Using Locality Surfaces to Characterize the SPECint 2000 Benchmark Suite

Elizabeth S. Sorenson
leb@pel.cs.byu.edu

J. Kelly Flanagan
kelly@cs.byu.edu

Performance Evaluation Laboratory
Computer Science Department
Brigham Young University
Provo, UT 84602

Abstract

Researchers have developed many different methods for evaluating locality, however there exists no standard method of evaluation that incorporates all aspects of temporal and spatial locality. In this paper we introduce an advanced form of a locality surface that incorporates all aspects of temporal and spatial locality into one three-dimensional surface. Examining a locality surface for a particular workload gives significant information about sequential, temporal, and loop constructs. This gives researchers a new avenue for characterizing workloads and examining locality quantitatively. We use our new locality surface to characterize the SPECint 2000 benchmark suite.

1 Introduction

Locality, or the principle that programs tend to use a small section of memory most of the time, has long been studied in conjunction with memory hierarchy design and performance evaluation [2] [7]. It is well known that caches perform well due to the high degree of memory reference locality exhibited by most programs during execution. Locality is therefore a useful method for characterizing workloads in terms of the workload's potential cache performance. There should exist a measure of locality which helps to identify the characteristics of a workload that impact memory hierarchy performance. However, "No generic or abstract or intrinsic characterization of locality on any modern benchmark suite has been attempted. Ideally it should be possible to arrive at a locality index or locality function for every application" [9]. In this paper we describe such a locality function and use it to characterize the SPECint 2000 benchmark suite.

Locality is usually divided into two types: temporal locality and spatial locality. A high degree of *temporal locality* describes a situation where referenced items are referenced again soon. A high degree of *spatial locality* describes a situation where items close together in memory tend to be referenced close together in time [7]. Most researchers examine temporal and spatial locality separately. We will show that this separation is not necessary.

1.1 Traditional Locality Measures

Some researchers simply use cache miss ratios to determine how much locality is exhibited by a particular workload [11]. However, rather than using locality to understand cache performance, these researchers are using cache performance to infer locality; low miss rates denote high amounts of locality, and high miss rates denote low amounts of locality.

Other researchers use a series of two-dimensional graphs to evaluate locality information [10] [1]. One graph may contain information about a workload's temporal locality, calculated as the distribution of the number of items between successive references to the same item. Conte and Hwu in [1] use a two-dimensional graph of the number of unique items between successive references to evaluate locality. Jacob, et al. call this the *stack distance* [8].

Other graphs may contain limited information about spatial locality. Some measures include finding the average distance in memory between each set

of successive references, or information about the average length of sequential runs of references. Conte and Hwu examine spatial locality by examining "the probability that between references to the same item, a reference to an item x units away occurs" [1].

Because many programs spend a majority of their processing time in loops, some researchers have focused their locality research specifically on loops. Additional terminology has been created. For example, in [10], McKinley and Teman split locality into four types: self-spatial, group-spatial, self-temporal, and group-temporal. In [12], the researchers choose to differentiate the terms *reuse* and *locality*. Again, these researchers rely on a series of two-dimensional graphs, only there are more elements to study and therefore more graphs.

Each of these methods has its uses and strengths. However, all of these methods involve separating the analysis of temporal and spatial locality. In addition, none of these methods include all aspects of locality.

2 The Locality Surface

In [4] [5], Grimsrud, et al. introduce the idea of a *locality surface*. Rather than using a series of two-dimensional graphs, Grimsrud includes all locality information on one three-dimensional graph using the following equation:

$$L(\vec{T}, s, d) = Pr\left(\vec{T}[t_0] + s = \vec{T}[t_0 + d] \wedge \vec{T}[t_0] + s \notin \{\vec{T}[t_0+1] \ldots \vec{T}[t_0+d-1]\}\right).$$

In the equation, L represents the resulting locality function; $\vec{T}$ represents a time-ordered trace of memory reference events; s designates the *stride*, or the distance in memory between two given items; d designates the *delay*, or the distance in time between two items; and t_0 designates a location in the trace. This equation results in a graph that is essentially a three-dimensional histogram based on two variables: stride and delay.

Locality is computed using this equation in the following manner. For each reference in the trace, find the first occurrence of every other memory reference, and calculate the stride and delay between the two references. For example, in the simplified trace in Figure 1, at time 0 the processor references memory word 4, and time 1 memory word 5, etc. Figure 1 shows all the locality events between the first referenced memory word and the first instance of each subsequent referenced word. Between the reference at time 0 and the reference at time 6, for instance, the delay is 6 (time 6 minus time 0) and the stride is 5 (memory word 9 minus memory word 4). After recording the stride/delay pairs associated with the memory reference at time 0, continue with time 1, etc., throughout the trace. The resulting histogram for Figure 1 is shown in Table 1.

