A Component Model of Spatial Locality

Xiaoming Gu
Intel China Research Center
xiaoming@cs.rochester.edu

Ian Christoper Tongxin Bai
Department of Computer Science
University of Rochester
{ichrist2,bai}@cs.rochester.edu

Chengliang Zhang
Microsoft Corporation
chengzh@microsoft.com

Chen Ding
Department of Computer Science
University of Rochester
cding@cs.rochester.edu

Abstract

Good spatial locality alleviates both the latency and bandwidth problem of memory by boosting the effect of prefetching and improving the utilization of cache. However, conventional definitions of spatial locality are inadequate for a programmer to precisely quantify the quality of a program, to identify causes of poor locality, and to estimate the potential by which spatial locality can be improved.

This paper describes a new, component-based model for spatial locality. It is based on measuring the change of reuse distances as a function of the data-block size. It divides spatial locality into components at program and behavior levels. While the base model is costly because it requires the tracking of the locality of every memory access, the overhead can be reduced by using small inputs and by extending a sampling-based tool. The paper presents the result of the analysis for a large set of benchmarks, the cost of the analysis, and the experience of a user study, in which the analysis helped to locate a data-layout problem and improve performance by 7% with a 6-line change in an application with over 2,000 lines.

Categories and Subject Descriptors C.4 [*Computer Systems Organization*]: Performance Of Systems—measurement techniques

General Terms Measurement, Performance

Keywords Spatial locality, Reuse distance

1. Introduction

Given a fixed access order, the effect of caching and prefetching depends on the layout of program data — whether the program has good *spatial locality* or not. Conventionally, the term may mean three different effects at the cache level. Here a memory block is a unit of memory data that is loaded into a cache block when being accessed by a program.

ISMM'09, June 19–20, 2009, Dublin, Ireland.

- Intra-block spatial locality — Successive memory operations access data from the same memory block, resulting in cache-block reuse.
- Inter-block spatial locality — Program operations access memory blocks that do not map to the same cache set, avoiding cache conflicts.
- Adjacent-block spatial locality — The program traverses memory contiguously, maximizing the benefit of hardware prefetching.

Intra-block and adjacent-block locality also plays a critical role in lower levels of memory hierarchy such as virtual memory and file systems where spatial locality manifests as usage patterns of memory pages and disk sectors instead of cache blocks. In this paper we focus on modeling intra-block spatial locality in a way that can be extended to adjacent-block locality. For brevity, we use the term spatial locality to mean intra-block spatial locality unless we specify otherwise.

The preceding notions of spatial locality are not quantitative enough for practical use. In particular, a programmer cannot use them to measure the aggregate spatial locality, to identify locations in a program that may benefit from locality improvement, and to identify the potential by which spatial locality can be improved.

Numerous techniques have been developed to improve spatial locality. Example models include loop cost [18] at the program level, and access frequency [22], pairwise affinity [6], hot streams [8], and hierarchical reference affinity [32, 35] at the trace level. Most techniques show how to improve locality but not how much locality can be improved. When a program does not improve, there is no general test to check whether it is due to the limitation of our technique or whether the spatial locality is already perfect and admits no improvement.

Another common metric is miss rate — if a new data layout leads to fewer cache misses, it must have better spatial locality. It turns out that miss rate is not a complete measure because one can improve spatial locality without changing the miss rate (see Section 2.4). A more serious limitation is that the metric evaluates rather than predicts: a programmer cannot easily judge the quality of a data layout without trying other alternatives. Changing data layout for large and complex code is time consuming and error prone. After much labor and with or without a positive result, the programmer returns to the starting point facing the same uncertainty. The problem is worse with contemporary applications because much of the code may come from external libraries. Poor

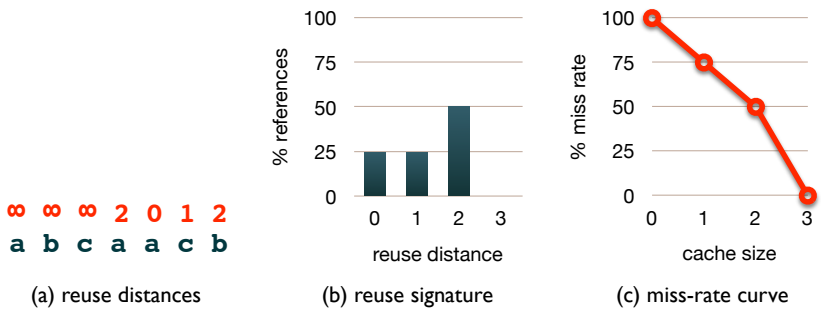


Figure 1. Example reuse distances, reuse signature, and miss-rate curve

spatial locality may arise inside a library or from the interaction between programmer code and library code.

In this paper, we define spatial locality based on the distance of data reuse. Figure 1 illustrates reuse distance as our chief locality metric. In an execution, the *reuse distance* of a data access is the number of distinct data elements accessed between this and the previous access to the same data. Figure 1(a) shows an example trace and the reuse distance of each element. The concept was defined originally by Mattson et al. in 1970 as one of stack distances [17]. The histogram of all reuse distances in an execution trace forms its *reuse signature*, as shown in Figure 1(b) for the example trace. Reuse signature can be used to calculate the miss rates for fully associative LRU cache of all sizes [17] and can be used to estimate the effect of limited cache associativity [27]. The miss rate of all cache sizes can be presented as a miss-rate curve, as shown in Figure 1(c) for the example trace.

The basic idea of the paper is as follows. A reuse signature includes the effect of both temporal and spatial locality. If we change the granularity of data and measure the reuse signature again, temporal locality should stay the same because the access order is the same. Any change in the reuse signature is the effect of spatial locality. Our new spatial model is based on this observation. To measure intra-block spatial locality, we change data-block size from half cache-block size to full cache-block size. To estimate adjacent-block spatial locality, we change data-block size from cache-block size to twice of that size.

