A Component-based Definition of Spatial Locality

Xiaoming Gu, Chen Ding
Computer Science Department
University of Rochester
{*xiaoming,cding*}*@cs.rochester.edu*

Chengliang Zhang
Microsoft Corporation
chengzh@microsoft.com

Abstract

The data layout of a program is critical to performance because it determines the spatial locality of the data access. Most quantitative notions of spatial locality are based on the overall miss rate and leave three questions not fully answered: how much can the locality of a given data layout be improved, can a data layout be improved if the miss rate cannot be lowered, and can the overall spatial locality be decomposed into finer components? This paper describes a new definition of spatial locality that addresses these questions. The model is based on online profiling and off-line analysis. It has been used to analyze 7 SPEC2000 benchmarks and 1 SPEC2006 benchmarks. Among their 18 components, it finds 5 components that have a significant problem of poor spatial locality.

1 Introduction

Caching and prefetching are the two most effective ways for hardware to improve memory performance. Both depend critically on the layout of the program data—whether the program has good *spatial locality*. Most quantitative notions of spatial locality are based on the miss rate—whether a new data layout leads to a fewer number of cache misses. The miss rate, however, is not enough to fully understand this important concept from the perspective of the software.

First, the miss rate is the overall effect from all computations and data. It does not show whether a program has different locality among its components. For example, different pieces of data may have different locality, or the same piece of data may have different locality when accessed by different parts of computation. While powerful compiler techniques have been developed to identify these components for regular scientific code, the miss rate is still widely used for programs not amenable to compiler analysis such as those with input-dependent control flows and indirect data accesses.

Second, it is uncertain whether the miss rate can be reduced without trying other choices. Changing the data layout for large, complex code is time consuming and error prone. Yet the effort is inconclusive at best. It may fail to change the miss rate, then the effort is wasted. If it leads to an improvement, it is still unclear whether more improvements are possible. In both cases, the programmer, after much labor, returns to the starting point facing the same uncertainty. Many studies found that a technique improved some programs but not some others. There is no general test to show whether the lack of improvement in a program is due to the limitation of the technique or the lack of room for improvement.

Third and last, when prefetching is considered, the quality of two data layouts may differ even though they incur the same number of cache misses. A concrete example was described by White et al. in 2005 [39]. They studied the effect of data layout transformations in a large (282 files and 68,000 lines C++), highly tuned and hand optimized mesh library used in the Lawrence Livermore National Lab. A data packing transformation, consecutive packing [8], placed the data objects based on the order of their access. Since mesh entities were too large for spatial reuse within a cache block, the transformation did not reduce the number of (L1/L2) cache misses. However, it produced a sequence of addresses more amenable to stream prefetching, so it increased the number of useful prefetches by 30% and reduced the load latency from 3.2 cycles to 2.8 cycles (a 7% overall performance gain), as measured using hardware counters on an IBM Power3 machine [39].

In addition, White et al. found that two other transformations, iteration blocking and register tiling [25], although reducing the number of loads and branches by 20% and 9%, resulted in a higher load latency of 4.4 cycles because they interfered with hardware prefetching. Because of prefetching, not all cache misses are equal. Those misses that are incurred on consecutive memory blocks are less costly

because the data can be more effectively prefetched by hardware or software.

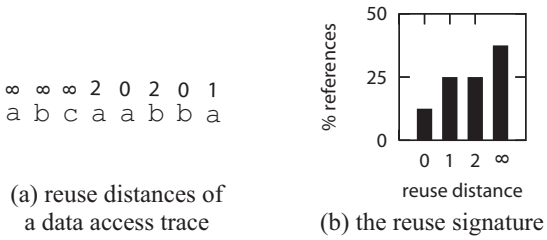
In this paper, we redefine spatial locality based on the distance of data reuse. In particular, it examines the change of reuse distance for different block sizes. When the data block size is the same as the cache block size, the spatial locality will measure the utilization of cache blocks. When data blocks are larger than cache blocks, the spatial locality will reflect the effect of consecutive hardware prefetching. The new model is based on components, which are groups of memory accesses that have similar reuse distances. We will use the new model in conjunction with metrics of the temporal locality and component size to identify components that are likely to have a problem of poor spatial locality.

Spatial locality was first modeled using the notion of working set [2]. Recently, Weinberg et al. defined a spatial locality score by varying the size of "look back" windows in time [38]. The model here is defined by varying the size of data blocks in memory, and it has finer granularity than the score of the whole-program. As we will show through experiments on a wide range of benchmarks, most of their components do not have a serious problem of spatial locality, but some non-trivial components do. The new model can be used to automatically identify these components and measure the specific effect of a program transformation. For example in *swim*, an improvement to the spatial locality of 7.3% memory references leads to 10% and 14% overall performance gain on two modern machines.

The model uses only program-level metrics and does not model machine-dependent factors. For example, it predicts whether a data layout would benefit more than another from prefetching but it does not predict the performance difference on a specific machine. It assumes a fixed computation order, so the spatial locality is determined entirely by the data layout. The analysis is based on the postmortem study of execution traces, which has two limitations. First, the components we identify are groups of memory accesses in a trace. They are not components in code or data. Correlating between locality and program components remains our future work. The other problem is that the trace changes when a program takes a different input. In this work, we measure the spatial locality of multiple inputs to examine whether and how the high-level locality metrics change with the input.