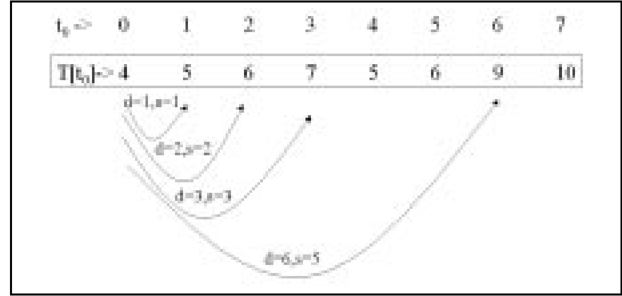

Figure 1: A simple memory reference trace with the locality dependencies between the first memory address and the first instance of each subsequent address. T represents the trace; t_0 is the time, or location, in the trace; and $T[t_0]$ is an address into memory.

Stride

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

Table 1: Resulting locality information for the simple trace in Figure 1 using Grimsrud's method.

Grimsrud's locality surface for a real workload is shown in Figure 2. For visualization purposes, the surface is displayed on a log scale in both the stride and delay axis. Grimsrud does this in the stride direction by averaging across all the affected bins; in the delay direction he sums all the affected bins. Grimsrud also divides each stride/delay bin by the total number of references in the trace. Two views of the same surface are typically displayed for easier identification of the height and location of various features.

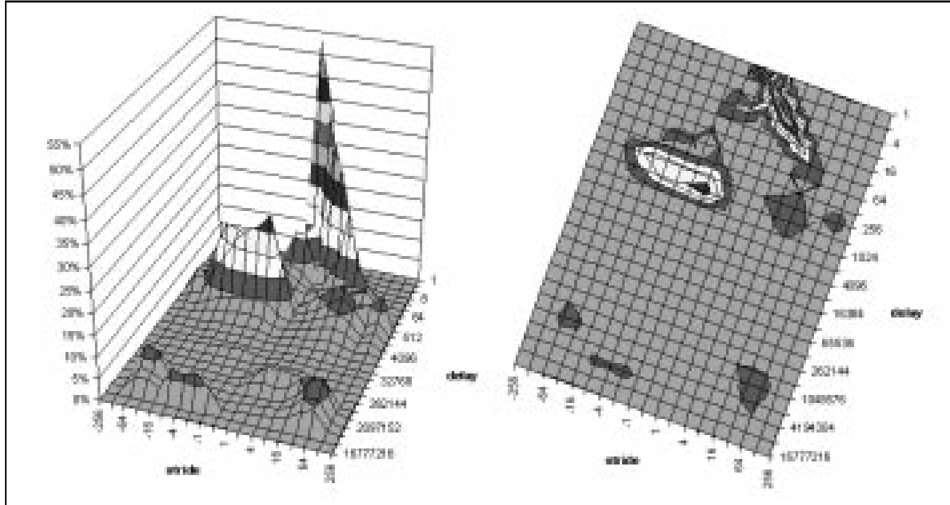

Figure 2: Example of Grimsrud's locality surface for the instruction fetches of the workload *gzip*.

The most dramatic advantage of this method is the unification of temporal and spatial locality into one locality measure. Temporal locality becomes a special case of locality that occurs when stride equals zero. For this reason, we often refer to the stride = 0 axis as the *temporal axis*.

A significant weakness with this locality surface is the delay measure. Grimsrud calculates the delay by counting the total number of references between the two items of interest. As noted earlier, other researchers [1] [8] find that a stack distance, or unique count, is better suited to the study of caches and therefore the calculation of locality.

2.1 Improved Locality Surface

We have improved Grimsrud's locality surface by using a stack distance when calculating the delay. In the example from Figure 1, the delay between the memory references at time 0 and time 6 now becomes 4 because there are 4 unique references (5, 6, 7, and 9) between time 0 and time 6.

This introduces a locality calculation problem illustrated by a trace consisting of the memory references shown in Figure 3. Notice that using a stack distance for delay means that each reference from time 1 to time 1000 has the same stride/delay relationship with the reference at time 1001, namely stride 4 and delay 1. However, the spatial locality calculated from this trace should be no different than if the trace only consisted

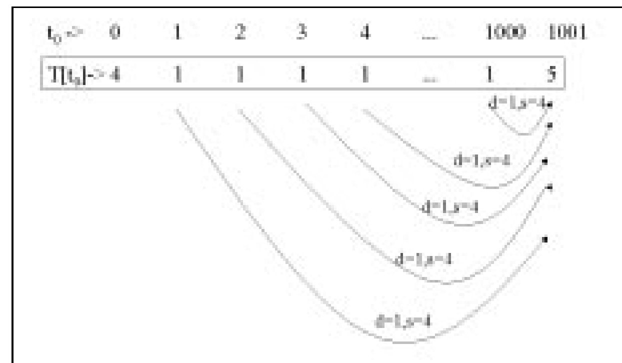

Figure 3: Sample trace that demonstrates a problem with using a stack distance delay measure with Grimsrud's locality function. All the references from time 1 to time 1000 have the same stride/delay relationship with the reference at time 1001, namely delay = 1 and stride = 4.

of three references, namely 4, 1, 5. (The difference between the two traces should only involve temporal locality.) So we add an additional requirement to our new locality surface which we call *limited effect*. This means that when calculating the stride/delay relationships between a given memory reference and its successors, we only use the successors that come before a repeated reference to the given memory reference. In the example in Figure 3, we would calculate the

