Program Locality Analysis Using Reuse Distance

YUTAO ZHONG
George Mason University
XIPENG SHEN
The College of William and Mary
and
CHEN DING
University of Rochester

On modern computer systems, the memory performance of an application depends on its locality. For a single execution, locality-correlated measures like average miss rate or working-set size have long been analyzed using *reuse distance*—the number of distinct locations accessed between consecutive accesses to a given location. This article addresses the analysis problem at the program level, where the size of data and the locality of execution may change significantly depending on the input.

The article presents two techniques that predict how the locality of a program changes with its input. The first is approximate reuse-distance measurement, which is asymptotically faster than exact methods while providing a guaranteed precision. The second is statistical prediction of locality in all executions of a program based on the analysis of a few executions. The prediction process has three steps: dividing data accesses into groups, finding the access patterns in each group, and building parameterized models. The resulting prediction may be used on-line with the help of distance-based sampling. When evaluated on fifteen benchmark applications, the new techniques predicted program locality with good accuracy, even for test executions that are orders of magnitude larger than the training executions.

The two techniques are among the first to enable quantitative analysis of whole-program locality in general sequential code. These findings form the basis for a unified understanding of program

The article contains material previously published in the 2002 Workshop on Languages, Compilers, and Runtime Systems (LCR), 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), and 2003 Annual Symposium of Los Alamos Computer Science Institute (LACSI).

The authors were supported by the National Science Foundation (CAREER Award CCR-0238176 and two grants CNS-0720796 and CNS-0509270), the Department of Energy (Young Investigator Award DE-FG02-02ER25525), IBM CAS Faculty Fellowship, and a gift from Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding organizations.

Authors' addresses: Y. Zhong, George Mason University, Fairfax, VA; email: yzhong@cs.gmu.edu; X. Shen, College of William and Mary, Williamsburg, VA; email: xshen@cs.wm.edu; C. Ding, University of Rochester, Rochester, NY; email: cding@cs.rochester.edu.

DOI 10.1145/1552309.1552310 http://doi.acm.org/10.1145/1552309.1552310

locality and its many facets. Concluding sections of the article present a taxonomy of related literature along five dimensions of locality and discuss the role of reuse distance in performance modeling, program optimization, cache and virtual memory management, and network traffic analysis.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Optimization, compilers*

General Terms: Measurement, Languages, Algorithms

Additional Key Words and Phrases: Program locality, reuse distance, stack distance, training-based analysis

ACM Reference Format:

Zhong, Y., Shen, X., and Ding, C. 2009. Program locality analysis using reuse distance. ACM Trans. Program. Lang. Syst. 31, 6, Article 20 (August 2009), 39 pages.
DOI = 10.1145/1552309.1552310 http://doi.acm.org/10.1145/1552309.1552310

1. INTRODUCTION

Today's computer systems must manage a vast amount of memory to meet the data requirements of modern applications. Because of fundamental physical limits—transistors cannot be infinitely small and signals cannot travel faster than the speed of light—practically all memory systems are organized as a hierarchy with multiple layers of fast cache memory. On the software side, the notion of *locality* arises from the observation that a program uses only part of its data at each moment of execution. A program can be said to conform to the 80-20 rule if 80% of its execution requires only 20% of its data. In the general case, we need to measure the active data usage of a program to understand and improve its use of cache memory.

Whole-program locality describes how well the data demand of a program can be satisfied by data caching. Although a basic question in program understanding, it has eluded systematic analysis in the past due to two main obstacles: the complexity of program code and the effect of program input. In this article, we address these two difficulties using training-based locality analysis. This analysis examines the execution of a program rather than analyzing its code. It profiles a few runs of the program and uses the result to build a statistical model to predict how the locality changes in other runs. Conceptually, training-based analysis is analogous to observation and prediction in the physical and biological sciences.

The basic runtime metric we measure is *reuse distance*. For each data access in a sequential execution, the reuse distance is the number of *distinct* data elements accessed between the current and previous accesses to the same datum (the distance is infinite if no prior access exists). It is the same as the LRU stack distance defined by Mattson et al. [1970]. As an illustration, Figure 1(a) shows an example access trace and its reuse distances. If we take the histogram of all (finite) reuse distances, we have the *locality signature*, which is shown in Figure 1(b) for the example trace. For a fully-associative LRU cache, an access misses in the cache if and only if its reuse distance is greater than the cache size. Figure 1(c) shows all nonzero miss rates of the example execution on all cache sizes. In general, a locality signature captures the average locality of an