Our model monitors the change of every reuse distance. The precision allows an analysis tool to identify components of spatial locality. We consider two types of components. *Program components* are divided by program constructs such as functions and loops. An analysis can identify causes of poor spatial locality in program code. *Behavior components* are divided by the length of reuse distance. An analysis can focus the evaluation of spatial locality on memory references that have poor temporal locality, which is useful since these are the references that cause cache misses.

Measuring the change in every reuse distance is costly. The paper explores two ways of ameliorating the problem. The first is using small input sizes, and the second is using sampling.

The new model has a number of limitations. It assumes a fixed computation order and does not consider computation reordering, which can significantly improve spatial locality in both regular and irregular code (e.g. [11, 18, 29]). The behavior reported in training runs may or may not happen in actual executions. The location of a locality problem does not mean its solution. In fact, optimal data layout is not only an NP-hard problem but also impossible to approximate within a constant factor (if P is not NP) [21]. We intend our solution to be a part of the toolbox used by programmers.

The rest of the paper is organized as follows. Section 2 describes the new model. Section 3 describes the profiling analysis for the new model. The result of evaluation is reported in Section 4, including the cost of the analysis and the experience from a user study. Finally, Section 5 discusses related work and Section 6 summarizes.

2. Component Model of Spatial Locality

We define spatial locality by the change of reuse distance as a function of data-block sizes. Consider contiguous memory access, which has the best spatial locality for sequential computation. Assume we traverse an array twice, and the data-block size is one array element. The reuse distance of every access in the second traversal is equal to the array size minus one. If we double the data-block size, the reuse distance is reduced to zero for every other memory access because of spatial reuse. Next we describe a model based on measuring the change of reuse distance.

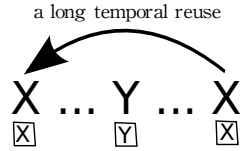
2.1 Effective Spatial Reuse

In our analysis, reuse distance is measured for different data-block sizes. We refer to them as *measurement block sizes* or *measurement sizes* in short. Our model is based on the change of reuse distance when the measurement size is doubled. Without loss of generality, consider data x and y of size b that belong to the same $2b$ block. Consider a reuse of x and its reuse distance. The reuse distance may change in two ways when the measurement size doubles from b to $2b$. The difference is whether y is accessed between the two x accesses. We call such y access an *intercept*.

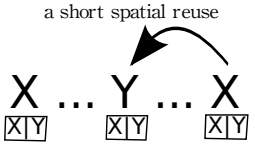
- No intercept — If y is not accessed between the two x accesses, the reuse distance is changed from the number of distinct b-blocks to the number of distinct $2b$-blocks between the two x accesses.
- Intercept — If y is accessed one or more times in between, the reuse distance is changed to the number of distinct $2b$-blocks between the last y access and the second x access.

Without intercepts, the reuse distance, measured by the number of distinct data blocks, can be reduced at most to half of its original length when the measurement size is doubled. The distance does not actually decrease if it is measured by the number of bytes. If the reuse of x is a miss in cache of b-size blocks, it likely remains a miss in cache of $2b$-size blocks.

In comparison, an intercept can shorten a reuse distance to any length. The best case is zero as it happens for accesses in a contiguous data traversal as mentioned earlier. Figure 2 shows an example intercept. At block size b, the two x accesses are connected by a temporal reuse. At block size $2b$, the intercept causes a spatial reuse and shortens the original reuse distance.



(a) The original reuse when data-block size is b



(b) The shortened reuse when data-block size is doubled to $2b$

Figure 2. Example spatial reuse. Data X and Y have size b and reside in the same $2b$ block. When the data-block size is $2b$, the original reuse in Part (a) is shortened by an intercept as shown in Part (b).

An *effective spatial reuse* is one whose reuse distance is reduced sufficiently so the access is changed from a cache miss to a cache hit. We consider two criteria for effective spatial reuse.

- Machine-independent criterion — A memory access has effective spatial reuse if its reuse distance is reduced by a factor of 8 or more when the measurement size doubles. The threshold is picked because 8 is a power of two and close to being an order of magnitude reduction.
- Machine-dependent criterion — An access has effective spatial reuse if its reuse distance is reduced below a cache-size threshold, for example, converting an L1 cache miss into an L1 hit.

2.2 Spatial-locality Score

A reuse signature is a pair $< R, P >$, where R is a series of bins with consecutive ranges of reuse distance, $r_i = [d_i, d_{i+1})$, and P is a series of probabilities p_i. Each $< r_i, p_i >$ pair shows that p_i portion of reuses have distances between d_i and d_{i+1}. In statistical terms, a randomly selected reuse distance has probability p_i to be between d_i and d_{i+1}. We use logarithmic bin sizes, in particular, $d_{i+1} = 2d_i$ ($i \geq 0$).

We use a *distribution map* to record how the distribution of reuse distances changes from one measurement size to another. Numerically it is a matrix whose rows and columns consist of bins of reuse distances. Each cell p_{ij} is a probability showing that a b-block distance in the ith distance bin (r_i^b) has probability p_{ij} to become a $2b$-block distance in the jth distance bin (r_j^{2b}). When read by rows, the distribution map shows the spread of distances in a b-block bin into $2b$-block bins. When read by columns, the map (with additional bookkeeping) shows what distances in b-block bins fall into the same $2b$-block bin.

Taking a row-based view of a distribution map, we can calculate the probability for a memory access in bin i to have effective spatial reuse. The best case (or the highest probability) is 0.5 in contiguous access, because half of the data accesses have effective spatial reuses. The spatial-locality score, *SLQ*, is this probability normalized to the best case. Normally the locality score takes a value between 0 and 1. Zero means no spatial reuse, and one means perfect spatial reuse.

For machine-independent scoring, the accesses with effective spatial reuse are whose reuse distance is reduced by a factor of 8 or more. The locality score is defined as follows.

$$SLQ(i) = \frac{\sum_{j=0\ldots i-3} p_{ij}}{0.5} \quad (1)$$

The definition is machine independent and allows spatial-locality scoring based on very small inputs. Usually small inputs are not effective in cache simulation studies. Program data may fit in cache for a small input, making memory problems invisible. A slight change in input size may cause a large change in cache performance, if a large group of reuse distances cross the cache-size threshold. The machine-independent scoring avoids the sensitivity to particular cache sizes and enables efficient analysis through the use of small program inputs.