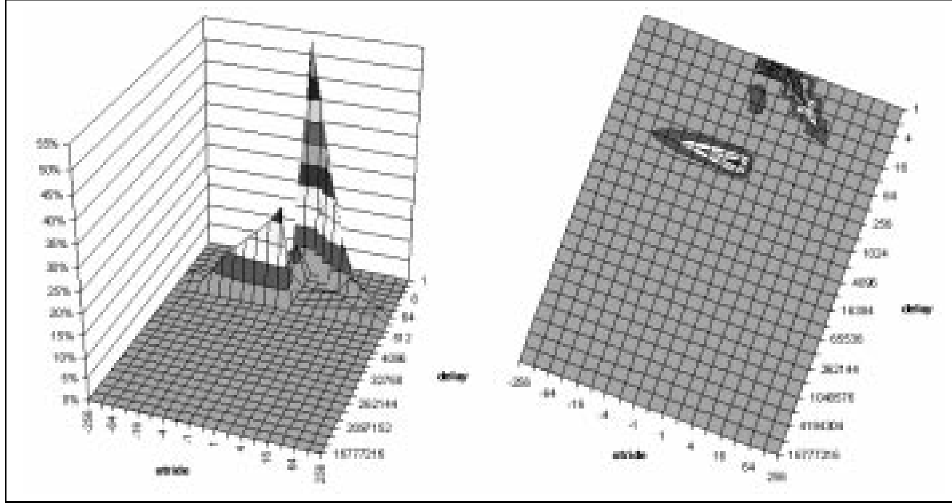

Figure 4: Example of our new locality surface that uses the stack distance and limited effect. This surface is calculated from the same workload used in Figure 2, namely the instruction fetches of *gzip*.

Stride

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

Table 2: Resulting locality information for the simple trace in Figure 1 using our new locality measure with stack distance and limited effect.

stride/delay from time 0 to time 1 and to time 1001, from time 1 to time 2, from time 2 to time 3, etc. and only calculate to time 1001 from time 0 and time 1000. The new locality histogram for the example trace in Figure 1 is shown in Table 2.

This limited effect requirement more closely simulates the function of a cache. When a particular memory item is submitted a second time to a cache, the second reference to the item has more impact on following items that the first reference. In fact, the impact of the second reference replaces the impact of the first.

Figure 4 shows our new locality surface, computed for the same workload trace that we used in Figure 2. Notice how the new surface has sharper features that are closer to the origin. The maximum delay of the surface is now a function of the stack distance of the trace, or the number of unique references in the trace, rather than the total length of the trace. The new surface is more closely tied to how a cache functions and should provide more useful information when studying caches and characterizing workloads in terms of memory hierarchy performance.

3 Characterizing Workloads Using Locality Surfaces

We now discuss some of the characteristics of the different features on a locality surface. To do this, we have created synthetic traces with several different kinds of memory accesses: sequential, random, temporal, and looping. We then created locality surfaces for each of these synthetic traces so we know how each memory access pattern is displayed on our surface. This enables the identification and quantification of each of these characteristics in real workloads.

3.1 Sequential References

First we will examine what a locality surface looks like for a simple sequence of memory references. The code fragment in Figure 5 creates a synthetic trace of

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

Figure 5: Code fragment to create a synthetic trace of sequential memory references.

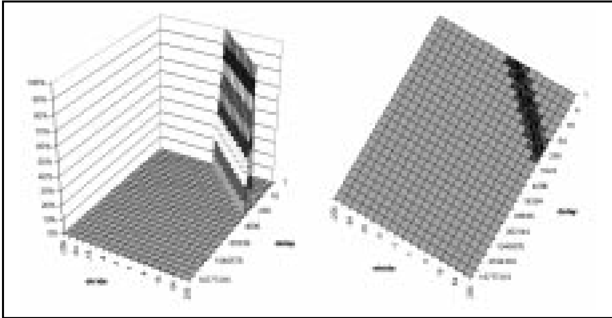

Figure 6: Locality surface for the sequential references generated by the code in Figure 5.

sequential references and Figure 6 shows the resulting locality surface. Sequential references create a ridge on the locality surface where stride = delay. 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.

Real workloads generally contain several different sequential runs of various lengths. The rate at which the ridge decays as stride and delay increase would demonstrate the distribution of the lengths of the various sequential runs.

3.2 Random References

The code fragment in Figure 7 creates a uniformly distributed list of random references. The locality surface of these references is shown in Figure 8. Most of the volume of the surface is around a delay of 1 million, the same number as the number of references in

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

Figure 7: Code fragment that creates a synthetic trace of uniformly distributed random memory references.

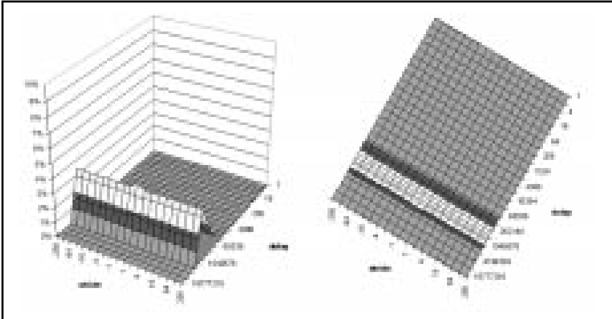

Figure 8: Locality surface for a series of uniformly distributed random numbers generated by the code in Figure 7.

the trace. This is because we are using the same log averaging in the stride direction and summing in the delay direction that Grimsrud used. Clear visualization prohibits the plotting of a 1 million by 512 surface without log binning; a locality surface would look relatively flat using a random sequence without binning. Notice the slight spike of the surface along the temporal axis, where the stride = 0. This is due to the limited effect requirement.