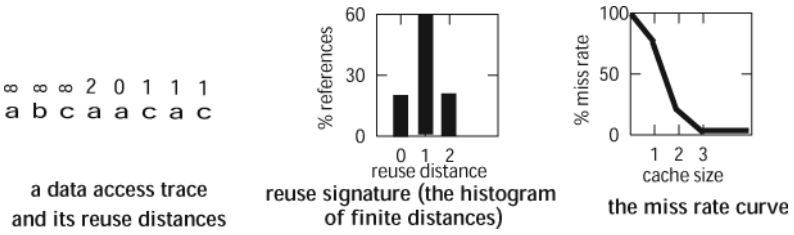


Fig. 1. Example reuse distances, locality signature, and miss rate curve.

execution from the view of the hardware as the miss rate in caches of all sizes and all levels of associativity [Mattson et al. 1970; Smith 1976; Hill and Smith 1989] and from the view of the operating system as the size of the working sets [Denning 1980].

At the program level, locality analysis is hampered by complex control flows and data indirection. For example, pointer usage obscures the location of the datum being accessed. With reuse distance, we can avoid the difficulty of code analysis by directly examining the execution or, more accurately, the locality aspect of the execution. Compilers may make local changes to a program, for example, by unrolling a loop. Modern processors, likewise, may reorder instructions within a limited execution window. These transformations affect parallelism but not cache locality. The unchanging locality cannot be seen in the reuse distance since the number and the length of long reuse distances stay the same with and without the transformations. As a direct measure, reuse distance is unaffected by coding and execution variations that do not affect locality.

Furthermore, reuse distance makes it possible to correlate data usage across training executions. Since a program may allocate different data (or the same data in different locations) between runs, we cannot directly compare data addresses, but we may find correlations in their reuse distances. More importantly, we can partition memory accesses by decomposing the locality signature into subcomponents with only short- or long-distance reuses. As we shall see, programs often exhibit consistent patterns across inputs, at least in some components. As a result, we can characterize whole-program locality by defining common patterns and identifying program components that have these patterns.

A major difficulty of training-based analysis is the immense size of execution traces. A small program may produce a long execution, in which a modern processor may execute billions of operations a second. Section 2 addresses the problem of measuring reuse distance. We present two approximate algorithms: one guarantees a relative precision and the other an absolute precision. Since data may span the entire execution between uses, a solution must maintain some representation of the trace history. The approximate solutions use a data structure called a *scale tree*, in which each node represents a time range of the trace. By properly adjusting these time ranges, an analyzer can examine the trace and compute approximate reuse distance in effectively constant time regardless of the length of the trace. Over the past four decades, there has been a

steady stream of solutions developed for the measurement problem. We review the other solutions in Section 2.3 and present a a new lower-bound result in Section 2.4.

The key to modeling whole-program locality is prediction across program inputs. Section 3 describes the prediction process, which first divides data accesses into groups, then identifies statistical patterns in each group, and finally computes parameterized models that yield the least error. Pattern analysis is assisted by the fact that reuse distance is always bounded and can change at most as a linear function of the size of the data. We present five prediction methods assembled from different division schemes, pattern types, and statistical equations. Two methods are *single-model*, which means that a locality component, that is, a partition of memory accesses, has only one pattern. The other three are *multimodel*, which means that multiple patterns may appear in the same component. These offline models can be used in online prediction using a technique called distance-based sampling.

The new techniques of approximate measurement and statistical prediction are evaluated in Section 4 using real and artificial benchmarks. Section 4.1 compares eight analyzers and shows that approximate analysis is substantially faster than previous techniques in measuring long reuse distances. Section 4.2 compares five prediction techniques and shows that most programs have predictable components, and the accuracy and efficiency of prediction increase with additional training inputs and with multimodel prediction. On average, the locality in fifteen test programs can be predicted with 94% accuracy. Programs that are difficult to predict include interpreters and scientific code with high-dimension data. Interestingly, because reuse distance is execution-based, our analyses can reveal similarities in inherent data usage among applications that do not share code.

Our locality prediction techniques are examples of a broader approach we call *behavior-based program analysis*. Conventional program analysis identifies invariant properties by examining program code. Behavior analysis infers common patterns by examining program executions. Section 5 discusses related work in locality analysis using program code and behavior metrics including reuse distance, access frequency and data streams. Locality analysis has numerous uses in performance modeling, program improvement, cache and virtual memory management, and network caching. Section 6 presents a taxonomy that classifies the uses of reuse distance into five dimensions—program code, data, input, time, and environment. Many of these uses may benefit from the fast analysis and predictive modeling described in this article.