2 Background

The *reuse distance* of a memory access is the number of distinct data elements accessed between this and the previous access to the same data. For example for the data access trace in Figure 1(a), the reuse distance of the second access of b is 2 because 2 distinct data elements appeared between this and the first access of b. The distribution of all reuse distances in the trace is called the *reuse signature*, shown in Figure 1(b). Reuse distance is the same as LRU stack distance defined by Mattson et al. in 1970 [23].



(a) reuse distances of a data access trace

(b) the reuse signature

Figure 1: Example reuse distances and their reuse signature

While the reuse signature is purely a software measure, it can be used to calculate the miss rate on a machine. In fully associative LRU cache, a memory access misses in cache if and only if its reuse distance is equal to or greater than the cache capacity (hence the name LRU stack distance). The effect of direct mapped and set associative caches can be estimated based on a set of probability equations by Smith in 1976 [32]. It is based on the assumption that a reuse has an equal probability finding the previous copy in any one of the cache sets. Past studies found it quite accurate [22,32]. Algorithms and tools for direct simulation of set-associative cache also exist [17,34].

3 A Component-based Definition of Spatial Locality

Spatial locality measures the quality of data layout for a given cache block size or larger data block sizes when prefetching is considered. It measures how locality changes as a function of the data block size. In particular, we define spatial locality as the change in reuse distance when the data block size changes from b_1 to twice the size $b_2 = 2b_1$.

Spatial reuse may shorten a reuse distance in two ways. Suppose data x, y of size b belong to the same $2b$ block, and the reuse distance happens between two accesses of x. If y is not accessed in between, the reuse distance may be shortened as the number of $2b$ size blocks may be less than the number of b size blocks.

If y is accessed in between, the reuse distance is definitely shortened since the reuse is "intercepted". The one reuse pair becomes two reuse pairs, and the latter pairs give the new reuse distance measured in $2b$ size blocks.

The example in Figure 2 shows an example of an intercept due to spatial reuse. Suppose each element of array d is of size b. When the block size doubles from b to $2b$, the reuse distance between two accesses of $d[1]$ is changed from a temporal reuse to a spatial reuse between the second $d[1]$ access and the intervening $d[2]$ access.

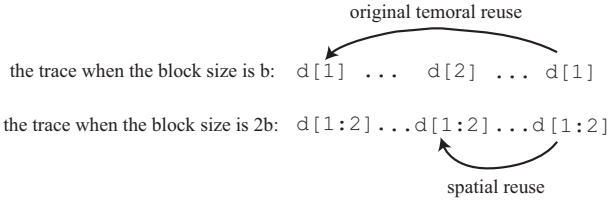


Figure 2: An example effect of spatial reuse

The shorter the new reuse distance compared to the original distance, the better the spatial locality, because it leads to fewer cache misses. In practice, the spatial reuse needs to reduce the length of reuse distance significantly to make a difference. In the absence of intercepts, a reuse distance can be shortened by at most a factor of two. Any larger reduction must come from intercepts. In Figure 2, the reduction depends on how close the access of $d[2]$ to the second access of $d[1]$.

We compute the effect of spatial locality for each bin. Here we use superscripts to represent the size of data blocks and subscripts to represent the index of the bin, so the size of ith bin when block size is b is represented as s_i^b. We partition the reuse distances of s_i^b into two portions *effective spatial reuse* and *ineffective spatial reuse* with percentages α_i^b and β_i^b ($\alpha_i^b + \beta_i^b = 1$) respectively. Effective spatial reuse includes reuse distances whose bins have been changed by a threshold factor, which is 3 in our experiments. Effectively it is an order of magnitude reduction in reuse distance. The remaining ones are considered ineffective spatial reuse.

To precisely measure α_i^b and β_i^b, we need to know how the reuse distance is changed by doubling the block size. For implementation we augment a basic reuse distance analyzer [9] by running two analyzers in parallel for two block sizes. For each memory access, the analyzer computes the reuse distance for the two block sizes and based on the difference, it classifies the access as having effective or ineffective spatial reuse. The original analyzer stores the sub-trace containing the last access of each block. The new analyzer stores two sub-traces, one for each block size. With the algorithm of compression tree [9], the space cost of each sub-trace is logarithmic to the data size, so twice the cost is not much higher than before.

We normalize the ratio of effective spatial reuse to a number between 0 and 1. We consider sequential access as the best case for spatial reuse. In that case, α equals to 0.5 since half of the reuse distances become 0. The data layout quality score SLQ is measured by how the actual reduction compares with the best-case reduction.

$$SLQ(s_i^b) = \frac{\alpha_i^b}{0.5} = 2\alpha_i^b. \quad (1)$$

Finally we define a *component* as data reuses of nearby bins that have a similar portion of effective spatial reuse. In this study we will manually examine the score for all bins and group them into components. In the future, we will consider using pre-defined ranges based on cache sizes, although the group composition will depend on the input of a program (if its reuse distance changes with the input).