3.3 Temporal References

The code fragment in Figure 9 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 10. 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 32,000. There

```
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 9: Code fragment that creates a synthetic trace with varying amounts of temporal locality.

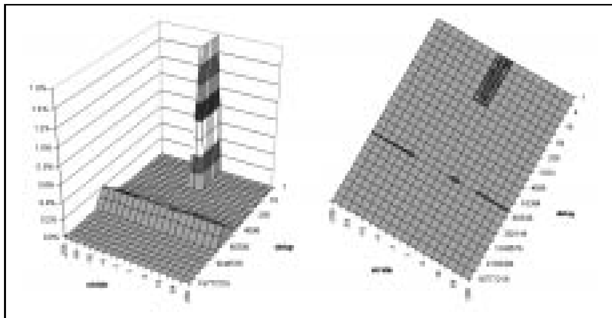

Figure 10: Locality surface for the synthetic trace with varying amounts of temporal locality generated by the code fragment in Figure 9.

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.

3.4 Looping References

Other than temporal and sequential features, the next most important feature to identify is a loop. Figure 11 shows a code fragment that creates five loops of equal frequency. The loops are 2, 16, 128, 1024, and 8492 references long. Figure 12 shows the resulting locality surface. The sequential ridge is due to the fact that each loop consists of sequential references. Notice the decay of the ridge due to the varying lengths

```
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 11: Code fragment that creates five sizes of loops with equal frequency.

of the sequential runs within the loops.

Looping structures are featured between the delay = -stride and stride = 0 axis. The location along the delay axis indicates the size of the loop in terms of the number of unique references in the loop. The loop of length two is almost hidden next to the sequential ridge. The height of the loop humps indicates the frequency of loops of that size. Qualitative predictions of cache performance for a particular workload can be performed by comparing the cache size with the size of the primary loops in a given workload. If the cache is not large enough to contain the major loops, cache performance will degrade.

4 The SPECINT 2000 Benchmark Suite

We used our new locality surface to evaluate the SPECint 2000 Benchmarks. Our trace data was collected using the BACH system [3] [6]. The traced system is a 450 MHz Pentium II based system with 256 Mbytes of memory running RedHat Linux version 6.0 with both L1 and L2 caches disabled. Each trace is over 100 million references long. The trace statistics

Workload	Instruction Fetches	Data Reads and Writes	Description
bzip2	69,573,178	34,388,558	Based on Julian Seward's bzip2 version 0.1, without the file I/O.
crafty	63,909,470	40,053,612	High-performance Computer Chess program with significant logical operations.
eon	51,519,737	42,444,848	Probabilistic ray tracer, with less memory coherence than a deterministic tracer, using the cook rendering algorithm.
gzip	74,751,517	29,209,958	Popular data compression program, using the Lempel-Ziv coding.
mcf	72,713,382	31,248,146	Single-depot vehicle scheduling in public mass transportation.

Table 3: Description of the SPECint 2000 benchmarks mentioned in this paper.

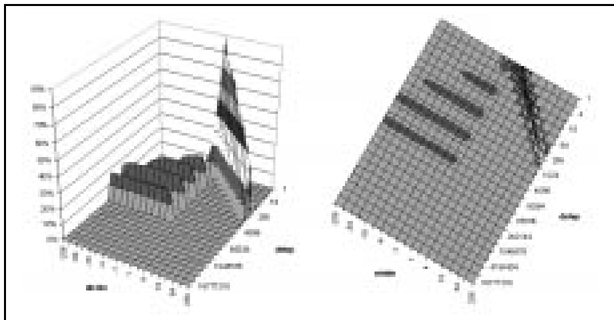

Figure 12: Locality surface that results from the synthetic trace of loops generated by the code in Figure 12.

are illustrated in Table 3.

We computed the locality surface for the instruction fetches and data separately because most level one caches are split instruction/data. A description of each of the benchmarks mentioned in this paper is also found in Table 3.

4.1 Locality of Instruction Fetches

We chose *bzip2*, *crafty*, and *eon* to represent the instruction fetches of the various workloads in the suite. The locality surfaces for the instruction fetches of these three workloads are shown in Figures 13, 14, and 15. Each of these surfaces exhibit a large temporal spike at a delay of 1. For example, *eon's* spike is at 61.7%, meaning that 61.7% of the words in the trace are repeats of the immediately previous word. This is due to the fact that when an instruction is fetched by a Pentium II based system eight bytes are transferred. These eight bytes may contain more than one instruction, but this eight byte word must be transferred for each instruction fetched within the word. Of course this usually results in L1 cache hits, and therefore is not a performance problem.