2. APPROXIMATE REUSE-DISTANCE MEASUREMENT

In our problem setup, a trace is a sequence of T accesses to N distinct data items. A reuse-distance analyzer traverses the trace and measures the reuse distance for each access. At each access, the analyzer finds the previous time the data was accessed and counts the number of different data elements accessed in between. To find the previous access, the analyzer assigns each access a logical time and stores the last access time of each datum in a hash table. In the worst

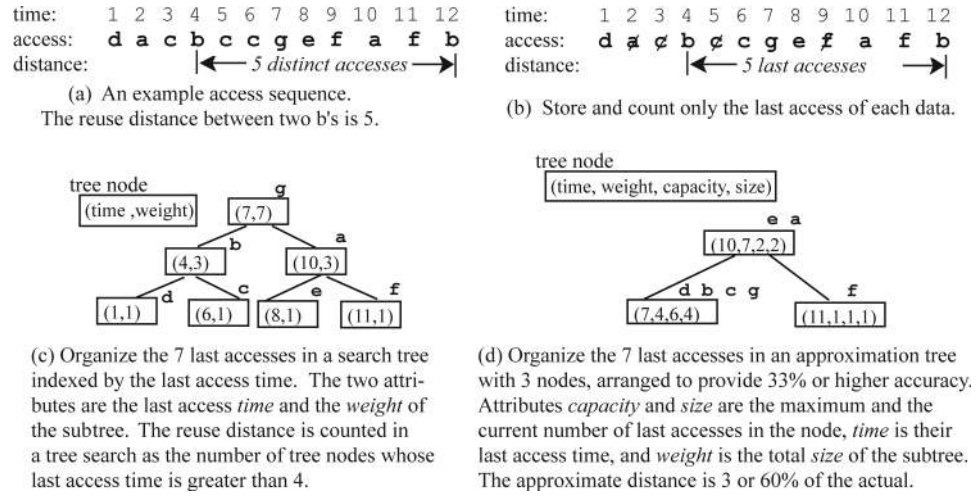


Fig. 2. An example illustrating the reuse-distance measurement. Part (a) shows a reuse distance. Parts (b) and (c) show its measurement by the Bennett-Kruskal algorithm and the Olken algorithm. Part (d) shows our approximate measurement with a guaranteed precision of 33%.

case, the previous access may occur at the beginning of the trace, the difference in access time is up to $T - 1$, and the reuse distance is up to $N - 1$. In large applications, T can be over 100 billion, and N is often in the tens of millions.

We use the example in Figure 2 to introduce two previous solutions and then describe the basic idea for our solution. Part (a) shows an example trace. Suppose we want to find the reuse distance between the two accesses of b at time 4 and 12. A solution has to store enough information about the trace history before time 12. Bennett and Kruskal [1975] discovered that it is sufficient to store only the last access of each datum, as shown in Part (b) for the example trace. The reuse distance is measured by counting the number of last accesses, stored in a bit vector rather than using the original trace.

The efficiency was improved by Olken [1981], who organized the last accesses as nodes in a search tree keyed by their access time. The Olken-style tree for the example trace has 7 nodes, one for the last access of each datum, as shown in Figure 2(c). The reuse distance is measured by counting the number of nodes whose key values are between 4 and 12. The counting can be done in a single tree search, first finding the node with key value 4 and then backing up to the root accumulating the subtree weights [Olken 1981]. Since the algorithm needs one tree node for each data location, the search tree can grow to a significant size when analyzing programs with a large amount of data.

While it is costly to measure long reuse distances, we rarely need the exact length. Often the first few digits suffice. For example, if a reuse distance is about one million, it rarely matters whether the exact value is one million or one million and one. Next we describe two approximate algorithms that extend the Olken algorithm by adapting and trimming the search tree.

The new algorithms guarantee two types of precision for the approximate distance, $d_{approximate}$, compared to the actual distance, d_{actual}. In both types, the

approximate distance is no greater than the actual distance. *Relative precision* means that the maximal error is no more than a constant fraction e of the actual distance. *Absolute precision* means that the maximal error b is a constant. Here the term "precision" means the portion of a value that can be reliably measured. We also use the term "accuracy" interchangeably. The formal definition of the two guarantees is as follows:

(1) Relative precision w/ max error e: $0 < e < 1$ and $0 \leq \frac{d_{actual} - d_{approximate}}{d_{actual}} \leq e$.
(2) Absolute precision w/ max error b: $b > 0$ and $0 \leq d_{actual} - d_{approximate} \leq b$.