4 Experimental Results

We have tested our model on seven programs from SPEC2000 [33] and one program from SPEC2006 [33]. The basic information about these benchmarks are given in Table 1. First are three floating-point benchmarks followed by four integer benchmarks come from SPEC2000. The last is a floating-point benchmark from CPU2006. To measure the effect of different inputs, we have collected results for all reference inputs of these programs. We also test two synthetic cases. All of the C programs are compiled using "gcc -O3", and the Fortran programs using "f95 -O5" using the GNU compiler (version 4.1.2). The 14 executions shown in Table 1 have different characteristics. The data size ranges from 1.2MB to 72MB, and the trace length, measured by the number of memory accesses, ranges from 7.7 billion to 400 billion.

We use the dynamic binary instrumentor Valgrind (version 3.2.2) [27] to collect the data access trace and measure the reuse distance using the tool described in Section 3. It measures reuse distance in near linear time with a guaranteed precision, which we set to be 99.9%. The cost of instrumentation and reuse distance analysis is up to several hundred times slowdown. For example, it takes 100 hours to analyze the 400 billion memory accesses of an *Milc* run, which takes less than 19 minutes without instrumentation.

programs	inputs	data size (bytes)	trace length
equake ref	<inp.in	5.0e+07	5.9e+10
art ref1	-scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 110 -starty 200 -endx 160 -endy 240 -objects 10	3.7e+06	1.1e+10
art ref2	-scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 470 -starty 140 -endx 520 -endy 180 -objects 10	3.7+06	1.2e+10
swim ref	< swim.in	2.0e+08	9.2e+10
swim.opt ref	< swim.in	2.0e+08	9.2e+10
gzip ref1	input.source 60	4.2e+07	1.5e+10
gzip ref2	input.log 60	3.9e+07	7.7e+9
gzip ref3	input.graphic 60	6.5e+07	2.4e+10
gzip ref4	input.random 60	7.4e+07	1.9e+10
gzip ref5	input.program 60	5.2e+07	2.6e+10
mcf ref	inp.in	8.0e+07	1.8e+10
crafty ref	< crafty.in	1.3e+06	5.0e+10
twolf ref	ref	1.1e+06	1.1e+11
milc ref	< su3imp.in	7.2e+08	4.0e+11

Table 1: The size of the 14 executions of 8 benchmarks

In the following analysis, we just focus on the components whose reuse distance are larger than 31, since now even the register set of a modern machine are larger than 31. We also cut off those components who contain less than 0.1% of memory accesses, considering them insignificant in program performance.

4.1 Spatial Locality of Synthetic Traces

We test sequential and random data traversal. As expected, in the reuse distance histogram from sequential access, the size of each long-distance bin is halved as the data block size is doubled, showing the best case spatial reuse (spatial locality quality score is 1). In contrast, in the histogram from random access, the spatial locality quality score is less than 0.13 for most block sizes.

4.2 A Case Study: Swim

Swim is a floating-point benchmark program from SPEC2000. It simulates shallow water using a two-dimensional grid, represented by a set of 14 arrays. We use two versions—the original version and the version after array regrouping, which is supposed to improve the spatial locality [30, 43].

Figure 3 shows the spatial locality score for both versions with 64-byte or 128-byte cache line. The score is show by the y-axis, and the reuse distance bin (equal or greater than 8) by the x-axis. The score for each bin is marked by a cross for the original version and by a downward triangle for the transformed version. The size of the bin is show by the size of the circle enclosing the mark.

With 64-byte cache line in both versions there are three bins (≥ 10) of a significant size: bin 12, 21 and 22. The spatial locality of the latter two bins is identical in two versions, shown by their perfect overlap. The spatial locality score of the first bin, which accounts for 4.6% or 3.1% of all memory accesses repectively in both versions, has been improved from 0.16 to more than 0.99. Our previous work shows that array regrouping improved the performance by 14% on IBM Power4 [30]. For this study, we compared Gcc-compiled 64-bit binaries on 3.2GHz Intel Xeon and observed 8.1% performance improvement. With the new spatial locality model, we now see that the improvement is caused by the better spatial locality for the about 4% memory references.