Notice a significant sequential ridge in each of the three featured workloads. The height, length, and decay rate of the ridge varies with the workload. This is typical of instruction fetches in general. Compare the ridge for *bzip2* and *crafty*. *Crafty's* sequential runs are often 64 words long, while *bzip2's* sequential ridge is essentially finished after 16 words. Sequential ridges such as those found in instruction fetches indicate the usefulness of long cache lines. Cache miss rates for various cache sizes and line sizes for *crafty's* instruction fetches are found in Table 4. The cache results indicate that increasing the line size improves performance more than increasing the cache size.

Each of the instruction fetches' locality surfaces also has one or more significant looping structures. *Bzip2* has a strong loop that is 64 unique words long, while *eon* exhibits a series of loops ranging from 8 unique words to 4096 unique words long. *Crafty's* looping structures appear to be somewhere between *bzip2* and *eon*. Cache performance would be significantly improved when the cache is large enough to contain the largest looping structure. For *bzip2*, a 64 word (1 Kbyte) cache should perform much better than a 32 word (512 byte) cache, because the smaller cache could not contain the entire loop simultaneously. Notice in Table 4 the sharper decrease in the miss rate as the cache size increases from 4 Kbyte to 32 Kbyte. This is a result of the wide looping structure from 1K to 4K word (8 Kbyte to 32 Kbyte) in the locality surface.

4.2 Locality of Data Reads and Writes

The locality surfaces for the data reads and writes of the *crafty* and *gzip* workloads are in Figures 16 and

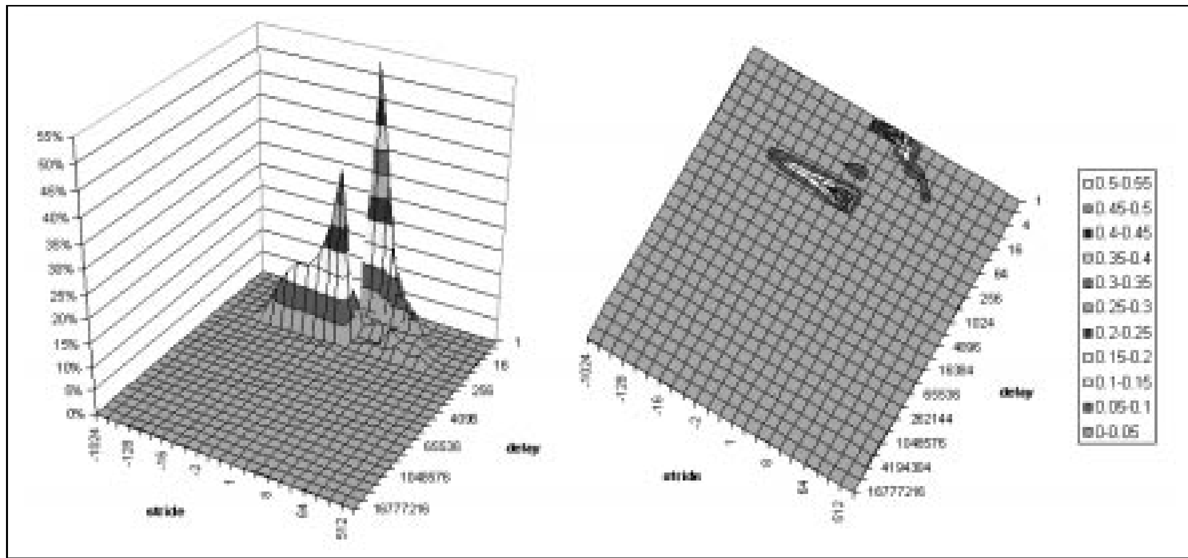

Figure 13: Locality surface for the instruction fetches of the *bzip2* workload.