Instead of using a tree node to store the last access of one data element as in the Olken algorithm, the approximate analysis uses a tree node to store a time range that may include the last accesses of multiple data elements. We call the new tree a *scale tree* and define the size of each node as the number of data elements last accessed in its time range. An example scale tree is shown in Figure 2(d), which stores the last accesses of 7 variables approximately in 3 tree nodes with sizes 2, 4, and 1, respectively (in comparison, the precise representation in Figure 2(c) requires 7 tree nodes). The size of the scale tree, measured by the number of tree nodes, equals N divided by the average node size. The error in approximation can be as large as the maximal node size. The Olken algorithm uses unit-size nodes and has full precision. The problem for the approximation algorithms is how to inflate the node size so the tree size is minimized while the measurement error is bounded.

In the following discussion, we do not consider the cost of finding the last access time. This can be performed by looking it up in a hash table, which has an $O(1)$ expected cost per access. The space cost is $O(N)$, although it can be reduced to a constant using multipass analysis [Bennett and Kruskal 1975].

2.1 Approximation with a Relative Precision

We describe the scale tree and its two types of operations. The first happens at every access to compute the reuse distance. The second happens periodically to compress the tree by coalescing the time ranges and reducing the number of tree nodes.

A node in a scale tree has 7 attributes, as defined in Figure 3. The time attribute is the end of its time range and also the search key. The size attribute is the number of data last accessed in the time range. The weight attribute is the total size of all the tree node's children. We assume the tree is a binary tree, so each node has left and right children. For the purpose of compression, we link tree nodes in a linear order using the *prev* attribute, by which each node is tied to the node of the immediately earlier time range. For example, the time range of node x is from $x.prev.time + 1$ to $x.time$. The last and most important attribute is *capacity*, which sets the upper bound of the node size and in turn determines the size of the tree and the precision of the approximation.

Let the current access be to datum d. The main routine, *ReuseDistance* shown in Figure 3, is called given as input the *last* and *current* access time. As the first step, it calls the subroutine *TreeSearchDelete*, which finds the host node

```
data declarations
    TreeNode = structure(time, weight, capacity, size, left, right, prev)
    root: the root of the tree
    e: the bound of the relative error

algorithm ReuseDistance(last, current)
    // inputs are the last and current access times
    TreeSearchDelete(last, distance)
    latest = new TreeNode(current, 1, 1, 1, ⊥, ⊥, ⊥)
    TreeInsert(latest)
    if (tree_size ≥ 4 * log_{1/(1−e)} root.weight + 4)
        TreeCompress(latest) // removes at least half of the nodes
      end if
    return distance
end algorithm

subroutine TraceSearchDelete(last, distance)
    node = root
    distance = 0
    while true
      node.weight = node.weight − 1
      if (last < node.time and node.prev exists and last ≤ node.prev.time)
          if (node.right exists)
            distance = distance + node.right.weight
          if (node.left not exists) break
          distance = distance + node.size
          node = node.left
      elseif (last > node.time)
          if (node.right not exists) break
          node = node.right
      else exit loop
      end if
    end while
    node.size = node.size − 1
end subroutine TreeSearchDelete
```

Fig. 3. Approximate analysis with a relative precision $1 - e$. Part I.

containing the last access of d and traverses the search path backward to calculate the approximate reuse distance using the sum of the subtree weights as in the Olken algorithm.

Since the last access of d is changed, *TreeSearchDelete* removes the last record by decrementing the *size* attribute of the host node and the *weight* attribute of all parent nodes. Subroutine *TreeInsert* is then called to add a new node into the tree representing the new last access, the current access. It rebalances the tree as needed. The insertion procedure is not shown since it depends on the type of search tree being used.

The loss of precision occurs at the host node. It contains a group of last accesses but we cannot know which is the last access of d. To prevent overestimating, we assume it is the last one in the group. The error is at most $size - 1$, which is at most $capacity - 1$, since $size \leq capacity$.

The initial capacity of a new node is 1. As tree nodes become dated, their capacity is adjusted by the subroutine *TreeCompression*, shown in Figure 4.

```
subroutine TreeCompress(n)
    // Initially n is the latest node in the tree
    distance = 0
    n.capacity = 1
    while (n.prev exists)
      if (n.prev.size + n.size ≤ n.capacity)
          // merge n.prev into n
          n.size = n.size + n.prev.size
          n.prev = n.prev.prev
          deallocate n.prev
      else
          distance = distance + n.size
          n = n.prev
          n.capacity = ⌊distance * e/(1−e)⌋ + 1
      end if
    end while
    Build a balanced tree from the list and update the root
end subroutine TreeCompress
```

Fig. 4. Approximate analysis with a relative precision $1 - e$. Part II.

It uses the *prev* link to traverse tree nodes in reverse chronological order and assigns the capacity of each node x to $distance * \frac{e}{1-e} + 1$, where e is the error bound ($0 < e < 1$), and *distance* is the number of distinct data accessed after x's time range. Since the maximal error at node x is $x.capacity - 1$, the maximal relative error is $\frac{x.capacity-1}{distance+x.capacity-1} = e$. *TreeCompression* will also merge adjacent time ranges as long as the combined size does not exceeds the capacity. The size of the tree is minimized for the error bound.

