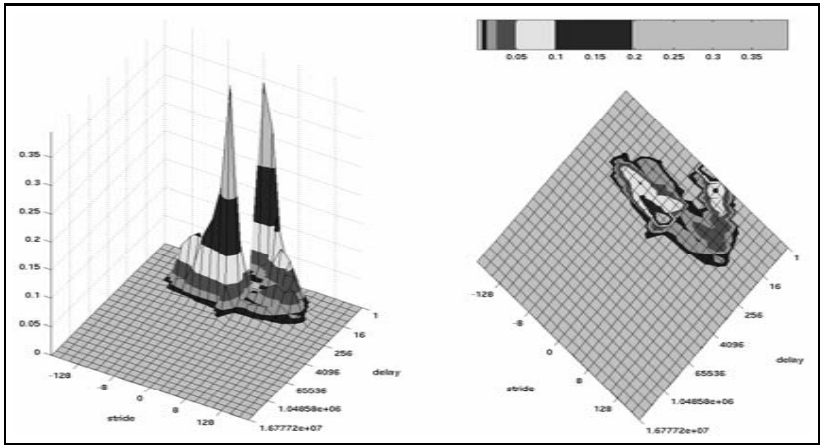
Data trace of *bzip2.g9* under Windows NT.



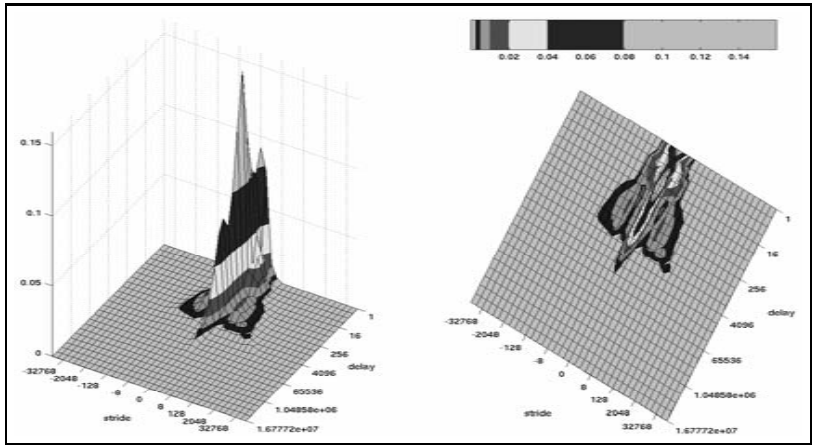
Instruction trace of *bzip2.p7* under Windows NT.



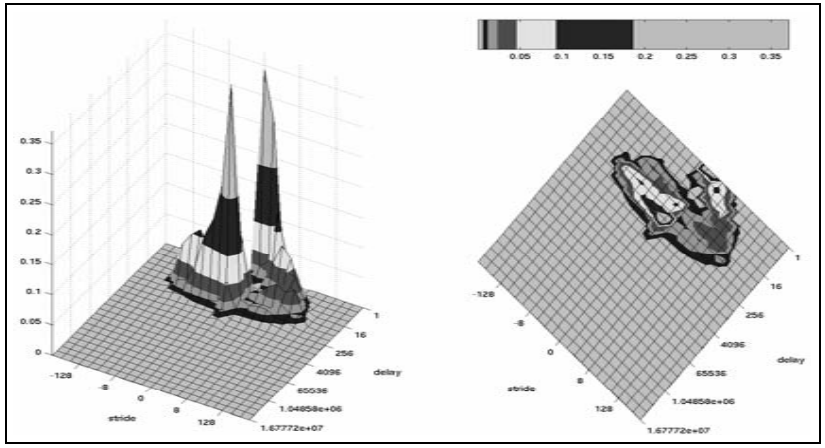
Data trace of *bzip2.p7* under Windows NT.



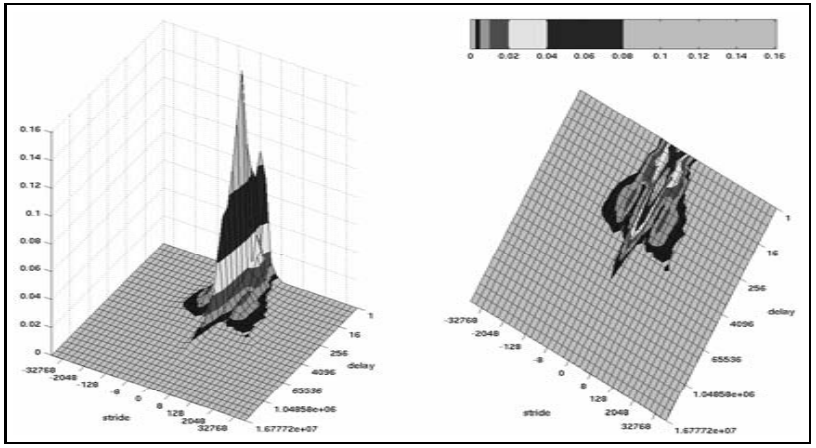
Instruction trace of *bzip2.p9* under Windows NT.



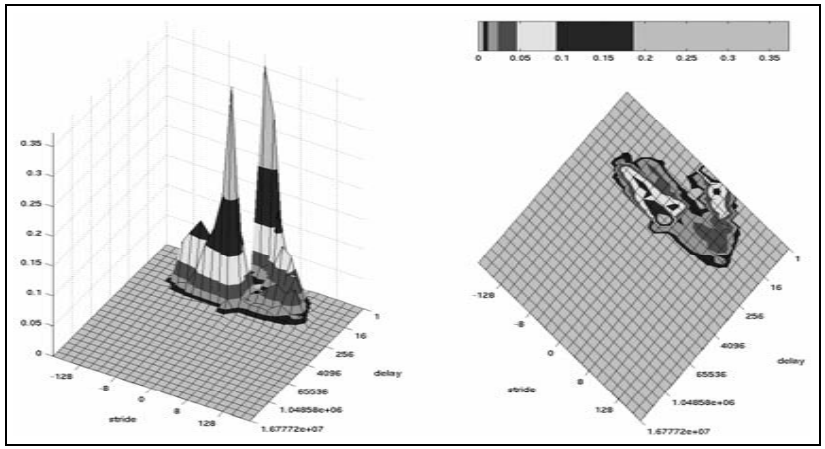
Data trace of *bzip2.p9* under Windows NT.



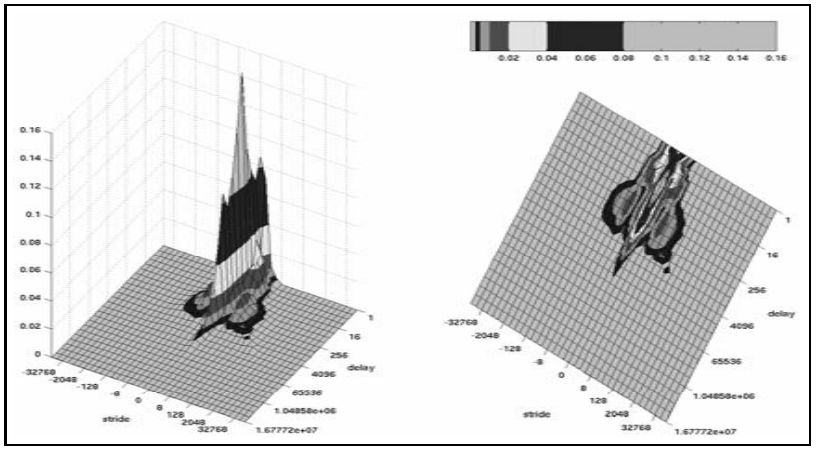
Instruction trace of *bzip2.s7* under Windows NT.



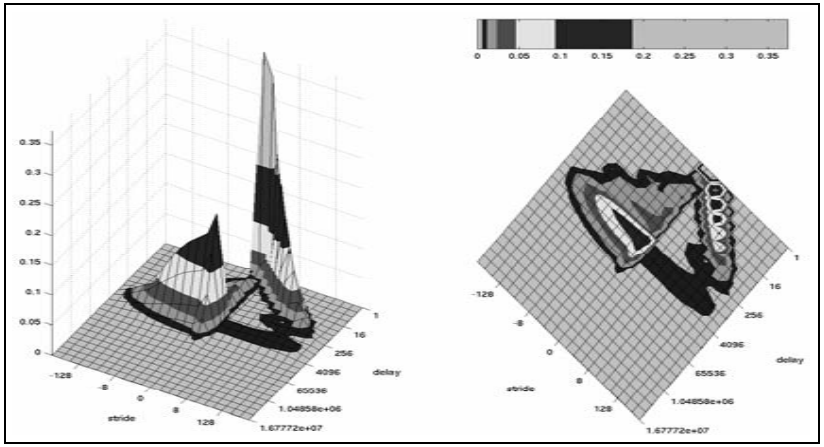
Data trace of *bzip2.s7* under Windows NT.



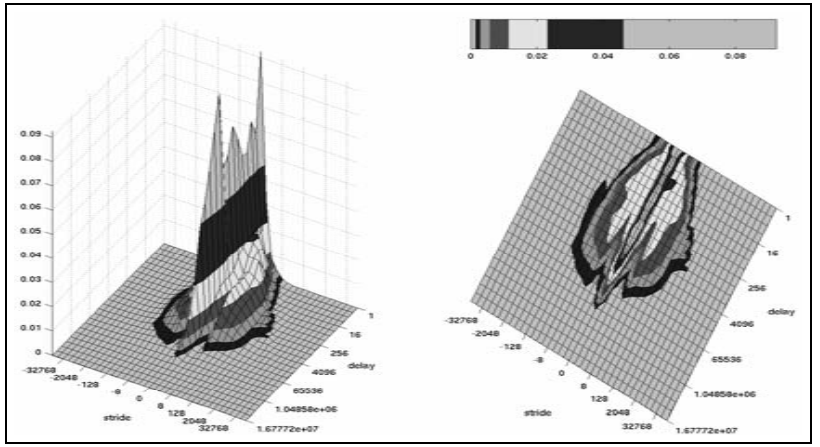
Instruction trace of *bzip2.s9* under Windows NT.



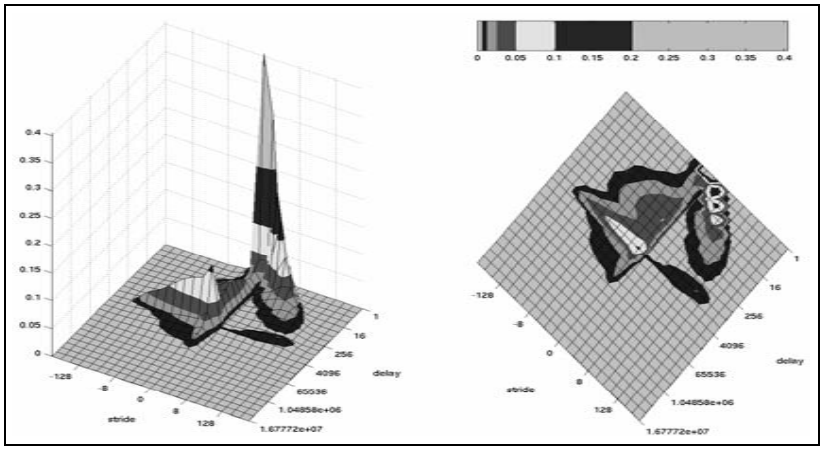
Data trace of *bzip2.s9* under Windows NT.



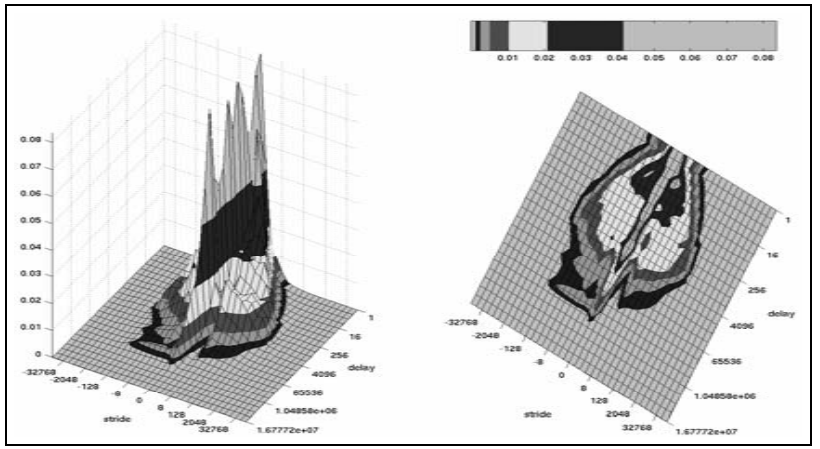
Instruction trace of *crafty* under Windows NT.



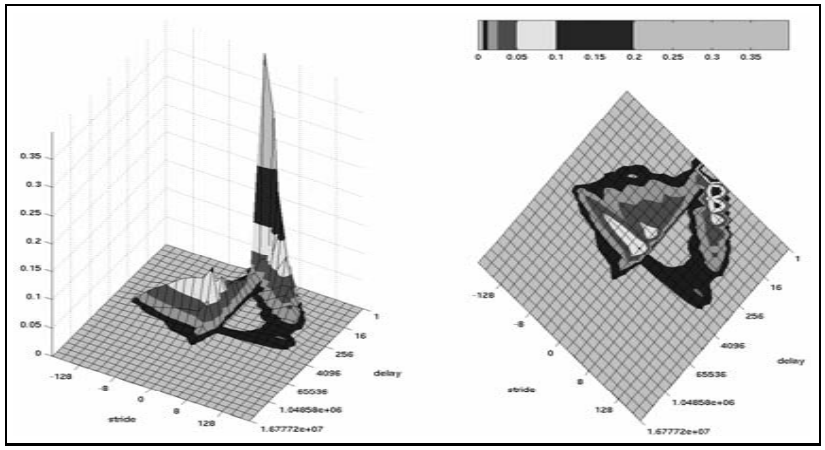
Data trace of *crafty* under Windows NT.



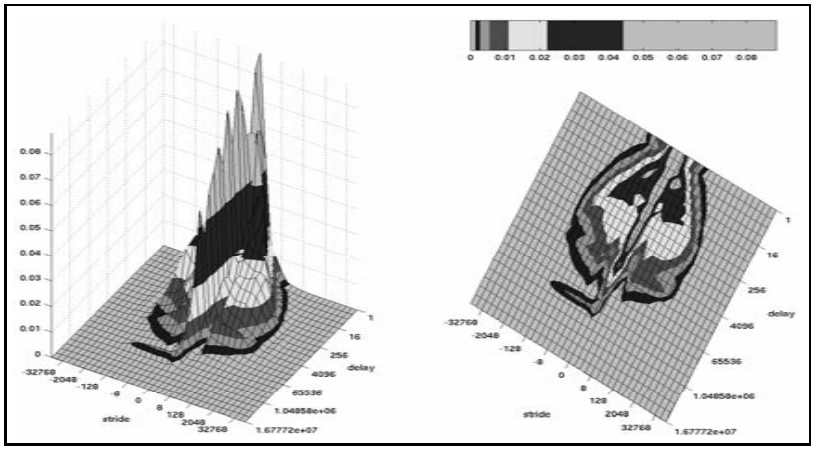
Instruction trace of *eon.cook* under Windows NT.



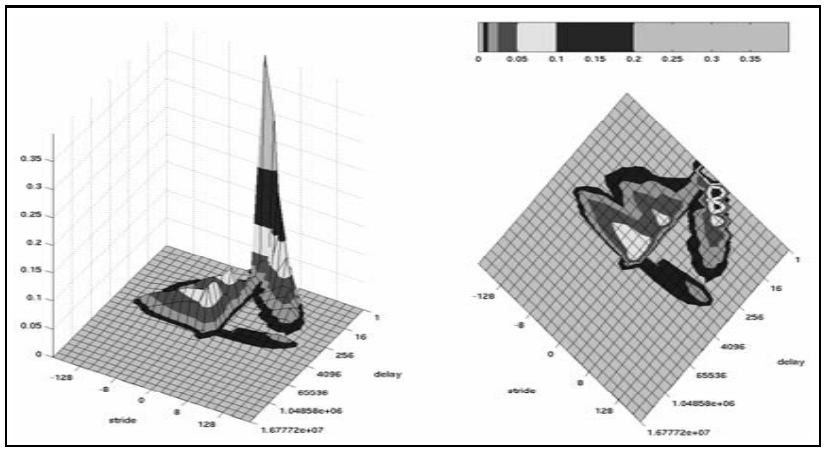
Data trace of *eon.cook* under Windows NT.



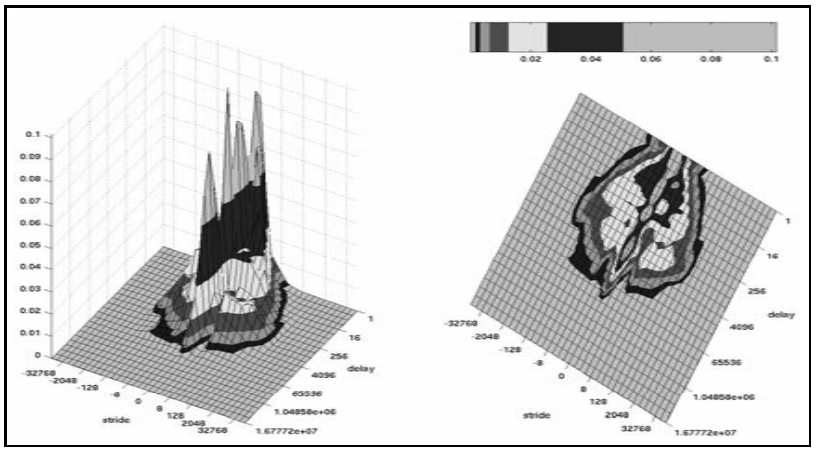
Instruction trace of *eon.kajiya* under Windows NT.



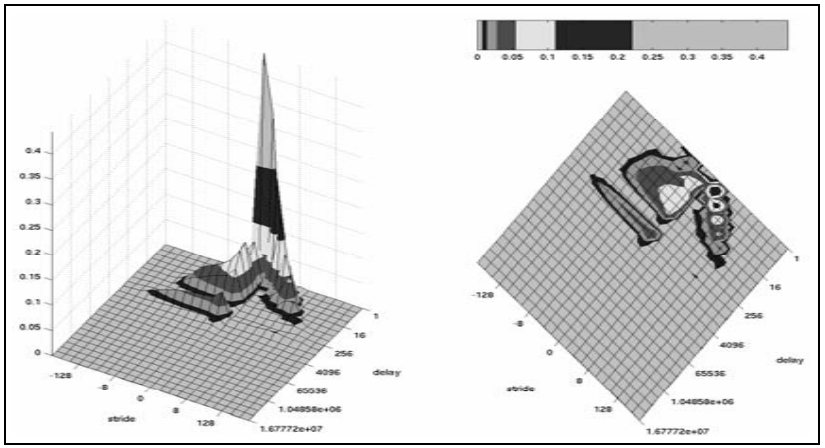
Data trace of *eon.kajiya* under Windows NT.



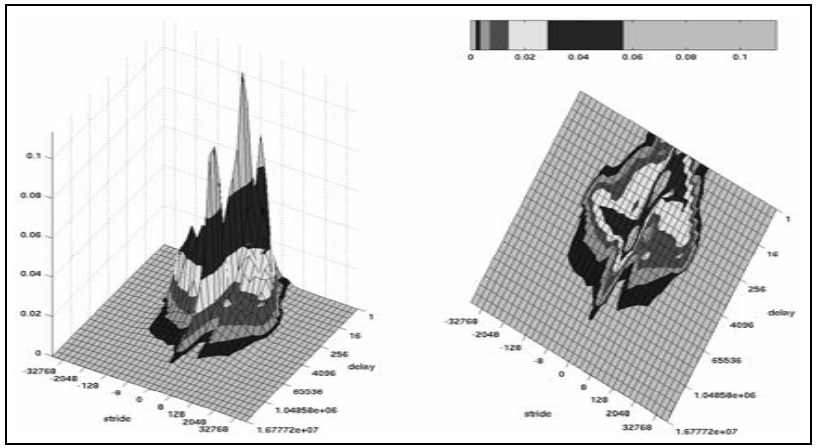
Instruction trace of *eon.rushmeier* under Windows NT.



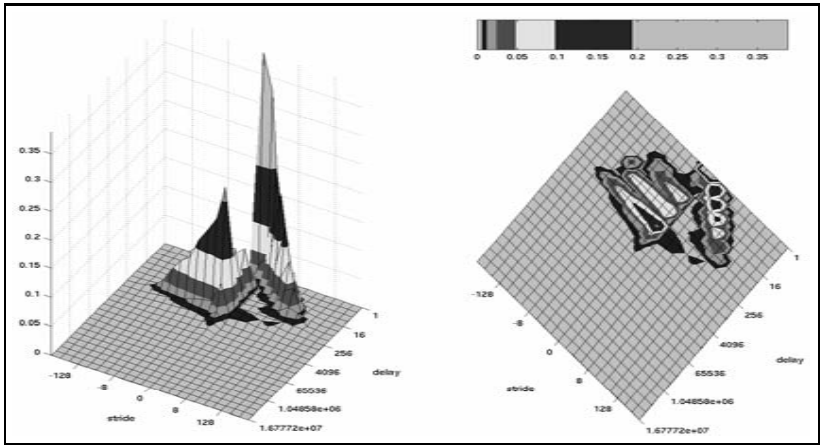
Data trace of *eon.rushmeier* under Windows NT.



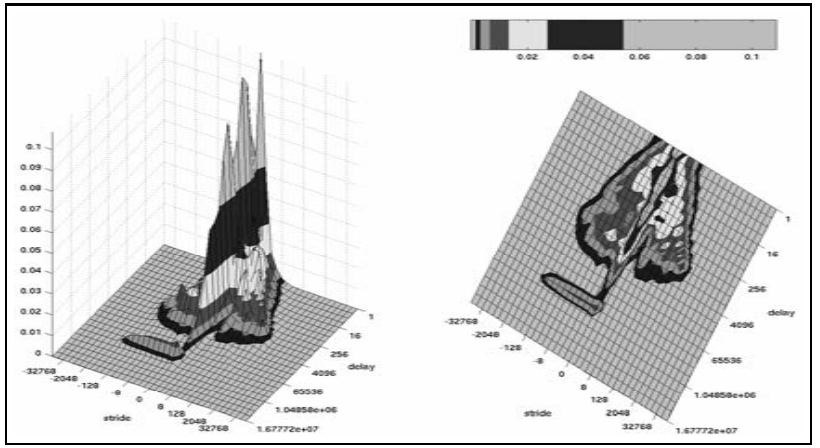
Instruction trace of *equake* under Windows NT.



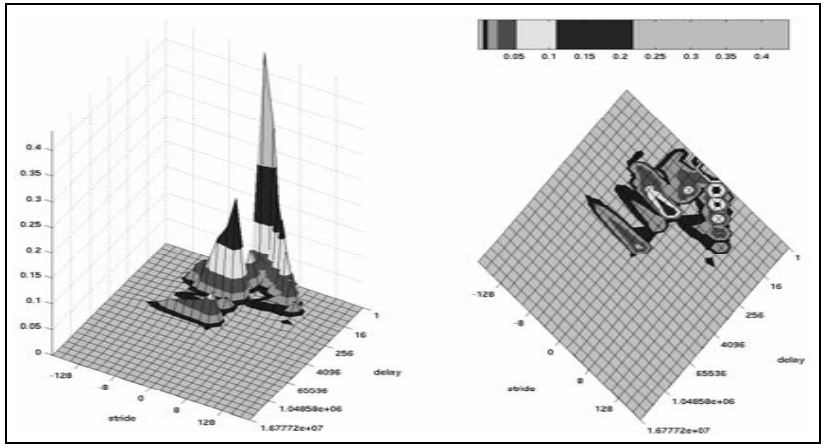
Data trace of *equake* under Windows NT.



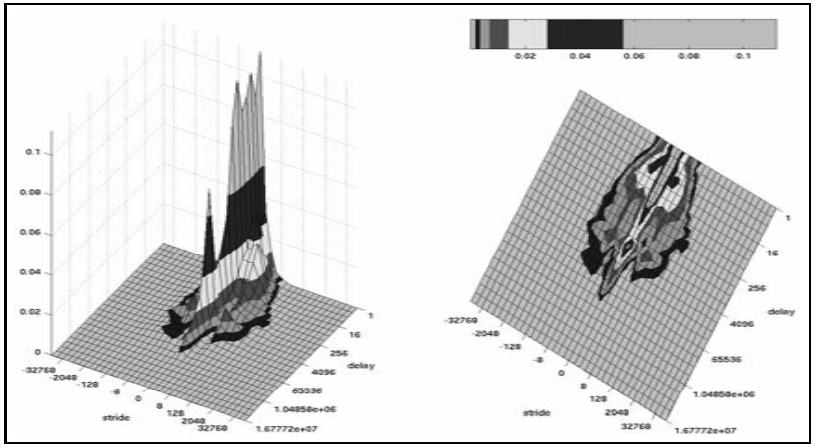
Instruction trace of *facerec* under Windows NT.



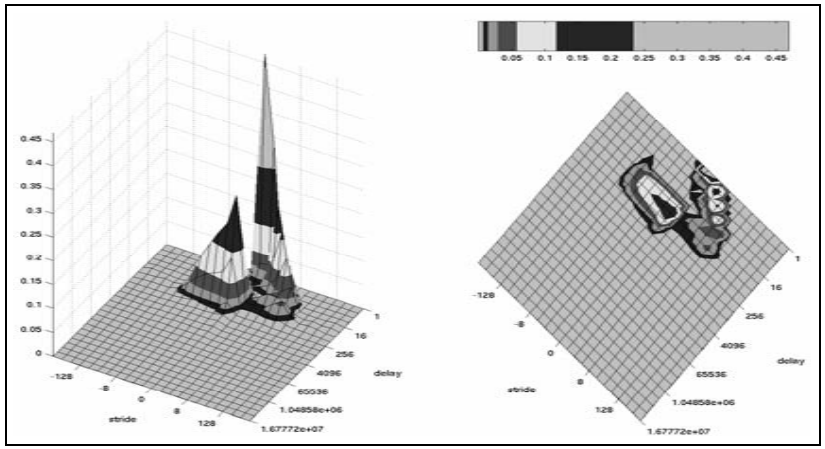
Data trace of *facerec* under Windows NT.



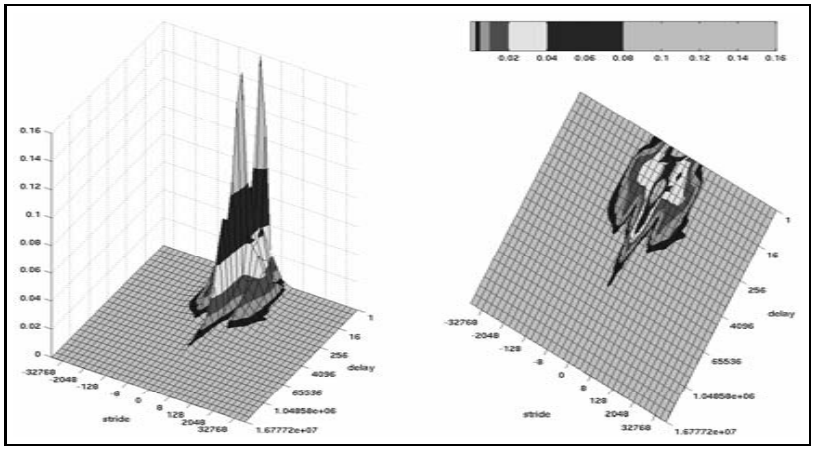
Instruction trace of *fma3d* under Windows NT.



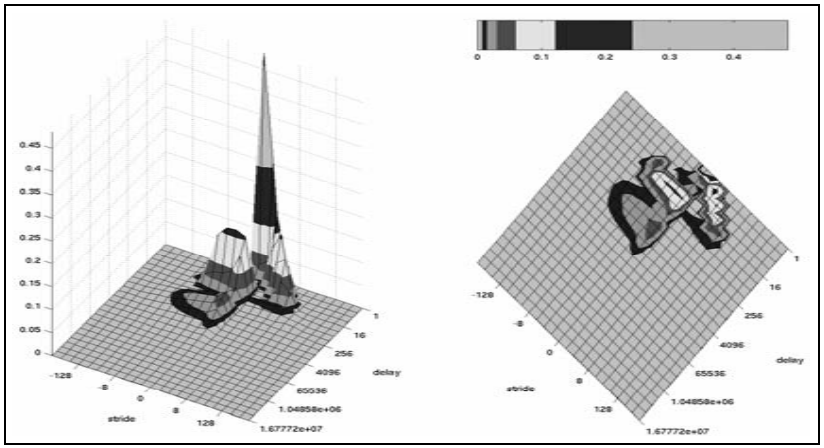
Data trace of *fma3d* under Windows NT.



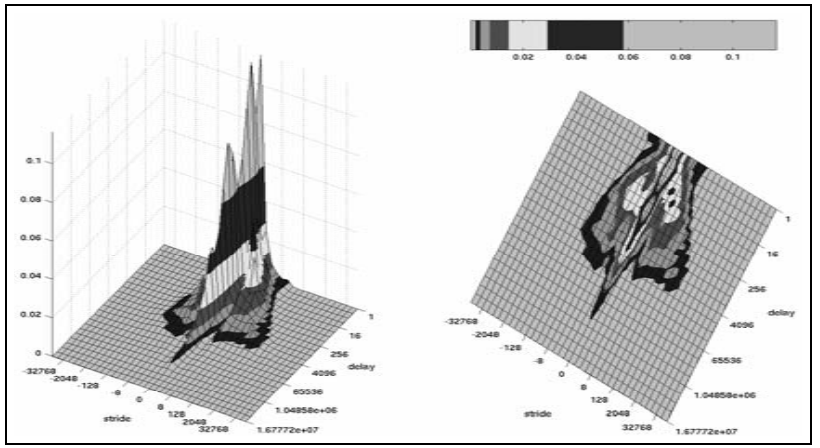
Instruction trace of *galgel* under Windows NT.



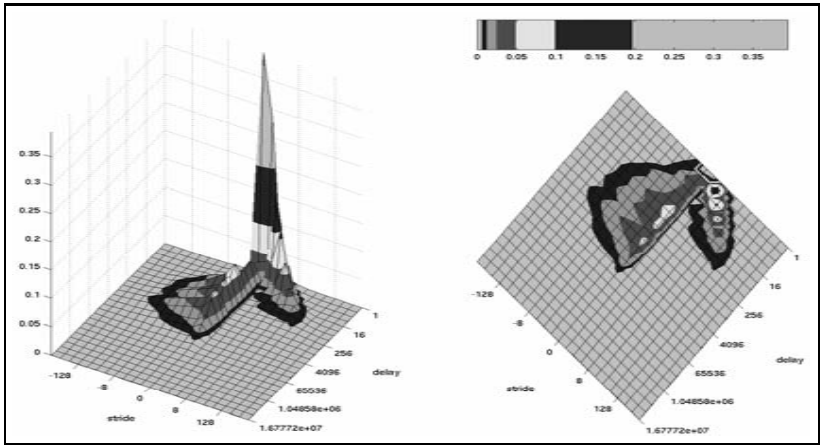
Data trace of *galgel* under Windows NT.



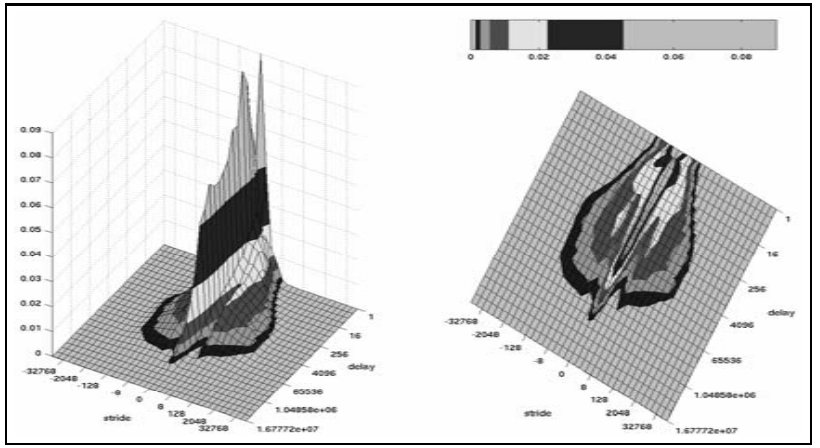
Instruction trace of *gap* under Windows NT.



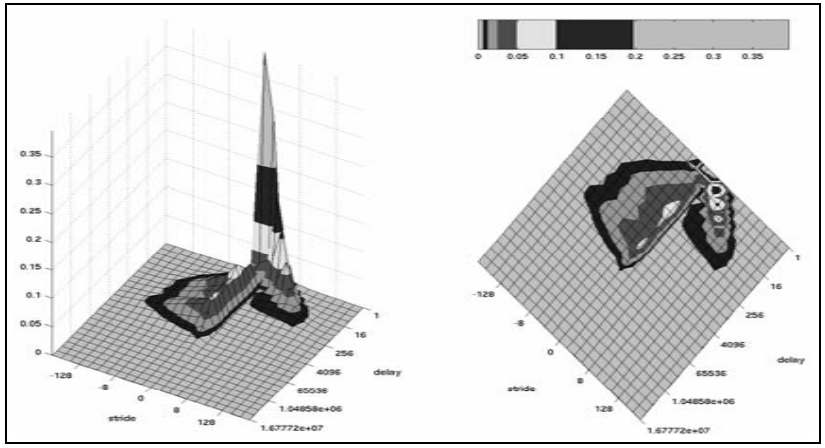
Data trace of *gap* under Windows NT.



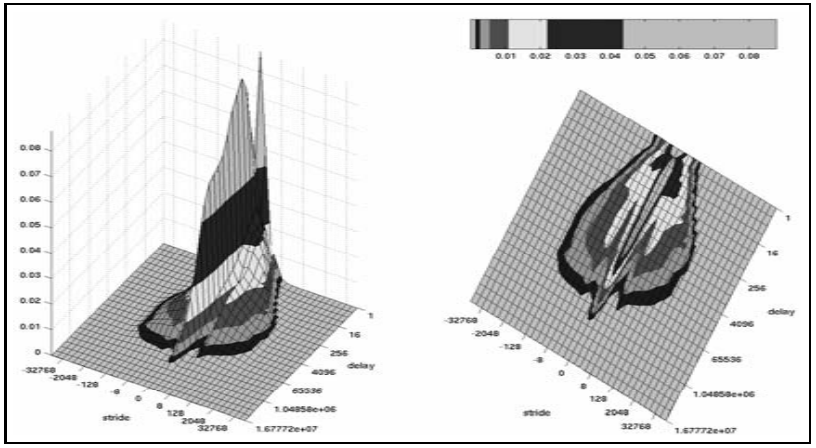
Instruction trace of *gcc.166* under Windows NT.