The locality score in the machine-dependent case can be defined similarly. The score is sensitive to the program and machine parameters, but the effect of spatial reuse is measured precisely when the parameters are fixed.

2.3 Spatial Locality Components

Spatial locality score can be defined for any sub-group of memory accesses in a program. A group of memory accesses is a *component* of the overall score. We consider two types of grouping.

Program components We measure the spatial locality score for program constructs such as functions and loops. We then rank program components by their contribution to poor spatial locality.

Behavior components We group memory accesses by their reuse distance. The length of reuse distance for b-block size is considered the temporal locality at this granularity. Spatial locality scoring can be done separately for accesses with different temporal locality. If we divide temporal and spatial locality into two groups, good and bad, we have four types of locality components: the first has good temporal and good spatial locality, the second has good temporal but poor spatial locality, the third has poor temporal but good spatial locality, and finally the last has poor temporal and poor spatial locality.

The division of behavior components and the scoring may use machine-dependent or machine-independent criteria.

- Machine-independent components — We define a trough as the bin whose size is smaller than its immediate left and right neighbors. A peak is the group of bins between any two closest troughs. We consider each peak in the reuse signature as a group. The effective spatial reuse is one whose reuse distance is reduced by a factor of 8.
- Machine-dependent components — Since the basic cache parameters are used by the programmer in performance analysis, it makes sense to compute spatial locality scores based on these parameters. We consider reuse distances between sizes of two consecutive cache levels a component (adding the last level as the cache of infinite size). The effective spatial reuse is one whose reuse distance is reduced below the smaller cache size.

2.4 Adjacent-block Spatial Locality

Miss rate is not a complete measure of spatial locality when prefetching is considered. The spatial locality quality for 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 [31]. 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 Laboratory, and found that a data transformation 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), without reducing the number of (L1/L2) cache misses [31]. In contrast, two other transformations, although reducing the number of

loads and branches by 20% and 9%, resulted in a higher load latency of 4.4 cycles because the transformations caused the misses to scatter in non-adjacent memory blocks and interfered with hardware prefetching.

The result from White et al. shows the effect of adjacent-block spatial locality. With prefetching, not all cache misses are equal. The misses on consecutive memory blocks cost less. If we view two consecutive memory blocks as a unit, then adjacent-block locality becomes an instance of intra-block spatial locality for the large block size. To evaluate the effect of data layout on hardware prefetching, we compute the same spatial locality score but based on memory blocks of size twice the size of cache block. The spatial-locality score can be used to measure adjacent-block spatial locality as it is for intra-block spatial locality.

2.5 All Block Size Score

Spatial locality is so far defined by the change of reuse signature between two measurement block sizes. We can measure the change for all possible block sizes and compute an aggregate metric by weighing the score from each pair of consecutive sizes with a linear decay. In particular, the score for all block sizes is defined as:

$$SLQ = \frac{\sum_{all\ b}[\sum_{all\ i} SLQ^b(i)p_i^b]2^{-b}}{\sum_{all\ b} 2^{-b}} \quad (2)$$

where $SLQ^b(i)$ is the spatial locality score of bin i for block size b, and p_i^b is the probability of bin i for block size b. The weighting ensures that the all-block-size score is between 0 and 1. We have conducted experiments in which the measurement block size ranges from 4 bytes for integer programs or 8 bytes for floating-point programs to 2^{13} or 8KB. The cumulative score, however, is difficult to interpret because of the weighing process. We discuss all block size results in Section 4.1.3.

3. Spatial Locality Profiling

Reuse distance analysis carries a significant overhead that renders its use largely impractical for relatively long running programs. With a typical slow down factor of a couple hundred, a five-minute program takes more than twenty nine hours. The overhead of large-scale analysis is too high for use in interactive software development cycles. We have developed two ways to reduce the analysis time: to use full analysis but on a smaller input or to use sampling. We use the sampling-based tool for interactive analysis. In our future work, we plan to parallelize the profiling analysis and improve its speed by using multiple processors [13].

3.1 Full Analysis

For full analysis we augment a reuse-distance analyzer by running two instances in parallel for two block sizes. For each memory access, the analyzer computes reuse distances for the two block sizes and based on the difference, it classifies a access as an effective spatial reuse or not an effective spatial reuse. A typical reuse-distance analyzer uses a hash table to store the last access time and a sub-trace to record the last access of each data element. Our new analyzer stores two hash tables and two sub-traces, one for each block size. With the compression-tree algorithm [9], the space cost of each sub-trace is logarithmic to the total data size. The hash table size is linear to the number of data elements being accessed, which is half as many for the larger block size as for the smaller block size. We have built full analysis in two tools — one at the binary level with Valgrind and the other at the source-level with Gcc.

The full-trace analysis itself does not show which part of the program is responsible for poor spatial locality. We have extended the locality model to identify program code and data with spatial-locality problems.

CCT-based program analysis During locality profiling, the analyzer determines for each memory access, whether it is an effective spatial reuse. In addition, the analyzer constructs a calling context tree [1] by observing the entering and exit of each function at run time, maintaining a record of the call stack, and attributing the access count for each unique calling context.

For spatial-locality ranking, the analyzer records two basic metrics. The first is size, measured by the number of memory accesses. The second is quality, measured by the portion of the memory accesses that are effective spatial reuses. The final results is about the calling contexts that have the worst quality with non-trivial size, measured in both inclusive and exclusive counts. The analyzer can take customized level one and level two cache sizes as parameters to find out functions with the worst spatial locality. The Valgrind-based tool has trouble recognizing some exits of some functions, which is required for CCT. Only the Gcc-based tool is implemented with CCT.