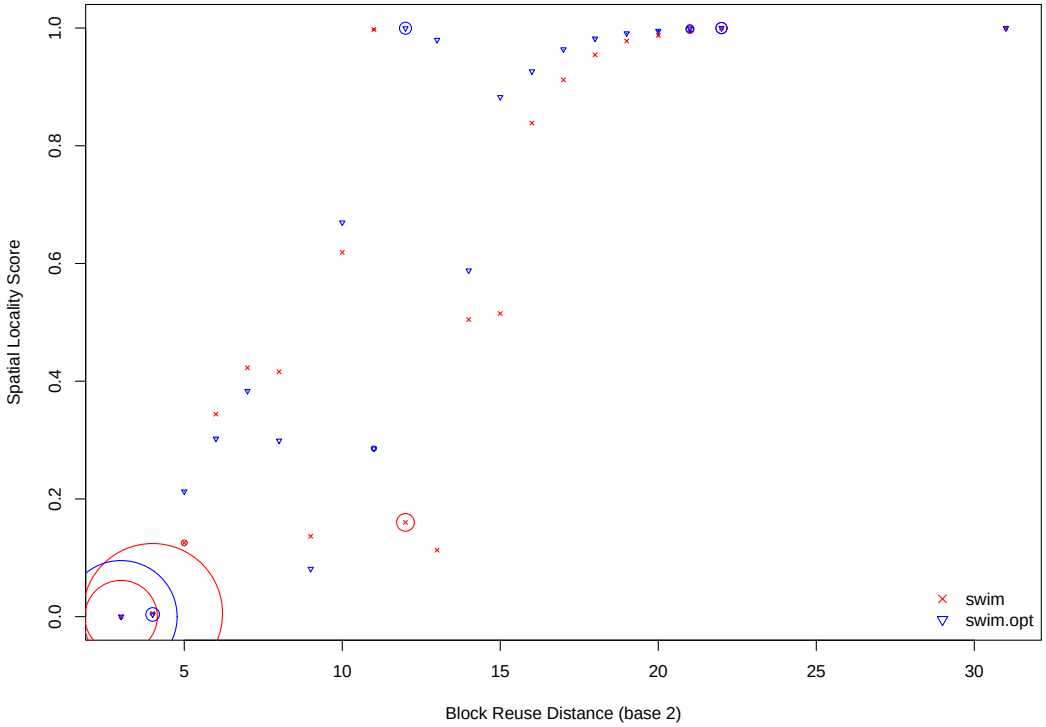
On the specific machine we tested with 64-byte cache line, the L1 cache size is 32K and L2 cache size is 1M. Let's assume fully-associative cache with cache block size 64, the predicted cache miss rates of the original swim benchmark are 10.4% and 5.33%. The cache miss rates for the optimized version are 9.7% and 5.33%. So the performance improvement mainly benefits from the lower L1 cache miss rate. However, we should notice that a reduction of 6.7% L1 cache miss rate can not explain the 8.1% performance improvement. In fact, as our spatial locality model predicts, the optimized version benefits from prefetching.

With 128-byte cache line, the two versions show more distinctive difference. The bin 12 in original program has no spatial locality, which accounts for 4.1% of all memory accesses. After array regrouping, this component is partly moved to bin 11 with spatial locality score 0.59. It means doubling block size may improve temporal locality and spatial locality together.

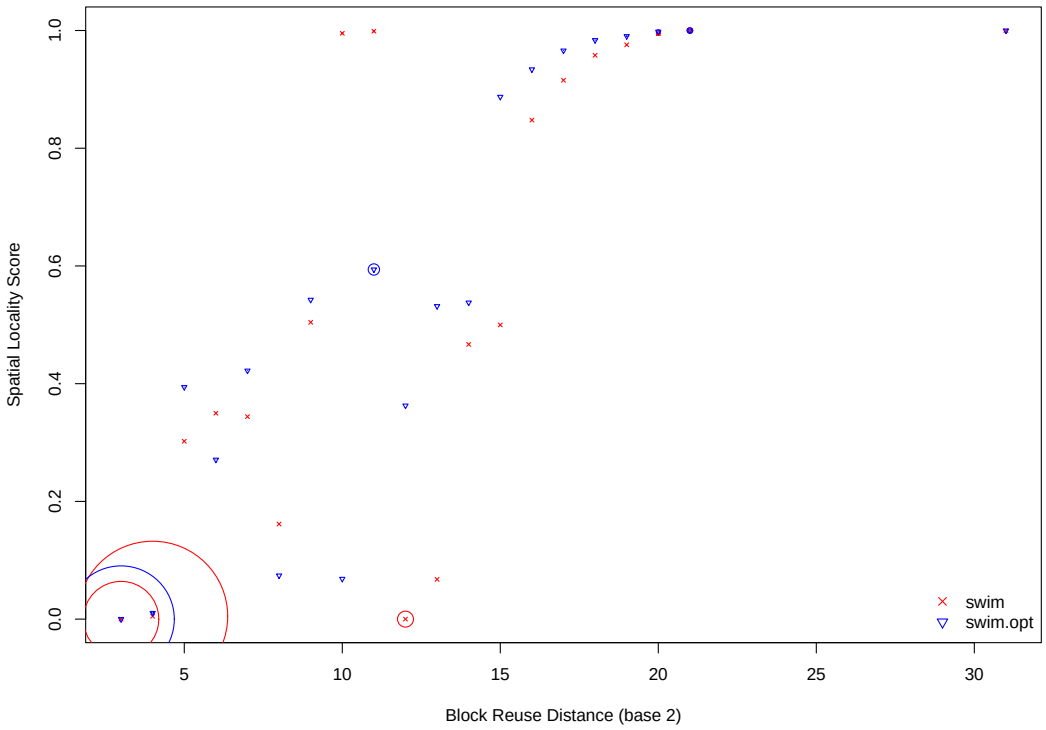
The program *Swim* demonstrates two unique features of the model. First, the model is based on components, so it can reveal different locality patterns within the same application, if different components are separated in their reuse distance. Second, the model is based on different data block sizes. It shows that the spatial locality of the same component may change significantly depending on the size of the data block.