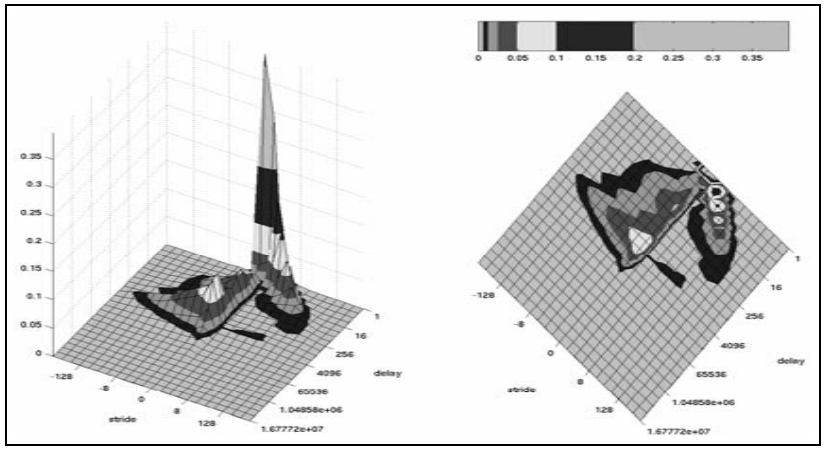
Data trace of *gcc.166* under Windows NT.



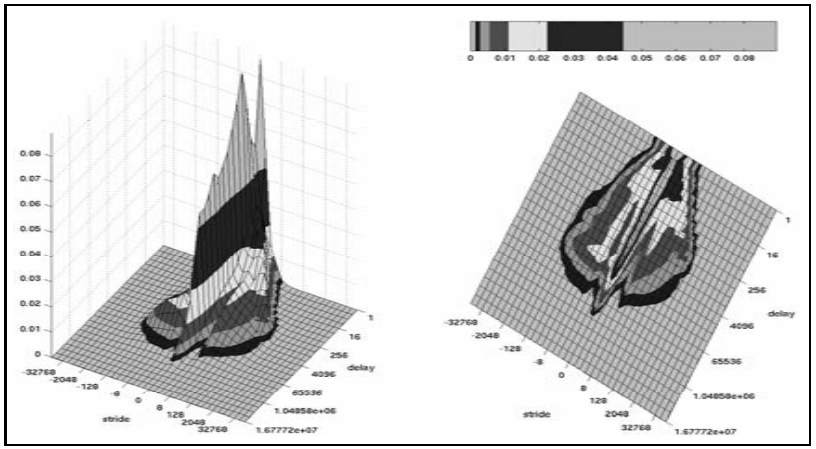
Instruction trace of *gcc.200* under Windows NT.



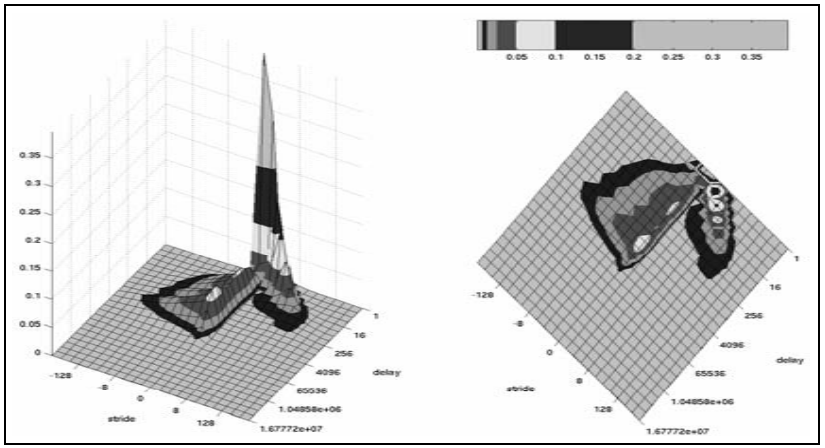
Data trace of *gcc.200* under Windows NT.



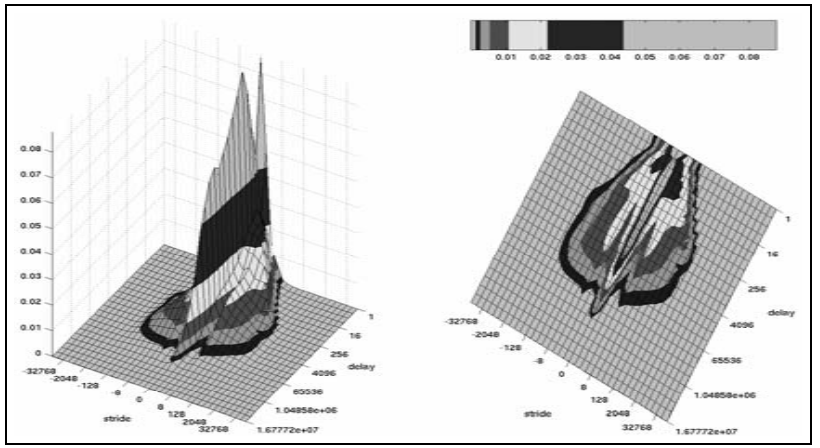
Instruction trace of *gcc.expr* under Windows NT.



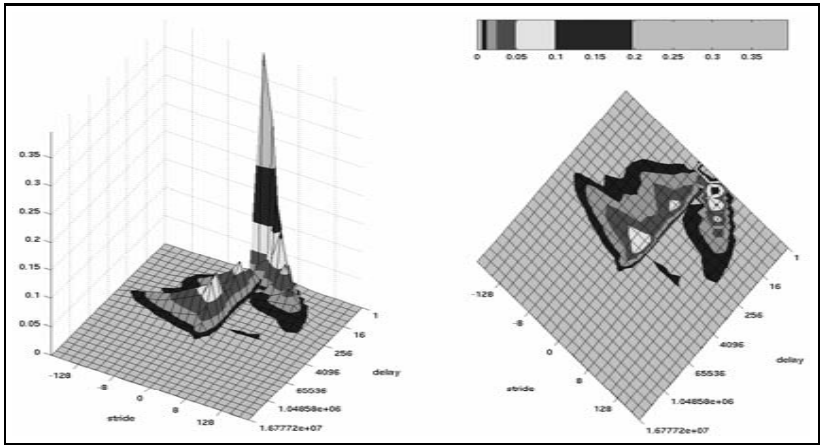
Data trace of *gcc.expr* under Windows NT.



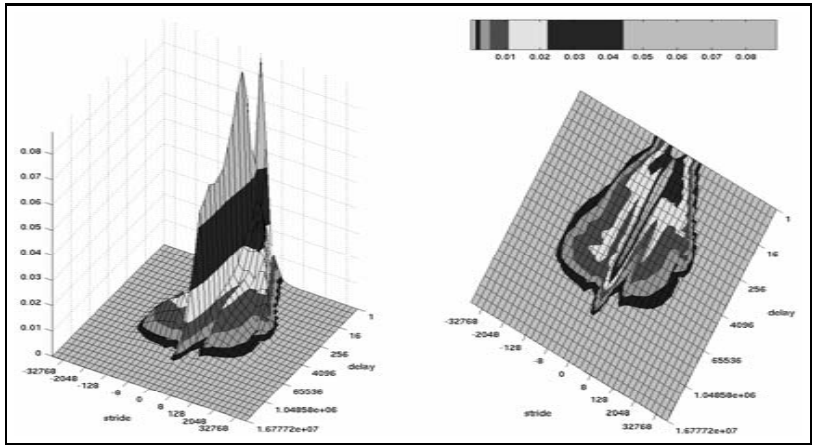
Instruction trace of *gcc.integ* under Windows NT.



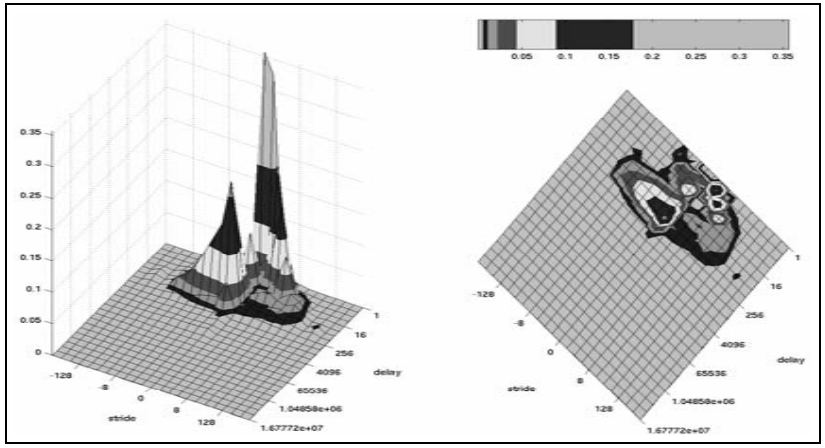
Data trace of *gcc.integ* under Windows NT.



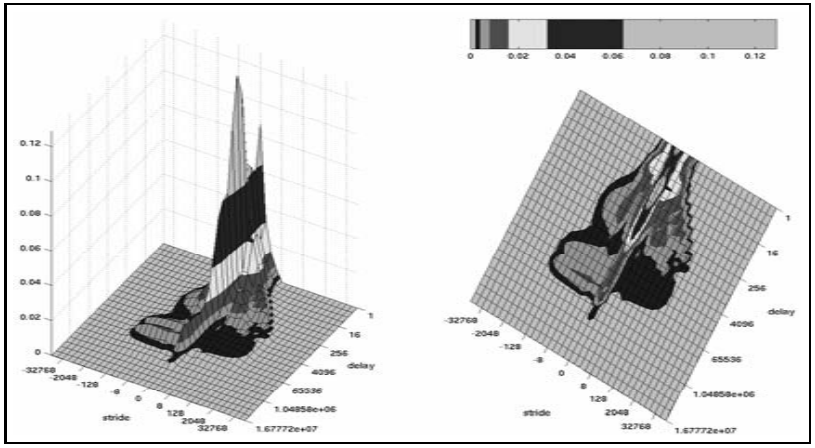
Instruction trace of *gcc.scilab* under Windows NT.



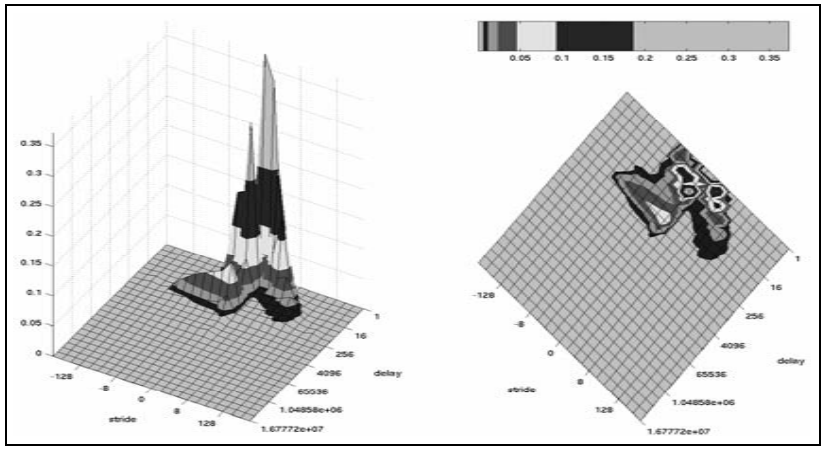
Data trace of *gcc.scilab* under Windows NT.



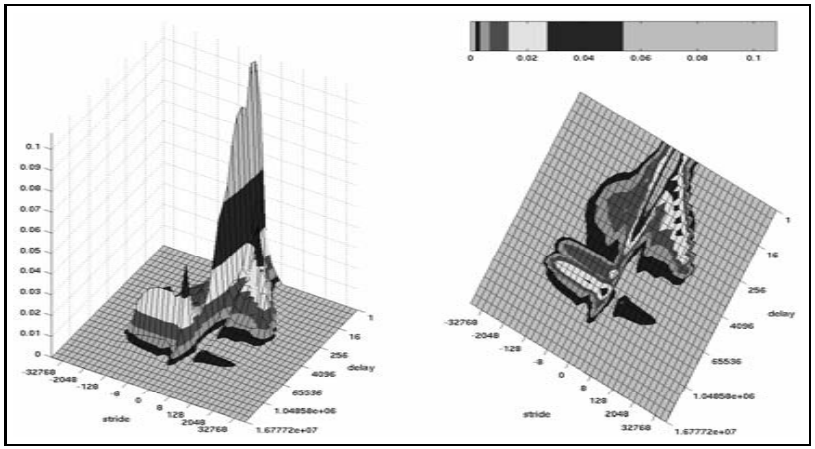
Instruction trace of *gzip.graphic* under Windows NT.



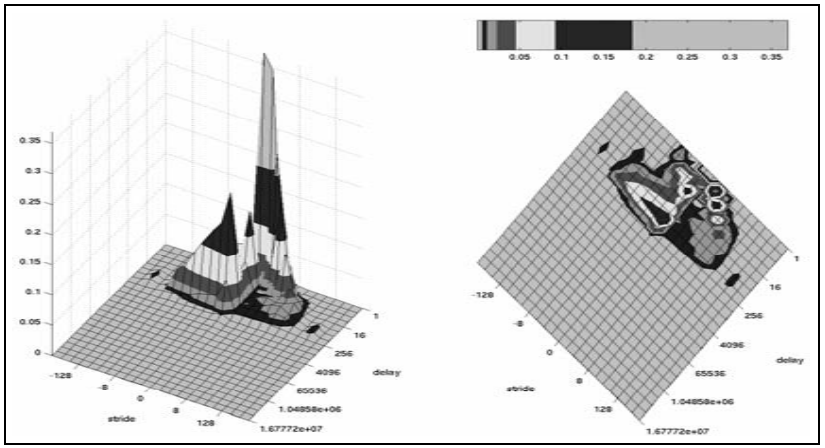
Data trace of *gzip.graphic* under Windows NT.



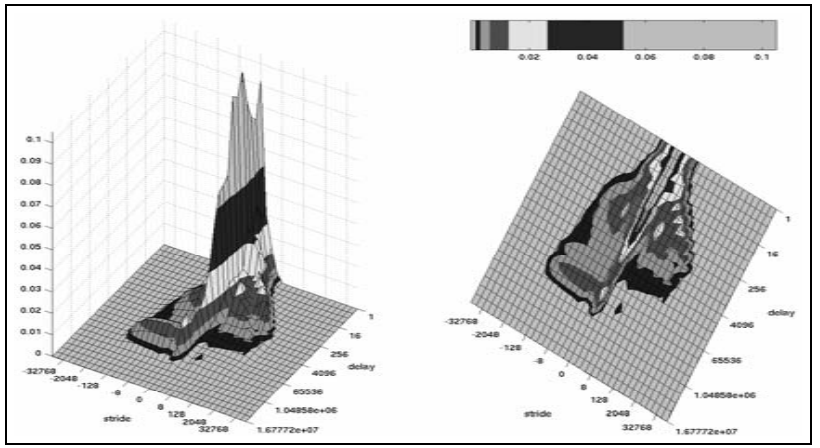
Instruction trace of *gzip.log* under Windows NT.



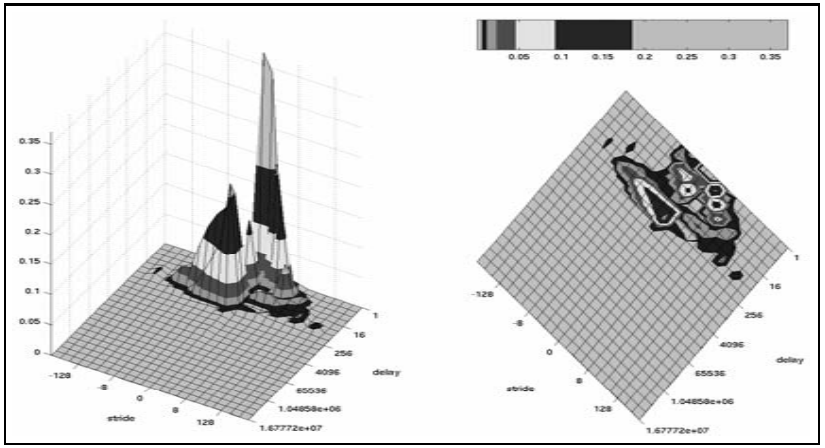
Data trace of *gzip.log* under Windows NT.



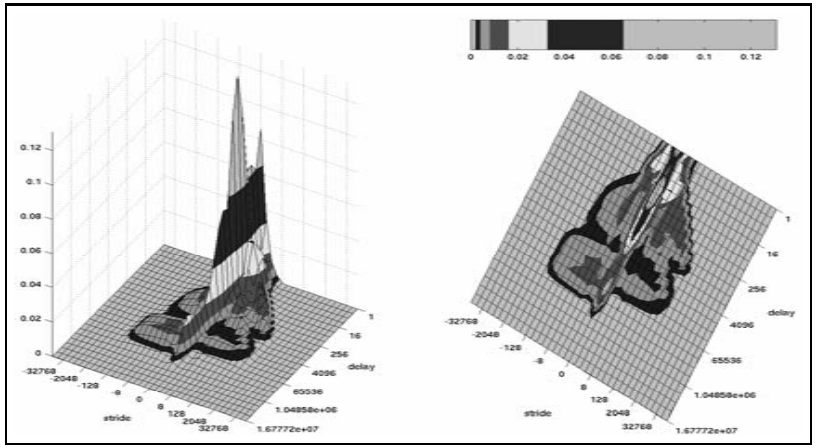
Instruction trace of *gzip.program* under Windows NT.



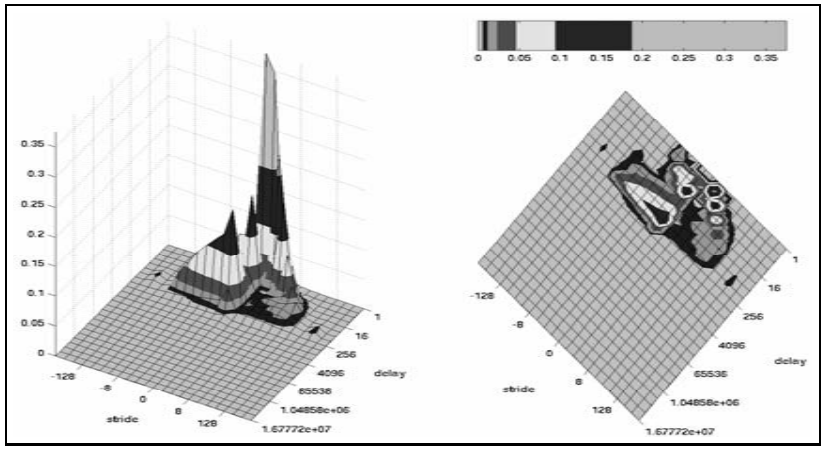
Data trace of *gzip.program* under Windows NT.



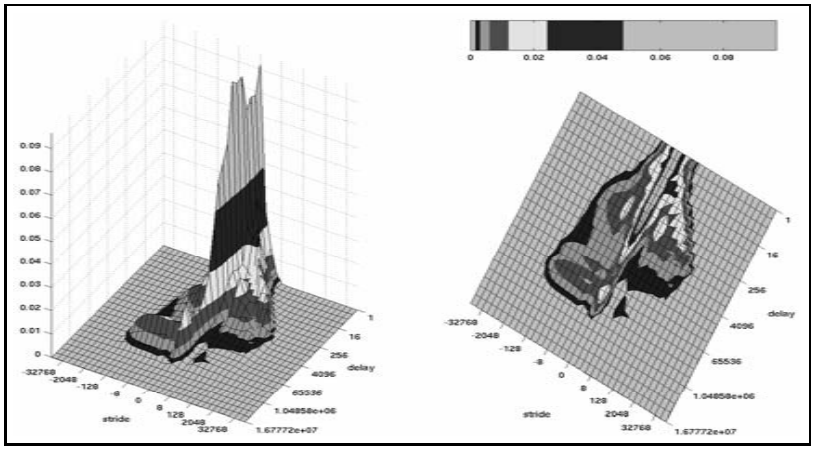
Instruction trace of *gzip.random* under Windows NT.



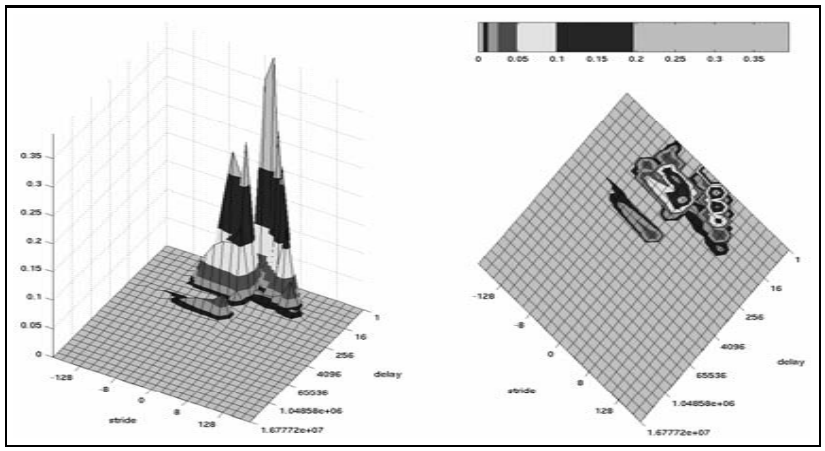
Data trace of *gzip.random* under Windows NT.



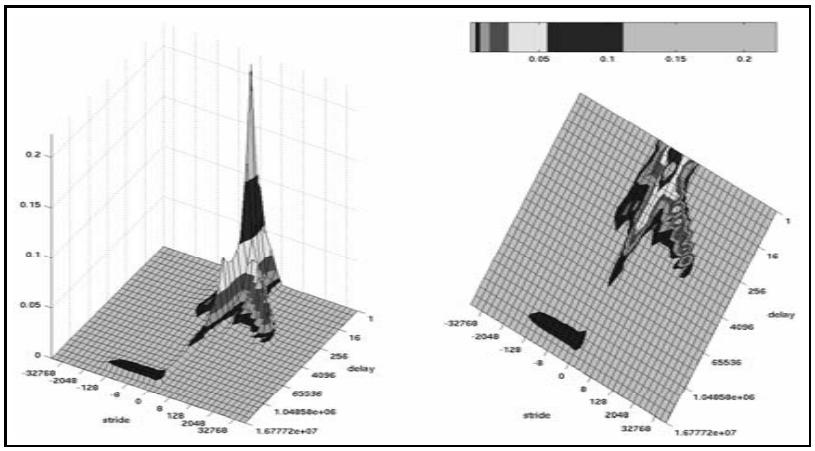
Instruction trace of *gzip.source* under Windows NT.



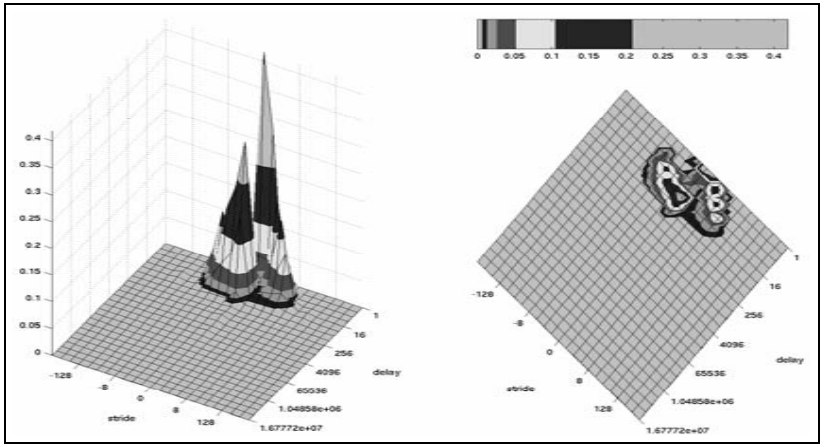
Data trace of *gzip.source* under Windows NT.



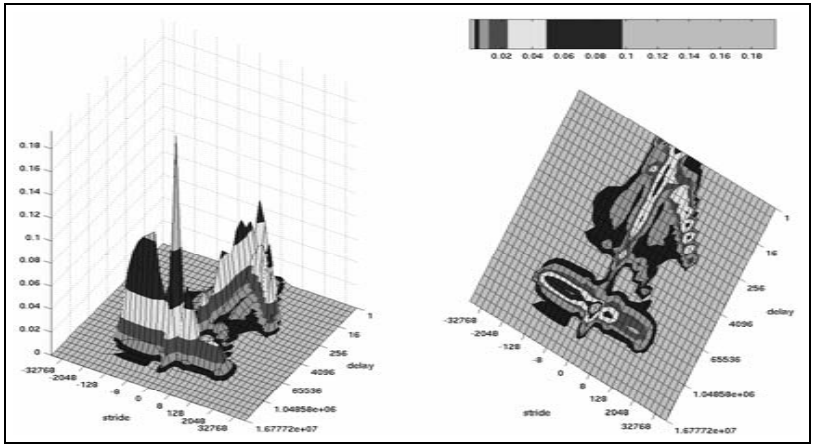
Instruction trace of *lucas* under Windows NT.



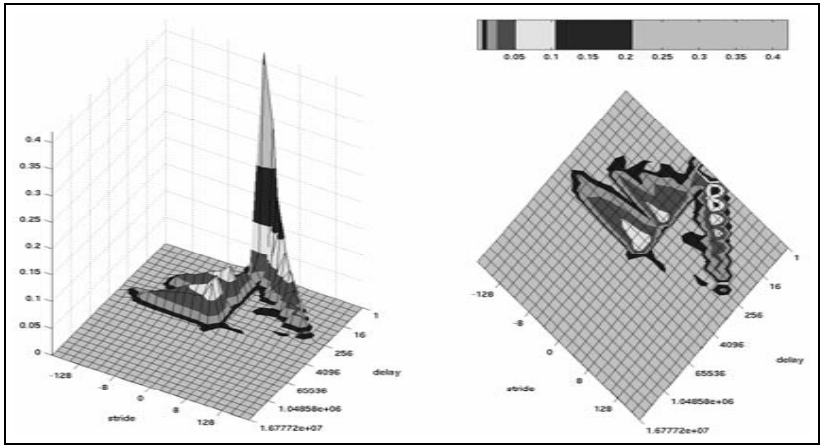
Data trace of *lucas* under Windows NT.



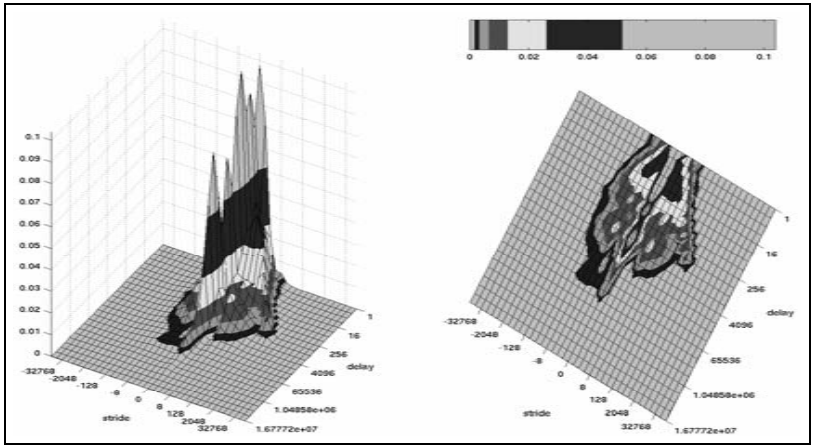
Instruction trace of *mcf* under Windows NT.



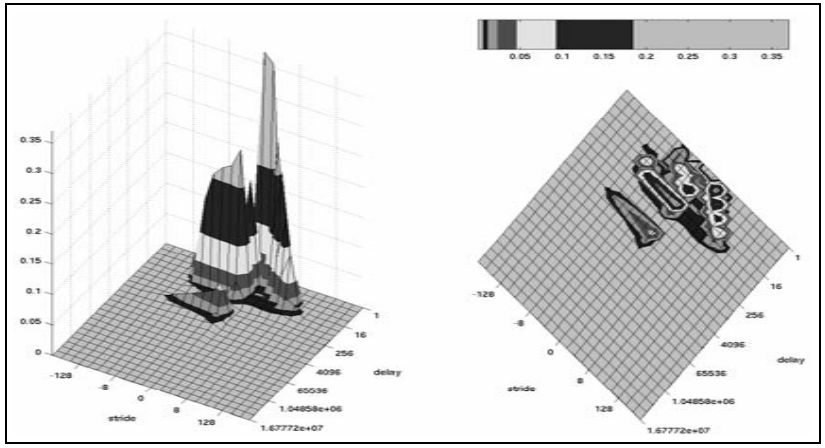
Data trace of *mcf* under Windows NT.



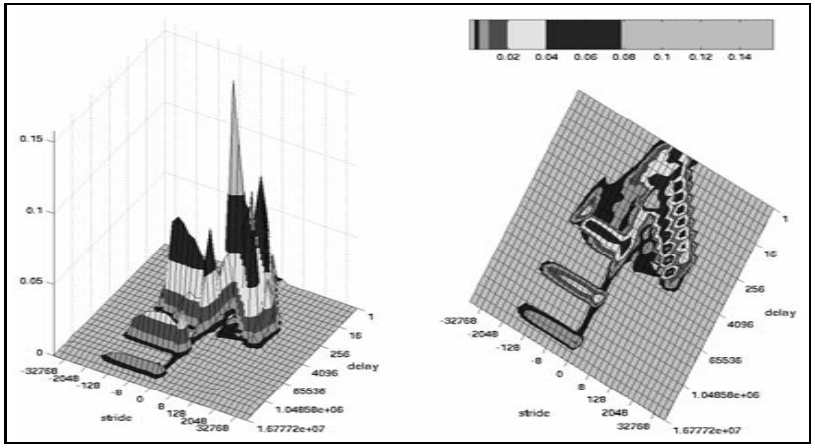
Instruction trace of *mesa* under Windows NT.



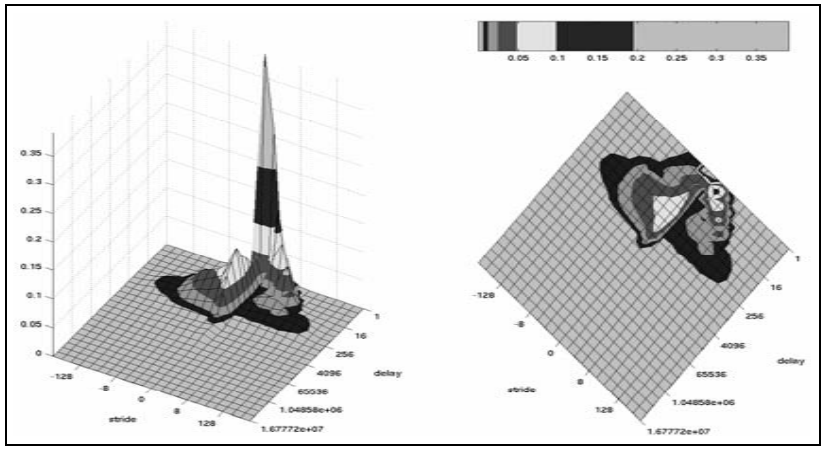
Data trace of *mesa* under Windows NT.



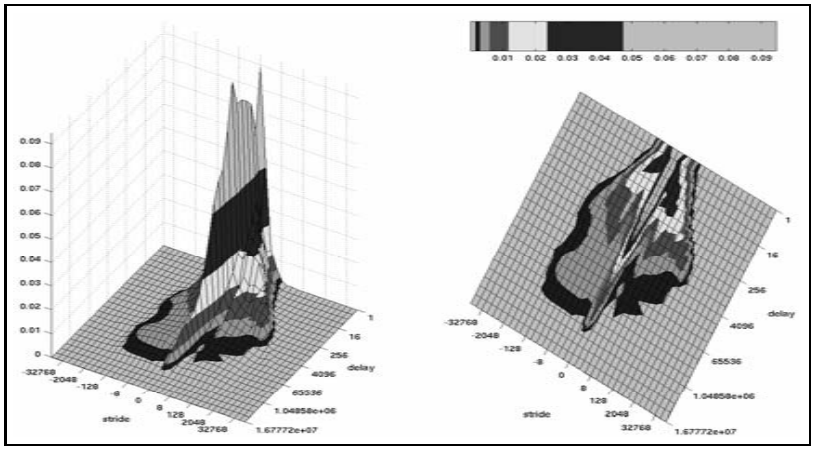
Instruction trace of *mgrid* under Windows NT.



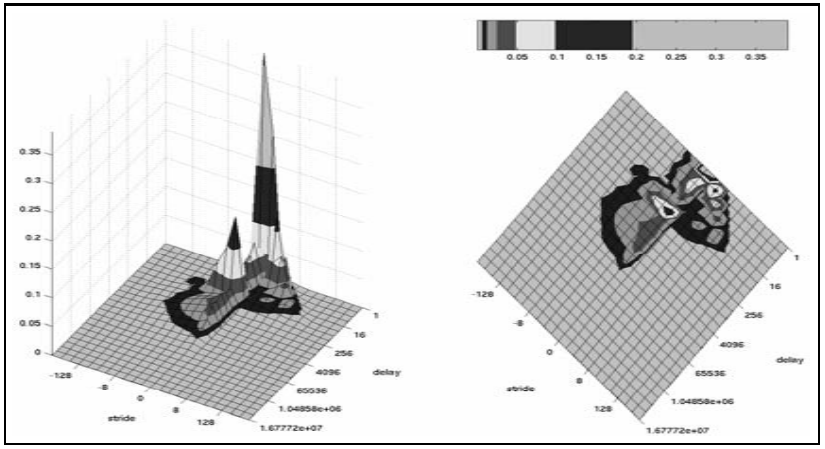
Data trace of *mgrid* under Windows NT.



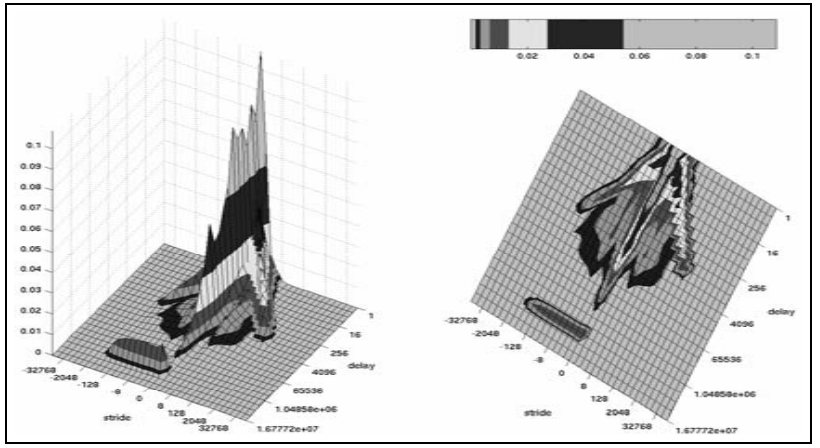
Instruction trace of *parser* under Windows NT.



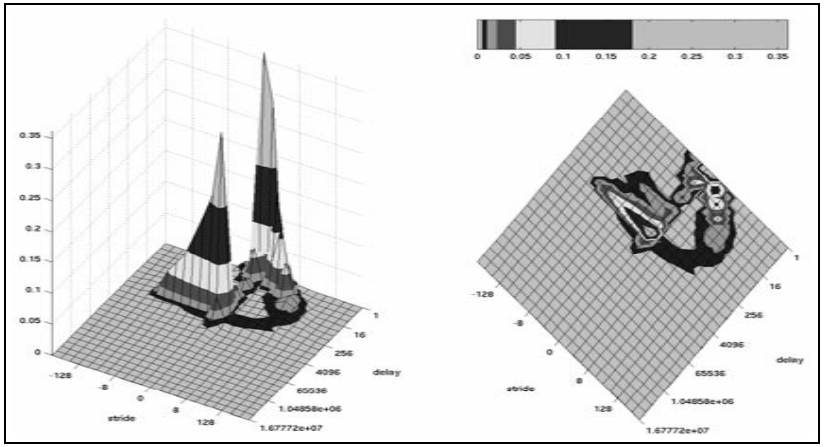
Data trace of *parser* under Windows NT.