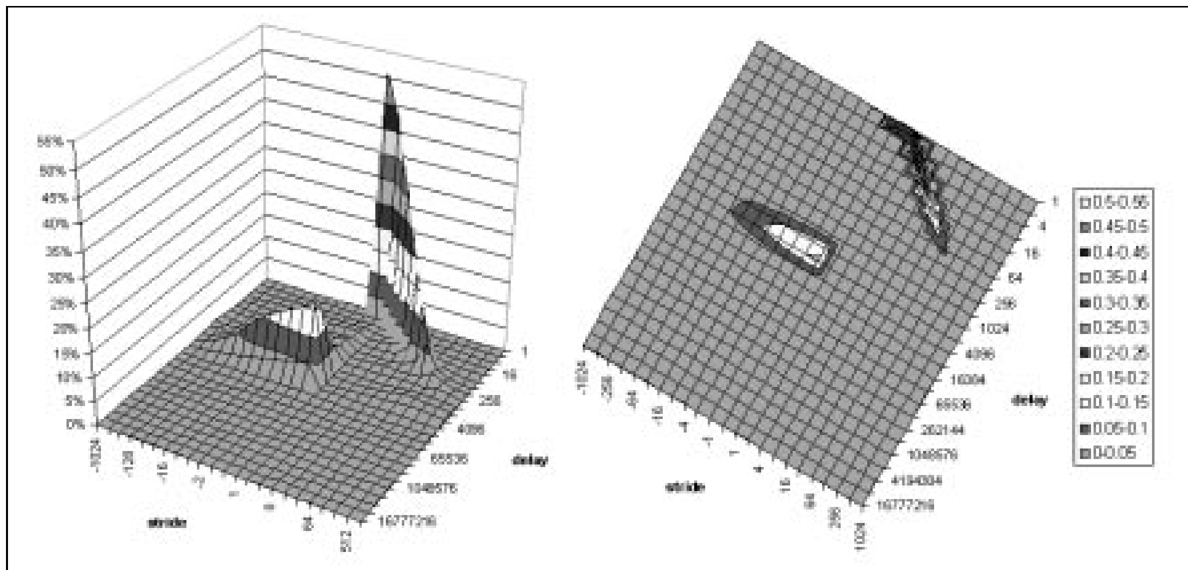

Figure 14: Locality surface for the instruction fetches of the *crafty* workload.

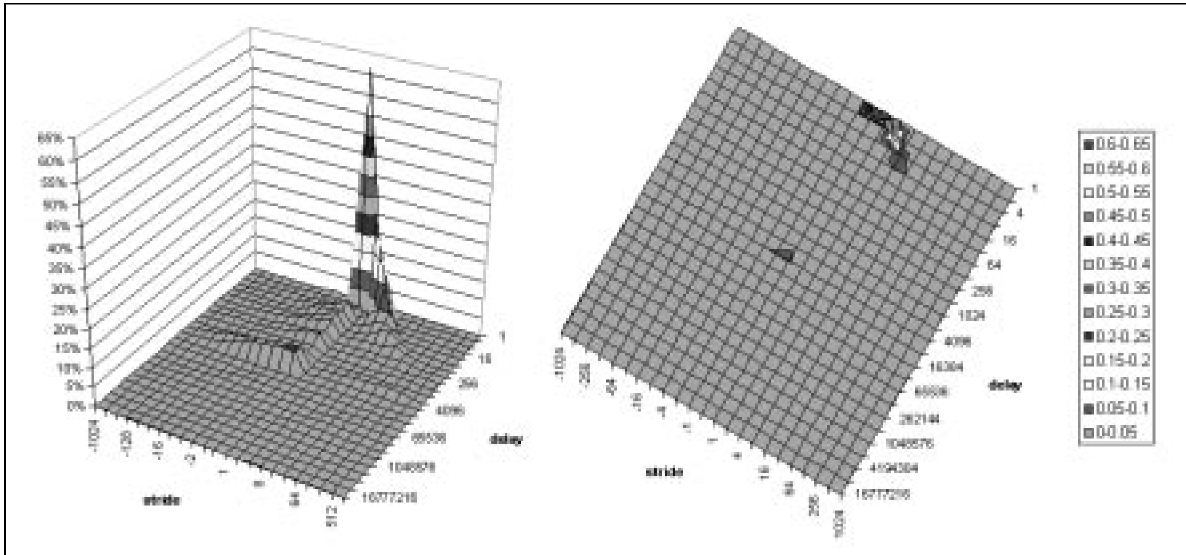

Figure 15: Locality surface for the instruction fetches of the *eon* workload.

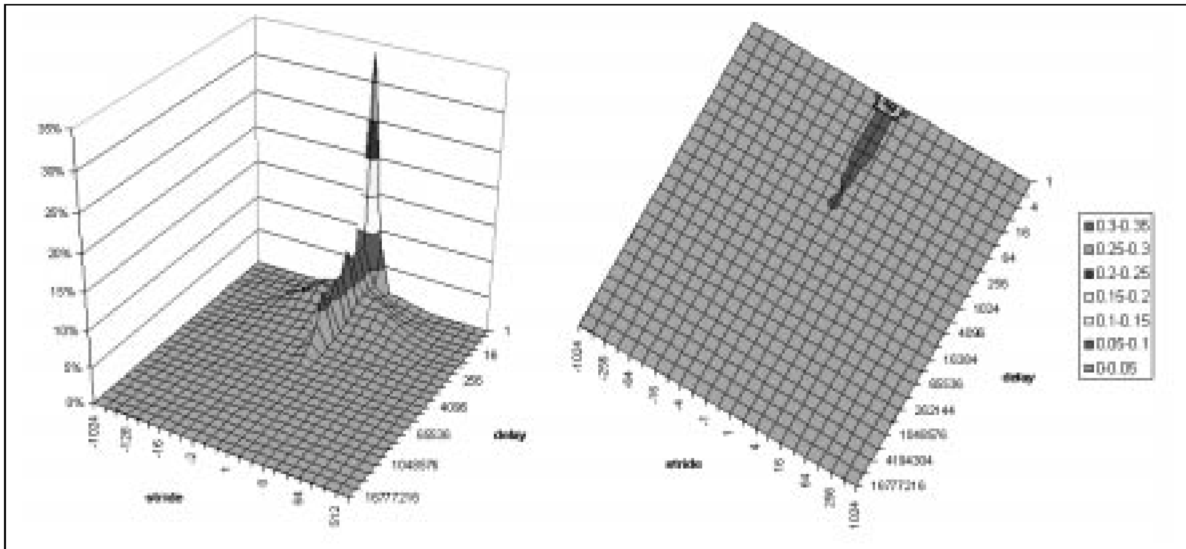

Figure 16: Locality surface for the data reads and writes of the *crafty* workload.