3.2 Sampling Analysis

The overhead of full analysis comes from recording every access, passing the information to the run-time analyzer, and then computing reuse distances. To reduce the cost, we have integrated the new model to a sampling-based tool — Suggestion of Locality Optimization (SLO), developed by Beyls and D'Hollander at Ghent University [4]. SLO uses reservoir sampling [14], which has two distinct properties. First, it keeps a bounded number of samples in reservoir, so the collection rate drops as a program execution lengthens. Second, locality analysis is performed after an execution finishes. The processing overhead is proportional to the size of the reservoir and independent of the length of the trace. SLO shows consistent analysis speed, typically within 15 minutes for our tests. In the current implementation, our addition makes it take twice as long.

4. Evaluation

This section first reports a series of measurements by the full analysis (Valgrind-based tool by default) and then discusses our experience from a user study.

4.1 Full analysis results

For full analysis we have both the dynamic binary instrumentor using Valgrind (version 3.2.2) [20] and the source-level instrumentor using the GCC compiler to collect data access trace and measure reuse distances using the analyzer described in Section 3.1. We set the precision of the reuse-distance analyzer to 99.9%.

We have applied our tools on all integer programs from SPEC2000 [28] that we could successfully build and run. In addition, we tested *swim* to evaluate the effect of a data-layout transformation and *milc* to try analysis on a larger program from the new SPEC2006 [28] suite. To measure the effect of different inputs, we have collected results for multiple reference inputs and different size inputs, in particular the test and train inputs used by the benchmark set. All of the C/C++ programs are compiled using the GCC compiler with the "-O3" flag, and the Fortran programs using "f95 -O5". The version of the GNU compiler is 4.1.2. The executions, 28 in total, have different characteristics, as shown in Table 1. The data size ranges from less than 1MB to over 80MB, and the trace length, measured by the number of memory accesses, ranges from 3.4 million to 400 billion.

programs	inputs	data size (bytes)	trace len.
art test	(test)	2.4e+6	5.9e+8
art train	(train)	2.7e+6	1.5e+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+6	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	3.7e+6	1.2e+10
bzip2 train	i.compressed	3.5e+7	1.6e+10
bzip2 ref	i.source	1.0e+8	2.2e+10
crafty ref	< crafty.in	1.3e+6	5.0e+10
equake ref	<inp.in	5.0e+7	5.9e+10
gzip test	i.compressed	9.3e+5	6.6e+8
gzip train	i.combined	1.1e+7	1.0e+10
gzip ref1	input.source 60	4.2e+7	1.5e+10
gzip ref2	input.log 60	3.9e+7	7.7e+9
gzip ref3	input.graphic 60	6.5e+7	2.4e+10
gzip ref4	input.random 60	7.4e+7	1.9e+10
gzip ref5	input.program 60	5.2e+7	2.6e+10
mcf test	test	2.8e+6	3.4e+6
mcf train	train	8.2e+7	2.2e+9
mcf ref	inp.in	8.0e+7	1.8e+10
milc ref	< su3imp.in	7.2e+8	4.0e+11
parser test	test	2.1e+7	7.9e+8
parser train	train	5.3e+7	2.0e+9
parser ref	ref	8.3e+8	7.9e+10
swim ref	< swim.in	2.0e+8	9.2e+10
swim.opt ref	< swim.in	2.0e+8	9.2e+10
twolf train	train	3.0e+6	3.4e+9
twolf ref	ref	1.1e+6	1.1e+11
vpr train	train	7.0e+5	2.6e+9
vpr ref	ref	3.8e+6	2.1e10

Table 1. The input, data size, and length of 28 executions of 11 benchmarks

4.1.1 All Benchmark Results

Our analysis has identified 16 components in the 28 executions of the 12 programs with ref inputs [1], including the two components (in the reuse signature) for each run of the 4 programs, *equake*, *mcf*, *swim* and *swim.opt*, and one for each of the other 8 programs. Figure 3 shows two weighted attributes for each spatial locality component: spatial locality score and temporal reuse distance. The temporal reuse distance results are based on the block size of 64 bytes and the spatial locality scores are based on the changes of the reuse signatures with block size doubled from 64 bytes to 128 bytes. In the names of components, we use 'c' for multiple components in a single input and 'r' for the same component in 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 3 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 in *gzip* and *art*.

Based on the summarized results, we classify the locality of the 16 components into four categories.

- Components with good temporal locality — Two components *crafty* and *equake-c1* (13% of 16) have good temporal locality because they have short reuse distances (shorter than 256 blocks or 16KB).
- Components with good spatial locality — Five components (31% of 16), *equake-c2, mcf-c2, swim-c2, swim.opt-c1* and *swim.opt-c2*, have almost perfect spatial locality (a score greater than 0.97).
- Components 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 (component sizes are shown in Figure 4),
  - It has long reuse distances (poor temporal locality), and
  - It is low in spatial locality quality (poor spatial locality).

  Seven components (44% of 16), *art, mcf-c1, milc, parser, swim-c1, twolf* and *vpr* meet these conditions. They contain between 5.13% to 33% accesses. Their reuse distance ranges from 64KB to 2MB. Their spatial locality score is between 0.250 and 0.657. *Art* has identical components with two inputs, suggesting a static data access pattern and a good chance for compiler optimization.
- Components with possible spatial locality problems — The remaining two components (13% of 16) meet some but not all three conditions. *Gzip* with different inputs has the low spatial locality scores of 0.140 and 0.387. However, the component in all inputs has relatively short reuse distances. While their sizes are from 5.82% to 21.5% of their references, almost all have a reuse distance of less than 8K blocks or half mega-bytes, which fits in the level-two cache of most modern machines. The component of *Bzip2* with the reference input has relatively long reuse distances, 12K blocks, and a low locality score, 0.32, but the size is only 2.3%, below our 5% threshold. It is interesting that the two compression programs appear in the same category. They are likely tuned by their designers to make the most use of cache, hence showing a borderline status.