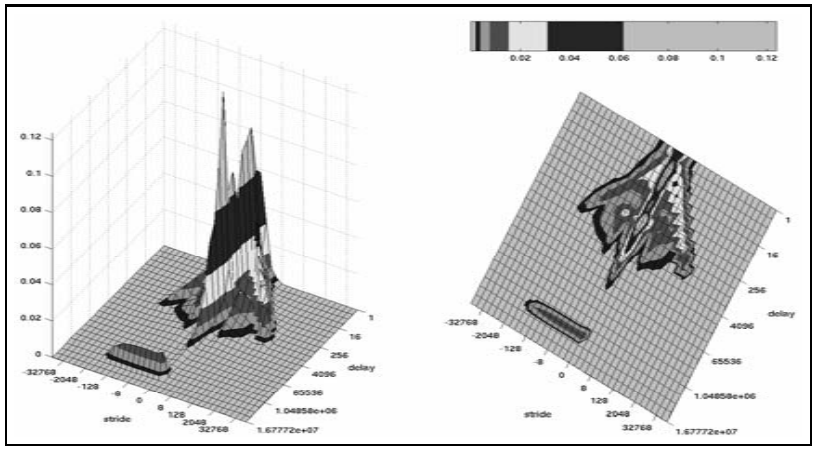
Instruction trace of *perlbmk.diffmail* under Windows NT.



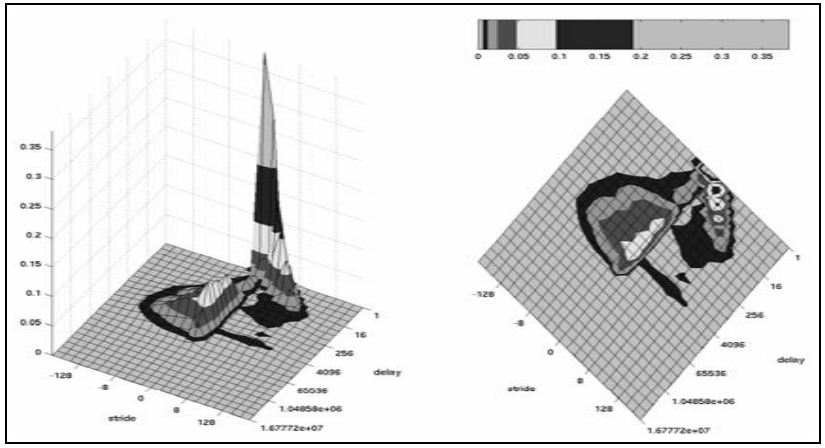
Data trace of *perlbmk.diffmail* under Windows NT.



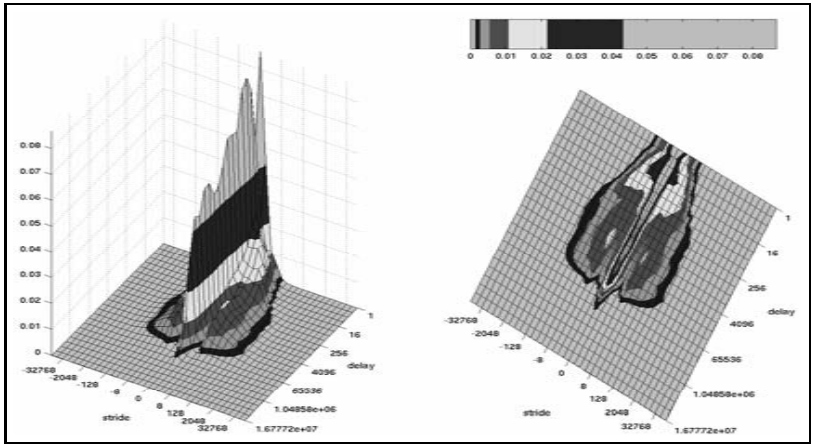
Instruction trace of *perlbmk.makerand* under Windows NT.



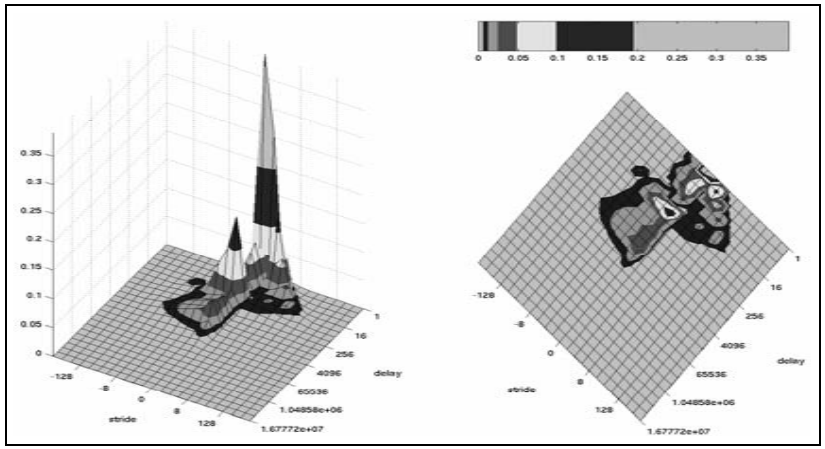
Data trace of *perlbmk.makerand* under Windows NT.



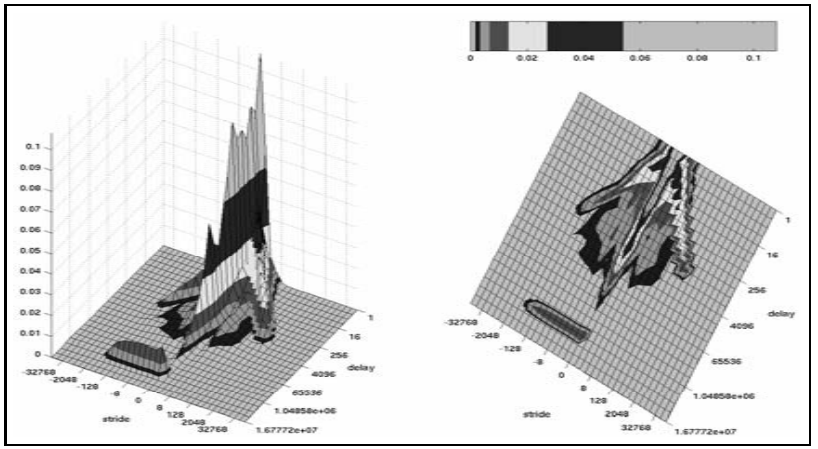
Instruction trace of *perlbmk.perfect* under Windows NT.



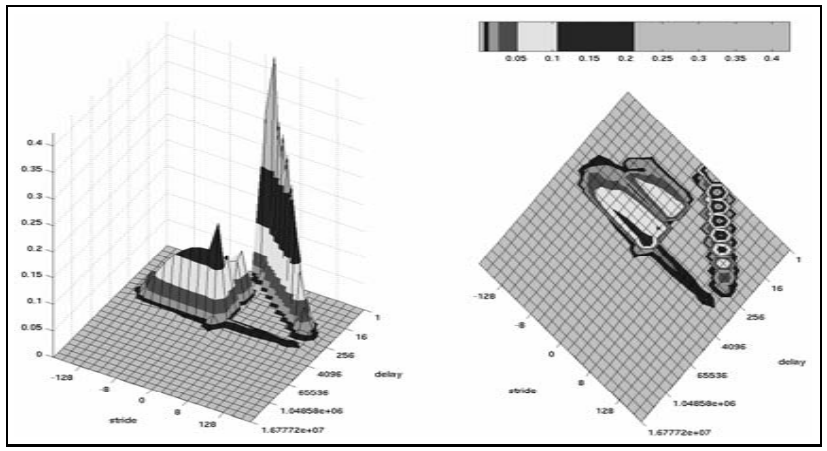
Data trace of *perlbmk.perfect* under Windows NT.



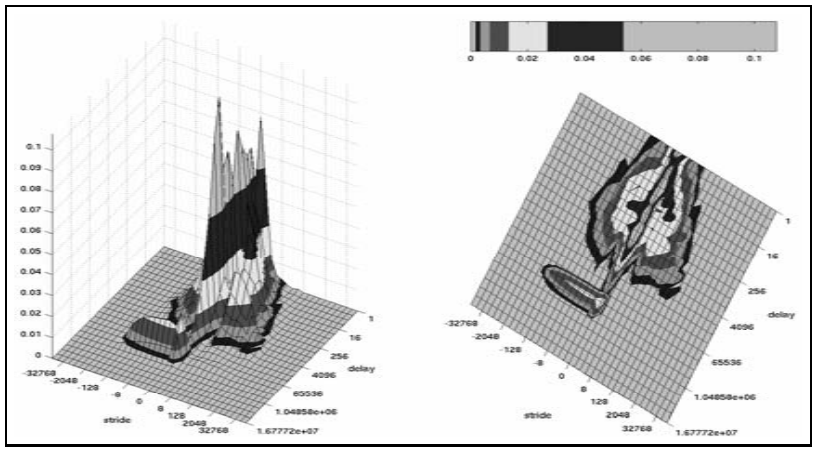
Instruction trace of *perlbmk.splitmail* under Windows NT.



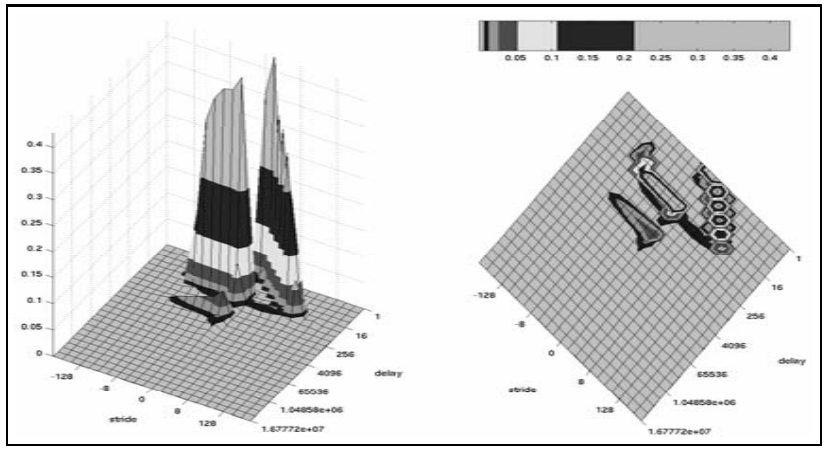
Data trace of *perlbmk.splitmail* under Windows NT.



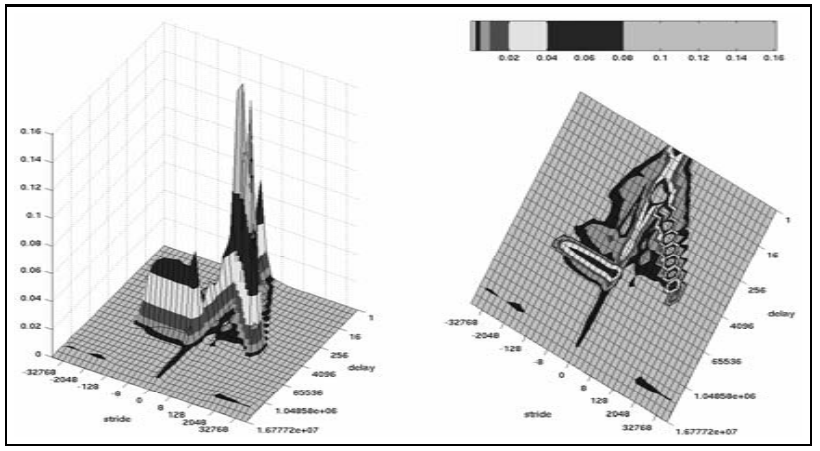
Instruction trace of *sixtrack* under Windows NT.



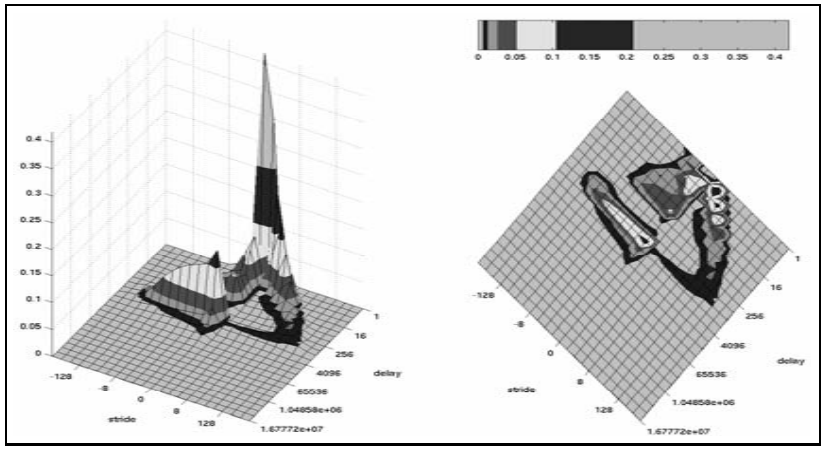
Data trace of *sixtrack* under Windows NT.



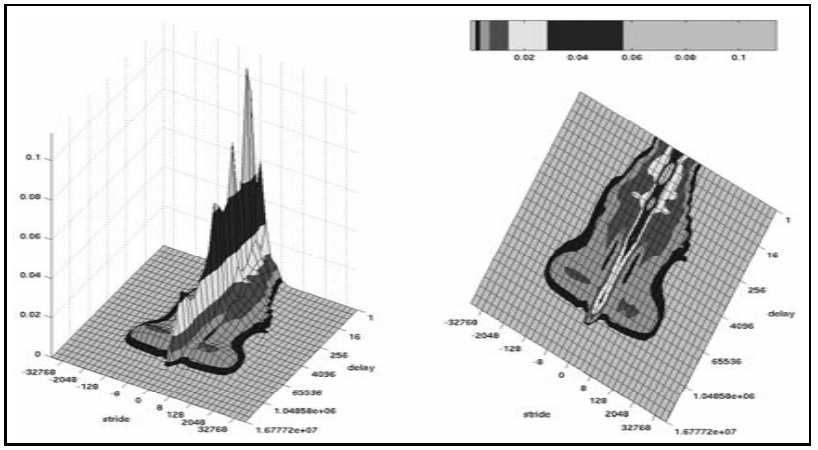
Instruction trace of *swim* under Windows NT.



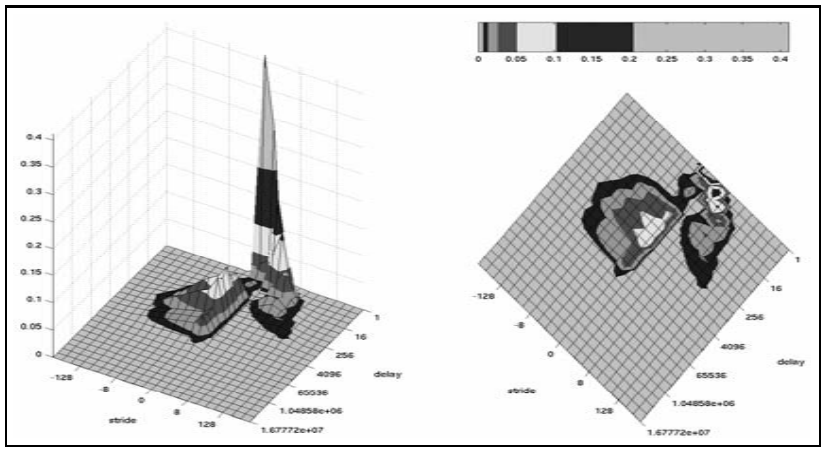
Data trace of *swim* under Windows NT.



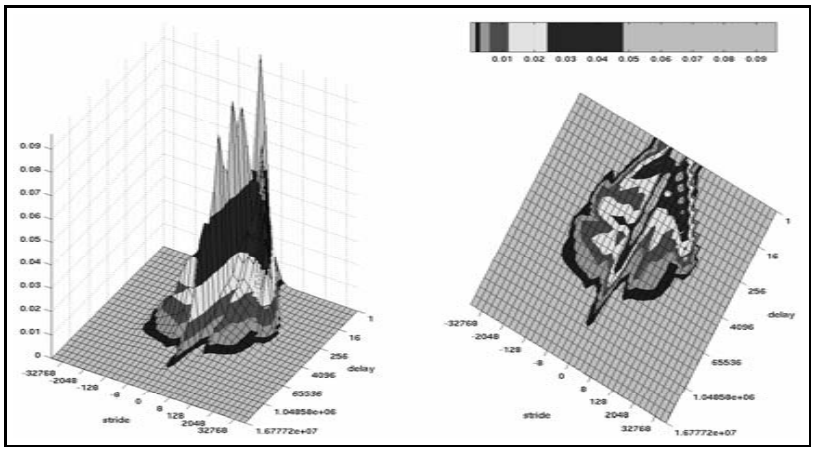
Instruction trace of *twolf* under Windows NT.



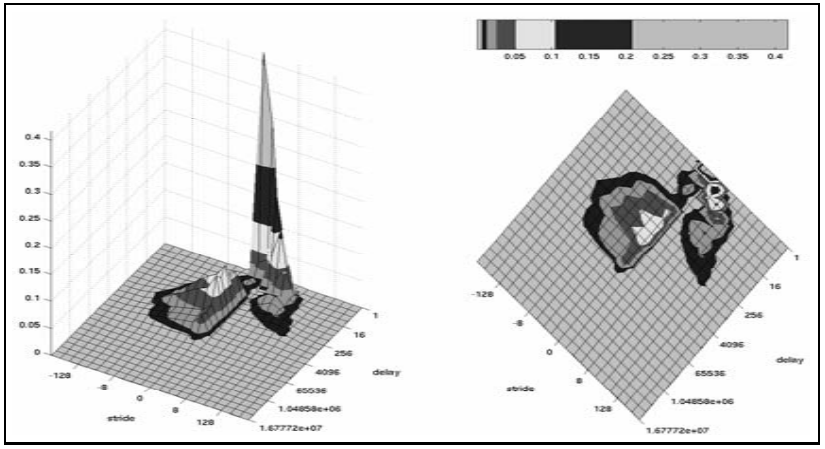
Data trace of *twolf* under Windows NT.



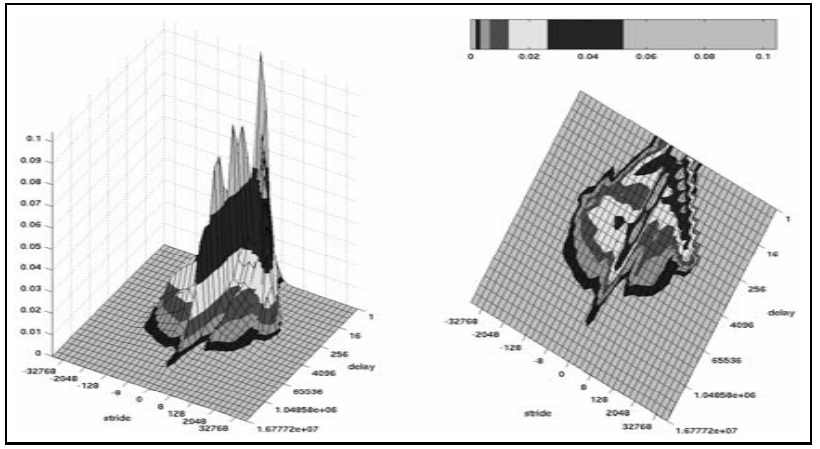
Instruction trace of *vortex.one* under Windows NT.



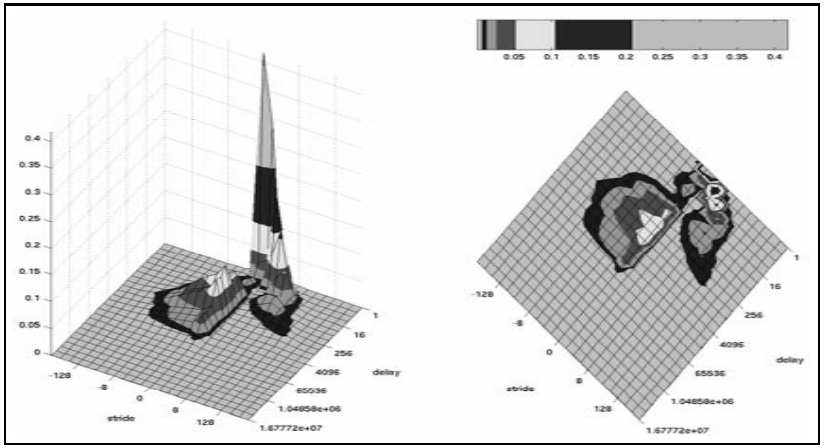
Data trace of *vortex.one* under Windows NT.



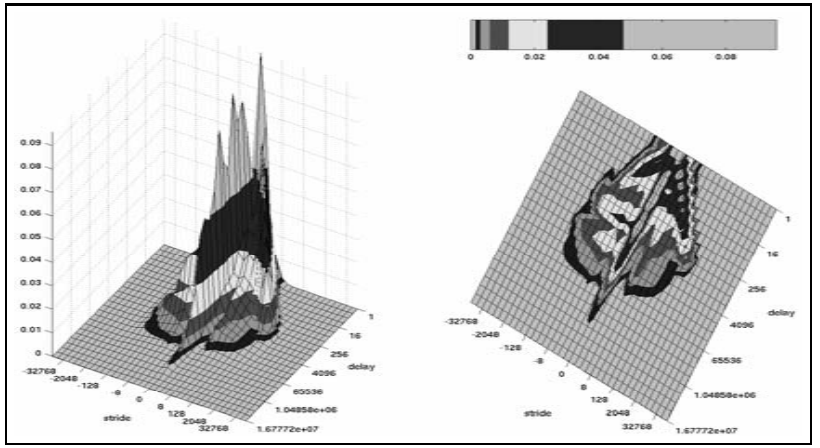
Instruction trace of *vortex.three* under Windows NT.



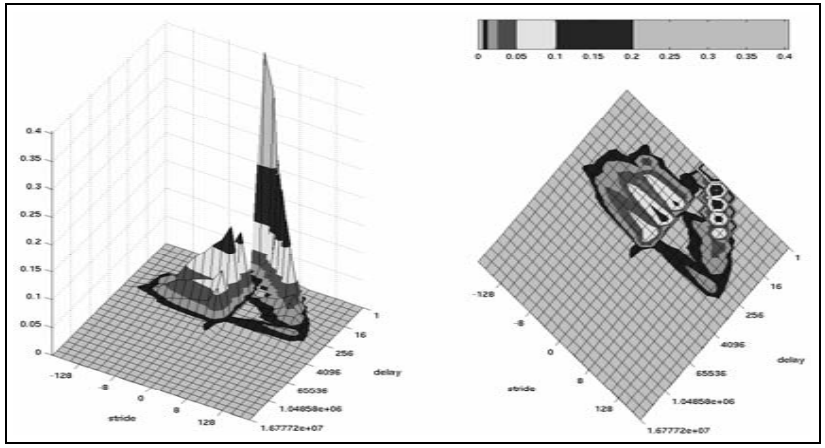
Data trace of *vortex.three* under Windows NT.



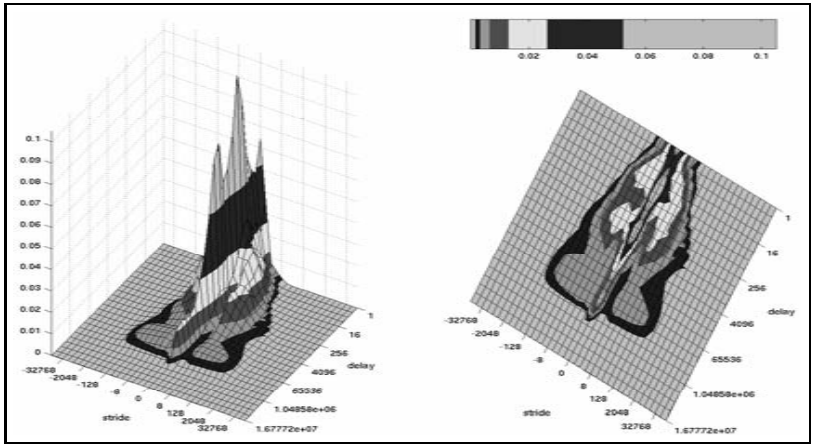
Instruction trace of *vortex.two* under Windows NT.



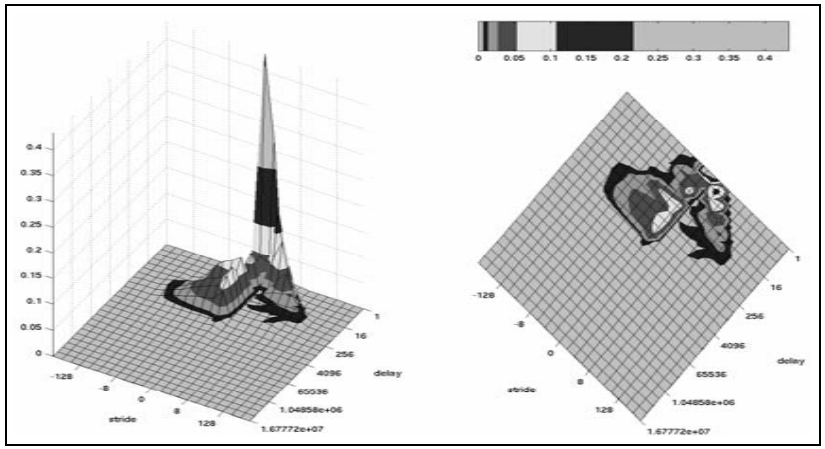
Data trace of *vortex.two* under Windows NT.



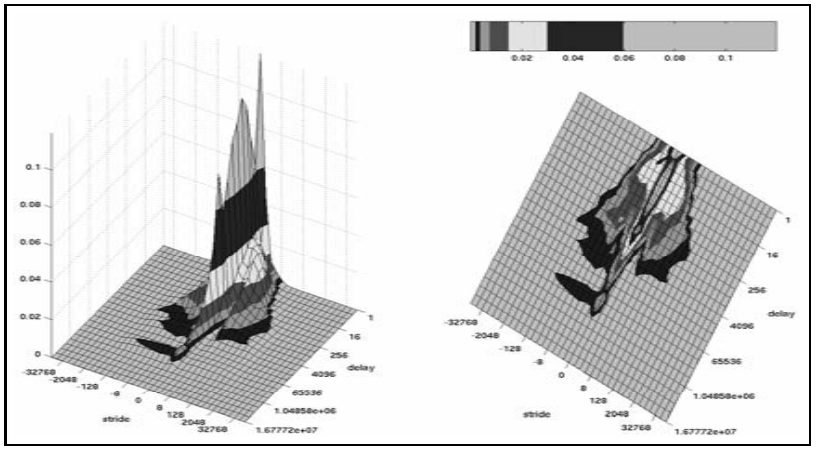
Instruction trace of *vpr.place* under Windows NT.



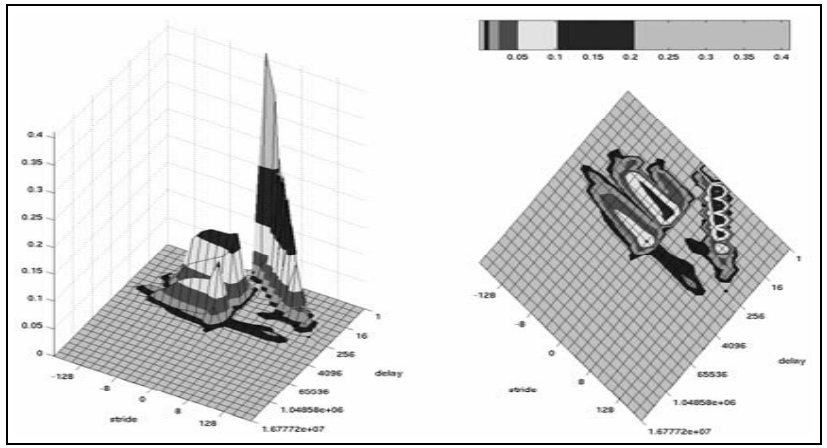
Data trace of *vpr.place* under Windows NT.



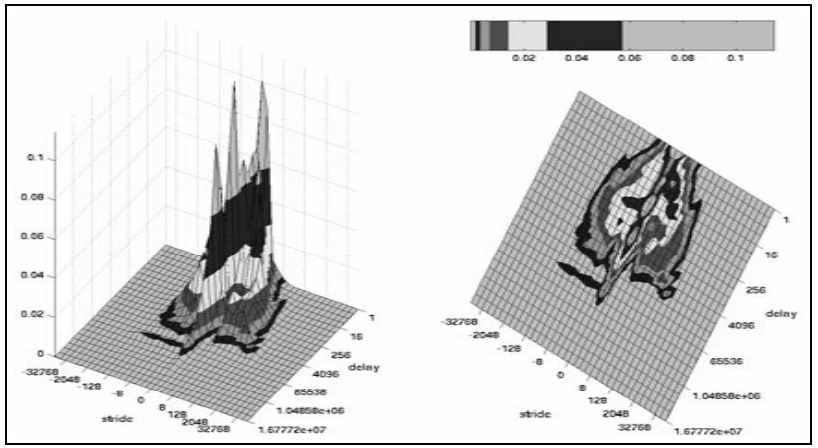
Instruction trace of *vpr.route* under Windows NT.



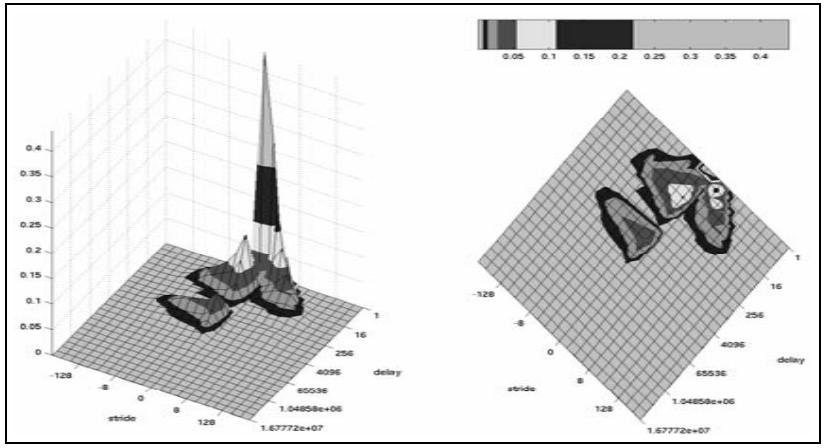
Data trace of *vpr.route* under Windows NT.



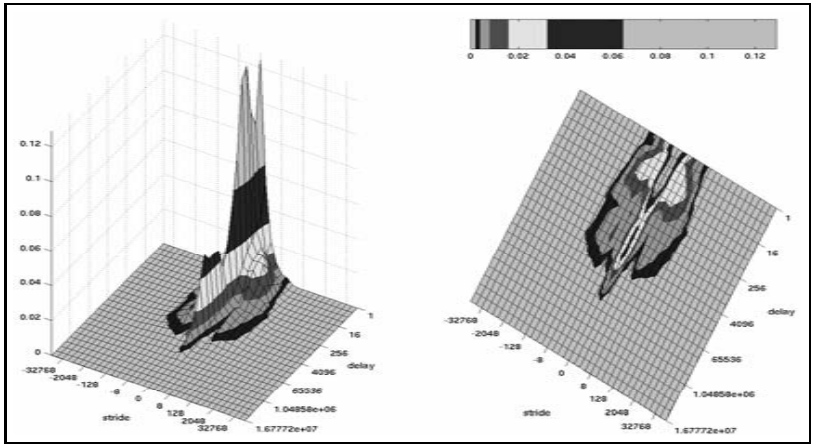
Instruction trace of *wupwise* under Windows NT.



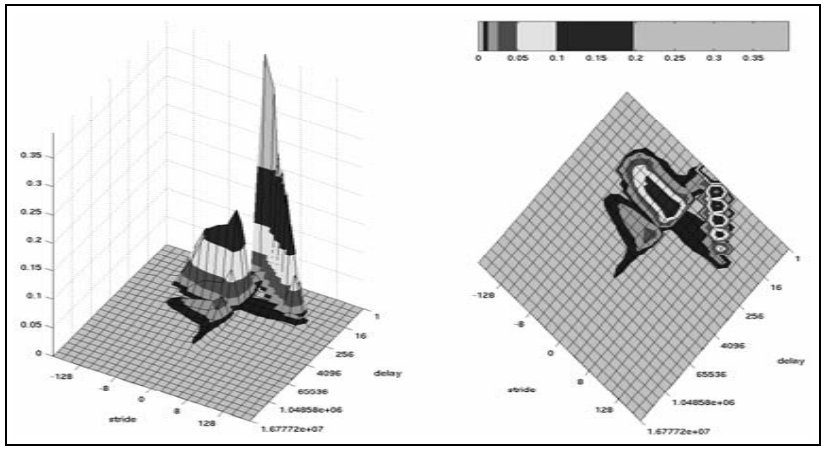
Data trace of *wupwise* under Windows NT.



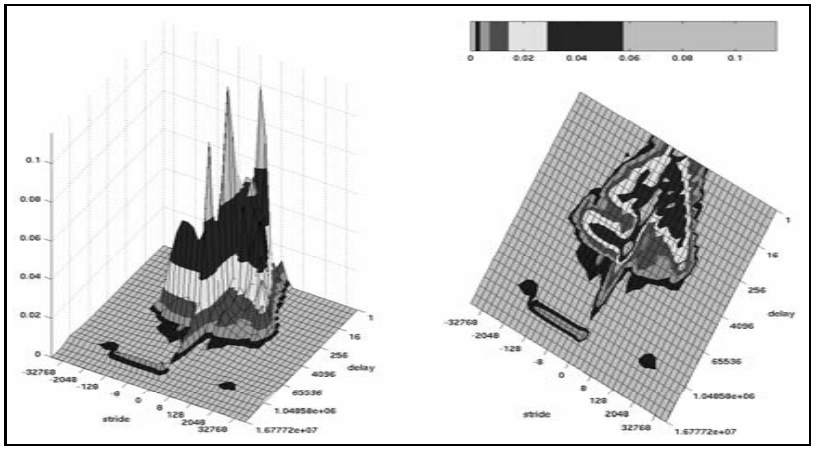
Instruction trace of *ammp* under Windows 2000.



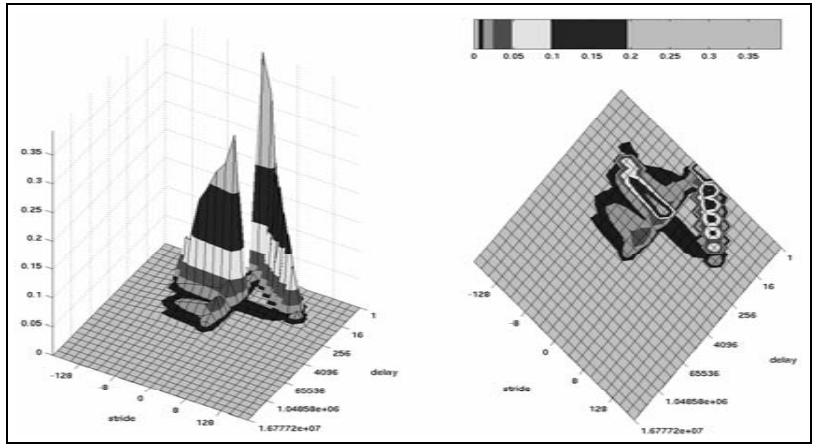
Data trace of *ammp* under Windows 2000.