In the main routine, *ReuseDistance*, tree compression is triggered when the number of tree nodes exceeds the threshold $4 \log_{\frac{1}{1-e}} N + 4$, where N is the number of distinct elements that have been accessed. The following theorem shows that the compression always removes at least half of the tree nodes.

THEOREM 1. *For a trace of T accesses to N data elements, the approximate reuse distance measurement with a bounded relative error e ($0 < e < 1$) takes $O(T \log^2 N)$ time and $O(\log N)$ space, assuming it uses a balanced tree.*

PROOF. Since the tree is compressed whenever it grows to $4 * \log_{\frac{1}{1-e}} N + 4$ nodes, the number of tree nodes cannot exceed $O(\log N)$. We next show that every time it is invoked, *TreeCompress* removes at least half of the tree nodes.

Since the compression routine marches backward in time, we number the compressed nodes in reverse chronological order as $n_0, n_1, \ldots$, and n_r, with n_0 being the latest node. Assume r is an odd number (if r were even, we could add a zero-size node). Consider each pair n_{2i} and n_{2i+1}, $i = 0, \ldots, \frac{r-1}{2}$. Let $size_i$ be the combined size of n_{2i} and n_{2i+1} and $sum_i = \sum_{j=0,\ldots,i} size_j$ be the total size of nodes up to and including n_{2i+1}.

Since the capacity of the node n_{2i} is set to $\lfloor sum_{i-1} * \frac{e}{1-e} \rfloor + 1$ by the algorithm, the combined size of the node pair, $size_i$, must be at least $\lfloor sum_{i-1} * \frac{e}{1-e} \rfloor + 2$; otherwise the ith node pair should have been merged into a single node. We now have $size_0 \geq 1$ and $size_i > sum_{i-1} * \frac{e}{1-e}$. Since $sum_i = size_i + sum_{i-1}$, by

induction we have $sum_i > (1+\frac{e}{1-e})^i$ or $i < \log_{\frac{1}{1-e}} sum_i$. Let $M_{compressed}$ be the size of the tree after compression. Since $M_{compressed} = r + 1 \leq 2i + 2$ and $sum_i = N$, $M_{compressed} < 2 * \log_{\frac{1}{1-e}} N + 2$. Comparing to the starting size, we see that each compression must cut out half of the tree nodes.

Now we consider the time cost. Assume that the tree is balanced and has M tree nodes ($M \leq 4 \log_{\frac{1}{1-e}} N + 4$). The time for the tree search, deletion, and insertion is $O(\log M)$ per access. Tree compression happens periodically after a tree growth of at least $2 \log_{\frac{1}{1-e}} N + 2$ or $M/2$ tree nodes. Since one tree node is added for each access, the number of accesses between successive tree compressions is at least $M/2$ accesses. Each compression takes $O(M)$ time because it examines each node in a constant time, and the tree construction from an ordered list takes $O(M)$ time. Hence the amortized compression cost is $O(1)$ for each access. The total time is therefore $O(\log M + 1)$, or $O(\log^2 N)$ per access. □

2.2 Approximation with Absolute Precision

For a cut-off distance c and a constant error bound b, the absolute-precision algorithm divides the access trace into two parts: the *precise trace* records the last c elements accessed, and the *approximate trace* stores older accesses in a tree where the capacity of each tree node is set to $b + 1$. As a result, the measurement is accurate for reuse distances up to c and approximate for larger distances with an error no more than b. Periodically, the algorithm transfers data from the precise trace to the approximate trace.

We have described a detailed algorithm and its implementation using a B-Tree for both the precise and approximate trace [Zhong et al. 2002]. Here we generalize it to a class of algorithms. The precise trace can use a list, a vector, or any type of tree, and the approximate trace can use any type of tree, with two requirements. First, the size of the precise trace is bounded by a constant. Second, a minimal occupancy of each tree node is guaranteed. To satisfy the first requirement, we transfer the last accesses of c data elements from the precise trace to the approximate trace when the size of the precise trace exceeds $2c$. To ensure minimal occupancy, we merge two consecutive tree nodes if their total size falls below the capacity of the succeeding node. The merge operation guarantees at least half utilization of the capacity b at each node. Therefore, the number of nodes in the approximate tree is at most $\frac{2N}{b}$.