4.1.2 The Effect of Input Size

Table 2 compares the locality components of different size inputs. All but one program show consistency in the component size, the spatial locality score, or both. The locality component in the three inputs of *art* all has a size of 33%, although interestingly the locality score decreases. The component in the two inputs of *vpr* has similar locality scores, although the size differs. Most programs, *gzip*, *mcf*, *parser* (train and ref), and *twolf* have similar component size and locality in all inputs. For example, the first component in the three inputs of *mcf* has a spatial locality score between 0.38 and 0.41, and the second component between 0.99 and 1.00. *Bzip2* is an exception, where both the size and locality score differ significantly between the train and reference inputs.

In comparison, the temporal locality is almost never similar among the inputs of any program except for *gzip*. In *parser*, the

[1] The two versions of $swim$ are different enough to be treated as two programs.

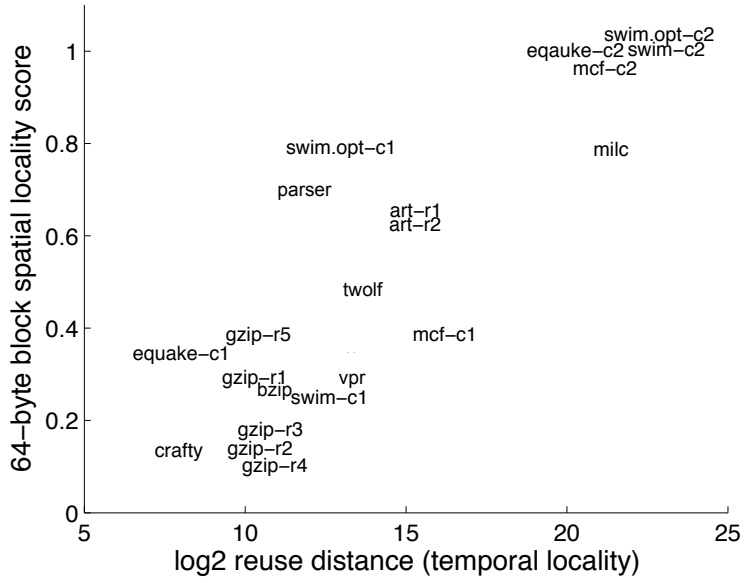


Figure 3. The spatial and temporal locality of spatial locality components. The y-axis shows the spatial-locality score, the higher the better. The score of one means perfect spatial locality. The x-axis shows the reuse distance in a logarithmic scale. The further to the right, the poorer is the temporal locality. Data layout transformations are most cost effective when targeted for components on the bottom-right half of the plot, which have poor spatial and temporal locality.

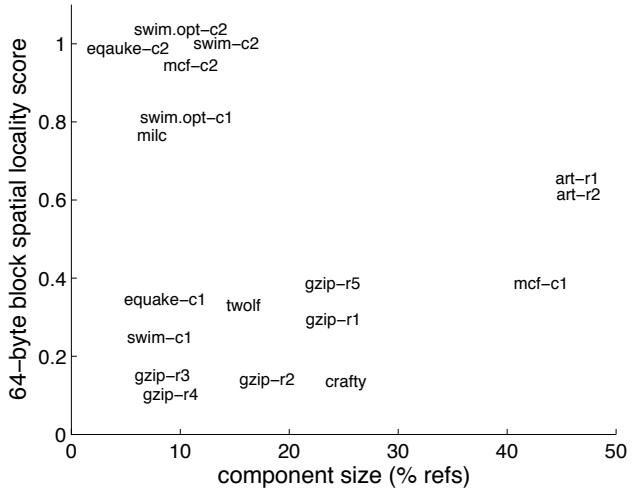


Figure 4. The size of spatial-locality components shown in Figure 3

average reuse distance sometimes decreases when the input size increases. The test inputs of *bzip2*, *twolf*, and *vpr* do not show any locality component in our analysis.

4.1.3 The Effect of Data Block Sizes

The preceding results are for a single change of data block size. We have examined the components for block sizes from 16 to 128.

As the size of data blocks increases, the spatial locality of the four components changes in three different patterns. *Art-r2* increases from 0.21 to 0.74, *swim-c1* and *gzip-r4* decrease from 0.7 to 0.13 and from 0.25 to 0.11 respectively, and *mcf-c1* alternates between 0.3 and 0.5, The lack of consistency may be due to the nature of the computations and the manual tuning by programmers. It suggests that spatial locality depends on the specified block size, which is in contrast to the stable locality quality for the same block size with different input sizes.

4.1.4 Effect of Array Regrouping for *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 designed to improve spatial locality [23, 35].

Figure 5 shows the spatial locality score for both versions when the measurement block size increases from 32 bytes to 64 bytes. 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. The plot does not group bins, so each bin is one component. There are two components with reuse distance larger than 32 blocks that are of a significant size, as pointed out on the graph.

The component model shows the effect of array regrouping on *Swim*. The first component, which accounts for 5.1% and 4.4% (bin 11 and 12) of memory accesses in two versions, has been improved from below 0.2 to close to perfect. The second component is almost identical (0.99) for the two versions. The early result shows that array regrouping improved performance by *14%* on IBM Power4 [23]. 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 due to better spatial locality in about 4% memory accesses.

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% at the two cache levels respectively. The cache miss rates for the optimized version are 9.7% and 5.33%. Hence the performance improvement mainly benefits from fewer L1 cache misses. However, we should point out that the 6.7% reduction in L1 miss rate may