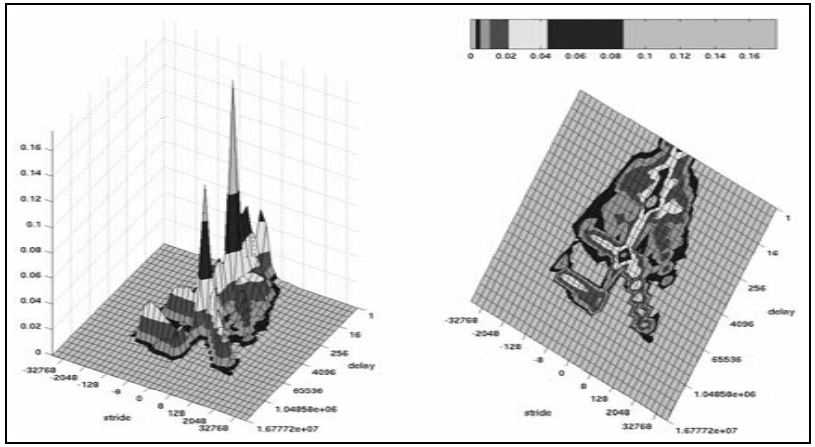
Instruction trace of *applu* under Windows 2000.



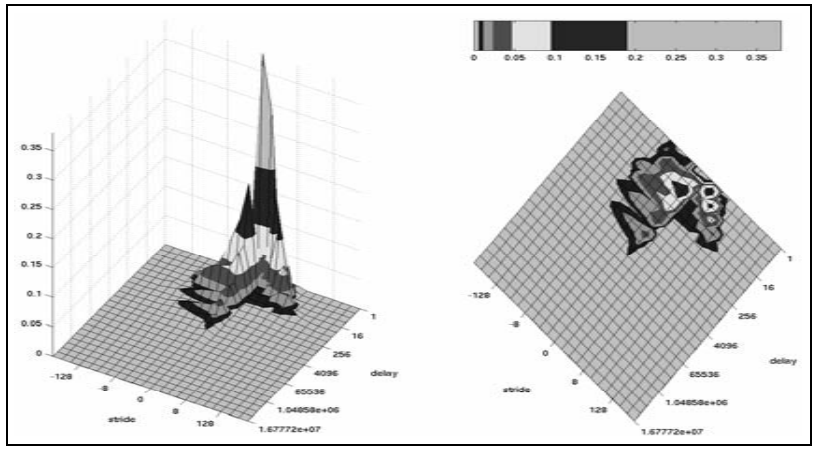
Data trace of *applu* under Windows 2000.



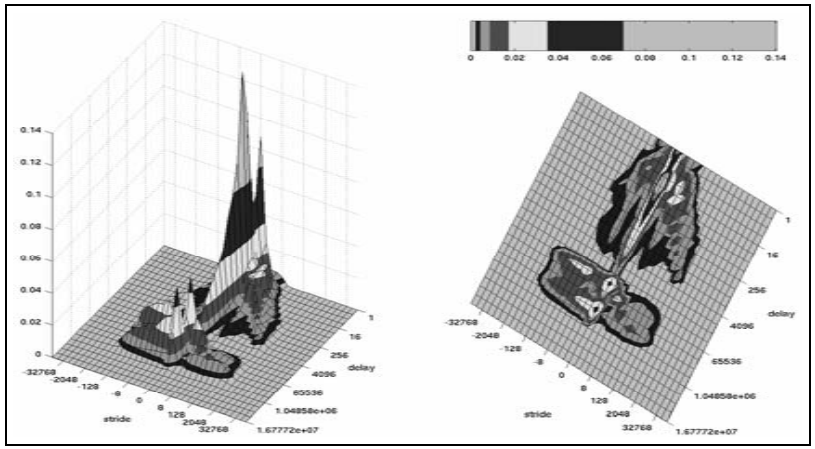
Instruction trace of *apsi* under Windows 2000.



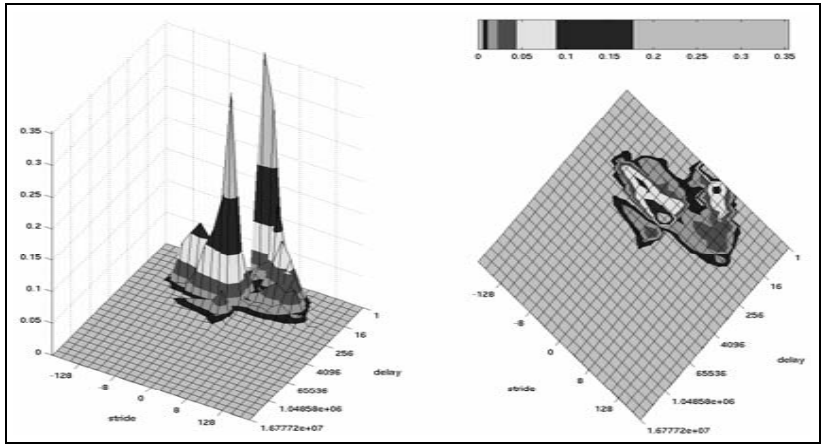
Data trace of *apsi* under Windows 2000.



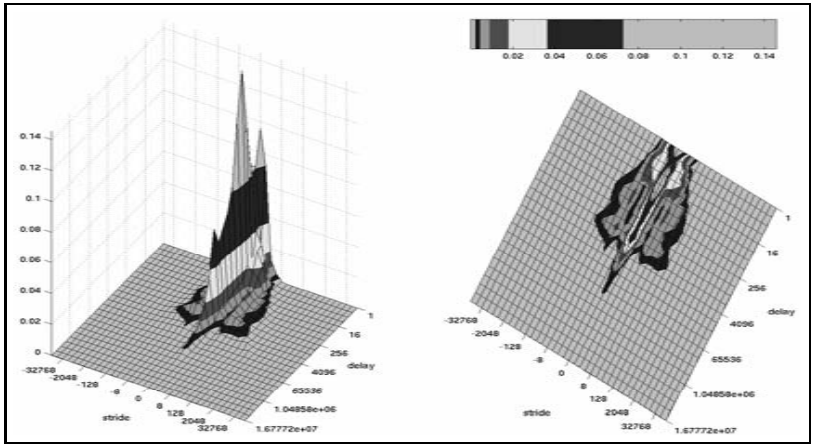
Instruction trace of *art* under Windows 2000.



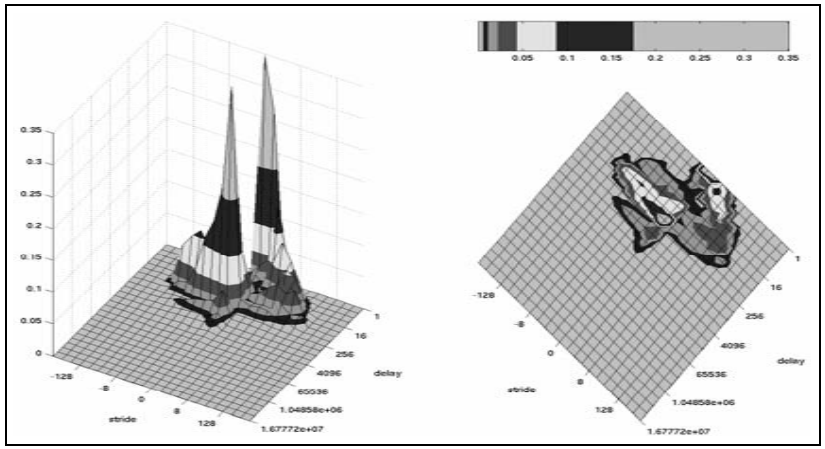
Data trace of *art* under Windows 2000.



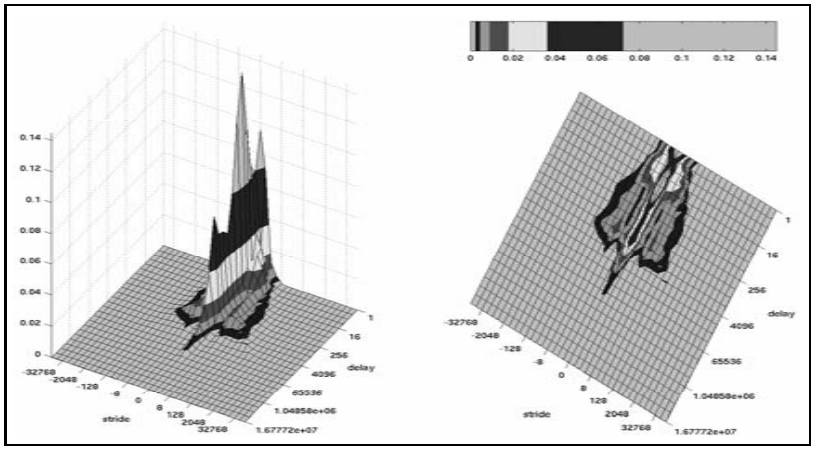
Instruction trace of *bzip2.g7* under Windows 2000.



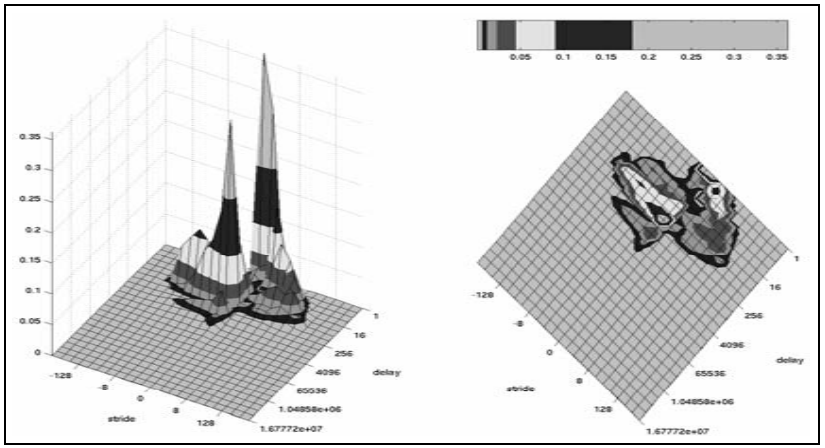
Data trace of *bzip2.g7* under Windows 2000.



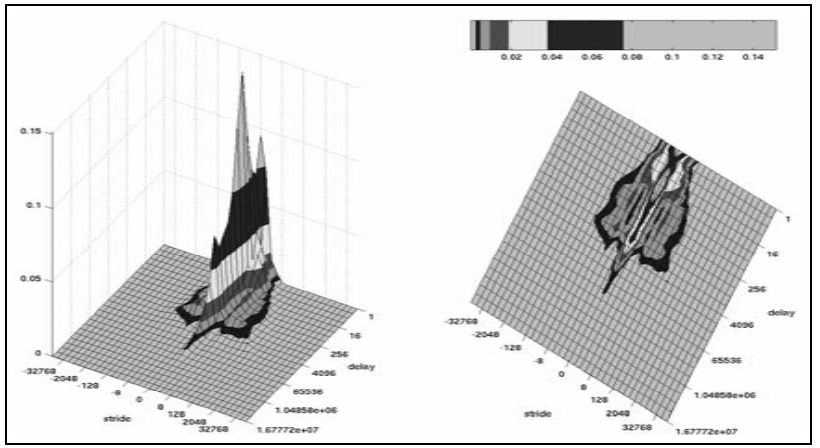
Instruction trace of *bzip2.g9* under Windows 2000.



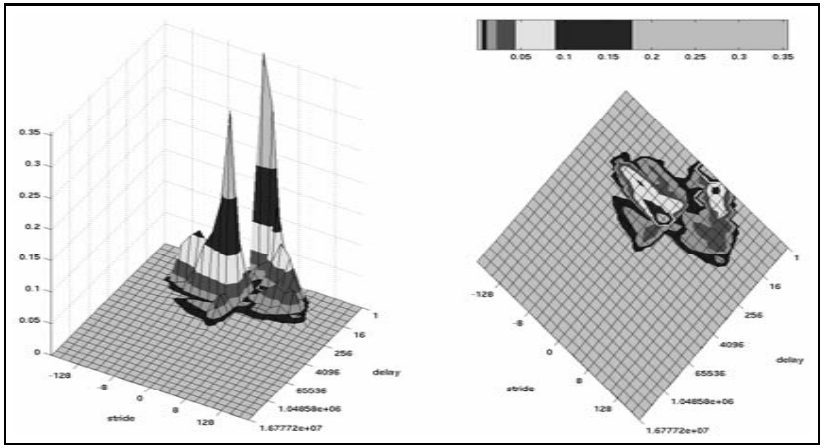
Data trace of *bzip2.g9* under Windows 2000.



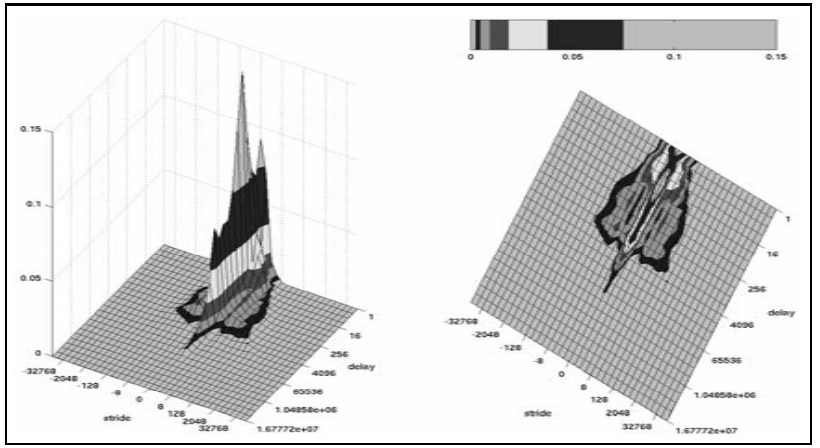
Instruction trace of *bzip2.p7* under Windows 2000.



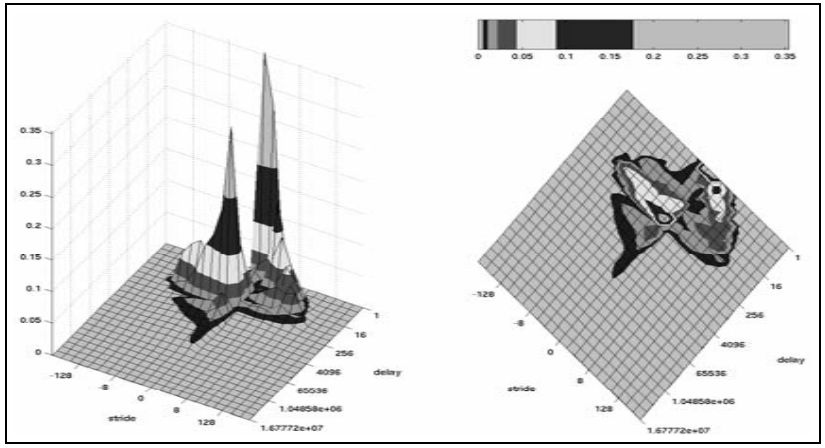
Data trace of *bzip2.p7* under Windows 2000.



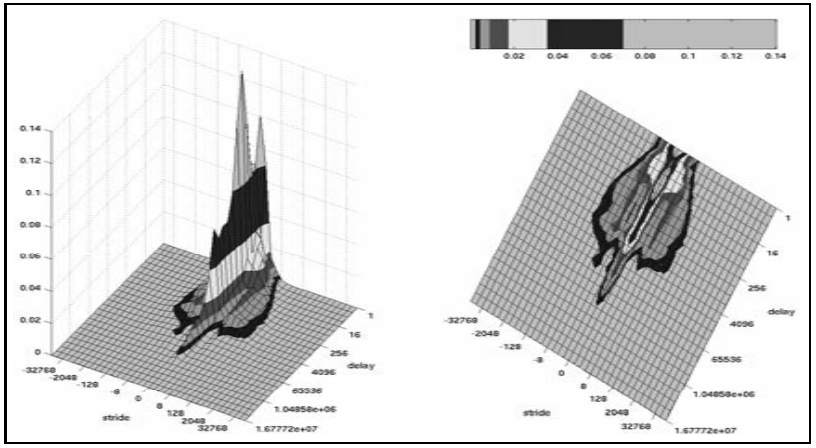
Instruction trace of *bzip2.p9* under Windows 2000.



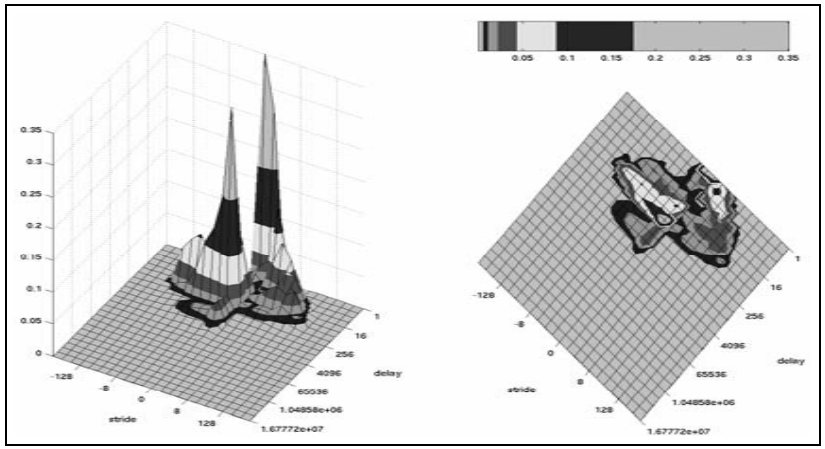
Data trace of *bzip2.p9* under Windows 2000.



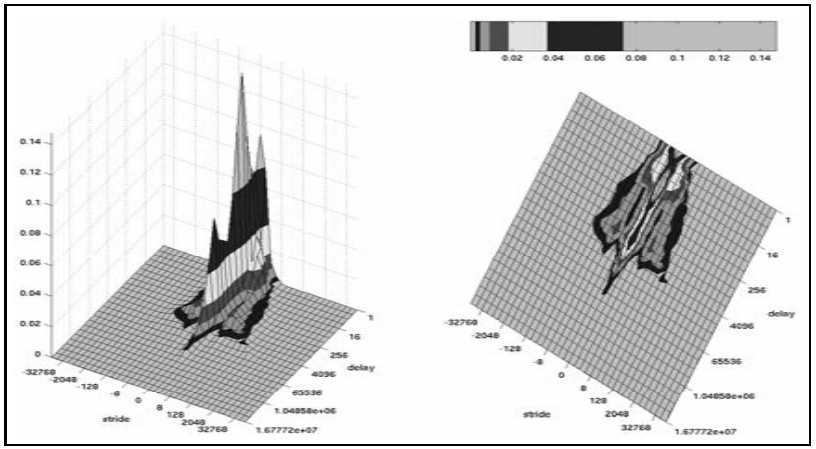
Instruction trace of *bzip2.s7* under Windows 2000.



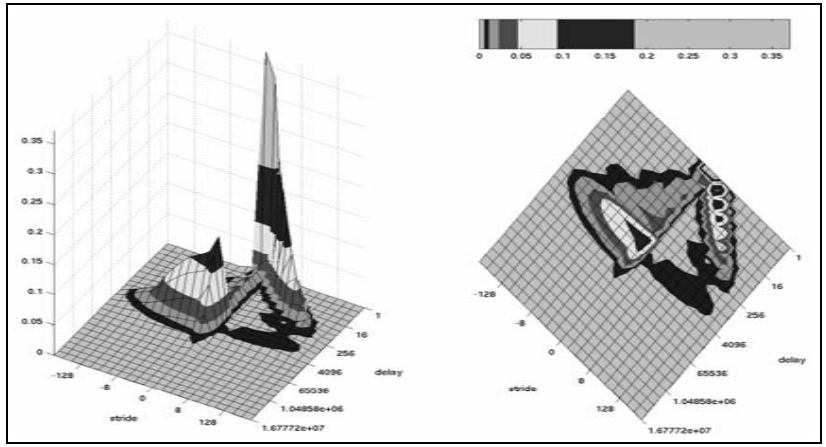
Data trace of *bzip2.s7* under Windows 2000.



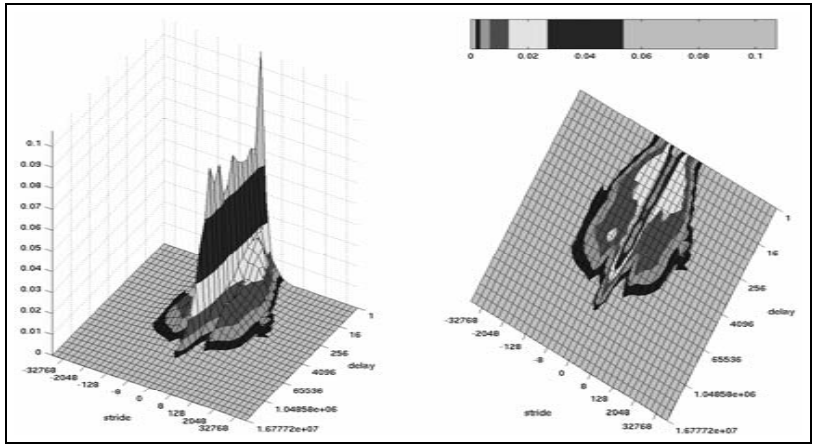
Instruction trace of *bzip2.s9* under Windows 2000.



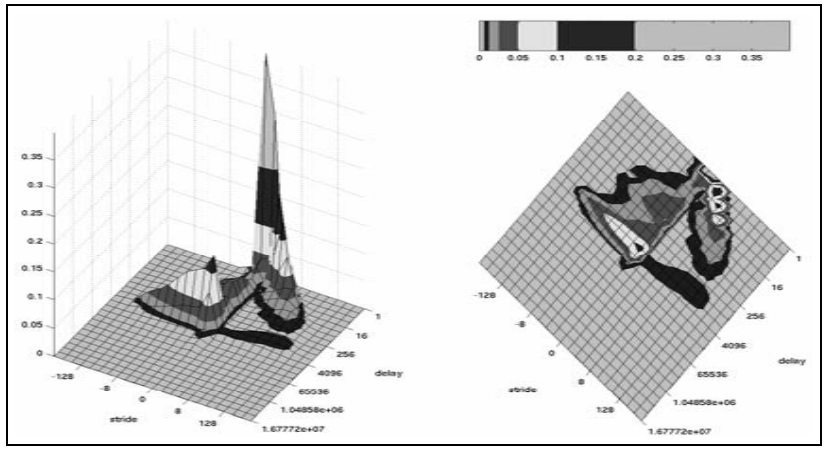
Data trace of *bzip2.s9* under Windows 2000.



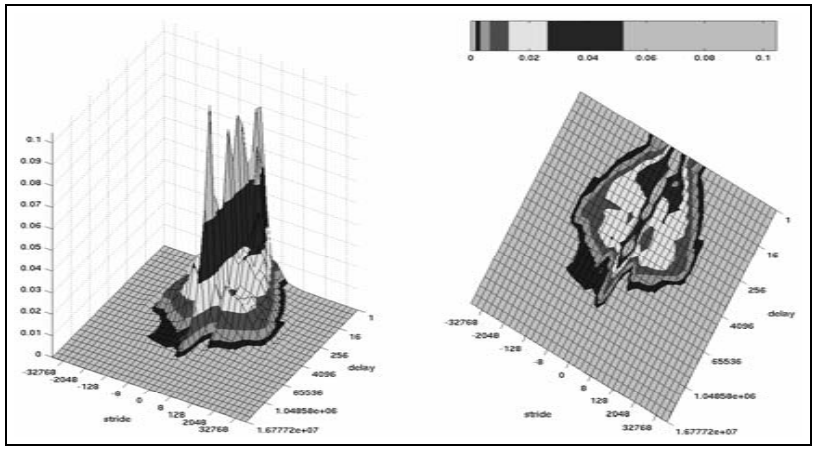
Instruction trace of *crafty* under Windows 2000.



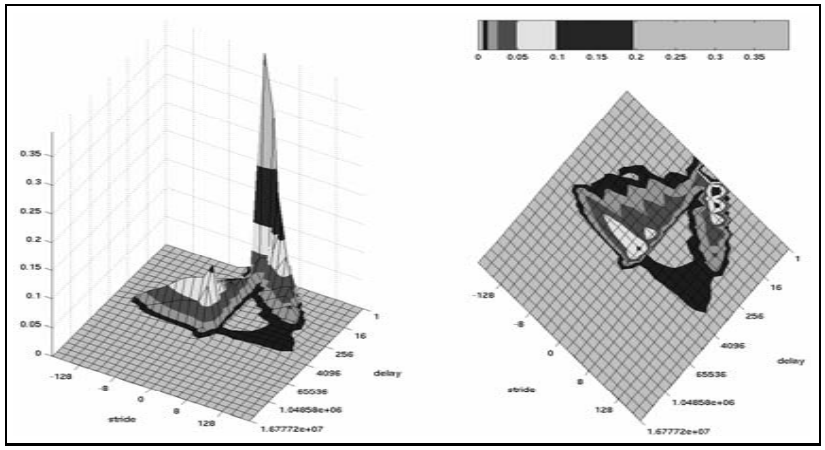
Data trace of *crafty* under Windows 2000.



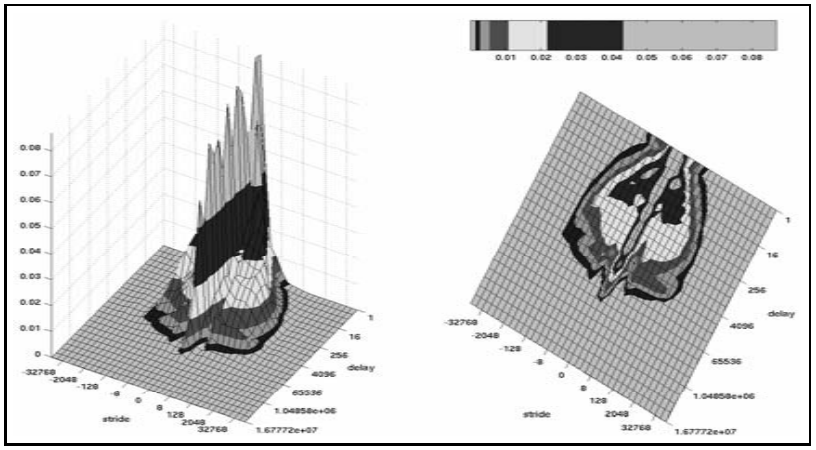
Instruction trace of *eon.cook* under Windows 2000.



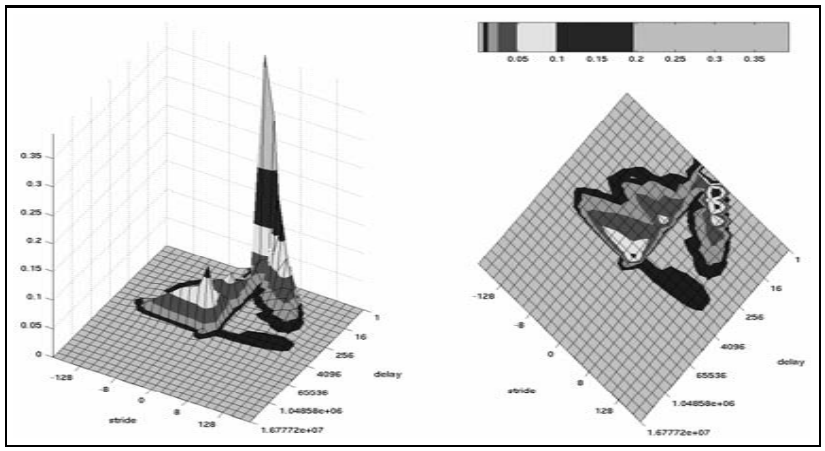
Data trace of *eon.cook* under Windows 2000.



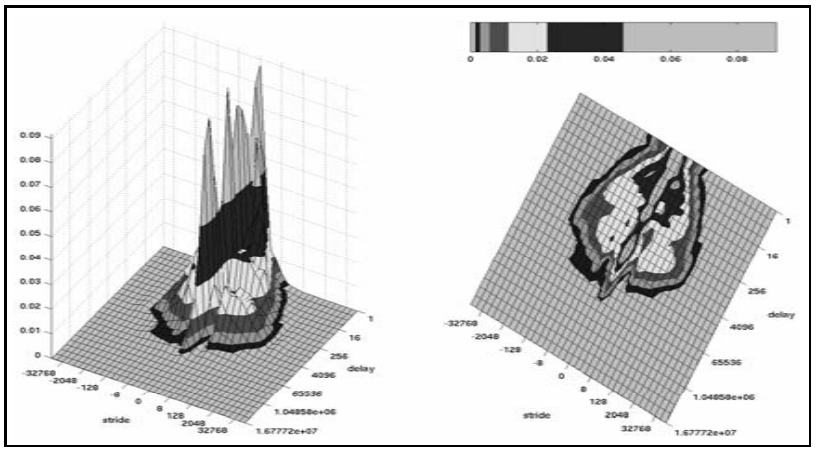
Instruction trace of *eon.kajiya* under Windows 2000.



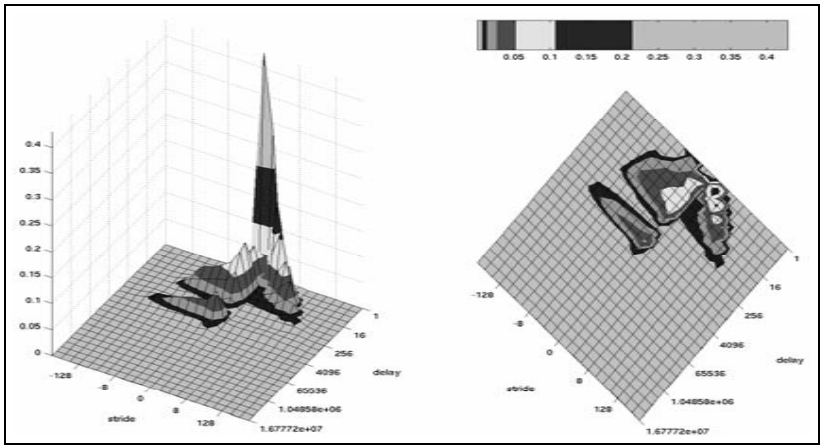
Data trace of *eon.kajiya* under Windows 2000.



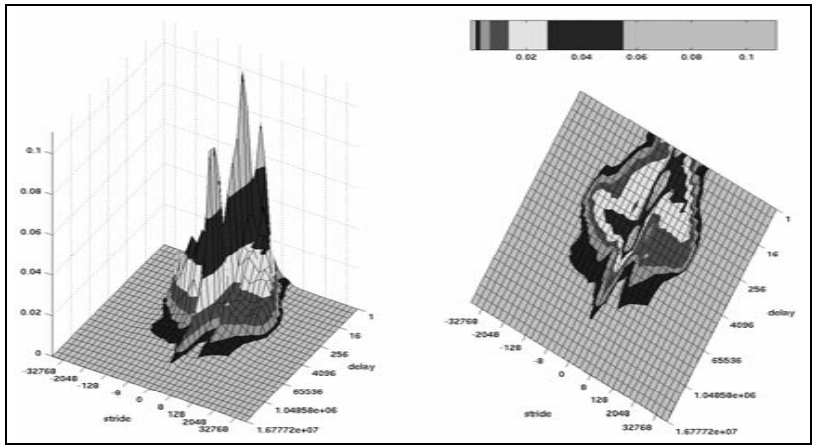
Instruction trace of *eon.rushmeier* under Windows 2000.



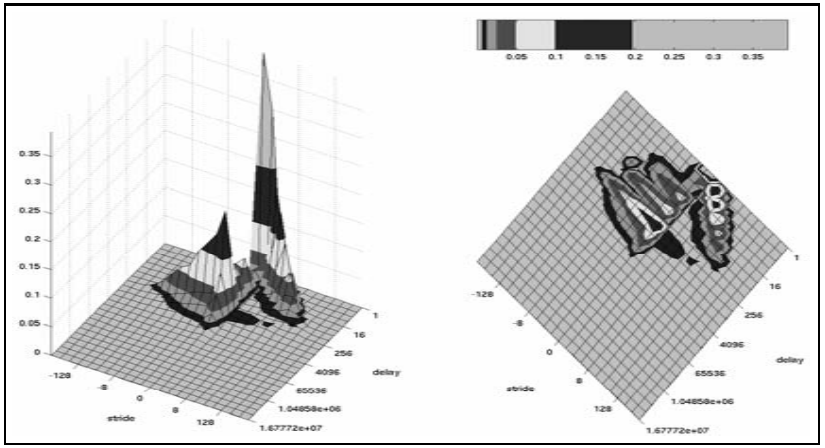
Data trace of *eon.rushmeier* under Windows 2000.



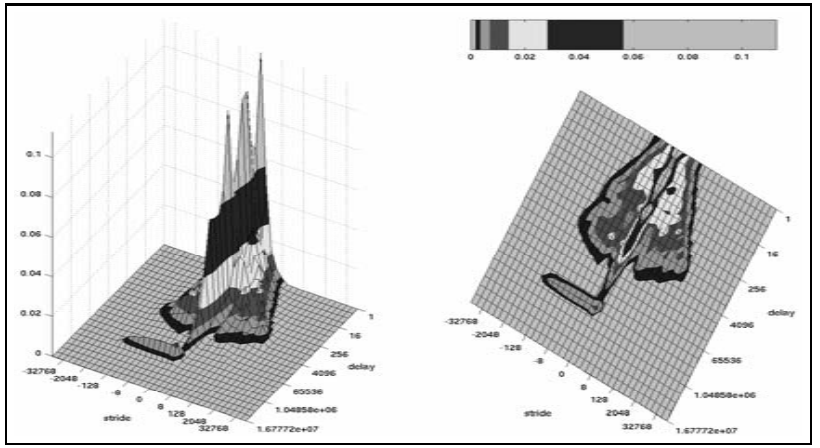
Instruction trace of *equake* under Windows 2000.



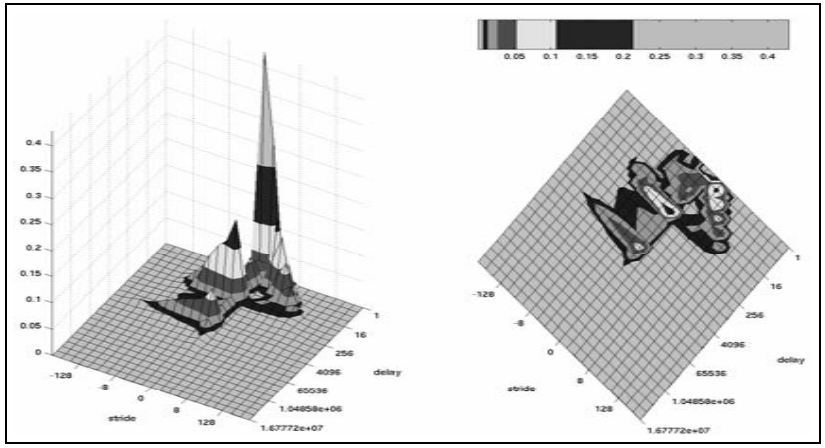
Data trace of *equake* under Windows 2000.