4.3 All Benchmark Results

Our analysis has identified 18 components in the 14 executions of the 8 programs, including the two components (on the reuse histogram) for each run of the 4 programs, *equake*, *mcf*, *swim* and *swim.opt*, and one for each of the other 10 executions. Figure 4 and



(a) 64-byte data block



(b) 128-byte data block

Figure 3: Spatial Locality Score Comparison of *Swim*: Original Program vs. Program after Array Regrouping

Figure 5 show three attributes of the spatial locality components: the spatial locality score, the temporal reuse distance and the size. All are for the block size of 64 bytes. For each component, we consider the weighted average of temporal reuse distance and spatial locality score. In the names of components, we use 'c' for multiple components in a single input and 'r' for multiple inputs with the same program. For example, *swim-c2* is the second component of the *swim* execution, and *gzip-r3* is the (only) component of the third input of *gzip*.

The x-axis of Figure 4 shows the weighted average reuse distance of each component. The range of the reuse distance differs from component to component and program to program. But different inputs of the same program show similar reuse distance as those of *gzip* and *art*. The x-axis of Figure 5 shows the sizes of the components as the percentage of the total references in a run.

Based on the summarized results, we classify the locality of the 18 components of the 7 SPEC benchmark programs in four categories.

Components with good temporal locality The components *crafty* and *equake-c1* (11% of 18) have good temporal locality because they have short reuse distances (shorter than 256 blocks or 16KB)

Components with good spatial locality The following 6 components (33% of 18), *equake-c2, mcf-c2, milc, swim-c2, swim.opt-c1* and *swim.opt-c2*, have good spatial locality for because all long-distance components have almost perfect spatial locality (a score greater than 0.78).

Executions with poor spatial locality A component has a serious spatial locality problem if it meets the following three conditions.

- The component has a significant size,
- It has long reuse distances (poor temporal locality), and
- It has low spatial quality score (poor spatial locality).

Five components (28% of 18), *art-r1, art-r2, mcf-c1, swim-c1* and *twolf* meet these conditions. They contain between 5.13% to 44.4% references. Their reuse distance ranges from 64KB to 2MB. Their spatial locality score is between 0.250 and 0.657. *Art* has identical components in two inputs, suggesting a good chance for compiler optimization.

Components with possible spatial locality problems Five components (28% of 18) of *gzip* with different input have the low spatial locality scores of 0.140 and 0.387. However, these component has relatively short reuse distances. While their sizes are from 5.82% to 21.5% of their references respectively, almost entire of them have the reuse distance less than 8K blocks or half mega-bytes, which fits in the level-two cache of most modern machines.

Other data block sizes The preceding results are for a single data block size, but they help us to focus on 10 out of the 18 components. We pick up 5 components and show them in Table 2 for block sizes from 16 to 128. The spatial and temporal locality are shown in term of their weighted average.

component	reuse distance	spatial locality	size (%)
16-byte blocks			
art-r2	16	0.21	62.18
gzip-r4	12	0.25	10.81
mcf-c1	16	0.32	63.76
swim-c1	13	0.7	7.33
twolf	13	0.44	29.9
32-byte blocks			
art-r2	15	0.4	55.54
gzip-r4	11	0.18	8.97
mcf-c1	16	0.55	55.7
swim-c1	12	0.44	5.86
twolf	13	0.39	20.7
64-byte blocks			
art-r2	14	0.66	44.44
gzip-r4	10	0.14	6.8
mcf-c1	15	0.39	40.63
swim-c1	11	0.25	5.13
twolf	12	0.33	14.24
128-byte blocks			
art-r2	13	0.74	29.84
gzip-r4	9	0.11	5.82
mcf-c1	14	0.56	33.63
swim-c1	11	0.13	4.76
twolf	11	0.29	10.32

Table 2: The spatial locality of 5 components for 4 data block sizes

The results show that the reuse distances of these components decreases by one when the cache block size is doubled. The component size decreases only slightly and far less than a 50% reduction. Conse-