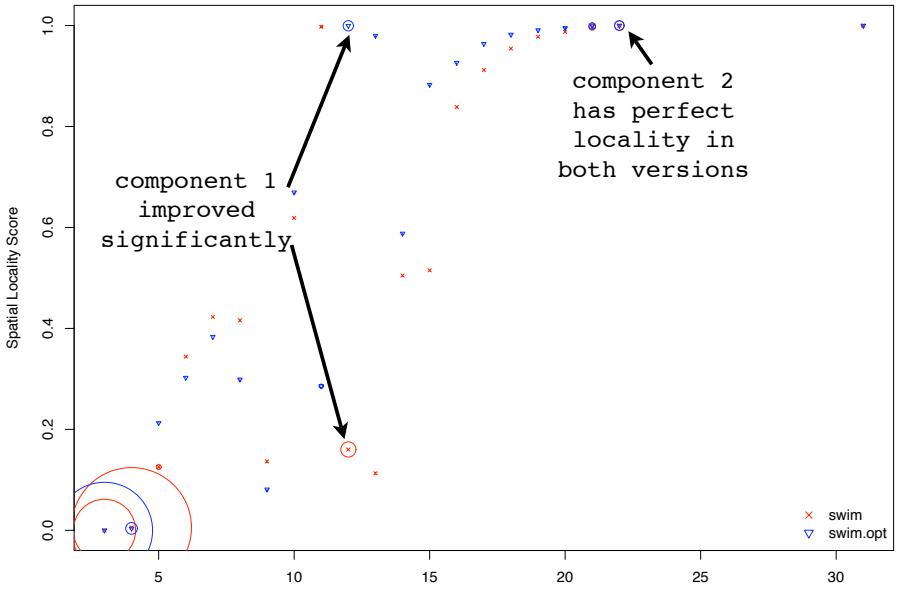


Figure 5. The effect of array regrouping on the spatial-locality score of each reuse-distance bin of *Swim*. The improvement comes mainly from better spatial locality for the first component.

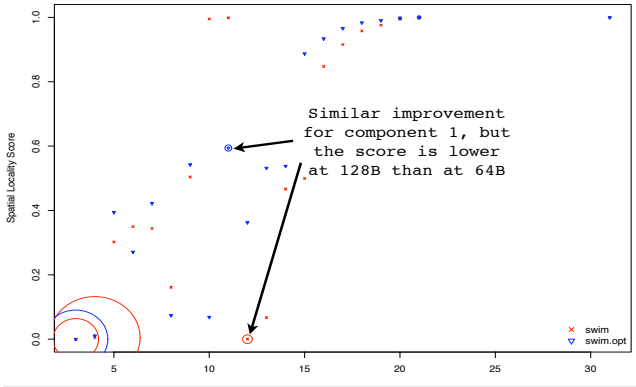


Figure 6. The effect of array regrouping on adjacent-block spatial locality, measured by the spatial-locality score when the measurement block size increases from 64 bytes to 128 bytes.

not completely explain the 8.1% performance improvement. Our spatial locality model at 128-byte block size shows good spatial reuse. This suggests that the optimized version also benefits from prefetching due to better adjacent-block locality. Figure 6 shows the effect of adjacent-block spatial locality. Most of the texts in the graph is too small to see, but they are the same as those in Figure 5.

The program *Swim* demonstrates three useful features of the model. First, the model is based on components, so it can reveal different locality patterns within the same application. Second, the model is based on different data block sizes. It can evaluate either cache-block reuse or prefetching effect. Finally, it shows the potential for improvements. After array regrouping, little opportunity remains for further improvement.

4.1.5 Analysis Time

The time cost of Gcc-based analyzer is around 350 times that of the normal execution, especially significant for long executions. For example, the reference input of *twolf* has 110 billion memory accesses, which takes 5 minutes 40 seconds in a normal execution but over 32 hours to analyze. However, as we have observed from the results in Section 4.1.2, we can identify locality components and their spatial locality quality using much smaller inputs. Table 3 compares the analysis time needed for a large-enough input and the time taken for the full analysis of the reference input.

The timing results show that at most the analysis time needed is around one hour in *twolf* and *vpr*. For other programs, *crafty* and *parser*, take half an hour; *art* and *gzip* use under 15 minutes; and *mcf* needs only 57 seconds with a very small number of accesses.

In the current implementation, we let the compiler insert a function call for each memory reference in a program. The purpose of the call is to store the data address in a buffer, and when the buffer is full, invoke reuse-distance computation in a batch. We are in the process of re-implementing the GCC-based instrumentor so it inserts inlined, and pre-optimized code instead of function calls.

4.1.6 A User Study

Computational methods are heavily used today in natural language translation (NLP) both in research and in publicly accessible (online) systems. Most methods build large-scale probabilistic models mapping the syntax and semantics structure from the source language to the target language. The translation quality depends completely on the structure and the parameters of the model, which are obtained through exhaustive training analysis over as many sentences as available. A corpus typically contains many articles in the two languages.

The NLP group at Rochester has built an analyzer [33], which is typically trained in 10 iterations, over 70,000 sentence pairs (in parallel) per iteration, at an average speed about 4 seconds per sentence pair per iteration on PC clusters (an improvement from over 1200 CPU hours per iteration reported in the original publication). For research the model is being improved as frequently as computationally possible. This analyzer consumes perhaps the most cycles on department computer servers.

Our effort was in part spurred by a request from the NLP group. They have hand-optimized the code, about 2200 lines in C++, as much as they could but were unsure about the memory perfor-

program		component size	spatial locality	temporal locality
art test		33%	1.00	23K
art train		33%	0.86	25K
art ref 1		33%	0.65	31K
bzip2 train		1.1%	0.74	23K
bzip2 ref		2.3%	0.32	12K
gzip test		3.1%	0.21	2.7K
gzip train		3.0%	0.25	2.5K
gzip ref 1		4.3%	0.33	2.0K
mcf test	1	22%	0.40	3.6K
	2	3.1%	1.00	66K
mcf train	1	37%	0.41	13K
	2	3.8%	0.99	1.1M
mcf ref	1	42%	0.38	33K
	2	2.1%	0.99	2.8M
parser test	1	1.3%	0.80	2.9K
	2	2.8%	1.00	38K
parser train	1	5.2%	0.80	14K
parser ref	1	5.3%	0.70	11K
swim train	1	5.1%	0.25	2.0K
	2	5.2%	1.00	446K
swim ref	1	5.1%	0.25	2.9K
	2	5.2%	0.99	2.1M
twolf train		8.0%	0.51	5K
twolf ref		8.0%	0.47	10K
vpr train		5.0%	0.21	3.3K
vpr ref		8.4%	0.27	8.2K

Table 2. Comparison of locality components in different size inputs

program	large enough input		ref input
	input	prof/exe time	prof/exe time
art	test	11m8s / 4s	6h23m / 5m9s
crafty	test	37m18s / 3s	18h56m / 1m15s
gzip	test	11m50s / 2s	4h24m / 32s
mcf	test	57s / 14s	8h34m / 5m50s
parser	train	38m8s / 6s	25h36m / 3m58s
twolf	train	67m52s / 9s	32h11m / 5m40s
vpr	train	61m30s / 8s	9h24m / 1m27s

Table 3. Comparison of analysis time between large enough inputs and the reference inputs

mance, which they recognized as the greatest factor in running time. Once we built the Gcc-based context sensitive analyzer, we applied the tool on their code the next day. Here is a short account of what happened on that day.