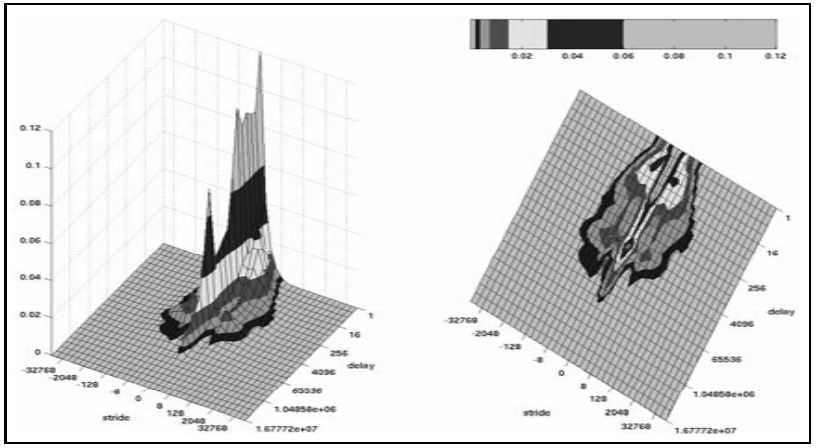
Instruction trace of *facerec* under Windows 2000.



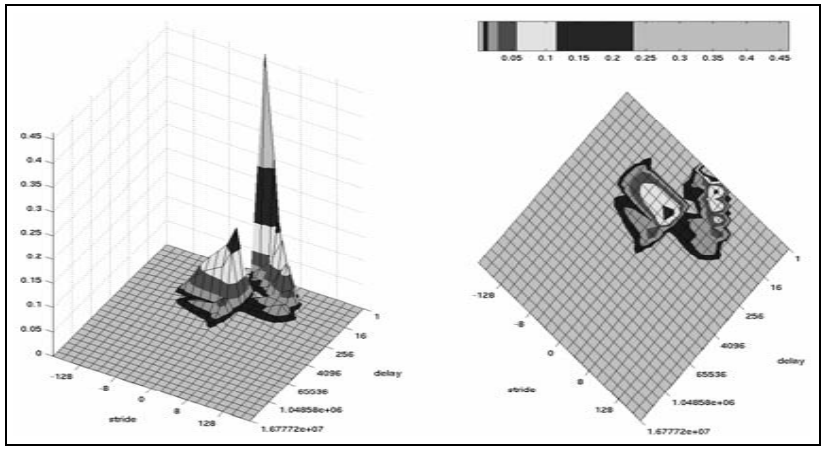
Data trace of *facerec* under Windows 2000.



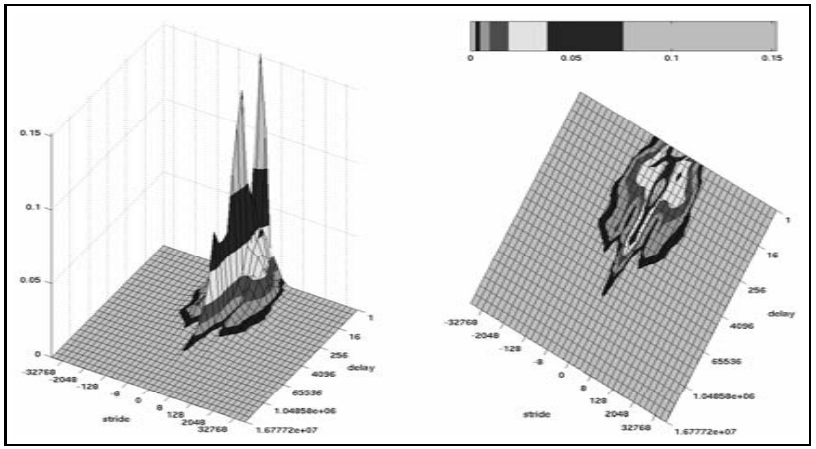
Instruction trace of *fma3d* under Windows 2000.



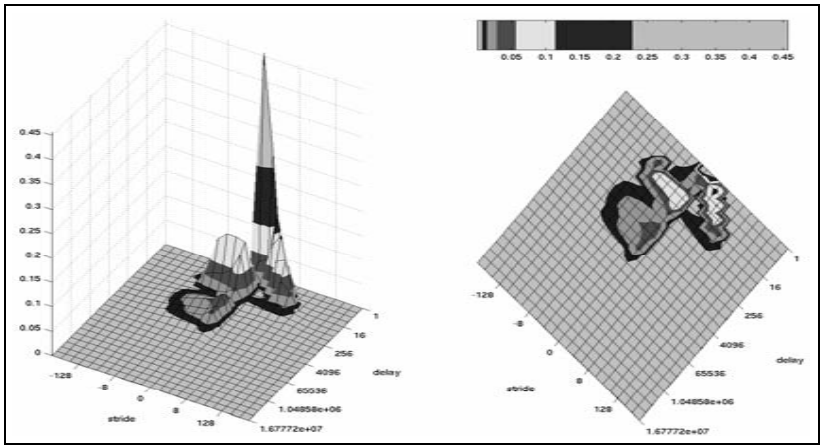
Data trace of *fma3d* under Windows 2000.



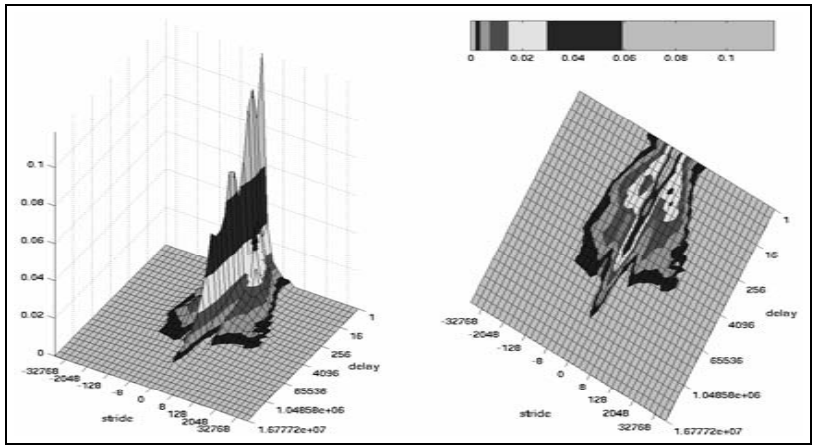
Instruction trace of *galgel* under Windows 2000.



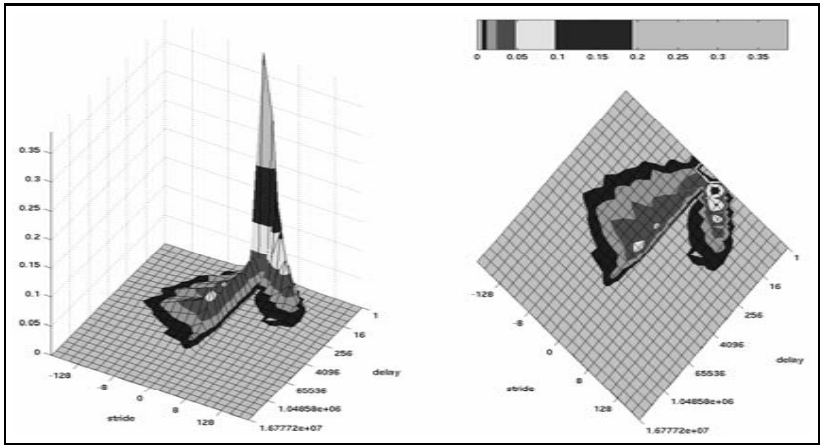
Data trace of *galgel* under Windows 2000.



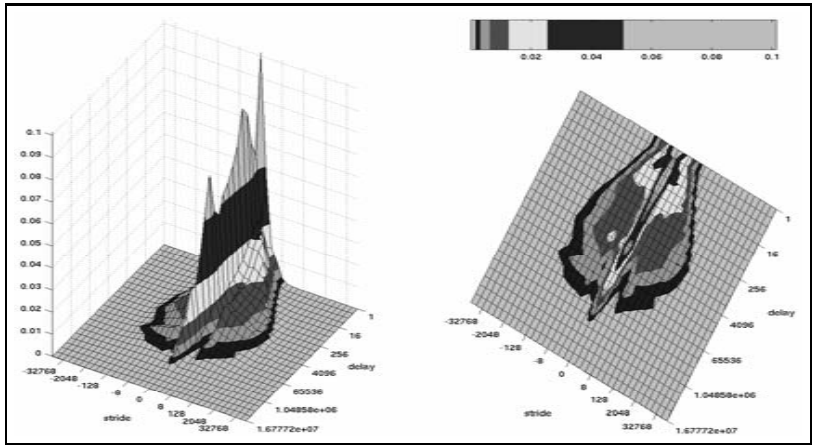
Instruction trace of *gap* under Windows 2000.



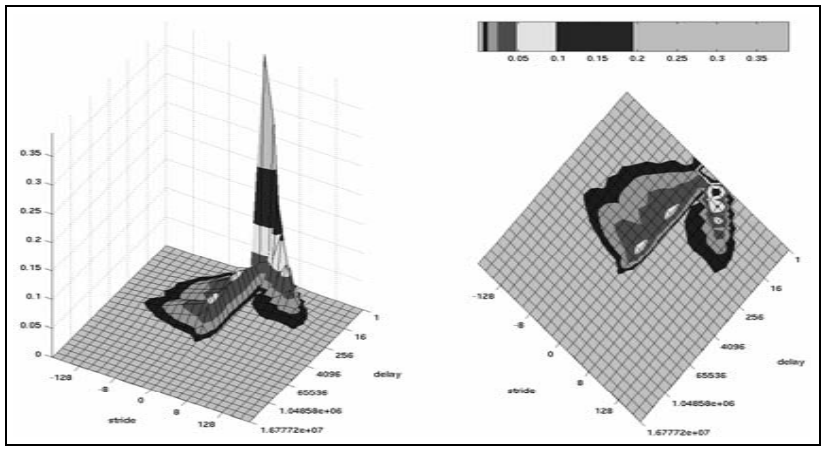
Data trace of *gap* under Windows 2000.



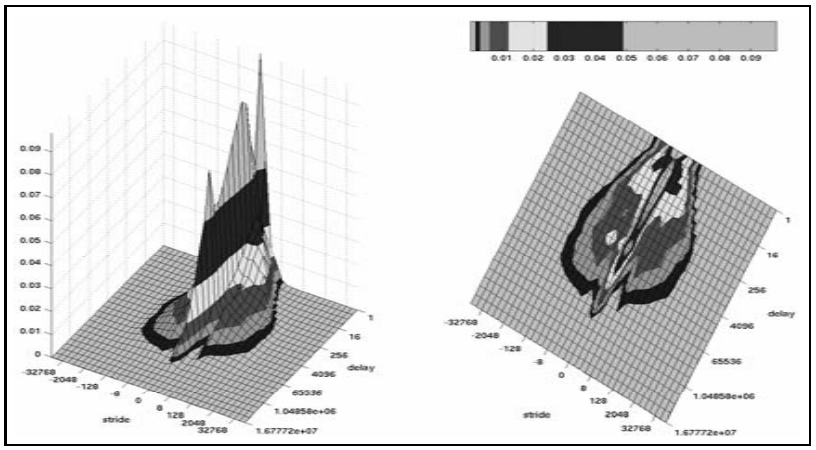
Instruction trace of *gcc.166* under Windows 2000.



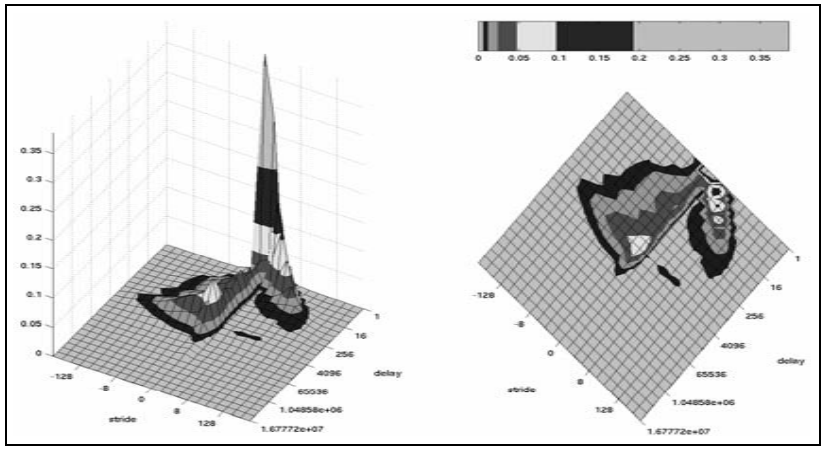
Data trace of *gcc.166* under Windows 2000.



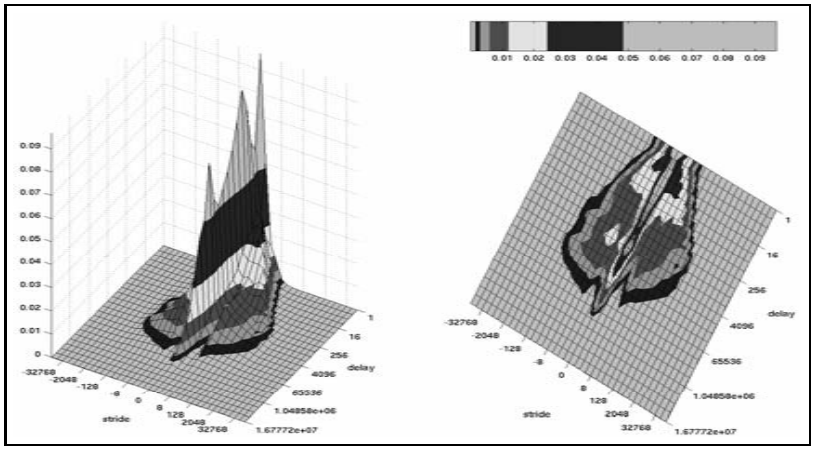
Instruction trace of *gcc.200* under Windows 2000.



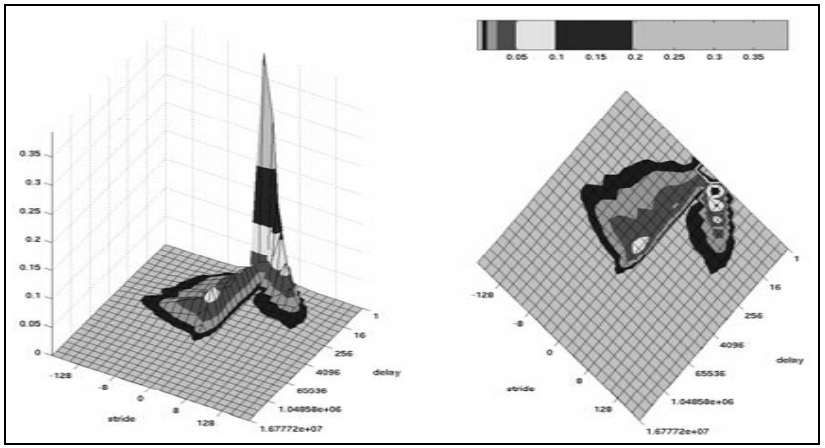
Data trace of *gcc.200* under Windows 2000.



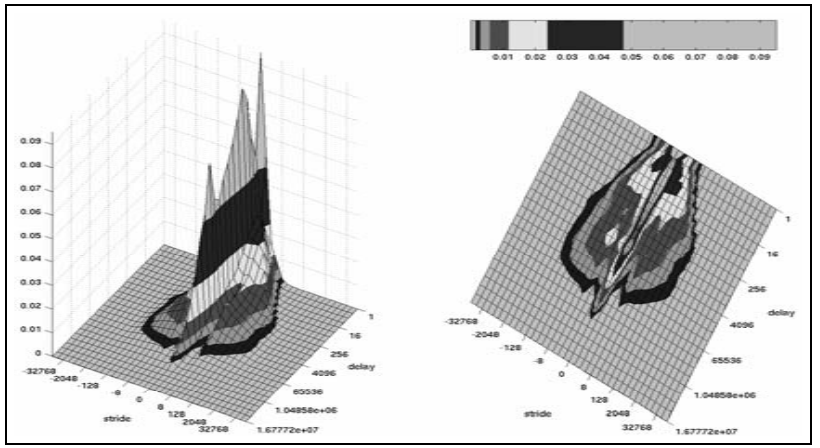
Instruction trace of *gcc.expr* under Windows 2000.



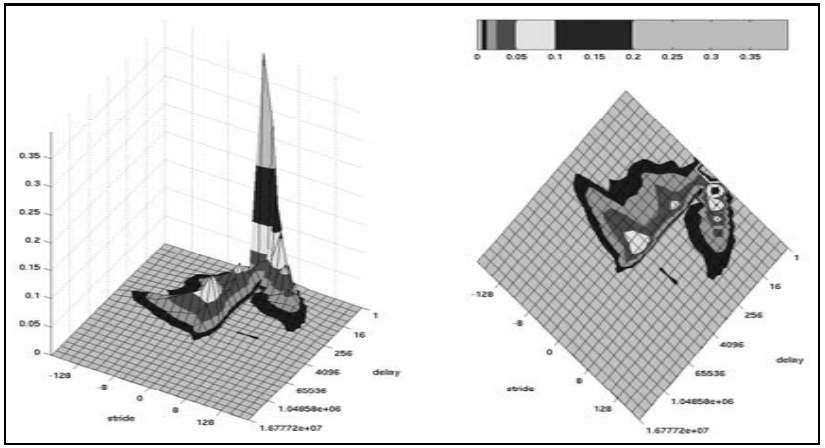
Data trace of *gcc.expr* under Windows 2000.



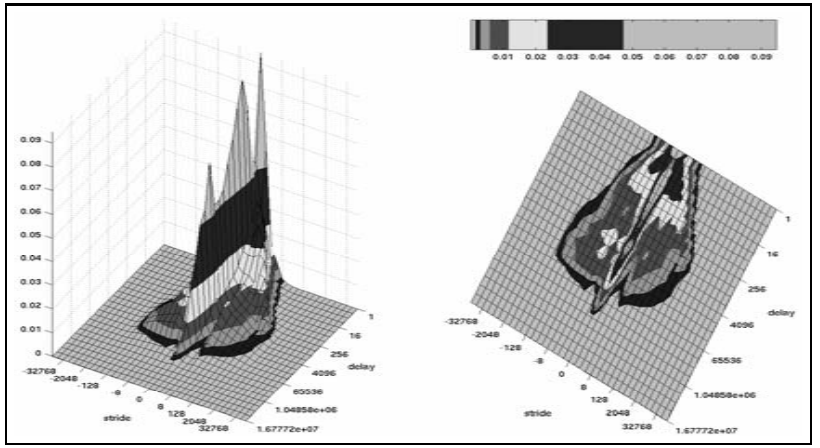
Instruction trace of *gcc.integ* under Windows 2000.



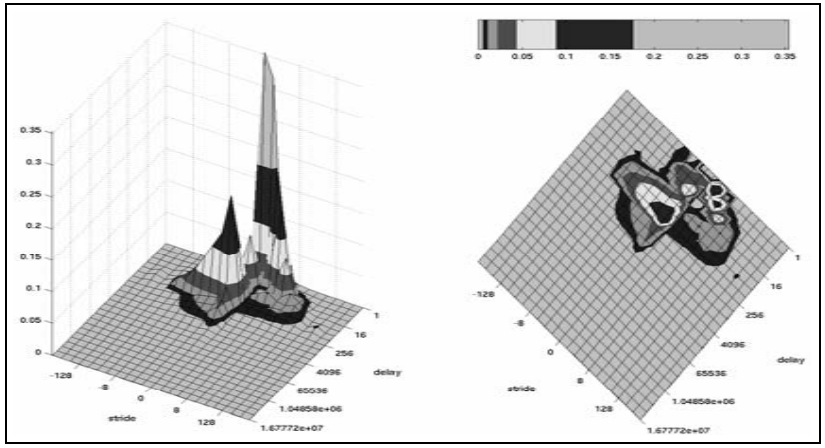
Data trace of *gcc.integ* under Windows 2000.



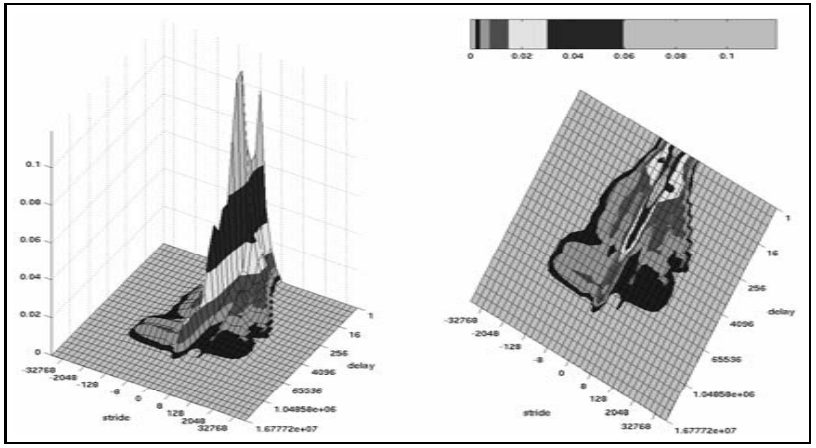
Instruction trace of *gcc.scilab* under Windows 2000.



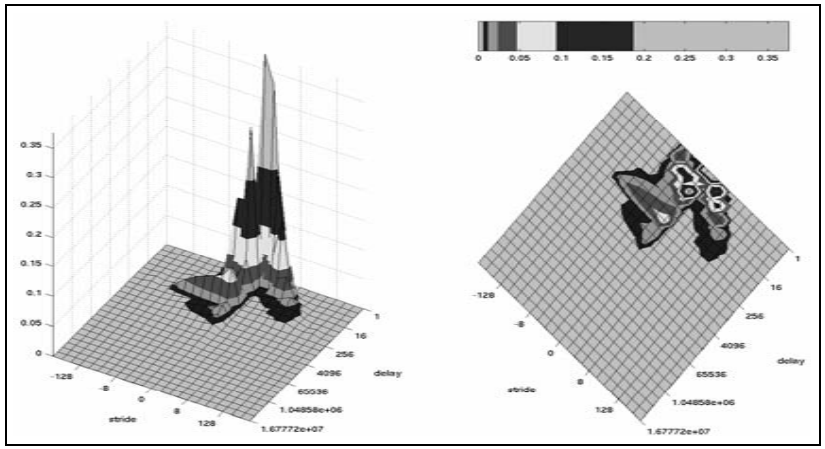
Data trace of *gcc.scilab* under Windows 2000.



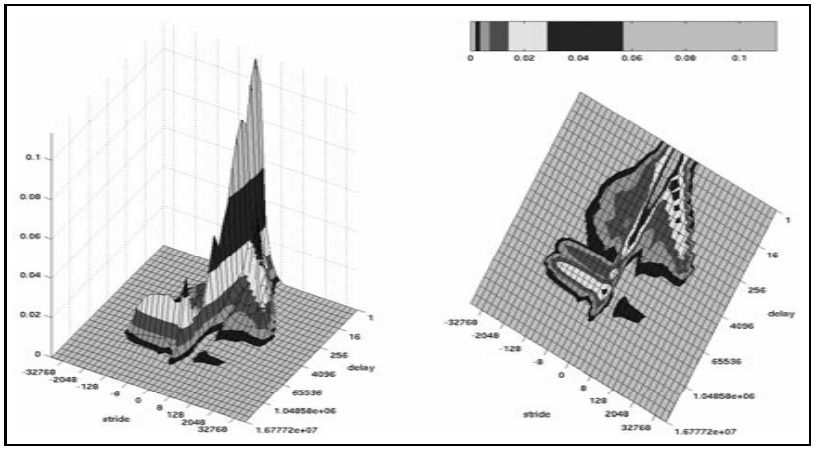
Instruction trace of *gzip.graphic* under Windows 2000.



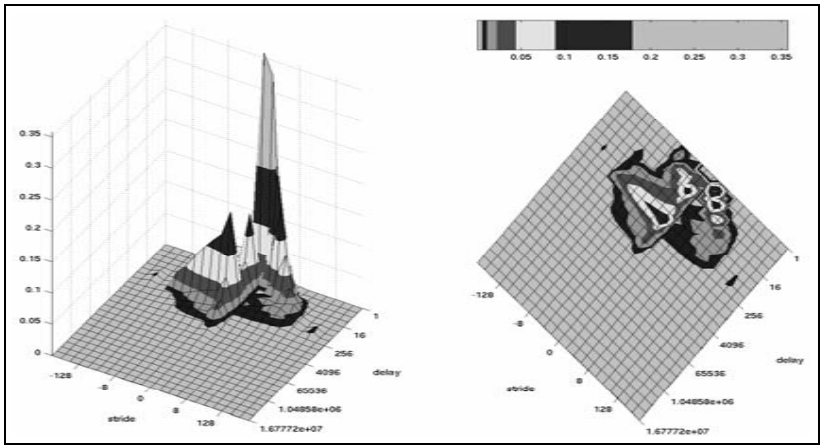
Data trace of *gzip.graphic* under Windows 2000.



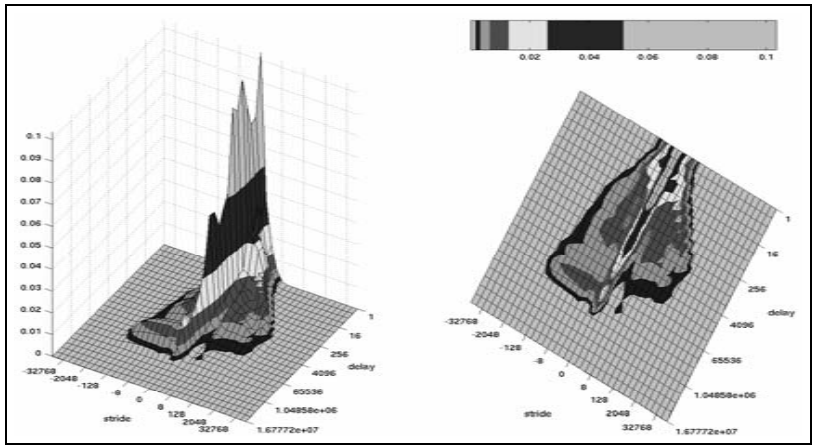
Instruction trace of *gzip.log* under Windows 2000.



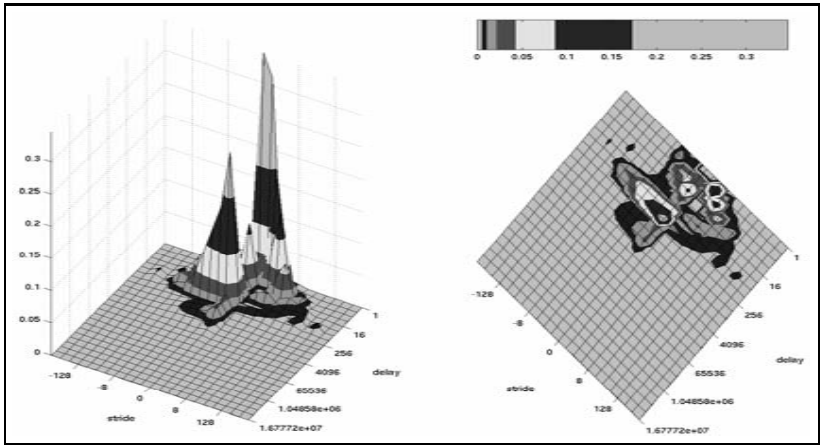
Data trace of *gzip.log* under Windows 2000.



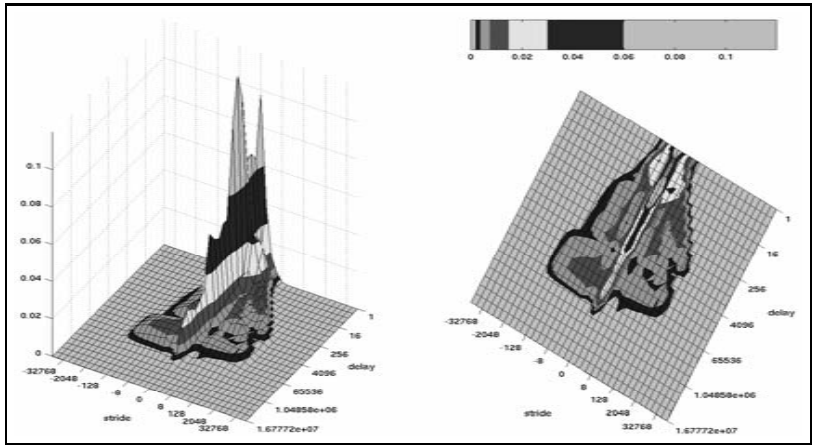
Instruction trace of *gzip.program* under Windows 2000.



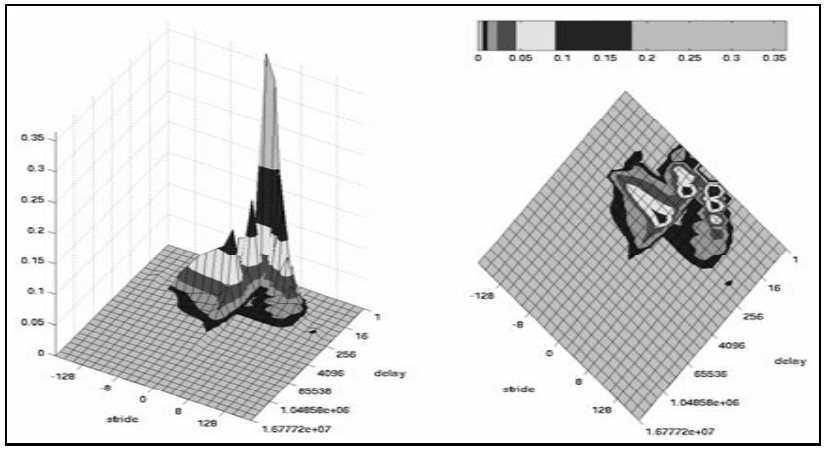
Data trace of *gzip.program* under Windows 2000.



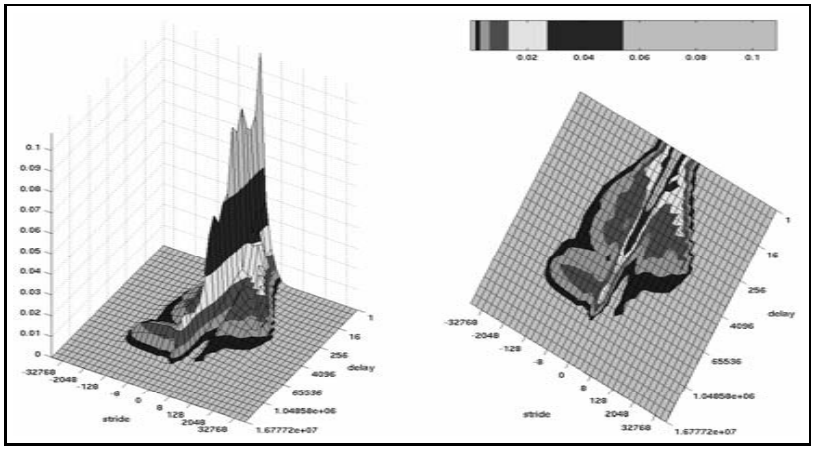
Instruction trace of *gzip.random* under Windows 2000.



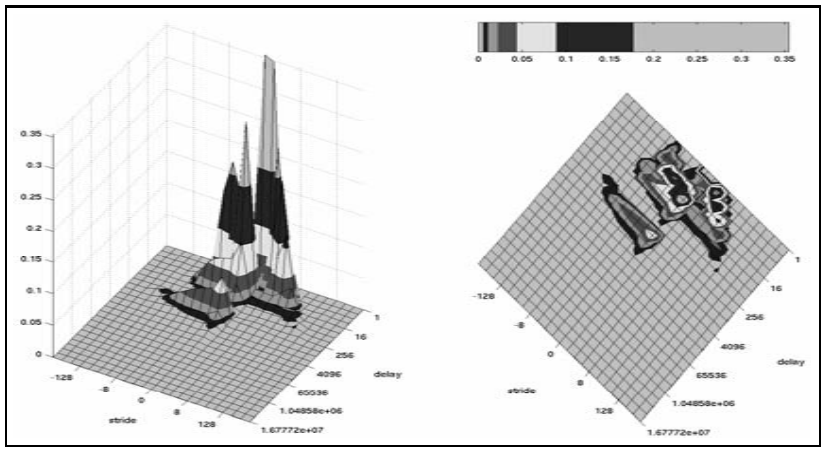
Data trace of *gzip.random* under Windows 2000.



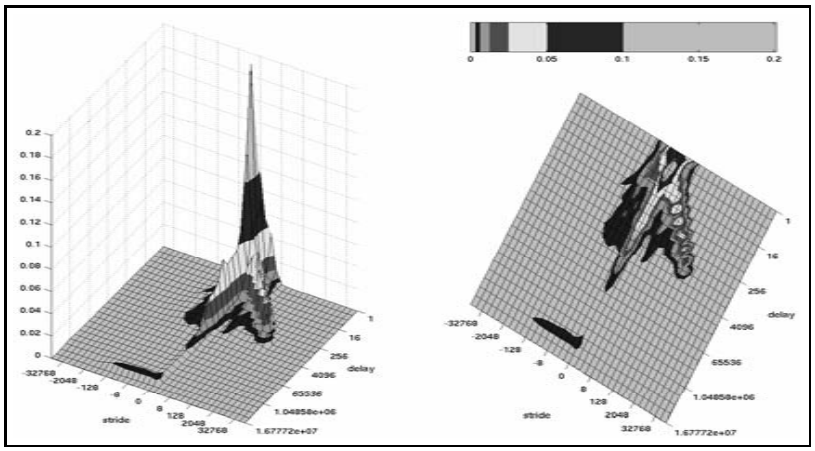
Instruction trace of *gzip.source* under Windows 2000.



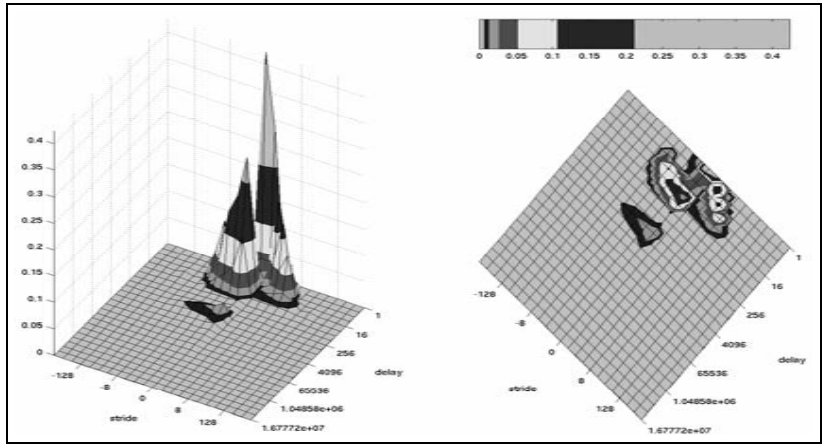
Data trace of *gzip.source* under Windows 2000.