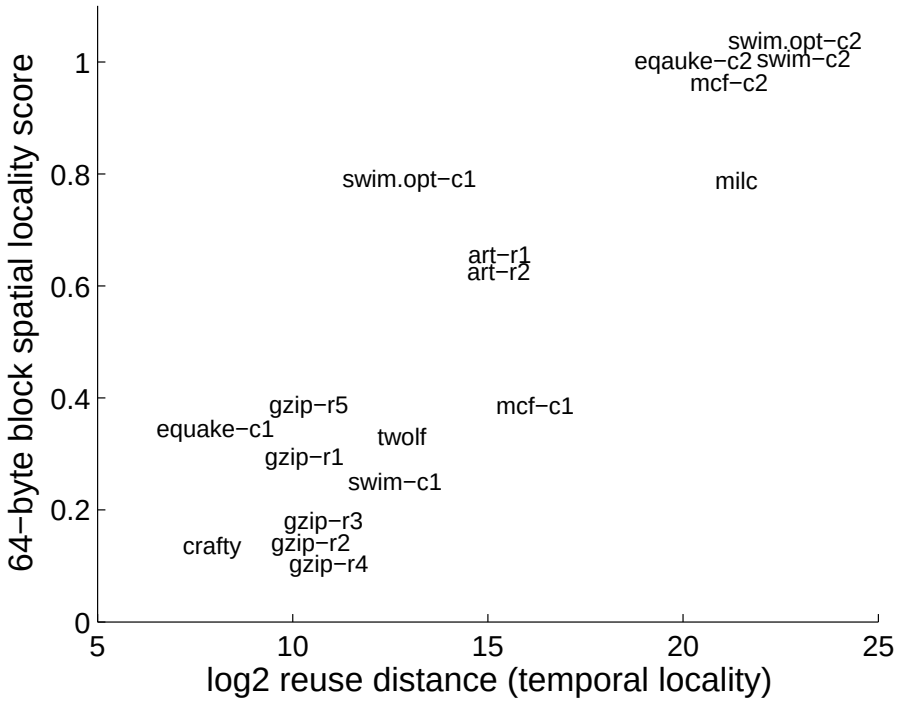


Figure 4: The spatial and temporal locality of spatial locality components.

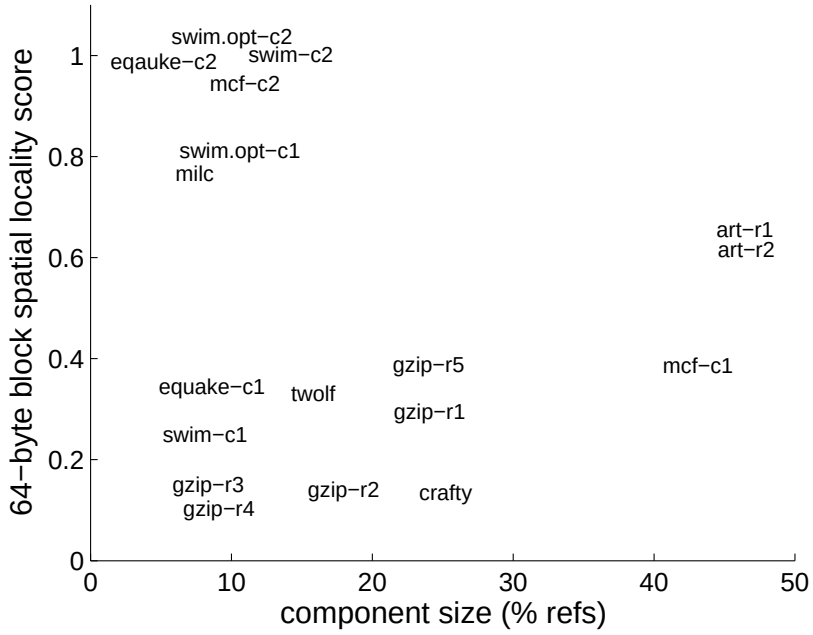


Figure 5: The spatial locality components and their sizes.

quently, the spatial locality score of these components remains low for different block sizes. This indicates that our spatial locality quality score is consistent for different block sizes.

Different inputs Table 3 compares the two components of *swim* with ref and train dataset as input. With a smaller input, the reuse distance of both components decreases. However, the spatial locality and size of both components do not change much. Similar conclusion can be drawn from the other benchmarks such as *art* and *gzip*. The spatial locality analysis can correctly identify the components and their spatial locality score, independent of the program's input. However, we do need an input larger enough to differentiate different components.

component	spatial locality	size (%)
ref input		
swim-c1	0.25	5.1
swim-c2	1	5.2
train input		
swim-c1	0.25	5.1
swim-c2	0.98	5.3

Table 3: Components of *Swim* for ref and train input

5 Possible Uses

User tuning Locality tuning is different from computation tuning. For example pipeline stalls can be addressed by reorganizing the code where the stalls happen. However, to remove cache misses one often needs to group data accesses from remote parts of a program. While commercial tools identify the frequency and location of cache misses, they do not identify program-level causes and solutions. For example, an Intel article on VTune explains memory tuning by showing data blocking in two example loops but does not explain how to handle other programs [19]. HPCView shows global data reuse, identifies data that cause cache conflicts, but it does not report on the degree of cache block reuse or the effect of hardware prefetching [18].

With the new metric, a profiling tool can identify data structures that exhibit poor spatial locality during some of the uses. It can identify where the data structure is defined, the data allocated and accessed. It is important to distinguish different access patterns because the same data may have excellent spatial locality in one part of a program but not in another part.

5.1 Superpage Management

On modern machines, the cost of TLB misses is a significant overhead, and the cost is expected to rise sharply as applications move to 64-bit address space. The idea of superpages was proposed in early 1990s to support large working sets by letting a program use pages up to tens of megabytes in size. Early studies explored basic allocation schemes (based on reservation [35] or relocation [29]) and special hardware support of non-continuous superpages [12, 35]. Restricted uses have been included in modern processors from IBM, Intel, and Sun and in operating systems including Linux, AIX, and Solaris.