We have implemented a splay tree [Sleator and Tarjan 1985] version of the algorithm and will use only the approximate trace ($c = 0$) in the analyzer for runtime locality analysis (i.e., distance-based sampling in Section 3.5) because the analyzer has the fastest speed, as shown later in Section 4.1.

2.3 Comparison with Related Concepts and Algorithms

Mattson et al. [1970] showed that buffer memory could be modeled as a stack, if the method of buffer management satisfied the *inclusion* property in that a smaller buffer would hold a subset of data held by a larger buffer. They showed that the inclusion property is satisfied when a buffer is managed by common replacement policies including least recently used (LRU), least frequently used

Table I. The Asymptotic Complexity of Reuse-Distance Measurement

Measurement Algorithms	Time	Space
trace as a stack (or list) [Mattson et al. 1970]	$O(TN)$	$O(N)$
trace as a vector (interval tree) [Bennett and Kruskal 1975; Almasi et al. 2002]	$O(T \log T)$	$O(T)$
trace as a search tree [Olken 1981] [Sugumar and Abraham 1993; Almasi et al. 2002]	$O(T \log N)$	$O(N)$
aggregate counting [Kim et al. 1991]	$O(Ts)$	$O(N)$
approximation using time [Berg and Hagersten 2004; Shen et al. 2007]	$O(T)$	$O(1)$
approx. w/ relative precision	$O(T \log^2 N)$	$O(\log N)$
approx. w/ absolute precision	$O(T \log \frac{N}{b})$	$O(\frac{N}{b})$

T is the length of execution, N is the size of program data, s is the number of (measured) cache sizes, b is the error bound.

(LFU), optimum (OPT), and a variant of random replacement. They defined a collection of *stack distances*. These concepts formed the basis of storage system evaluation and enabled much of the experimental research in virtual memory management in the subsequent decades.

Stack distance is also used extensively in studies of cache memory. But it is not a favorable metric in low-level cache design because it does not model issues such as write-backs, cache-line prefetch, and queuing delays. Some of the drawbacks have been remedied by techniques that modeling the effect of set associativity [Smith 1976; Hill and Smith 1989] and write-backs and subblocks for fully associative [Thompson and Smith 1989] and set-associative caches [Wang and Baer 1991].

Reuse distance is the same as the LRU stack distance. It is informative to use the shorter name here because our primary purpose is program analysis. Locality as a program property exists without the presence of buffer memory or caches, so the notion of the stack is immaterial. In addition, reuse distance can be measured directly and much more quickly using a tree (or a bit vector) instead of a stack.

Since 1970, there have been steady improvements in reuse distance measurement. We categorize previous methods by their organization of the trace. The first three rows of Table I show methods using a stack, a bit vector, and a tree. Mattson et al. [1970] gave the first algorithm, which used a stack. Bennett and Kruskal [1975] observed that a stack was too slow to measure long reuse distances in database traces. They used a bit vector and built an m-ary interval tree on it. They also showed how to make the hash table smaller using multi-pass analysis. Olken [1981] gave the first tree-based algorithm. He also showed how to compress the bit vector and improve the Bennett-Kruskal algorithm to the efficiency level of his tree-based algorithm. Sugumar and Abraham [1993] showed that a splay tree [Sleator and Tarjan 1985] had better memory performance and developed a widely used cache simulator, *Cheetah*. Almasi et al. [2002] showed that by recording the empty regions instead of the last accesses in the trace, they could improve the efficiency of vector and tree based methods by 20% to 40%. They found that the modified Bennett-Kruskal algorithm was faster than the Olken algorithm with AVL or red-black trees.

Kim et al. [1991] gave an algorithm that stores the last accesses in a list and embeds markers for cache sizes. It measures the miss rate precisely but not the reuse distance. The space cost is proportional to the largest cache size, which is N if we measure for caches of all sizes. Instead of reuse distance, the access distance, that is, the logical time between the two consecutive accesses to the same datum, has been used to estimate the miss rate for caches of all sizes in StatCache [Berg and Hagersten 2004, 2005] and to statistically infer the reuse distance in time-based prediction [Shen et al. 2007]. The two statistical techniques have a linear time cost but do not guarantee the precision of the result. In addition, Zhong and Chang [2008] used sampling analysis to reduce the constant factor in the cost of reuse-distance measurement.