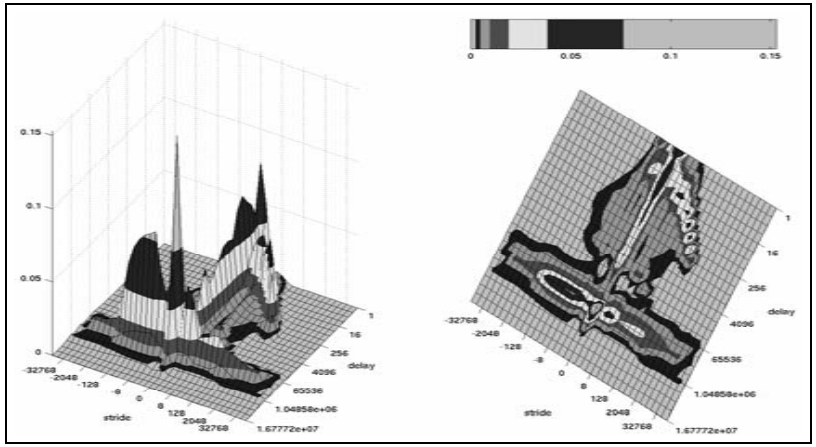
Instruction trace of *lucas* under Windows 2000.



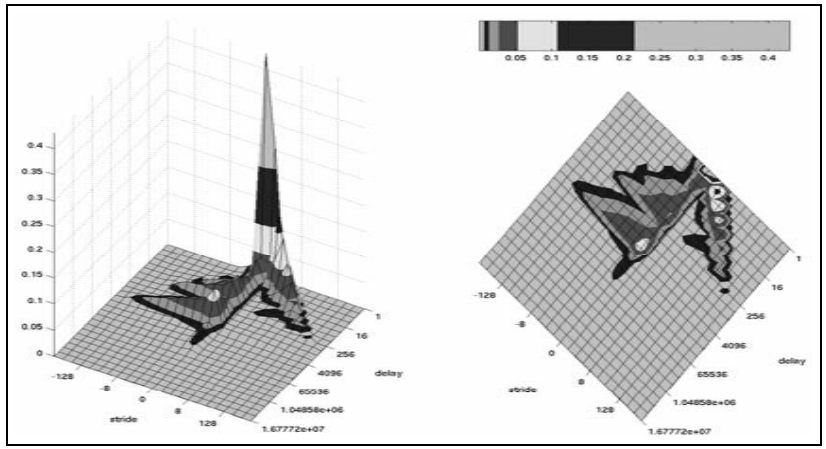
Data trace of *lucas* under Windows 2000.



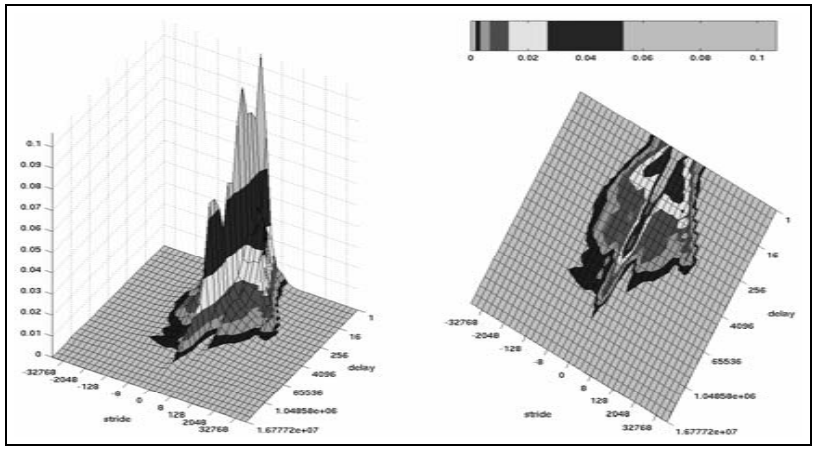
Instruction trace of *mcf* under Windows 2000.



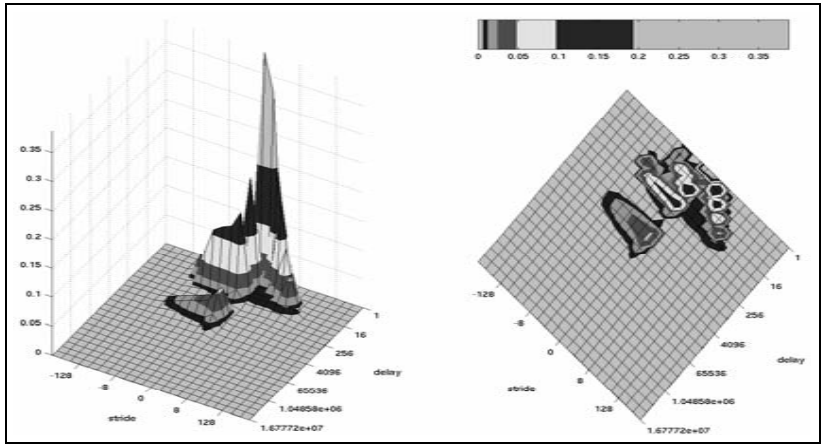
Data trace of *mcf* under Windows 2000.



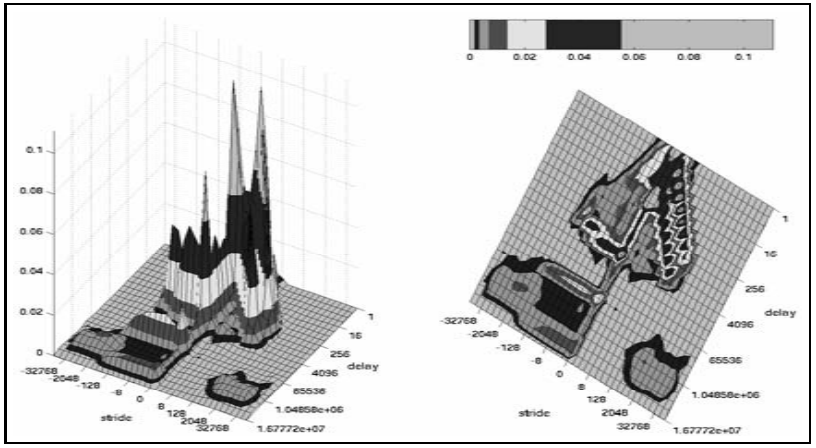
Instruction trace of *mesa* under Windows 2000.



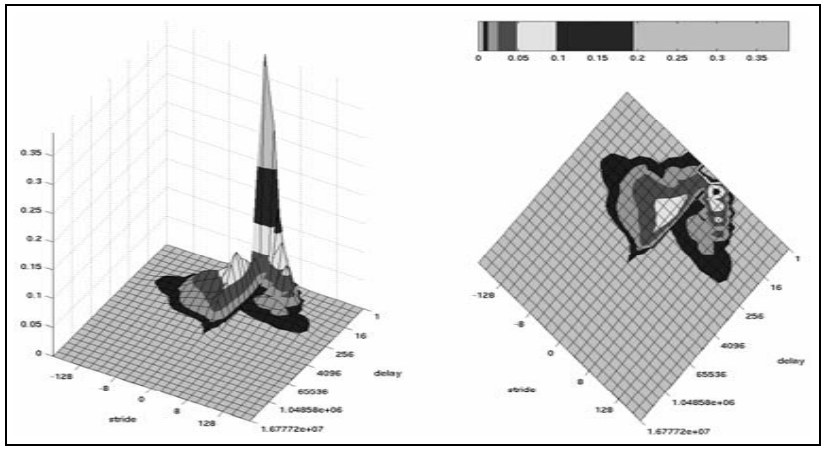
Data trace of *mesa* under Windows 2000.



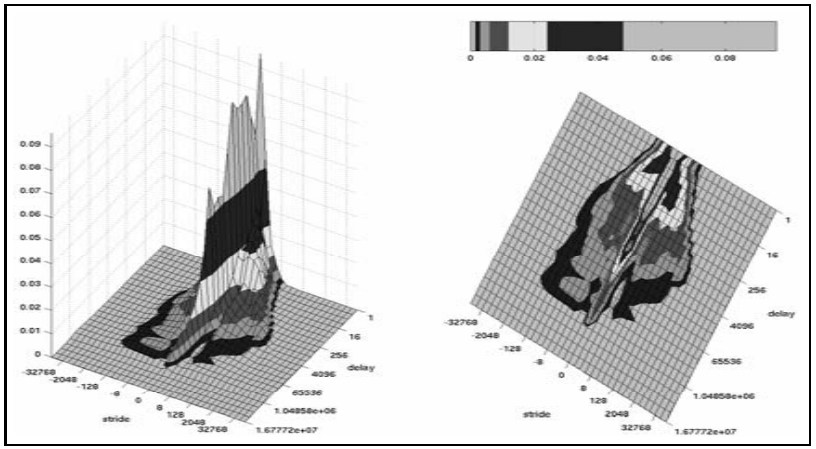
Instruction trace of *mgrid* under Windows 2000.



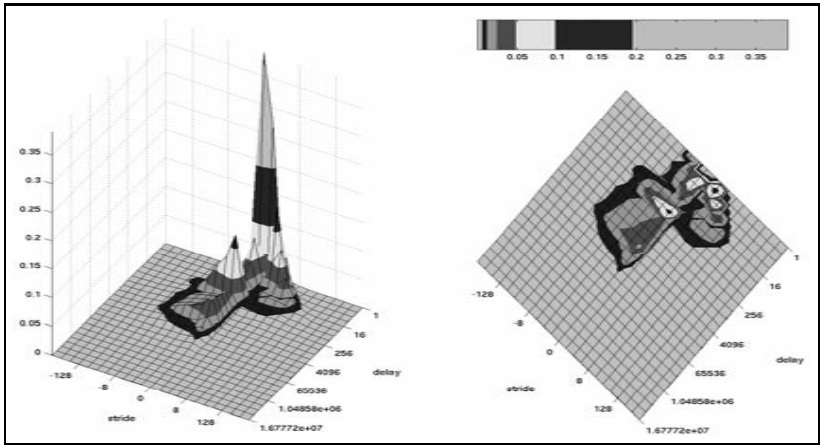
Data trace of *mgrid* under Windows 2000.



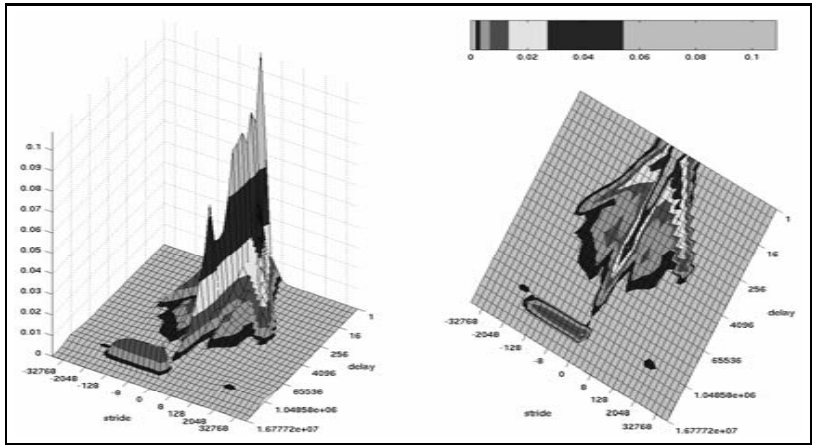
Instruction trace of *parser* under Windows 2000.



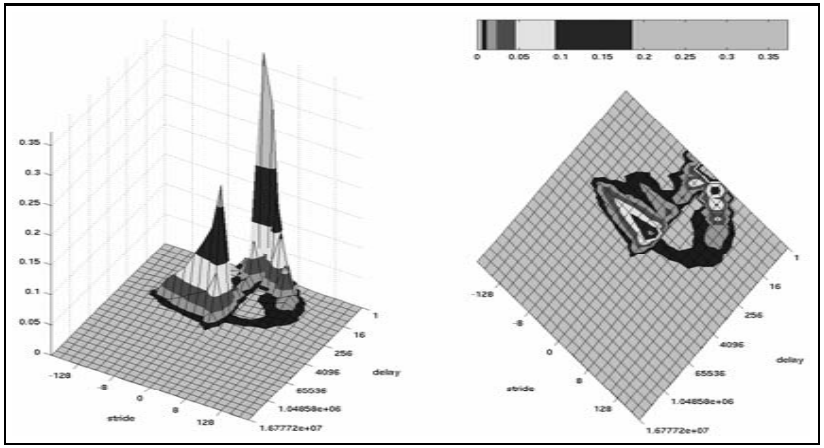
Data trace of *parser* under Windows 2000.



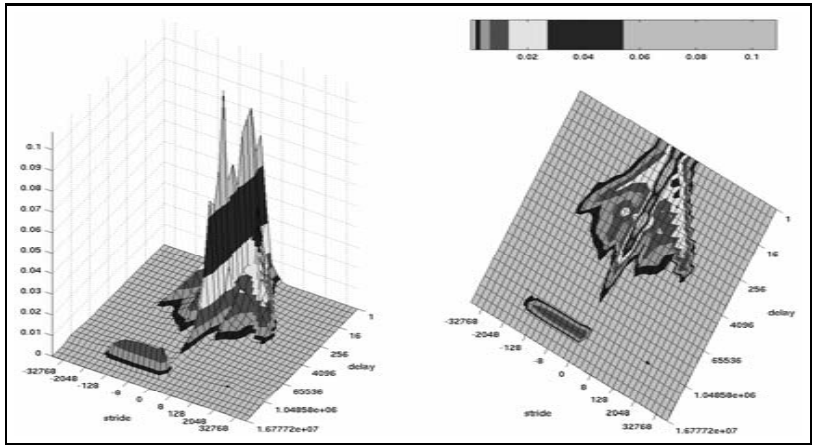
Instruction trace of *perlbmk.diffmail* under Windows 2000.



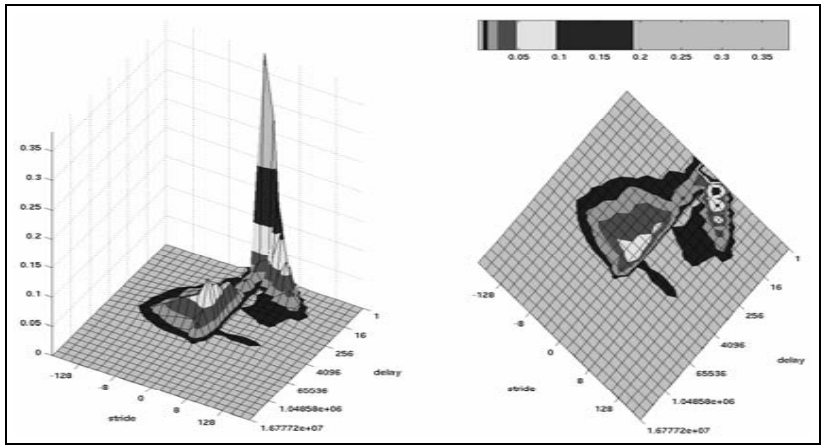
Data trace of *perlbmk.diffmail* under Windows 2000.



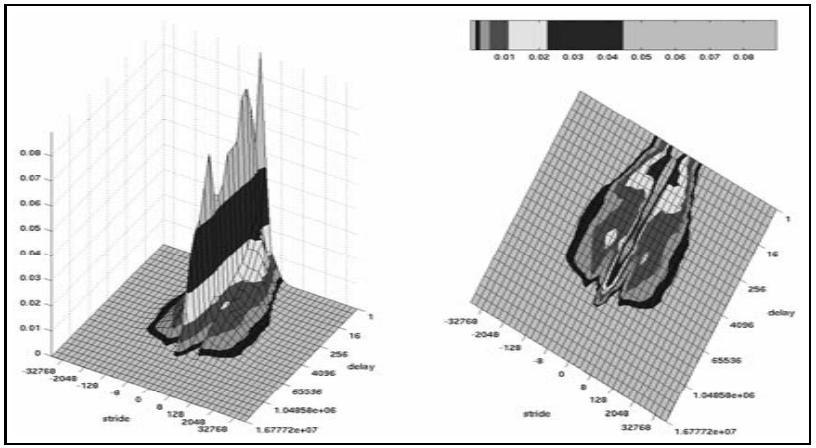
Instruction trace of *perlbmk.makerand* under Windows 2000.



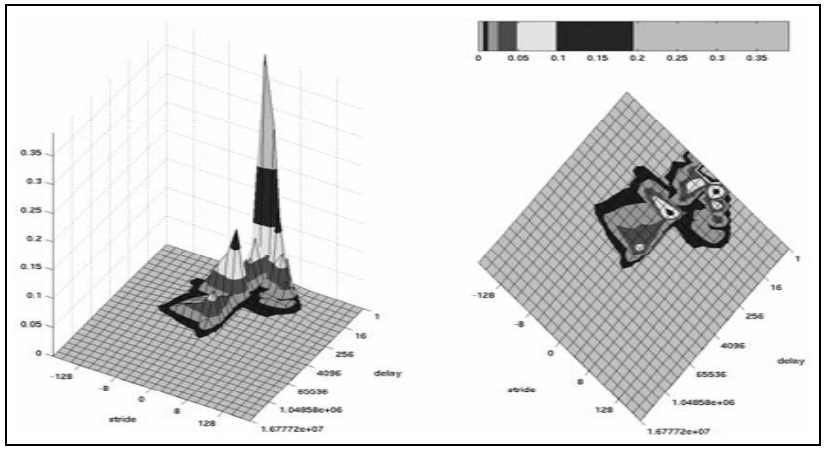
Data trace of *perlbmk.makerand* under Windows 2000.



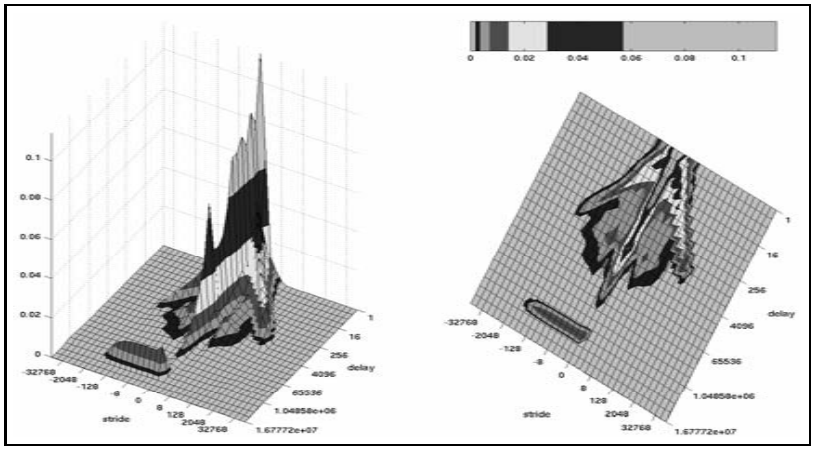
Instruction trace of *perlbmk.perfect* under Windows 2000.



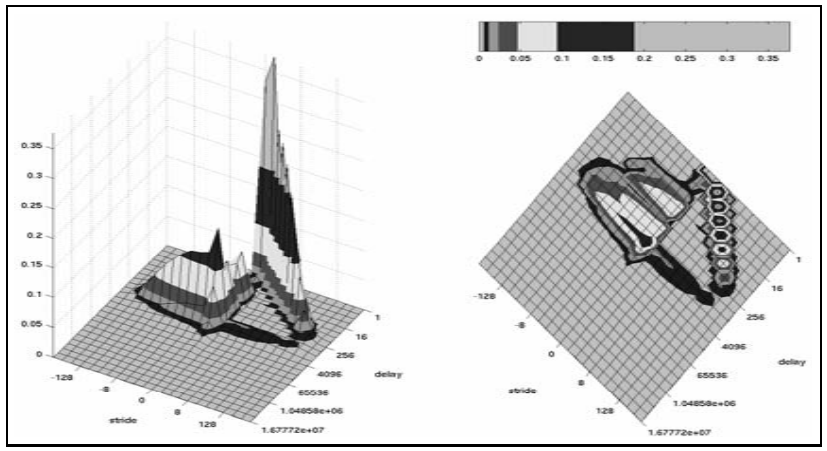
Data trace of *perlbmk.perfect* under Windows 2000.



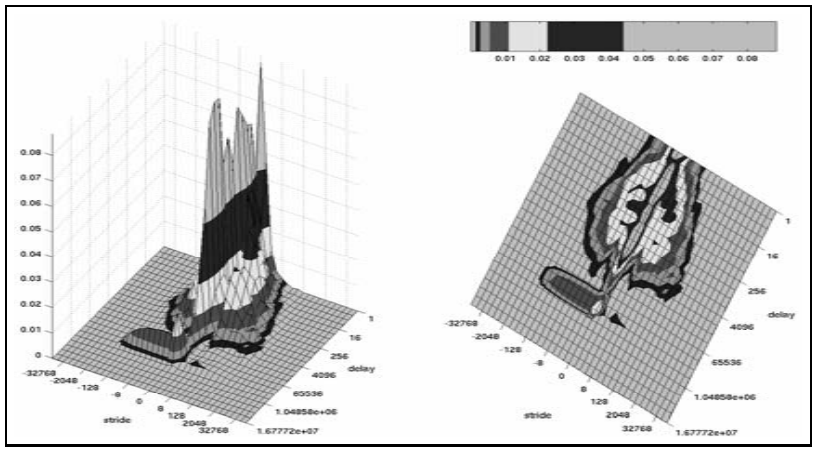
Instruction trace of *perlbmk.splitmail* under Windows 2000.



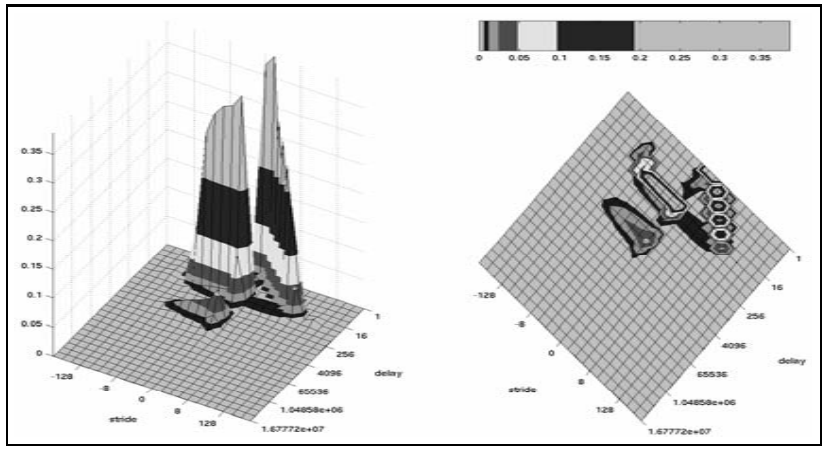
Data trace of *perlbmk.splitmail* under Windows 2000.



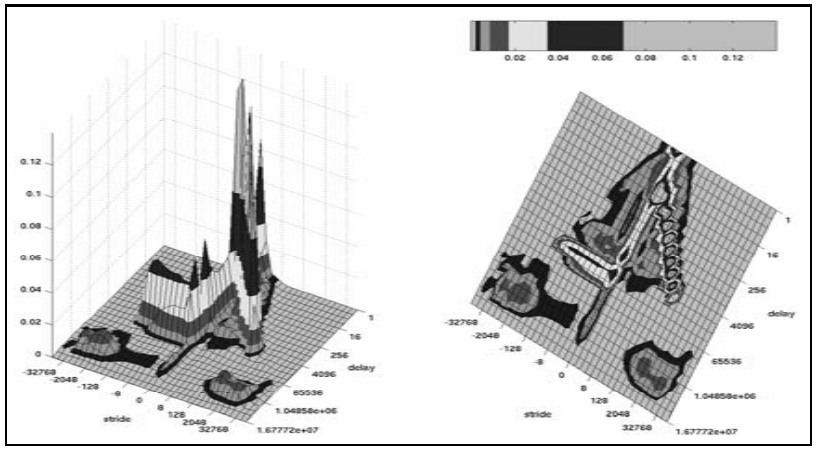
Instruction trace of *sixtrack* under Windows 2000.



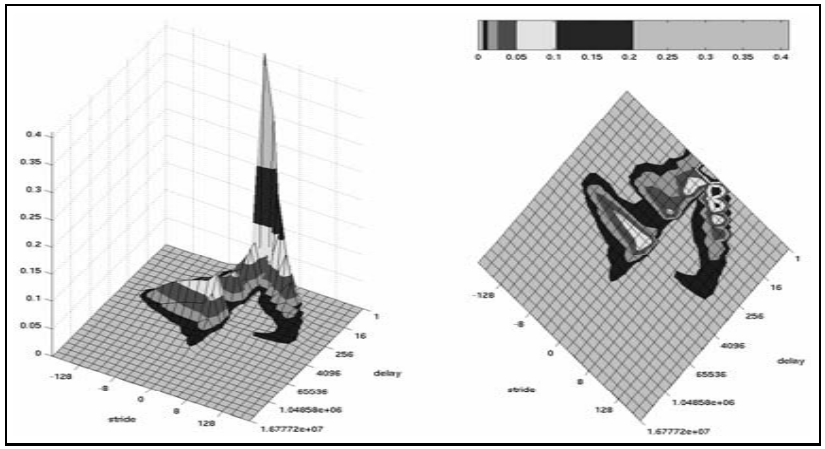
Data trace of *sixtrack* under Windows 2000.



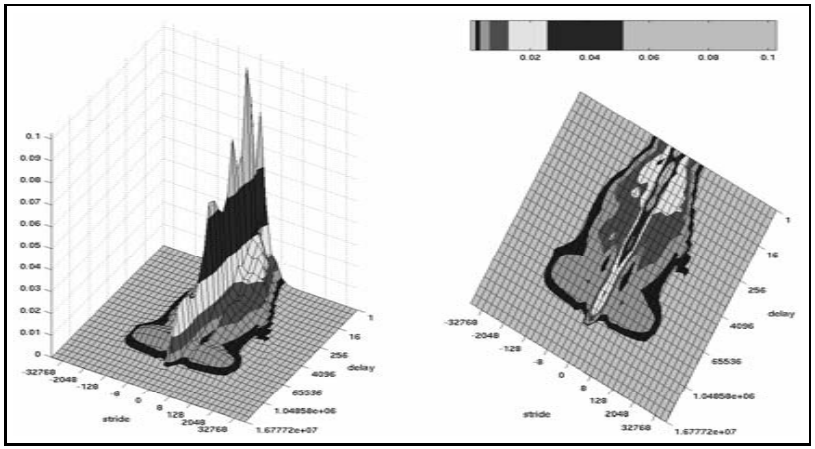
Instruction trace of *swim* under Windows 2000.



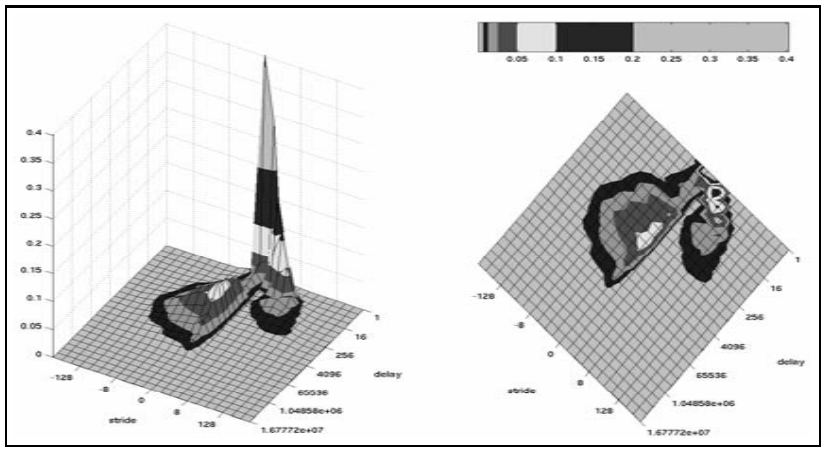
Data trace of *swim* under Windows 2000.



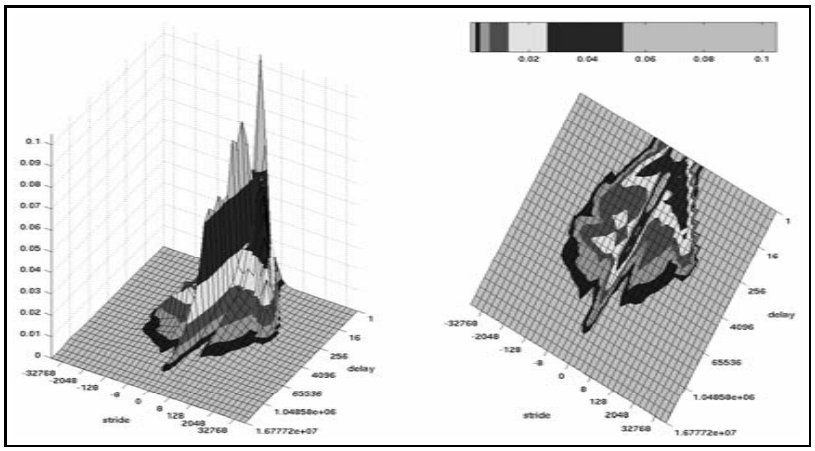
Instruction trace of *twolf* under Windows 2000.



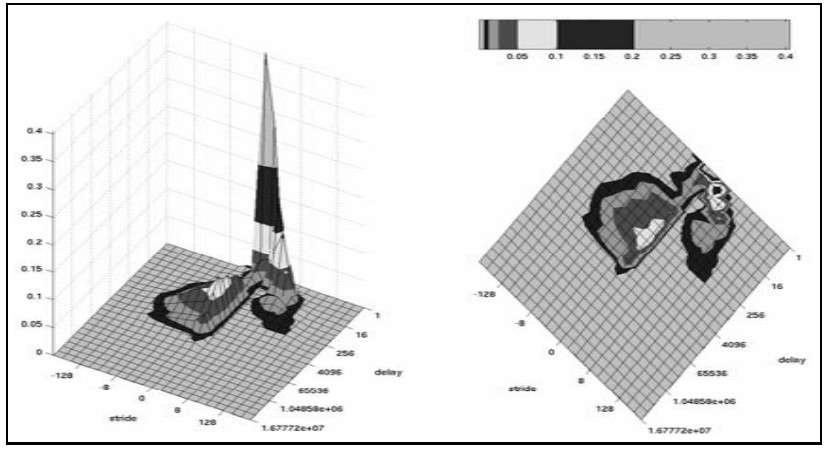
Data trace of *twolf* under Windows 2000.



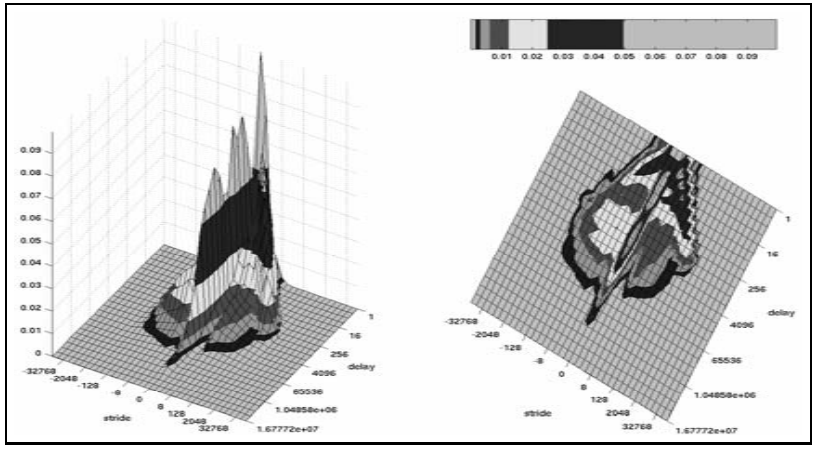
Instruction trace of *vortex.one* under Windows 2000.



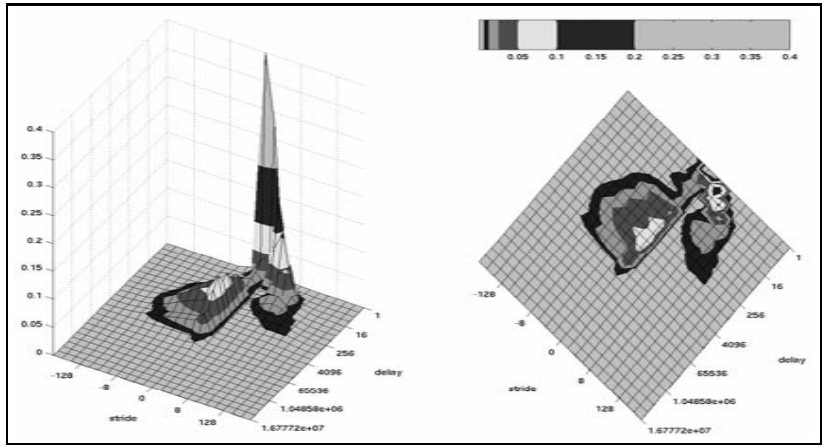
Data trace of *vortex.one* under Windows 2000.



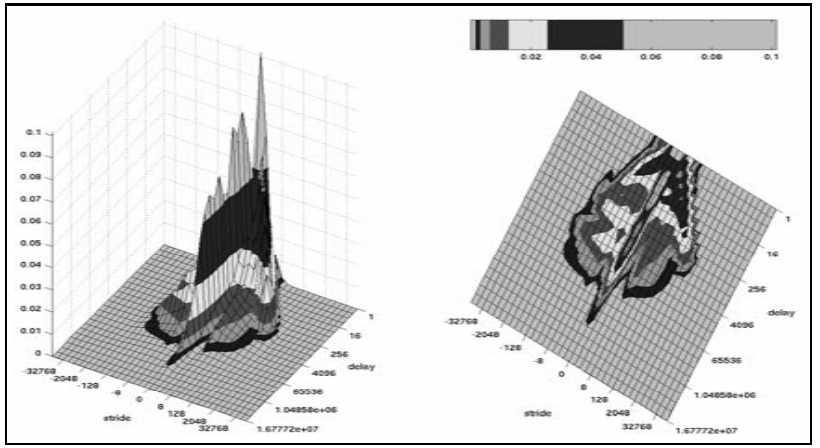
Instruction trace of *vortex.three* under Windows 2000.



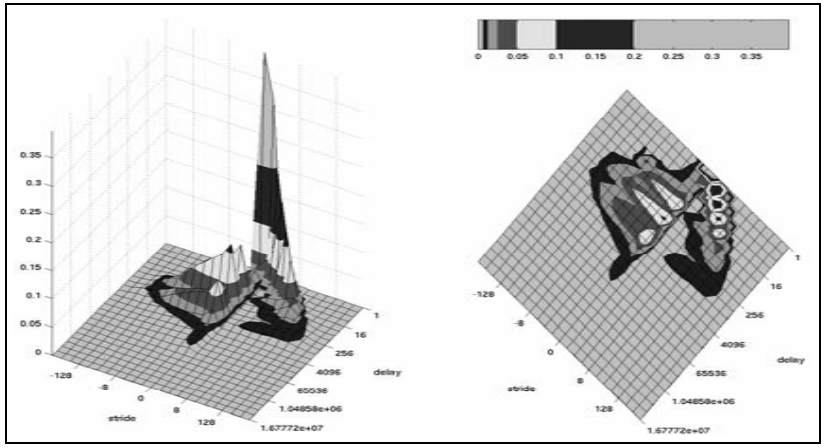
Data trace of *vortex.three* under Windows 2000.