Our analyzer, after hours of training in the previous evening, showed the ranked list [2]. The worst ranked function had about 10 statements, and less than 1% of their memory references had poor spatial reuse. The function was part of a library commonly used in NLP community to improve numerical stability and running speed by representing and computing floating point numbers using integer exponents. The poor spatial reuse was due to the access to different numbers and a table lookup. Working together with the NLP group, we reduced the table size by reducing the number of entries and reducing the size of each entry from 4-byte integer to 2-byte integer. The results differed only marginally — the reported likelihood numbers from the revised program were no more than $\frac{1}{2^{300}}$ different than the original. However, the running time was reduced from 40.1 seconds to 37.4 seconds for a 6-sentence run. An improvement of over 7% is obtained by only 6 lines of code change — all in the library code.

This user study demonstrates the practical value of a spatial-locality model. First, a small change in spatial locality may have significant performance impact. Second, trace-based model can be used to analyze programs of arbitrary size and complexity to capture aggregate and composite behavior. Most applications today use components from external sources, and the tool can analyze external code for users. Finally, the user interface assists a programmer who can improve an application based on high-level understanding and algorithmic changes that go beyond the limit of pure automatic techniques.

4.2 Sampling-based Tool

For sampling, we have integrated our spatial-locality analyzer into the SLO tool developed by Beyls and D'Hollander for temporal locality analysis [4]. We call the combined system SLOR. The spatial component reuses the original implementation of reservoir sampling. The methods to determine the number of samples to skip are nearly identical. The samples, on the other hand, are completely different for spatial locality analysis. SLO samples individual memory accesses, while SLOR collects samples of consecutive basic blocks for spatial locality analysis.

We have built a graphical user interface (GUI) to interactively display spatial locality information for users. It is based on the GUI system of SLO, which displays temporal locality results including reuse paths and suggestions of computation transformation [4]. The temporal results are still retained under the tab "Temporal", as shown in the upper left corner of the screen shot in Figure 7.

To present spatial locality, we have added two more tabs. The "Spatial" tab, selected in the screen shot, shows the list of ten program statements with the worst spatial locality. The ranking can be parameterized by cache sizes with or without a calling context tree, which a user can specify in text fields. The ranking is shown by the first column, colored by different degrees of redness. The table shows the location of statements, the spatial-locality score, and the contribution of these statements to the total number of poor spatial reuses. When a user selects one of the statements, the relevant code is displayed. In this example, three of the worst ten statements appear in the same loop in *Mcf*, bringing attention to the small program piece in the midst of thousands lines of code.

5. Related Work

Spatial locality was first modeled using the notion of working set. Bunt and Murphy considered two choices [5]. By examining different page sizes, the first model quantified the change in reuse signatures 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, which means that the smaller the working set is, the better the spatial locality. Somewhat similar to the first model, many studies have examined the effect of different page sizes and cache block sizes. Weinberg et al. defined a spatial locality score ranging from 0 or worst to 1 or best, which is based on physical closeness of data elements accessed in each time window of size w [30]. It uses a combination of the working set and the spatial distance. Murphy and Kogge estimated spatial locality by the portion of data used in 64-byte memory blocks in each interval of 1000 instructions [19].

Berg and Hagersten defined spatial locality without using fixed-size windows but by the change in the miss rate when the cache-block size increases [3]. To enable fast measurement, they used sampling and approximated reuse distance using time. Our model

[2] The study was done before we implemented sampling analysis.

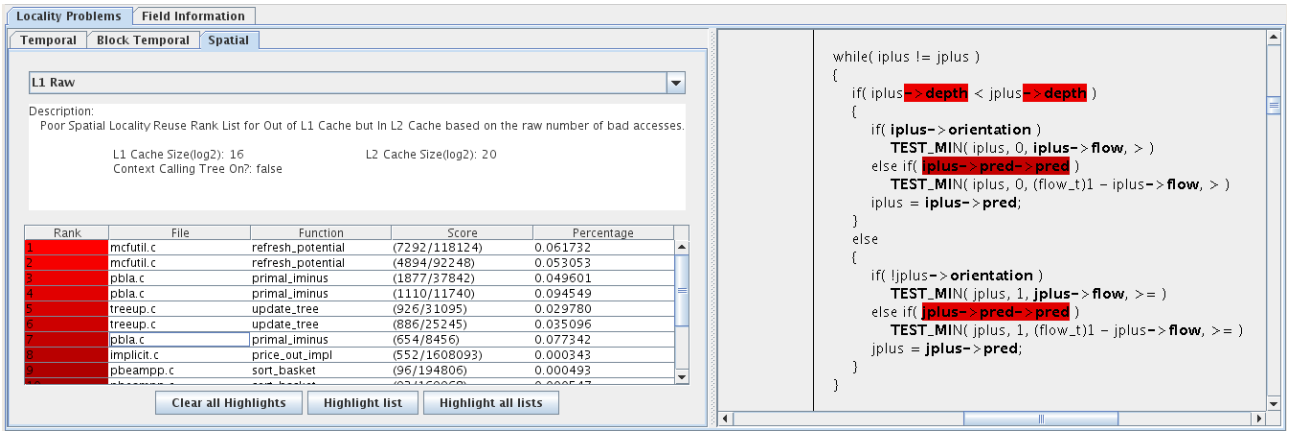


Figure 7. The GUI interface of the new analysis displaying the spatial-locality ranking of program code. The list of ten program statements in *Mcf* that contribute most to the poor spatial locality are shown in the left half of the screen shot. A user can view these statements in program code, as shown in the right half of the screen shot.

uses the same high-level idea but defines spatial locality by components rather than for the whole program. As result, we record the change of individual reuse distance and identify behavior components based on the length of the reuse distance. Berg and Hagersten found that *Swim* had good overall spatial locality, while we show in this paper that the program has a component with poor spatial locality, which could be improved and lead to significant performance gain.