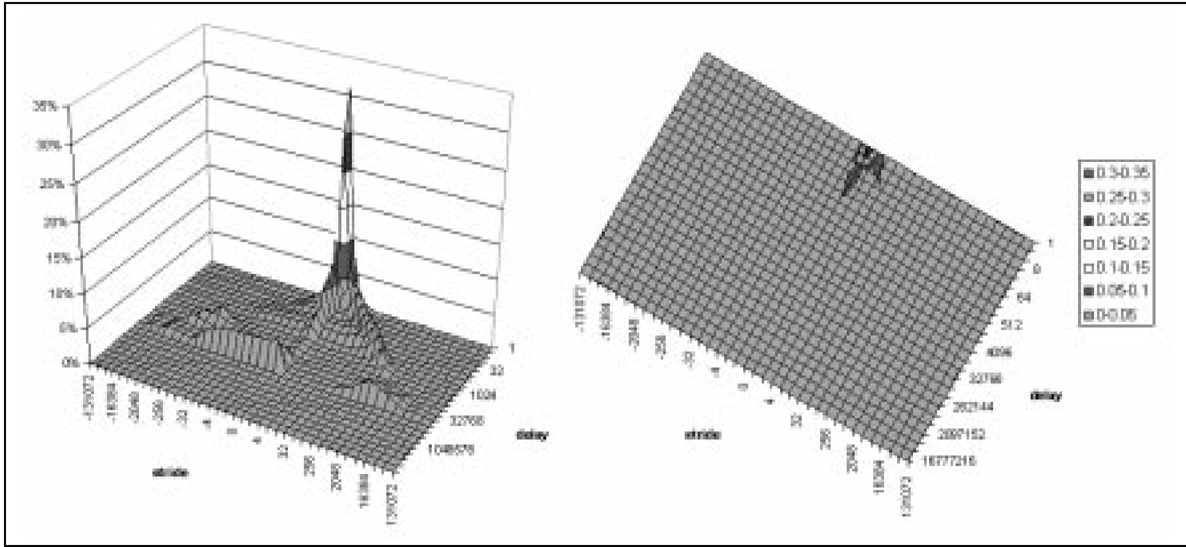

Figure 17: Locality surface for the data reads and writes of the *gzip* workload.

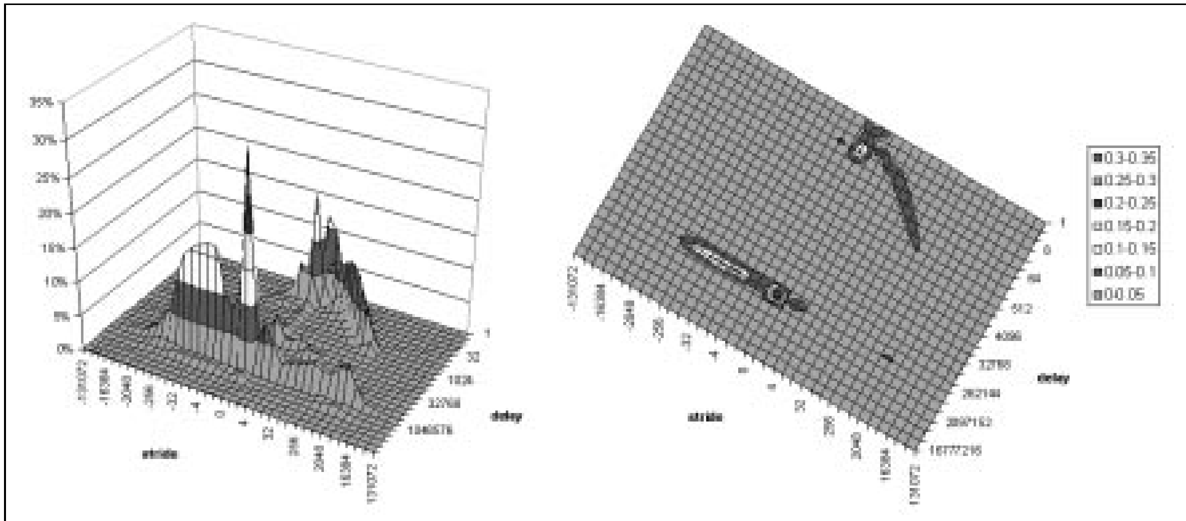

Figure 18: Locality surface for the data reads and writes of the *mcf* workload.

Line Size (bytes) / Cache Size (bytes)

	8	16	32	64
1 K	41.09%	22.60%	13.45%	8.07%
2 K	38.42%	21.15%	12.50%	7.42%
4 K	31.51%	17.63%	10.59%	6.39%
8 K	22.14%	12.41%	7.59%	4.76%
16 K	14.63%	8.31%	5.20%	3.34%
32 K	6.69%	3.88%	2.51%	1.70%

Table 4: Cache miss rates obtained from simulation for the instruction fetches of the *crafty* workload. The associated locality surface, in Figure 14, predicts that increasing line size will result in better performance improvement than increasing the cache size. Note how doubling the line size improves performance more than quadrupling the cache size, as predicted.