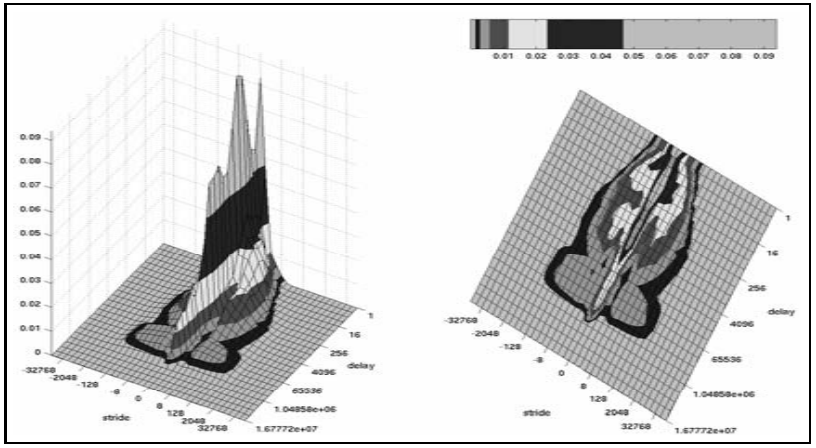
Instruction trace of *vortex.two* under Windows 2000.



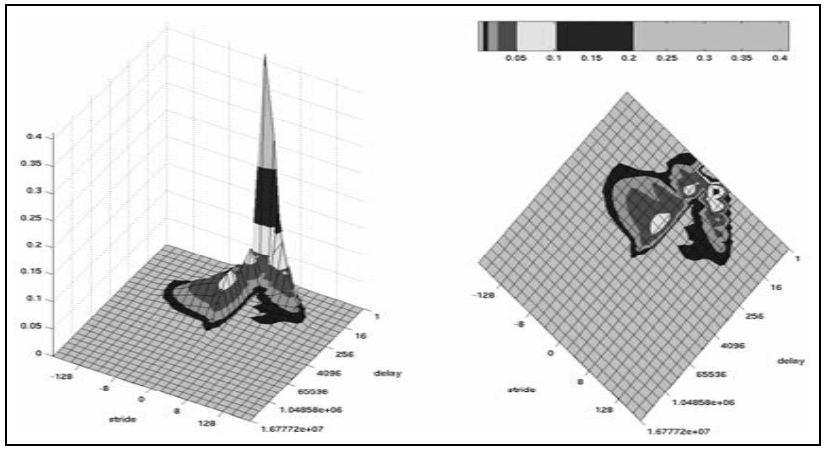
Data trace of *vortex.two* under Windows 2000.



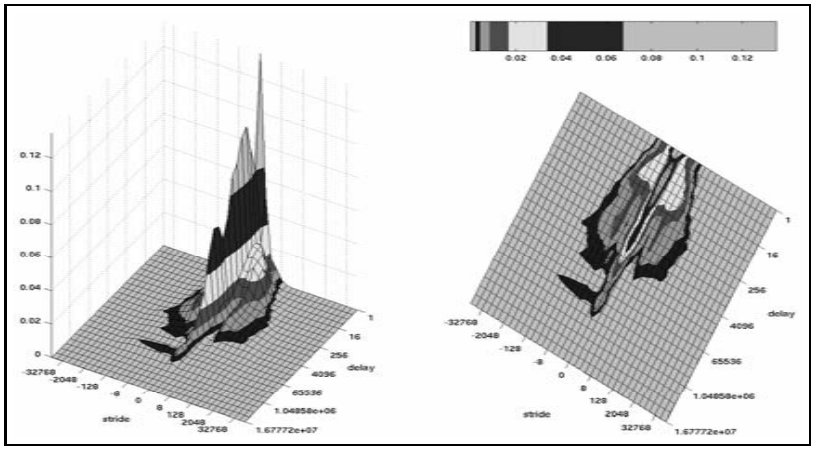
Instruction trace of *vpr.place* under Windows 2000.



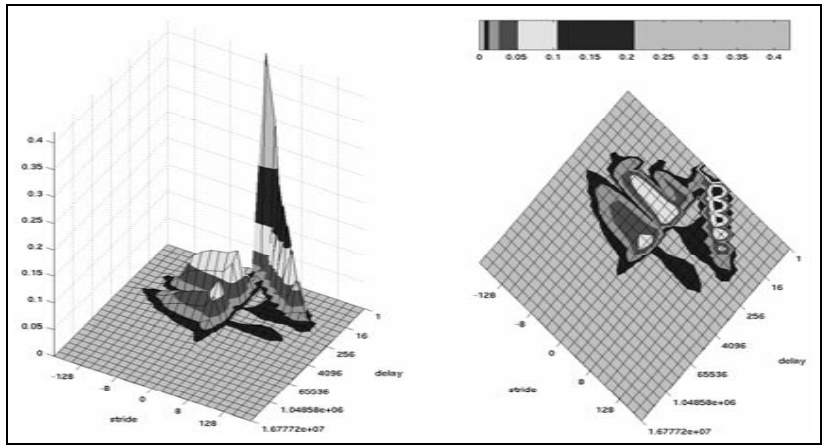
Data trace of *vpr.place* under Windows 2000.



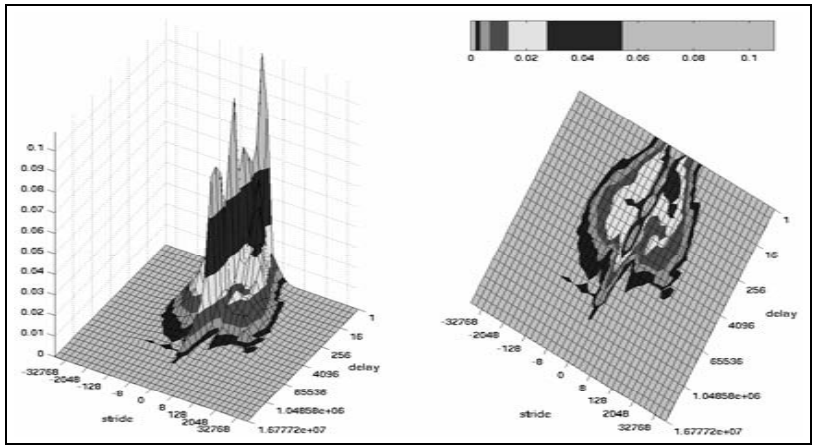
Instruction trace of *vpr.route* under Windows 2000.



Data trace of *vpr.route* under Windows 2000.



Instruction trace of *wupwise* under Windows 2000.



Data trace of *wupwise* under Windows 2000.

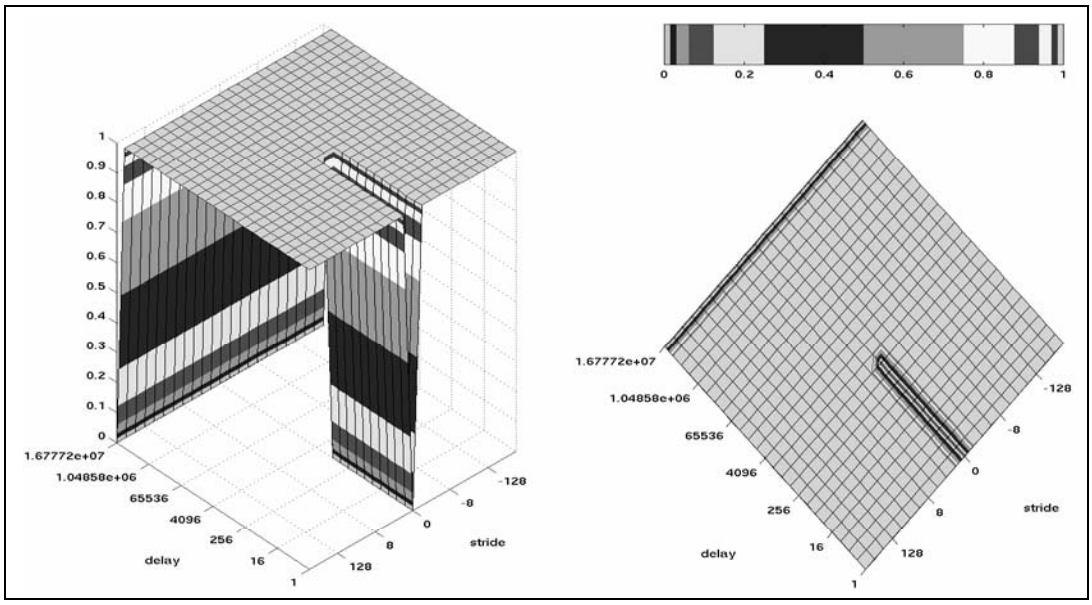
Appendix C

Cache Characterization Surfaces

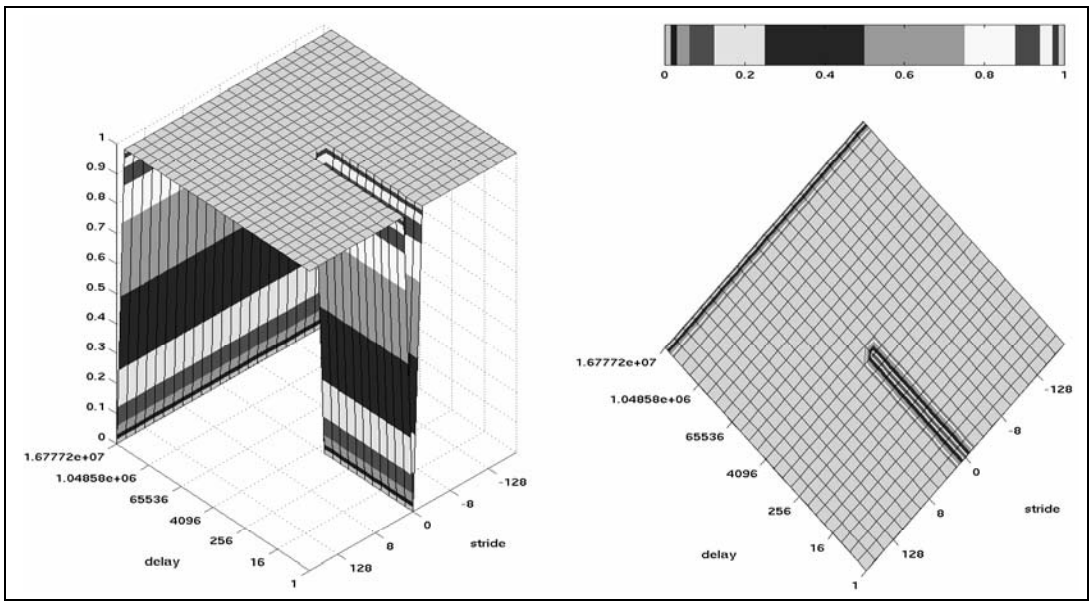
This appendix contains all the cache characterization surfaces created for this dissertion. They were created using the method described in Section 5.3 and using the parallel multiple cache characterization surface algorithm outlined in Figure 9.11.

The surfaces are ordered by cache case. (See Section 5.1 for a description of each of the cache cases.) First are the Case One caches, ordered by cache size. Next are the Case Two caches, ordered primarily by line size. Next are the Case Three caches, ordered by associativity. Lastly are the Case Four caches, also ordered by associativity.

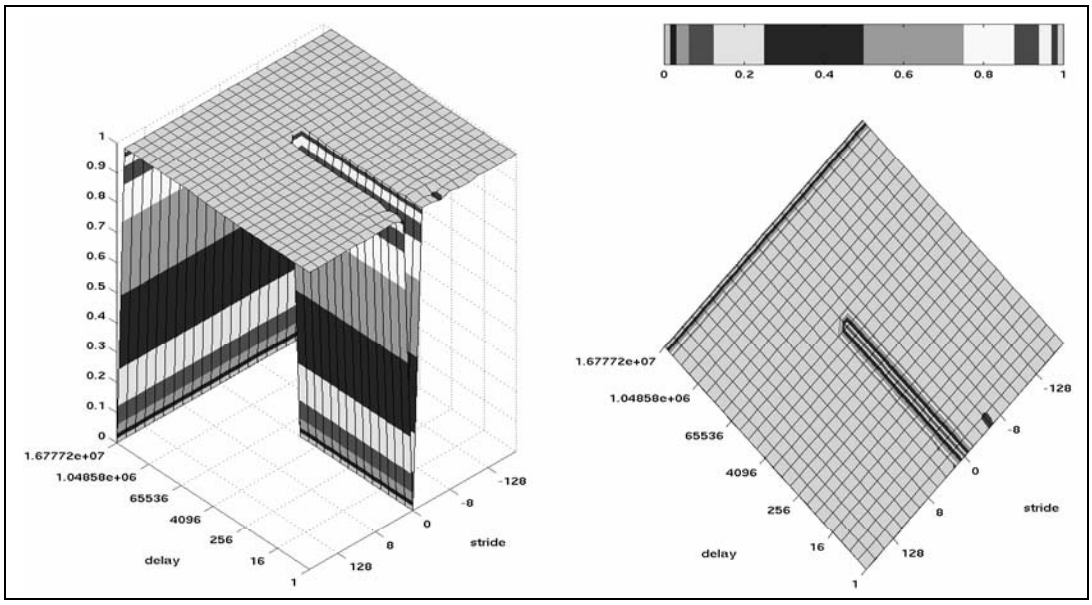
As discussed in Section 5.3.2, the cache size and line size for these surfaces are actually relative to the granularity. For example, the first cache characterization surface in this appendix is labeled as a surface for an 8 Kbyte fully associative cache with 8 byte lines. In reality, it is a cache characterization surface for a cache where the cache line size matches the input trace granularity and the cache size is 1024 times the granularity. All of these surfaces are labeled as if the input trace has an 8-byte granularity.



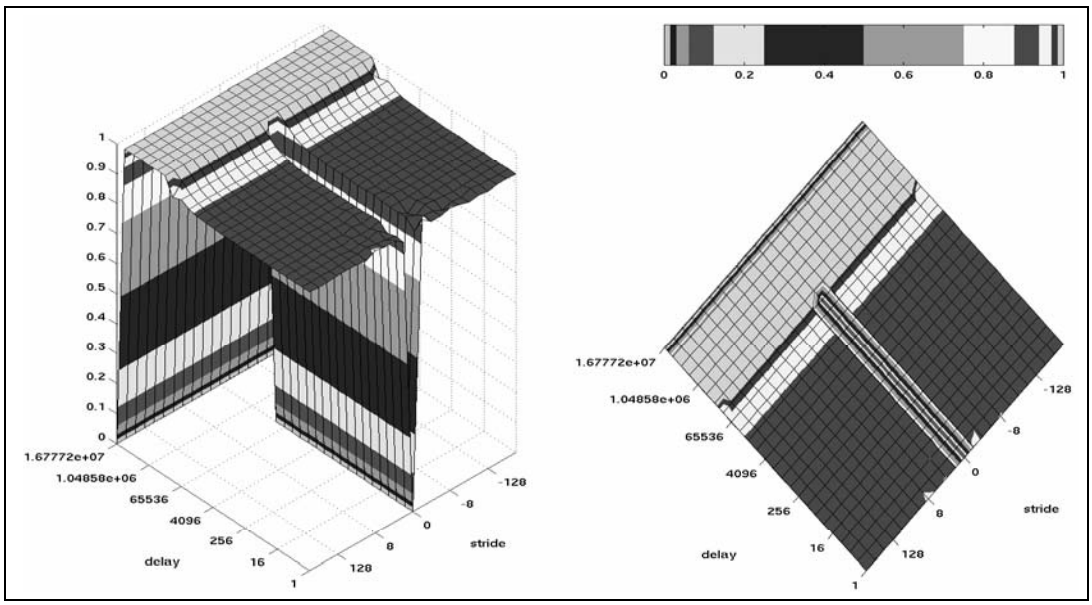
A 8 Kbyte fully associative cache with 8-byte lines.



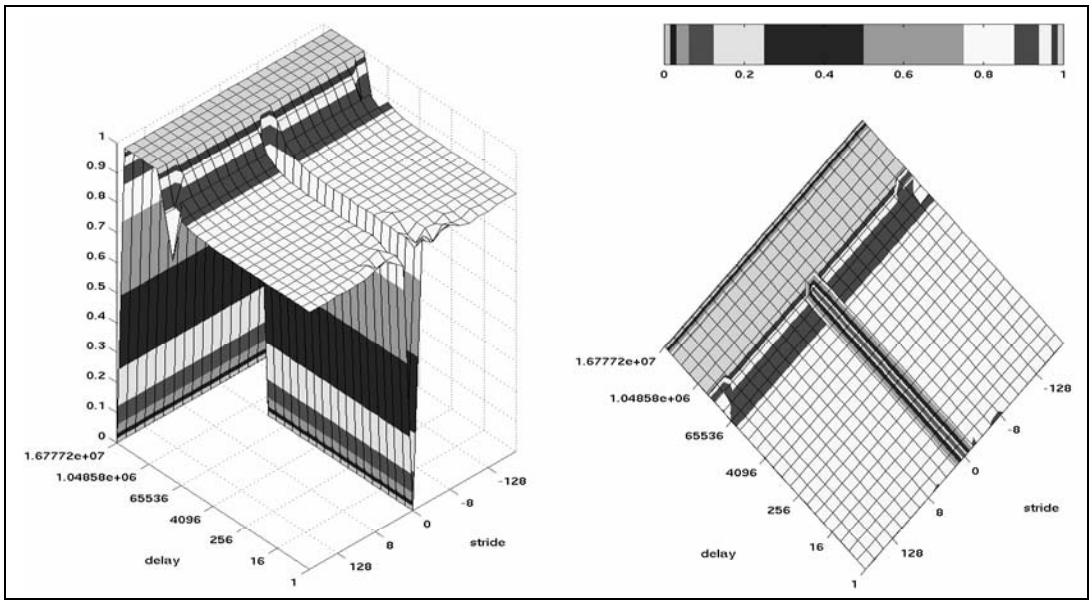
A 16 Kbyte fully associative cache with 8-byte lines.



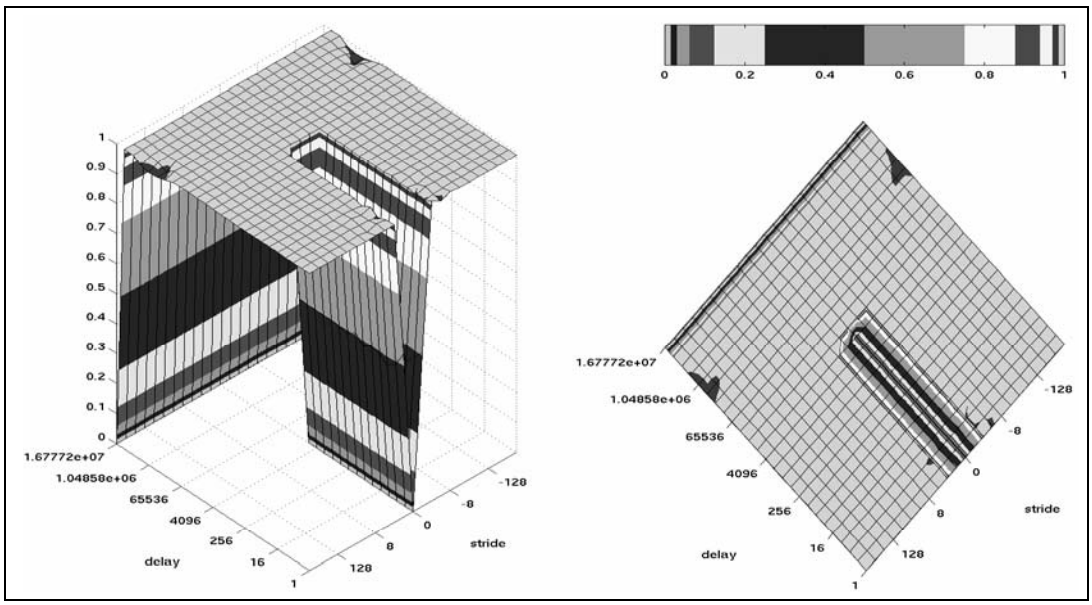
A 128 Kbyte fully associative cache with 8-byte lines.



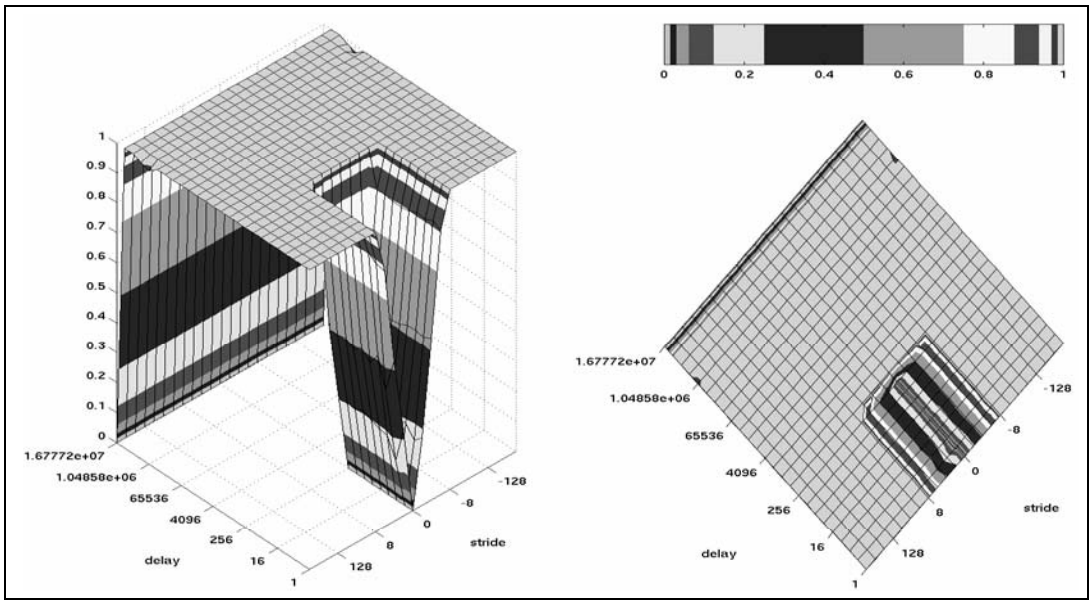
A 1 Mbyte fully associative cache with 8-byte lines.



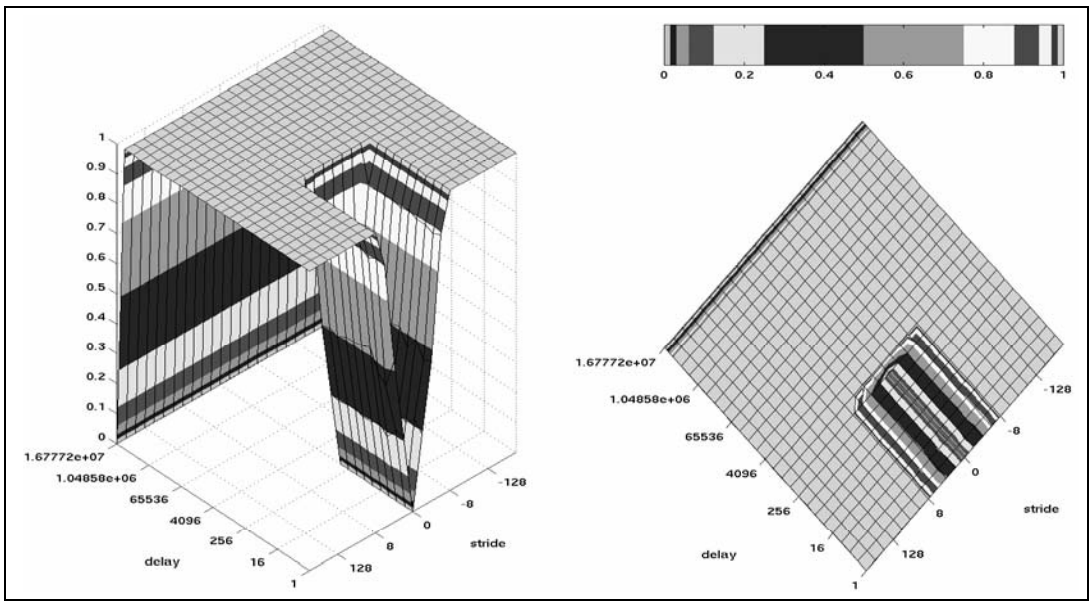
A 2 Mbyte fully associative cache with 8-byte lines.



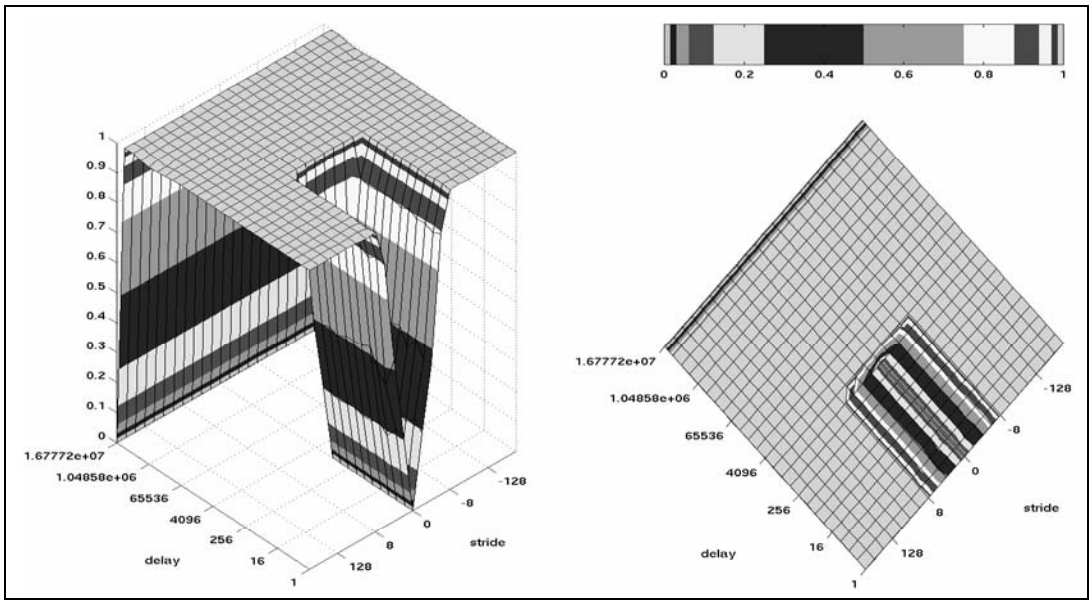
A 128 Kbyte fully associative cache with 16-byte lines.



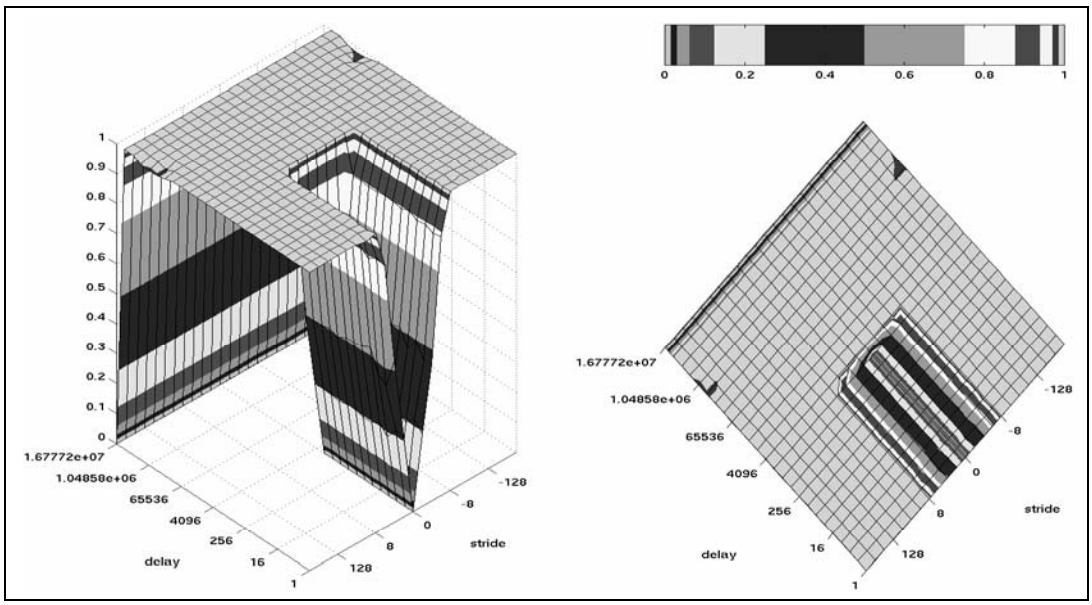
A 8 Kbyte fully associative cache with 32-byte lines.



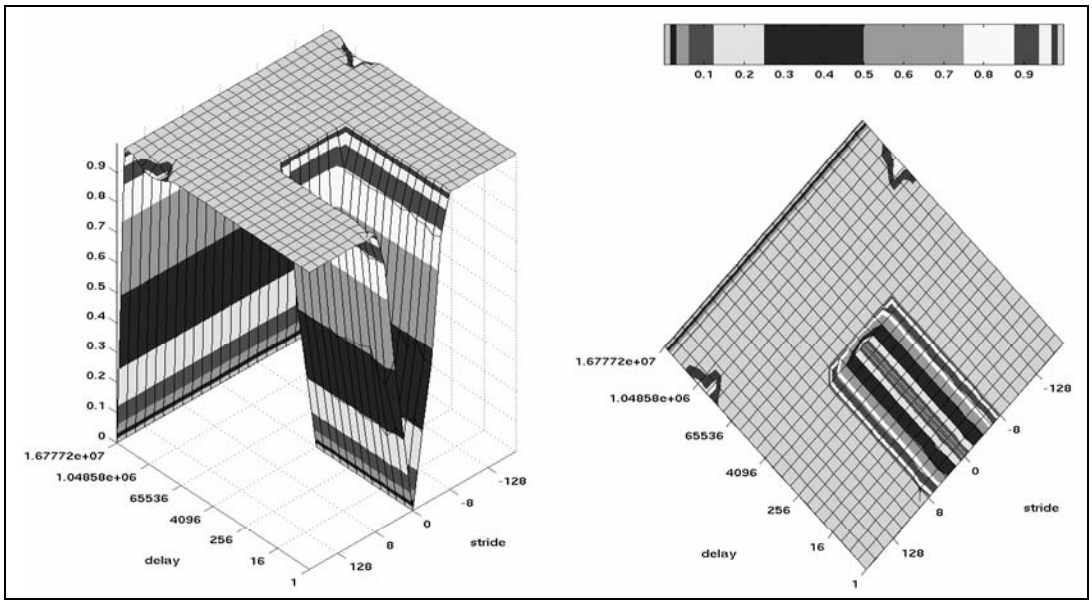
A 16 Kbyte fully associative cache with 32-byte lines.



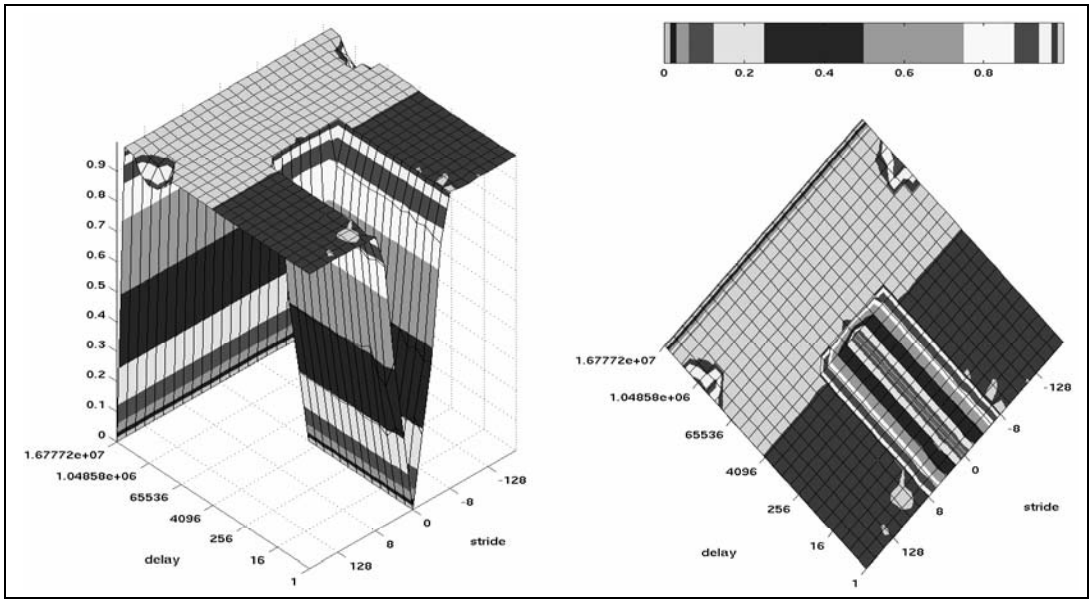
A 32 Kbyte fully associative cache with 32-byte lines.



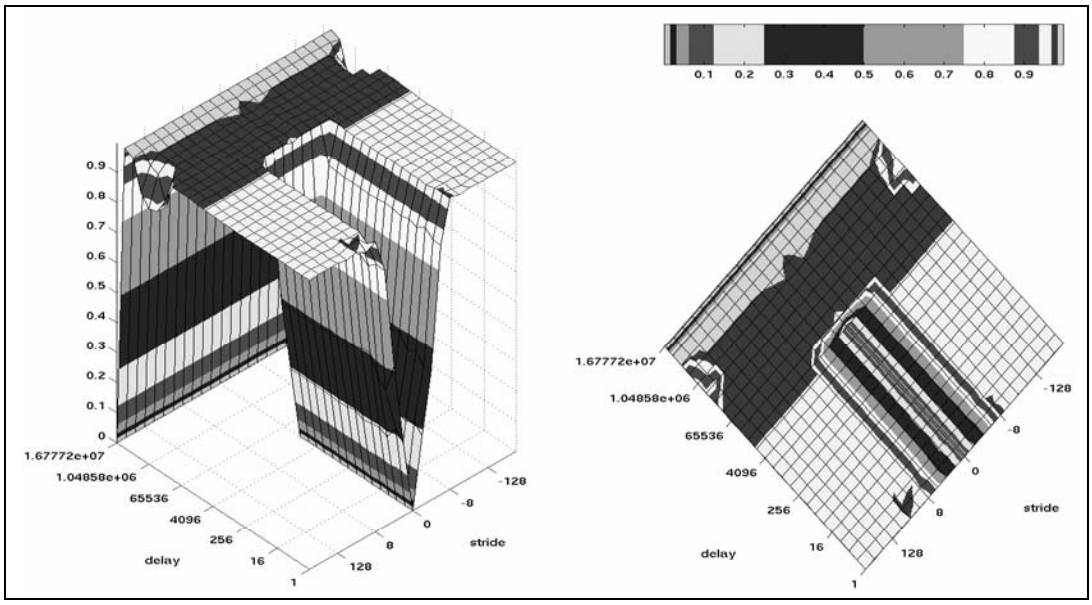
A 64 Kbyte fully associative cache with 32-byte lines.