The literature on algorithm design has two related problems: finding the number of distinct elements in a sequence of m elements each of which is between 0 and n, and finding the number of 1's in a window of m binary digits. The goal of streaming algorithms is to solve these problems incrementally without storing the entire sequence. Alon et al. [1996] gave a simple proof (Proposition 3.7) of the previously known result that any such algorithm must use $\Omega(n)$ memory bits. For counting the number of 1's over a sliding window of size m, Datar et al. [2002] gave a deterministic algorithm with optimal space complexity $O(\log^2 m)$ bits. They extended it to count the number of distinct values in a sliding window "with an expected relative accuracy of $O(\frac{1}{\sqrt{n}})$ using $O(n \log^2 m)$ bits of memory", based on probabilistic counting [Flajolet and Martin 1983]. The relative precision algorithm in this paper can solve the same sliding-window problem deterministically with constant relative precision using $O(n \log m)$ bits in the hash table and $O(\log n \log m)$ bits in the scale tree.

The approximate measurement is asymptotically faster than exact algorithms. The space cost of the search tree is reduced from linear to logarithmic. The time cost per access, $O(\log^2 N)$, is effectively constant for any practical data size N. The improvement is important when analyzing a program at the data-element granularity like we do in program locality analysis. Next we show a lower bound result for the space cost, which suggests that approximation is necessary to obtain this level of efficiency.

2.4 A Lower Bound Result

The following theorem gives the minimal space needed by an exact algorithm.

THEOREM 2. *The space cost for accurately measuring reuse distance is* $\Omega(N \log N)$ *bits, where* N *is the largest reuse distance.*

PROOF. The trace may contain accesses to N distinct data elements. Assuming prior to a logical time k, all N elements have been accessed, the reuse distance at k depends on the relative order of the last accesses of N elements. The number of possible orders is the number of permutations of N elements or $N!$.

An accurate method must be able to distinguish between any two different permutations. Otherwise, let us assume that there exists an accurate measurement that does not distinguish between two permutations Q and R, and datum x is last accessed at a different point in Q than in R. If given the two traces Qx

and Rx, the method would not be able to show that the reuse distance of the last access (of x) is different in the two traces. This contradicts the assumption that the method is accurate. Since an exact method must distinguish between all $N!$ permutations, it must store $\Omega(\log N!)$ or $\Omega(N \log N)$ bits. □

The lower bound result has two significant implications. First, the $\Omega(N \log N)$ lower bound differs from the $\Omega(N)$ lower bound of counting the number of 1's in a sliding window [Alon et al. 1996], so the problem of reuse distance measurement is inherently harder than the sliding window problem in streaming. Second, we observe that in the method of Olken [1981], both the hash table and the search tree have $O(N)$ entries of $O(\log T)$ bits per entry, so the space cost, $O(N \log T)$ bits, is close to optimal. To match the time efficiency of the approximate algorithm, an exact algorithm must process $O(N)$ items of information in $O(\log^2 N)$ steps for each access, which seems improbable. Hence the lower bound result suggests that we may not improve exact measurement much beyond Olken's result. For a greater efficiency we may have to resort to approximation, as we have done using the scale tree.

3. LOCALITY PREDICTION

Locality prediction has three steps: dividing reuse distances into groups, analyzing their length in training executions, and constructing a statistical model to predict their length in all executions. The only parameter of the model is the *input size*. In Section 3.5, we define the input size computationally using a technique called *distance-based sampling*. In most cases it is equivalent to N, the size of the data touched by an execution. We therefore use the terms input size and program data size interchangeably.

3.1 Decomposing the Locality Signature

As we divide reuse distances into groups, it is desirable to control the range of reuse distances in a group and the size of the group. Metaphorically speaking, the range and the size can be considered the two dimensions that control an inspection lens's resolution. The range should not be too large because it may include reuse distances representing different locality. The size should not be too small because it would increase the computational cost without improving accuracy.

We represent the locality signature using two types of histograms. In a *reuse-distance histogram* (or *distance histogram*), the x-axis gives the length of reuse distance in consecutive ranges or bins, and the y-axis shows the percentage of all reuse distances that fall in each range. For each bin in the histogram, we call the range of reuse distances its *width* and the frequency of reuse distances its *size*. The width may grow in a *linear scale*, for example, $[0, 2k), [2k, 4k), [4k, 6k), \cdots$; a *logarithmic scale*, for example, $[0, 1), [1, 2), [2, 4), [4, 8), \cdots$; or a *log-linear scale*, for example, the ranges below 2048 are logarithmic and the rest are linear. Figure 5(a) shows the logarithmic scale histogram of a fluid dynamics simulation program.