Previous studies managed program code, heap, and stack separately but the same policy is applied within each category. The new metric may help to refine the management of heap data, dividing them based on their usage. In particular, it can be used to determine the best superpage size. As the component quality varies among data regions, so do the benefit and the cost of making a superpage. An off-line analysis can examine the tradeoffs between the TLB miss reduction and the memory increase for fine-grained data regions and then develop new heuristics so that the available physical memory can be used to make the most gain. This will augment the previously developed systems, which assumed enough physical memory in their experiments [5, 26].

5.2 Data-based Cache Hints

One of the new extensions in computer architecture, in particular on IBM Power 5 and Intel Itanium processors, are cache hints. For a memory instruction, the hint indicates whether the loaded cache block should be kept in a level of cache or be replaced to make room for later data. It allows software-steered cache management on a real machine. Beyls and D'Hollander used reuse distance-based models to insert cache hints and obtained 9% performance improvement on average on Itanium for a set of integer and floating-point benchmarks. The results compared favorably with a set-based analysis in the compiler they developed [1]. One limitation of profiling-based analysis is that the behavior of a program may change in the actual execution. Fang et al. gave a solution, which constructed a model from multiple training runs and used the model to predict the change of reuse distance in other runs [11]. Empirically, they found over 90% accuracy on average for the per-instruction miss rate for both integer and floating-point Spec2k benchmarks for both fully and set (one-way and up) associative cache.

During an execution, a load instruction may access data with different locality, so the best cache hint may change at different times. Since the new metric can identify different patterns for different data at different parts of a program, it may be used for data-based cache hint insertion.

6 Related Work

Spatial locality was first modeled using the notion of working set. Bunt and Murphy considered the spatial locality in two ways [2]. By examining different page sizes, the first model quantified the change in the reuse signature in terms of its fit to a Bradford-Zipf distribution. The second model measured the frequency when a group of h pages were accessed by n consecutive times. The locality increased with h, and the smaller the working set, the faster the locality increased. Somewhat similar to the first model, many studies have examined the effect of different page sizes and cache block sizes. Recently, Weinberg et al. defined a spatial locality score, which is based on the closeness of data elements accessed in each time window of size w [38]. It is a combination of the working set and the spatial distance.

Like the spatial locality score, the quality here ranges between 0 and 1 with 0 being the worst locality and 1 being the best. Unlike previous studies, the locality here is defined on components rather than the whole program. The model is based on the reuse distance rather than time windows.

Ding and Zhong used a similar component-based analysis for predicting the change of whole-program locality across data inputs. They divided all data accesses of a program into a fixed number of bins and modeled the pattern in each part by examining the reuse signature from two different runs [9]. Shen et al. improved their method by allowing mixed pattern inside each bin and by using linear regression on more than two inputs [31]. They reported an average accuracy of over 94% when predicting the (change in) reuse signature in a new input. The technique was later used to predict the cache miss rate across program inputs [42]. Marin and Mellor-Crummey gave an adaptive method based on recursive division for partitioning the data accesses of a program [21]. They augmented the model to predict not just the miss rate but program performance and to consider non-fully associative cache [22, 32]. Fang et al. improved the precision of the method of Ding and Zhong by using a linear distribution (rather than a uniform distribution) inside each bin (for more on this study see Section 5.2) [11].

While these studies developed parameterized models for different access patterns, the goal was to better model their combined effect rather than to study them individually. They did not distinguish between temporal and spatial locality.

Dependence analysis analyzes data reuses in loop nests and can estimate the number of capacity misses in scientific programs [6,10,14,28]. Other researchers used various types of array sections to measure data access in loops and procedures [3, 4, 16, 20, 36]. Wolf and Lam [40] and McKinley et al. [24] used notions of self and group reuse to distinguish between temporal and spatial reuse in a loop body. One limitation of dependence analysis is that it does not model cache interference caused by the data layout. An early technique used efficient heuristics [13]. Recent studies used precise (though worst-case super-exponential) methods [1,7,15] or their fast approximation through sampling [37, 41]. These methods are more powerful but they are limited to programs written in loop nests with regular array subscripts. Past profiling-based techniques can measure the miss rate for general program executions, but they do not predict how locality changes with program inputs (for examples see [17, 23, 32]).

7 Summary

In this paper, we have built a quantitative model of spatial locality. The new model considers both caching and prefetching. It is hardware and input independent. The new model can reveal different locality pattern within the same application, which we call components. We have tested our model on 8 SPEC benchmark programs. Among their 18 components, 2 of them have good temporal locality, 6 of them have good spatial locality. Four programs contain significant components of poor spatial locality, and one program maybe have spatial locality problems. The new model may have uses in user performance tuning, superpage management, and cache hint insertion.

References

[1] K. Beyls and E. D'Hollander. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 51(4):223–250, 2005.