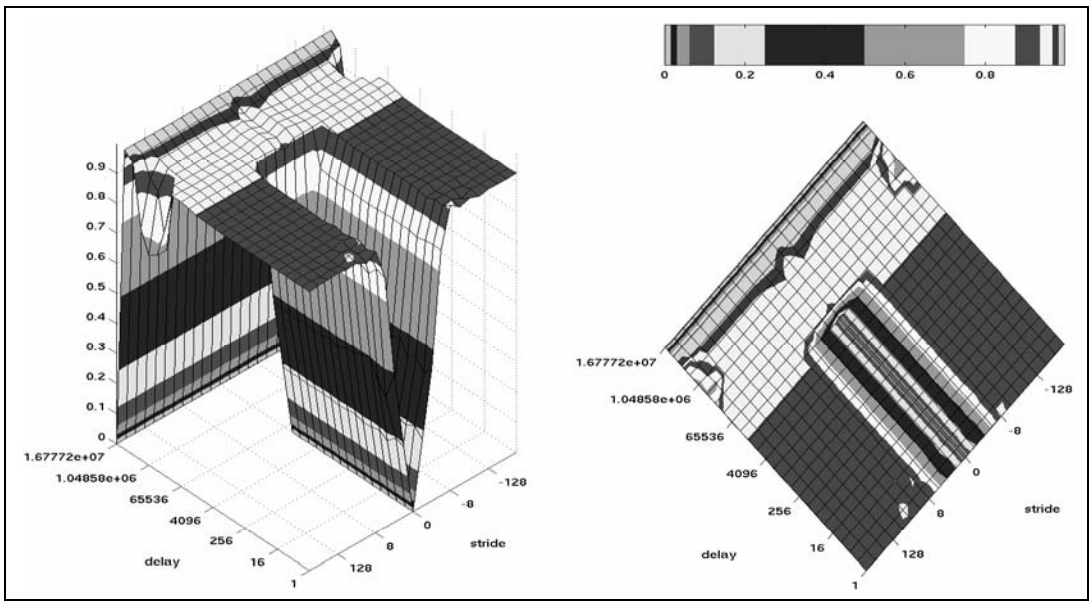
A 128 Kbyte fully associative cache with 32-byte lines.



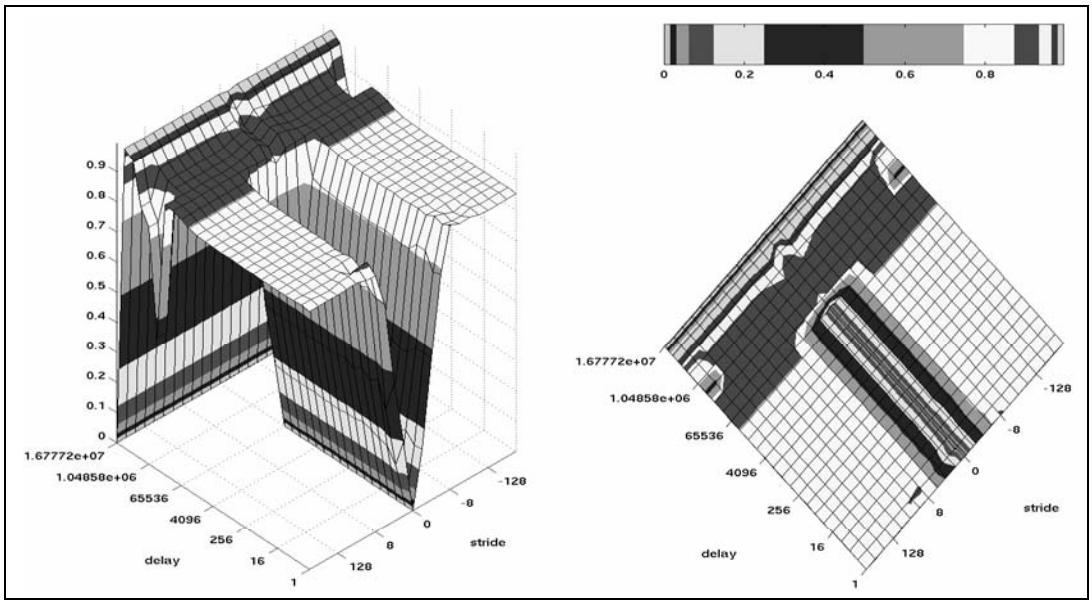
A 256 Kbyte fully associative cache with 32-byte lines.



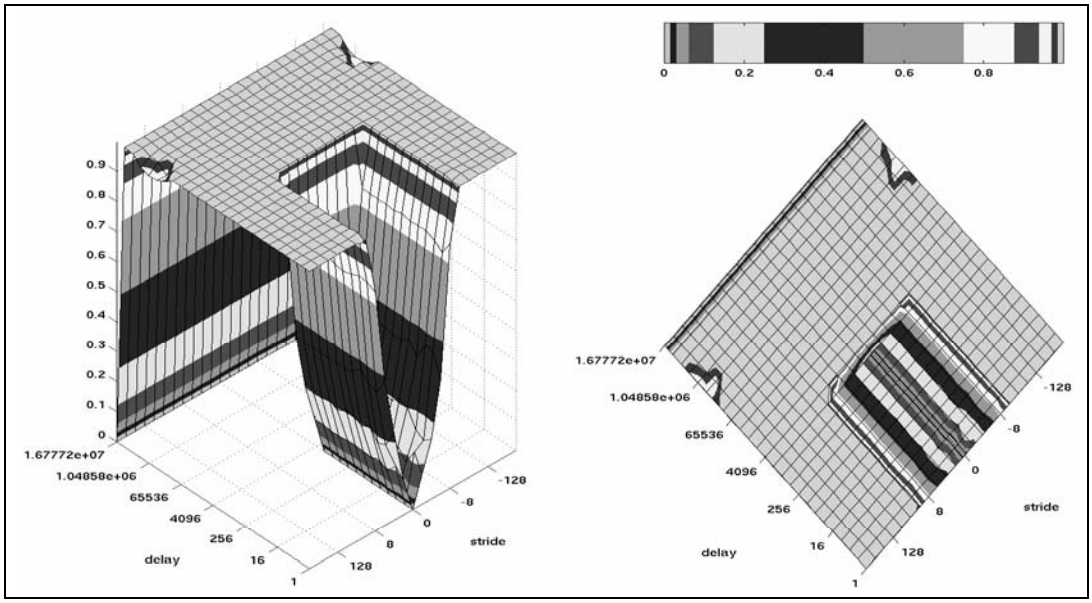
A 512 Kbyte fully associative cache with 32-byte lines.



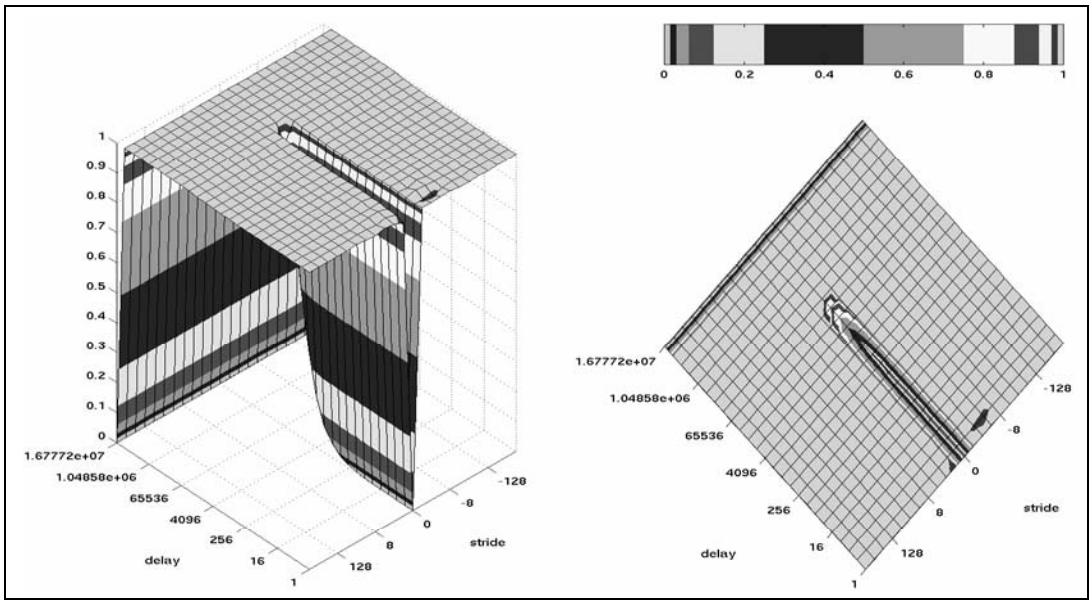
A 1 Mbyte fully associative cache with 32-byte lines.



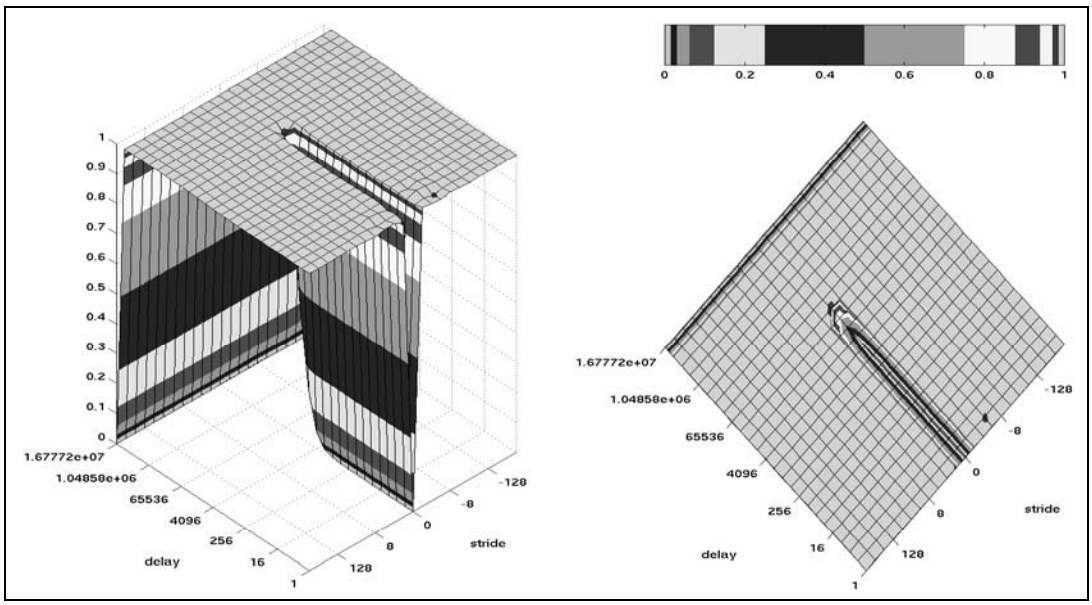
A 2 Mbyte fully associative cache with 32-byte lines.



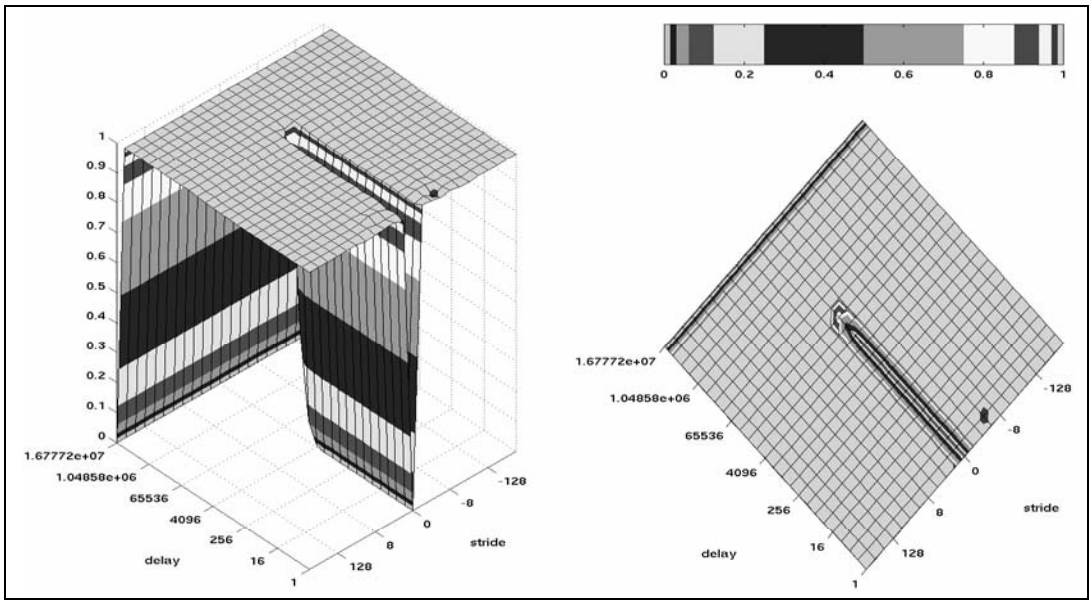
A 128 Kbyte fully associative cache with 64-byte lines.



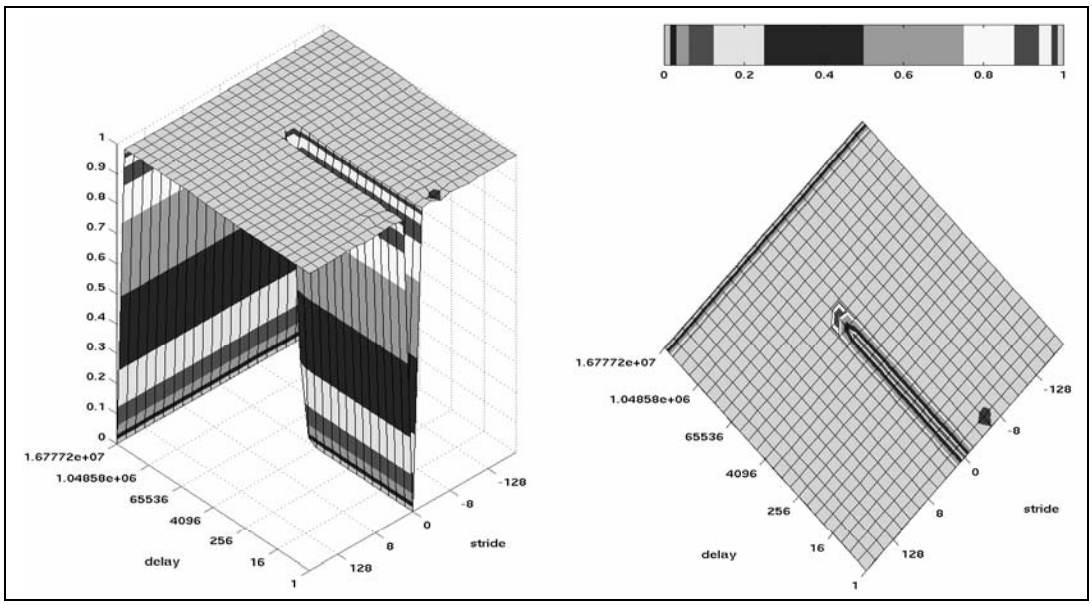
A 128 Kbyte direct mapped cache with 8-byte lines.



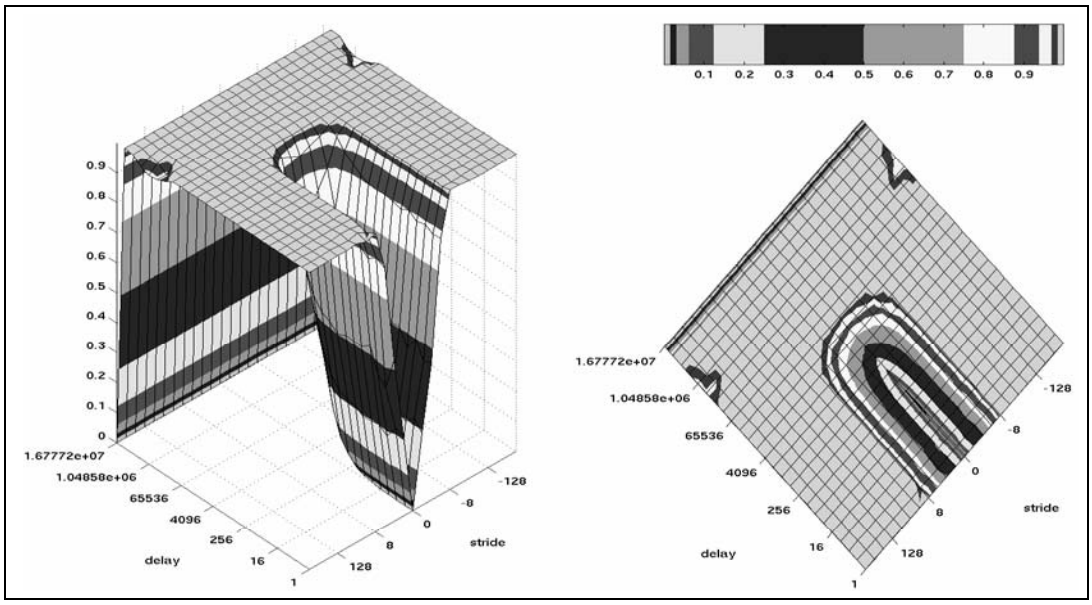
A 128 Kbyte 2-way associative cache with 8-byte lines.



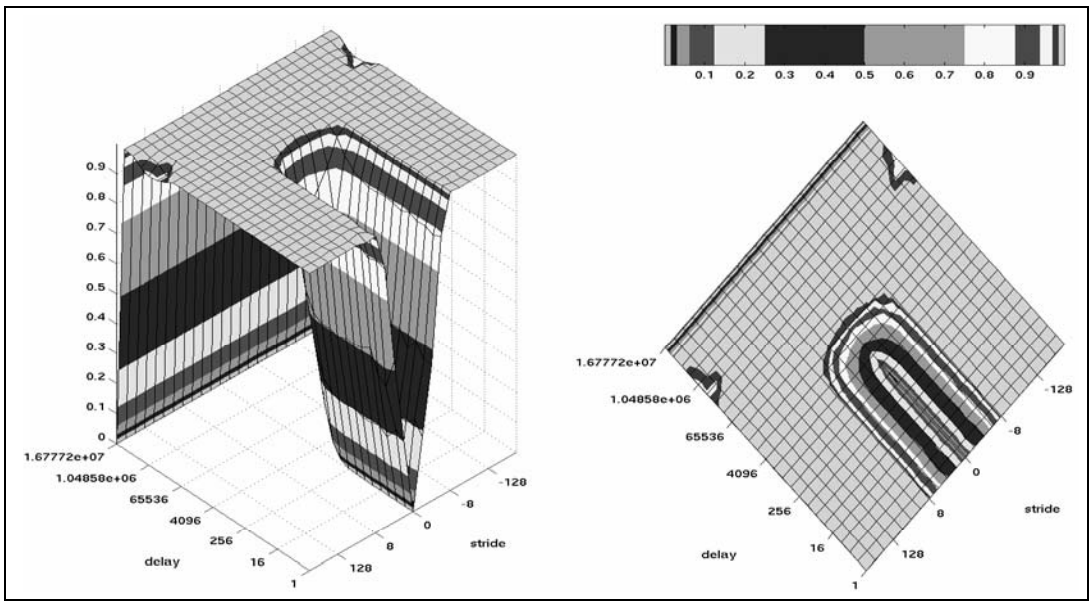
A 128 Kbyte 4-way associative cache with 8-byte lines.



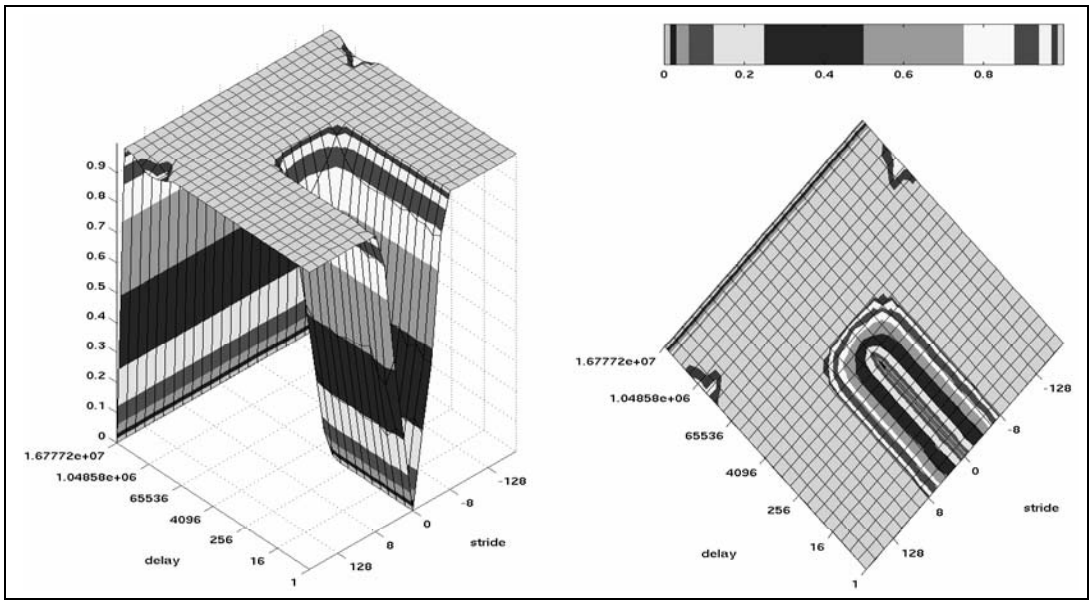
A 128 Kbyte 8-way associative cache with 8-byte lines.



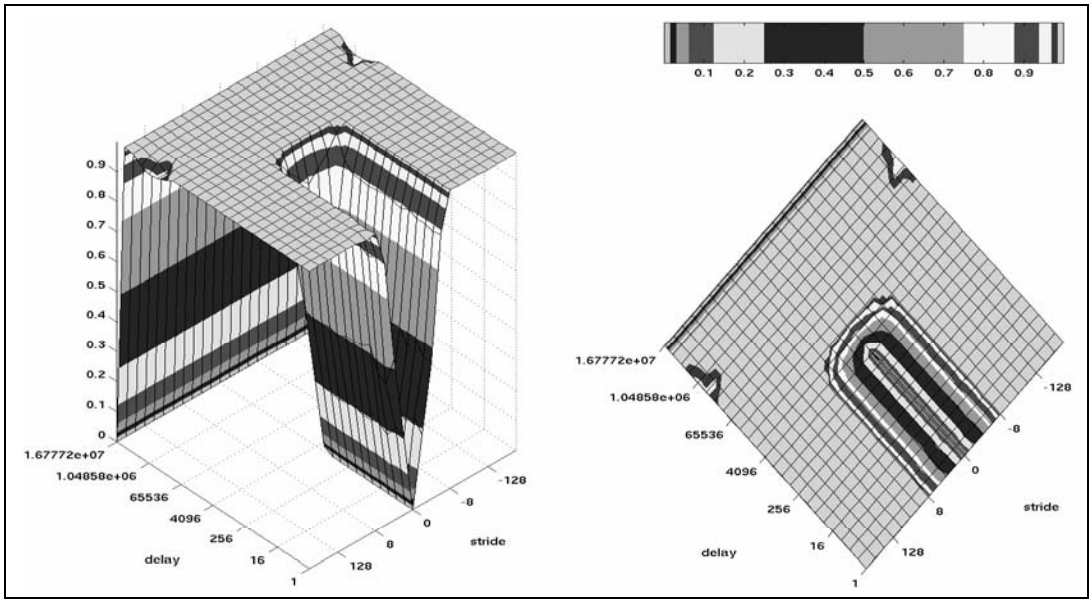
A 128 Kbyte direct mapped cache with 32-byte lines.



A 128 Kbyte 2-way associative cache with 32-byte lines.



A 128 Kbyte 4-way associative cache with 32-byte lines.



A 128 Kbyte 8-way associative cache with 32-byte lines.

Bibliography

[1] BYU trace repository. http://traces.byu.edu, 2002.

[2] Processors. http://www.intel.com/support/processors/sb/cs-001840-prd24.htm, 2002.

[3] Standard performance evaluation corp. http://www.spec.org/cpu2000/, 2004.

[4] A. Agarwal and A. Gupta. Temporal, processor and spatial locality in multiprocessor memory references. In S. K. Tewksbury, editor, *Frontiers in Computing Systems Research*, volume 1. Plenum Press, New York, NY, 1990.

[5] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215, May 1989.

[6] A. Agarwal and S. D. Pudar. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 179–190. ACM Press, 1993.

[7] A. Agarwal, R. L. Sites, and M. Horowits. ATUM: A new technique for capturing address traces using microcode. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 119–127. IEEE, 1986.

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

[9] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In *ACM-SE 42: Proceedings of the 42nd Annual Southeast Regional Conference*, pages 267–272, New York, NY, 2004. ACM Press.

[10] J. Albert. Albegraic properties of bag data types. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 211–219, September 1991.

[11] G. Almasi, C. Cascaval, and D. A. Padua. Calculating stack distances efficiently. In *Proceedings of the Workshop on Memory System Performance*, pages 37–43. ACM Press, 2002.

[12] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, pages 92–103, December 1996.

[13] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[14] O. Aven, E. Coffman, and Y. Kogan. *Stochastic Analysis of Computer Storage*. Reidel, Amsterdam, 1987.

[15] E. Berg and E. Hagersten. Statcache: A probabilistic approach to efficient and accurate data locality analysis. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2004.

[16] K. Beyls and E. H. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of PDCS'01*, August 2001.

[17] A. Borg, R. E. Kessler, and D. W. Wall. Generation and analysis of very long address traces. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 270–279, May 1990.

[18] R. E. Bryant and D. O'Hallaron. *Computer Systems: A Programmer's Perspective*, section 6.6.1, pages 512–516. Prentice Hall, Upper Saddle River, NJ, 2003.

[19] C. Cascaval and D. A. Padua. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 150–159. ACM Press, 2003.

[20] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 199–209. ACM Press, 2002.

[21] S. Cho, P.-C. Yew, and G. Lee. Access region locality for high-bandwidth processor memory system design. In *Proceedings of the 32nd International Symposium on Microarchitecture*, Haifa, Israel, November 1999.

[22] E. G. Coffman and P. J. Denning. *Operating Systems Theory*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973.

[23] W. J. Collins. *Data Structures*, chapter 8, pages 313–360. Addison-Wesley, Reading, MA, 1992.

[24] T. M. Conte and W. W. Hwu. Benchmark characterization for experimental system evaluation. In *Proceedings of the 1990 Hawaii International Conference on System Sciences (HICSS)*, volume I of *Architecture Track*, pages 6–18, 1990.

[25] J. Cook, R. L. Oliver, and E. E. Johnson. Examining performance differences in workload execution phases. In *Proceedings of the Fourth IEEE Annual Workshop on Workload Characterization*, pages 82–90, December 2001.

[26] N. E. Crosbie, M. Kandemir, I. Kolcu, J. Ramanujam, and A. Choudhary. Strategies for improving data locality in embedded applications. In *ASP-DAC '02: Proceedings of the 2002 Conference on Asia South Pacific Design Automation/VLSI Design*, page 631, Washington, DC, 2002. IEEE Computer Society.

[27] U. Dayal, N. Goodman, and R. H. Katz. An extended relational algebra with control over duplicate elimination. In *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 117–123. ACM Press, 1982.

[28] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[29] P. J. Denning and K. C. Kahn. A study of program locality and lifetime functions. In *Proceedings of the Fifth Symposium on Operating System Principles*, pages 207–216, 1975.

[30] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Pro-*

gramming Language Design and Implementation, pages 245–257. ACM Press, 2003.

[31] C. Dyreson. The relational algebra. Available: http://www.cs.jcu.edu.au/ftp/web/teaching/Subjects/cs3020/1998/foils/RMalgebra.html [accessed 8 January 2004], 1998.

[32] A. Fiat and A. R. Karlin. Randomized and multipointer paging with locality of reference. In *Proceedings of the Twenty-Seventh Annual ACM Symposium of Theory of Computing*, pages 626–634. ACM Press, 1995.

[33] J. K. Flanagan, B. Nelson, J. Archibald, and K. Grimsrud. BACH: BYU address collection hardware; the collection of complete traces. In *Proceedings of the 6th International Conference On Modeling Techniques and Tools for Computer Performance Evaluation*, September 1992.

[34] C. Fricker and P. Robert. A memory reference model for the analysis of cache memories. *Performance '90*, pages 255–269, 1990.

[35] J. L. Gersting. *Mathematical Structures for Computer Science*. W. H Freeman and Company, New York, NY, fifth edition, 2003.

[36] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239. ACM Press, October 1998.

[37] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[38] K. Grimsrud. *Visualizing Locality.* PhD thesis, Brigham Young University, 1993.

[39] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson. On the accuracy of memory reference models. In *7th International Conference Proceedings*, pages 369–388, May 1994.

[40] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson. Locality as a visualization tool. *IEEE Transactions On Computers*, 45(11), November 1996.

[41] K. Grimsrud, J. Archibald, M. Ripley, K. Flanagan, and B. Nelson. BACH: A hardware monitor for tracing microprocessor-based systems. *Microporcessors and Microsystems*, 17(6):443–459, October 1993.

[42] S. Grumbach and T. Milo. Towards tractable algebras for bags. *Journal of Computer and System Sciences*, 52(3):570–588, 1996.

[43] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Fransisco, CA, third edition, 2003.

[44] M. Hill. A case for direct-mapped caches. *IEEE Computer Magazine*, 21(12):25–40, December 1988.

[45] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[46] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*, chapter 2, pages 48–97. Computer Science Press, 1978.

[47] A. S. Huang and J. P. Shen. A limit study of local memory requirements using value reuse profiles. In *Proceedings of MICRO-28*, 1995.

[48] T. Inagaki, T. Onodera, H. Komatsu, and T. Nakatani. Stride prefetching by dynamically inspecting objects. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 269–277. ACM Press, 2003.

[49] B. L. Jacob, P. M. Chen, S. R. Silverman, and T. N. Mudge. An analytical model for designing memory hierarchies. *IEEE Transactions on Computers*, 45(10), October 1996.

[50] L. K. John, P. Vasudevan, and J. Sabarinathan. Workload characterization: Motivation, goals and methodology. In *Workload Characterization: Methodology and Case Studies*, Dallas, TX, November 1998.

[51] E. E. Johnson, J. Ha, and M. B. Zaidi. Lossless trace compression. *IEEE Transactions on Computers*, 50(2):158–173, 2001.

[52] K. L. Johnson. The impact of communication locality on large-scale multiprocessor performance. In *ISCA '92: Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 392–402, New York, NY, 1992. ACM Press.

[53] M. Kampe and F. Dahlgren. Exploration of the spatial locality on emerging applications and the consequences for cache performance. In *Proceedings of the Parallel and Distributed Processing Symposium*, pages 163–170, May 2000.

[54] M. T. Kandemir. A compiler technique for improving whole-program locality. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–192, New York, NY, 2001. ACM Press.

[55] W. Kent. Profile functions and bag theory. Technical Report HPL-SAL-89-19, Hewlett-Packard Laboratories, January 1989.

[56] A. Klausner and N. Goodman. Multirelations–semantics and languages. In *Proceedings of the 11th International Conference on Very Large Databases*, pages 251–258, 1985.

[57] H. Kuchen and K. Gladitz. Parallel implementation of bags. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 299–307. ACM Press, 1993.

[58] M. H. Lipasti, C. B. Wilerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems VII*, October 1996.

[59] M. E. S. Loomis. *Data Management and File Structures.* Prentice Hall, Englewood Cliffs, NJ, second edition, 1989.

[60] Y. Luo and L. K. John. Locality-based online trace compression. *IEEE Transactions on Computers*, 53(6):723–731, June 2004.

[61] A. Mahanti, D. Eager, and C. Williamson. Temporal locality and its impact on web proxy cache performance. *Performance Evaluation, Special Issue on Internet Performance Modeling*, 42(2/3):187–203, September 2000.

[62] K. S. McKinley and O. Temam. Quantifying loop nest locality using SPEC'95 and the perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4):288–336, November 1999.

[63] D. Nagle, R. Uhlig, and T. Mudge. Monster: A tool for analyzing the interaction between operating systems and computer architectures. Technical report, The University of Michigan, 1992.

[64] S. Ramanathan, R. Srinivasan, and J. Cook. Intristic data locality of modern scientific workloads. In *Proceedings of the Sixth IEEE Annual Workshop on Workload Characterization*, pages 77–85, October 2003.

[65] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patters. In *Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 148–157, March 2005.

[66] M. Salsburg. An analytical method for evaluating the performance of cache using the LRU process. *Computer Measurement Group Transactions*, pages 77–87, Winter 1994.

[67] F. J. Sanchez and A. Gonzalez. Data locality analysis of the SPECfp95. *Digest of Performance Analysis and its Impact on Design (PAID) Workshop*, pages 78–84, 1998.

[68] A. Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual Symposium on Computer Architecture*, pages 169–178, May 1993.

[69] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 165–176. ACM Press, 2004.

[70] J. P. Singh, H. S. Stone, and D. F. Thiebaut. A model of workloads and its use in miss-rate prediction for fully associative caches. *IEEE Transactions on Computers*, 41(7):811–825, July 1992.

[71] M. Sipser. *Introduction to the Theory of Computation*. PWD Publishing Company, 1997.

[72] A. J. Smith. A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Transactions on Software Engineering*, SE-4:121–130, March 1978.

[73] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

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

[75] J. Spirn. *Program Behavior: Models and Measurements*. Elsevier North-Holland, Inc., New York, NY, 1977.

[76] T. Stricker and T. Gross. Global address space, non-uniform bandwidth: A memory system performance characterization of parallel systems. In *Proceedings of the Third International Symposium on High Performance Computer Architecture*, February 1997.

[77] T. A. Sudkamp. *Languages and Machines*. Addison-Wesley, second edition, 1997.

[78] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of*

the ACM SIGMETRICS Conference on Measurement and Modeling Computer Systems, pages 24–35, May 1993.

[79] D. Thiebaut, J. L. Wolf, and H. S. Stone. Synthetic traces for trace-driven simulation of cache memories. *IEEE Transactions on Computers*, 41(4):388–410, April 1992.

[80] J. G. Thompson and A. J. Smith. Efficient stack algorithms for analysis of write-back and sector memories. *ACM Transactions on Computer Systems*, 7(1):78–117, February 1989.

[81] N. C. Thornock and J. K. Flanagan. A national trace collection and distribution resource. *ACM SIGARCH Computer Architecture News*, 29(3):6–10, 2001.

[82] D. N. Truong, F. Bodin, and A. Seznec. Accurate data distribution into blocks may boost cache performance. Technical Report RR-3174, 1997.

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

[84] unknown author. Core relational algebra. Available: http://www-db.stanford.edu/~ullman/fcdb/slides/slides5.pdf [accessed 3 March 2004], unknown year.

[85] M. G. Watson and J. K. Flanagan. Does halting make trace collection inaccurate? A case study using Pentium 4 performance counters and SPEC2000. In *Proceedings of the Seventh IEEE Annual Workshop on Workload Characterization*, October 2004.

[86] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Ontario, Canada, June 1991.

[87] Y. Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 210–221. ACM Press, June 2002.

[88] 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*, March 2002.

[89] Y. Zhong, S. G. Dropsho, and C. Ding. Miss rate prediction across all program inputs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 79–90, 2003.

[90] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 255–266. ACM Press, 2004.