Line Size (bytes) / Cache Size (bytes)

	8	16	32	64
1 K	15.60%	15.35%	16.00%	16.70%
2 K	11.60%	11.84%	12.67%	13.65%
4 K	8.10%	8.52%	9.49%	10.70%
8 K	4.10%	4.60%	5.43%	6.63%
16 K	2.43%	2.67%	3.16%	3.83%
32 K	1.09%	1.22%	1.45%	1.77%

Table 5: Cache miss rates obtained from simulation for the data reads and writes of the *crafty* workload. Notice that increasing the line size usually increases the cache miss ratio.

17. We also include the locality surface for the data of *mcf* in Figure 18 because of its interesting characteristics.

Again, there is a significant temporal spike at a delay of 1 for each of the three surfaces, however the spike is lower for the data reads and writes than for the instruction fetches. *Crafty* and *gzip* have spikes at 33.7% and 31.1% respectively. The data locality surfaces also display temporal locality at delays other than 1. *Crafty's* temporal ridge tapers off around 1024 words. Since this temporal ridge dominates the little amounts of sequential references present, cache configurations would give better results with smaller line sizes to make it easier for smaller caches to contain all the references. The cache results shown in Table 5 support this supposition. Increasing the line size typically worsens the cache miss rate.

Many of the SPECint 2000 benchmark suite's data references are similar to either *crafty* or *gzip*. *Crafty* has most of its mass close to the origin, without the significant looping structures featured in the instruction fetches. Except for the dominant temporal ridge, *gzip* spreads its mass more evenly throughout the surface. A minor loop is evident around 32K words, and a small amount of sequential runs are present, however these features do not dominate the surface.

The most interesting surface we found was *mcf's* data reads and writes. This is the only surface we have found from real workloads where the tallest portion of the surface is not in the delay = 1, stride = 0 bin. It also displays the largest working set of the SPECint 2000 benchmark suite, with 256K unique words. A cache at least 2 Mbyte in size would be needed to contain the entire working set. Figure 19 shows cache simulation results for *mcf's* reads and writes on 4-way associative caches with 8 byte lines. The size of the caches ranges from 1 Kbyte to 32 Mbyte. As predicted by the locality surface, there is a sharp drop in miss rate as the cache size increases past 2 Mbyte. Due to its reasonably sized sequential ridge, however, a larger line size might also be useful.

5 Conclusions and Future Work

We have developed a measure that effectively evaluates the locality characteristics of workloads. Using this method, it is possible to characterize workloads in terms of sequential runs, random effects, temporal locality, and looping structures simultaneously. The locality surface is comprehensive and straightforward to calculate. Currently, we are working on a method to use our locality surfaces to predict cache miss ratios for any traced workload and any cache configuration. We hope to find a cache characterization surface that, in conjunction with a locality surface, can predict the miss ratio of a particular cache and workload combination.

In the future, we believe that we can use locality surfaces to assist in specific memory hierarchy design

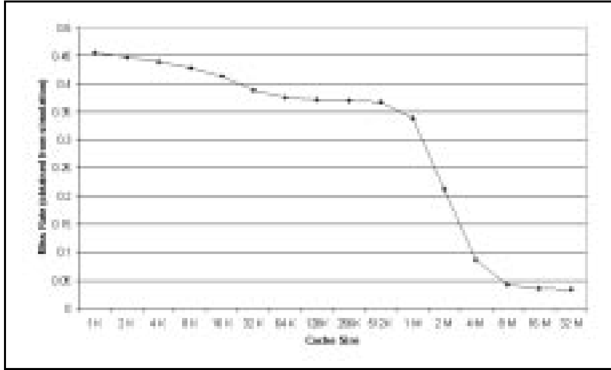

Figure 19: Cache miss rates obtained from simulation for the data reads and writes of the *mcf* workload. The caches all have 8 byte lines and are 4-way associative. Note the sharp decrease in miss rate around a cache size of 2 Mbyte, as predicted by the locality surface in Figure 18.

choices by examining locality at various stages in the hierarchy. In addition, we hope to be able to use a surface with indicated loops and sequential ridges and given amounts of temporal locality to create synthetic traces.

6 Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. 9807619.

References

[1] Thomas M. Conte and Wen mei 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.

[2] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[3] 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.

[4] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson. On the accuracy of memory reference models. In *7th International Conference Proceedings*, pages 369–388, Springer-Verlag, May 1994.

[5] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson. Locality as a visualization tool. *IEEE Transactions On Computers*, 45(11), November 1996.

[6] 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.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[8] Bruce L. Jacob, Peter M. Chen, Seth R. Silverman, and Trevor N. Mudge. An analytical model for designing memory hierarchies. *IEEE Transactions on Computers*, 45(10), October 1996.

[9] Lizy Kurian John, Purnima Vasudevan, and Jyotsna Sabarinathan. Workload characterization: Motivation, goals and methodology. In *Workload Characterization: Methodology and Case Studies*, Dallas, Texas, November 1998.

[10] Kathryn S. McKinley and Olivier Temam. Quantifying loop nest locality using spec'95 and the perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4):288–336, November 1999.

[11] 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.

[12] Michael E. Wolf and Monica 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.