[2] R. B. Bunt and J. M. Murphy. Measurement of locality and the behaviour of programs. *The Computer Journal*, 27(3):238–245, 1984.

[3] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceed-*

ings of the SIGPLAN '86 Symposium on Compiler Construction, Palo Alto, CA, June 1986.

[4] D. Callahan, J. Cocke, and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5(5):517–550, Oct. 1988.

[5] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski. Multiple page size modeling and optimization. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, St. Louis, MO, 2005.

[6] C. Cascaval and D. A. Padua. Estimating cache misses and locality using stack distances. In *Proceedings of International Conference on Supercomputing*, San Francisco, CA, June 2003.

[7] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, UT, 2001.

[8] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[9] C. Ding and Y. Zhong. Predicting whole-program locality with reuse distance analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

[10] T. Fahringer. Estimating cache performance for sequential and data parallel programs. In *Proceedings of HPCN*, Vienna, Austria, 1997.

[11] C. Fang, S. Carr, S. Onder, and Z. Wang. Instruction based memory distance analysis and its application to optimization. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, St. Louis, MO, 2005.

[12] Z. Fang, L. Zhang, J. Carter, S. McKee, and W. Hsieh. Reevaluating onlin superpage promotion with hardware support. In *Proceedings of HPC*, 2001.

[13] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, Aug. 1991. Springer-Verlag.

[14] K. Gallivan, W. Jalby, and D. Gannon. On the problem of optimizing data transfers for complex memory systems. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[15] 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), 1999.

[16] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[17] M. D. Hill. *Aspects of cache memory and instruction buffer performance.* PhD thesis, University of California, Berkeley, November 1987.

[18] J. Mellor-Crummey and R. Fowler and G. Marin and N. Tallent. Hpcview: A tool for top-down analysis of node performance. *Journal of Supercomputing*, pages 81–104, 2002.

[19] D. Levinthal. Black belt itanium 2 processor performance. Intel Software Network, http://www.intel.com/cd/ids/developer/asmona/eng/195487.htm?page=1.

[20] Z. Li, P. Yew, and C. Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):26–34, Jan. 1990.

[21] G. Marin and J. Mellor-Crummey. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of Joint International Conference on Measurement and Modeling of Computer Systems*, New York City, NY, June 2004.

[22] G. Marin and J. Mellor-Crummey. Scalable cross-architecture predictions of memory hierarchy response for scientific applications. In *Proceedings of the Symposium of the Las Alamo s Computer Science Institute*, Sante Fe, New Mexico, 2005.

[23] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[24] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[25] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach, California, October 1999.

[26] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 89–104, 2002.

[27] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007.

[28] A. Porterfield. *Software Methods for Improvement of Cache Performance*. PhD thesis, Dept. of Computer Science, Rice University, May 1989.

[29] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad. Reducing TLB and memory overhead using online superpage promotion. In *Proceedings of the International Symposium on Computer Architecture*, 1995.

[30] X. Shen, Y. Gao, C. Ding, and R. Archambault. Lightweight reference affinity analysis. In *Proceedings of the 19th ACM International Conference on Supercomputing*, Cambridge, MA, June 2005.

[31] X. Shen, Y. Zhong, and C. Ding. Regression-based multi-model prediction of data reuse signature. In *Proceedings of the 4th Annual Symposium of the Las Alamos Computer Science Institute*, Sante Fe, New Mexico, November 2003.

[32] A. J. Smith. On the effectiveness of set associative page mapping and its applications in main memory management. In *Proceedings of the 2nd International Conference on Software Engineering*, 1976.

[33] Spec cpu benchmarks. http://www.spec.org/benchmarks.html#cpu.

[34] R. A. Sugumar and S. G. Abraham. Multi-configuration simulation algorithms for the evaluation of computer architecture designs. Technical report, University of Michigan, August 1993.

[35] M. Talluri and M. D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Proceedings of the International Conference on Architectual Support for Programming Languages and Operating Systems*, 1994.

[36] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of CALL statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[37] X. Vera, N. Bernudo, J. Llosa, and A. Gonzalez. A fast and accurate framework to analyze and optimize cache memory behavior. *ACM Transactions on Programming Languages and Systems*, 26(2), March 2004.

[38] J. Weinberg, M. O. MCracken, A. Anavely, and E. Strohmaier. Quantifying locality in the memory access patterns of hpc applications. In *Proceedings of SC'05*, 2005.

[39] B. S. White, S. A. McKee, B. R. de Supinski, B. Miller, D. Quinlan, and M. Schulz. Improving the computational intensity of unstructured mesh applications. In *Proceedings of the 19th ACM International Conference on Supercomputing*, Cambridge, MA, June 2005.

[40] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[41] J. Xue and X. Vera. Efficient and accurate analytical modeling of whole-program data cache behavior. *IEEE Transactions on Computers*, 53(5), 2004.

[42] Y. Zhong, S. G. Dropsho, X. Shen, A. Studer, and C. Ding. Miss rate prediction across program inputs and cache configurations. *IEEE Transactions on Computers*, 56(3), 2007.

[43] 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 Conference on Programming Language Design and Implementation*, June 2004.