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 reuse signatures 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 [26]. They reported an average accuracy of over 94% when predicting the (change in) reuse signature for a new input. The technique was later used to predict the cache miss rate across program inputs [34]. Marin and Mellor-Crummey gave an adaptive method based on recursive division for partitioning the data accesses of a program [15]. They augmented the model to predict not just the miss rate but program performance and to consider non-fully associative cache [16, 27]. Fang et al. showed that a linear distribution (rather than a uniform distribution) inside each bin gave a better precision for integer code [10].

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.

Sampling-based measurement of reuse distance has been tested as part of a continuous program optimization system [7]. The sampling is made with hardware and operating system support. The sampling accuracy is checked using a statistical technique, the Hellinger affinity kernel [7]. Approximating reuse distance with time distance is used in the SLO tool [4] and systematically studied as a statistical problem [25]. Recent improvements including extension to arbitrary-scale histograms and implementation using the memory-management unit (MMU) [24].

Function and loop based sampling have been developed, where the program is cloned and the execution switches periodically between the normal execution in the original code and the slower execution in the instrumented clone [2, 12]. Our sampling scheme is based on basic blocks instead of high-level loop and function call constructs. As a result, the samples from the previous technique align with the program structure. The alternation with the original code makes analysis almost as efficient as uninstrumented execution. For statistical profiling of data accesses, however, we need to take samples at arbitrary times during execution and take samples of an arbitrary length. The block-based sampling is used in SLO [4]. We extended it to collect not just individual memory accesses but streams of accesses from consecutively executed basic blocks.

6. Summary

In this paper, we have presented a new model of spatial locality based on how reuse distance changes as a function of data-block size. We have defined machine-dependent and machine-independent score of spatial locality and divided the overall score into either program or behavior level components. The new model is implemented in three tools. The first two performs full-trace analysis using binary- and source-level intrumentation. The third uses sampling analysis based on the SLO tool.

Using these analyzers, we have identified 16 components from 11 commonly used benchmarks. Among these 2 have good temporal locality, 5 have good spatial locality, and 7 have poor spatial locality. We have examined the effect of inputs and data block sizes and shown that analysis time can be reduced by either using smaller inputs or sampling. Most benchmarks require no more than one hour of analysis time.

We have used the model-based tool to explain the effect of a data transformation and estimate the potential for further improvement. We have developed an interactive tool for program tuning. In a user study, the tool helped to identify a small routine that had poor spatial locality. The user was able to improve program performance by 7% by changing only 6 lines of code.

Acknowledgment

We wish to thank Hao Zhang and Dan Gildea at Rochester for providing the opportunity of our user study, Yaoqing Gao and Roch Archambault at IBM Toronto Lab for their advice and feedback, Linlin Chen and Guy Steele for the help with the paper submission, and the anonymous reviewers for their comments on the presentation.

Xiaoming Gu and Chengliang Zhang were students at the University of Rochester supported by the IBM Center for Advanced Studies Fellowship during the course of this study. Ian Christo-

pher was supported by an NSF REU grant (a supplement of CCR-0219848). Additional funding came from NSF (contract no. CNS-0720796 and CNS-0509270).

References

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, 1997.

[2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.

[3] E. Berg and E. Hagersten. Fast data-locality profiling of native execution. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 169–180, 2005.

[4] K. Beyls and E. D'Hollander. Discovery of locality-improving refactoring by reuse path analysis. In *Proceedings of HPCC. Springer. Lecture Notes in Computer Science Vol. 4208*, pages 220–229, 2006.

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

[6] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, Oct 1998.

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

[8] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.

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

[11] H. Han and C.-W. Tseng. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):606–618, 2006.

[12] M. Hirzel and T. M. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *Proceedings of ACM Workshop on Feedback-Directed and Dynamic Optimization*, Dallas, Texas, 2001.

[13] K. Kelsey, T. Bai, and C. Ding. Fast track: a software system for speculative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2009.

[14] K.-H. Li. Reservoir-sampling algorithms of time complexity o(n(1+log(n/n))). *ACM Transactions on Mathematical Software*, 20(4):481–493, December 1994.

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

[16] 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 Alamos Computer Science Institute*, Sante Fe, New Mexico, 2005.

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

[18] 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.

[19] R. C. Murphy and P. M. Kogge. On the memory access patterns of supercomputer applications: Benchmark selection and its implications. *IEEE Transactions on Computers*, 56(7):937–945, 2007.

[20] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[21] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *Proceedings of ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 2002.

[22] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, Oct 1998.

[23] X. Shen, Y. Gao, C. Ding, and R. Archambault. Lightweight reference affinity analysis. In *Proceedings of the 19th ACM International Conference on Supercomputing*, pages 131–140, Cambridge, MA, June 2005.

[24] X. Shen and J. Shaw. Scalable implementation of efficient locality approximation. In J. N. Amaral, editor, *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, pages 202–216, 2008.

[25] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 55–61, 2007.

[26] 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.

[27] 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.

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

[29] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–257, San Diego, CA, June 2003.

[30] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely. Quantifying locality in the memory access patterns of hpc applications. In *Proceedings of Supercomputing*, 2005.

[31] 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*, pages 341–350, Cambridge, MA, June 2005.

[32] C. Zhang, C. Ding, M. Ogihara, Y. Zhong, and Y. Wu. A hierarchical model of data locality. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Charleston, SC, January 2006.

[33] H. Zhang and D. Gildea. Stochastic lexicalized inversion transduction grammar for alignment. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 475–482, 2005.

[34] 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):328–343, March 2007.

[35] 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.