Alternatively, we can sort all reuse distances, divide them into equal-size partitions, line the groups up along the x-axis, and show the average reuse

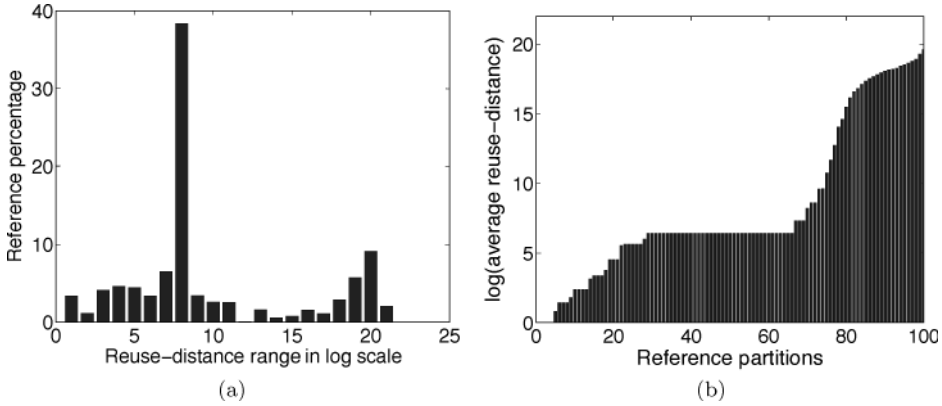


Fig. 5. Example histograms for program *SP* with input size 28^3. (a) The log-scale distance histogram shows the percentage of reuse distances (the y-axis) that fall into ranges of base-2 logarithmic scale (the x-axis). (b) The reference histogram shows the average reuse distance (the y-axis) for each 1% of reuse distances sorted by increasing length (the x-axis).

distance of each group on the y-axis. We call this a *reference histogram* and each bin a *reference partition*. Figure 5(b) shows the reference histogram in 100 partitions for the same example program. The first bin shows that the average length is 0 for the shortest 1% of reuse distances. The two histograms can be explained using nomenclature from probability theory. If we view reuse distance as a random variable, the distance histogram, for example, Figure 5(a), is the density function, and the reference histogram, for example, Figure 5(b), is the transpose of the cumulative density function.

The two types of histograms have complementary properties for behavior decomposition. With distance histograms, we can easily control the range of the reuse distances in each group but not the size of the group. With reference histograms, all groups have the same size but the range of reuse distances in a group can be arbitrarily large.

For locality prediction, the reference histogram has two important advantages over the distance histogram. First, it isolates the effect of nonrecurrent computations such as the initialization code before the main computation loop. When the input size is sufficiently large, the effect of the nonrecurrent computation diminishes into a single partition in the reference histogram. The second is to balance between information loss and modeling efficiency. When many reuse distances have a similar length, the reference histogram may divide them to increase precision. When a few reuse distances cover a wide spread in length, the reference histogram uses large ranges to reduce the number of groups. The size of the group determines the granularity and the cost of prediction. A group size of 1% means that we analyze only 100 bins, and the error in a bin does not affect more than 1% of the overall accuracy.

In our implementation, we generated the distance histogram using a log-linear scale. The bins' sizes were powers-of-2 up to 2048 and each remaining bin had a size of 2048. We compute the average distance of each bin and use it to convert the log-linear distance histogram to the reference histogram. Our

reference histogram has 1000 bins. We will evaluate prediction accuracy using the three types of histograms: the reference histogram, the log-linear distance histogram, and the logarithmic distance histogram. The last type is usually one or two orders of magnitude more compact than the first two types.

3.2 Constant, Linear, and Sublinear Patterns

Patterns are defined for each group of reuse distances. Let the groups be $\langle g_1, g_2, \ldots, g_B \rangle$ for one execution and $\langle \hat{g}_1, \hat{g}_2, \ldots, \hat{g}_B \rangle$ for another, where B is the number of groups. Let the average reuse distances of g_i and $\hat{g}_i$ be d_i and $\hat{d}_i$. Let s and $\hat{s}$ be the input size of the two executions. We find the closest linear function that maps the input size to the reuse distance. Specifically, we find the two coefficients, c_i and e_i, that satisfy the following two equations.

$$d_i = c_i + e_i * f_i(s) \tag{1}$$

$$\hat{d}_i = c_i + e_i * f_i(\hat{s}), \tag{2}$$

where f_i is the pattern function. Once we define common patterns f_i, the problem becomes one of linear regression.

Since *the largest reuse distance cannot exceed the size of program data*, the pattern function f_i can be at most linear and cannot be a general polynomial function. We consider the following five choices of f_i:

$$0;\ \ s;\ \ s^{1/2};\ \ s^{1/3};\ \ s^{2/3}.$$

We call the first, 0, the *constant pattern*. A group of reuse distances has a constant pattern if their average length does not change with the input. We call the second, s, the *linear pattern*. A bin i has a linear pattern if the average distance changes linearly with the program input size, i.e. $\frac{d_i - c_i}{\hat{d}_i - c_i} = e_i \frac{s}{\hat{s}}$, where c_i and e_i are constants. Constant and linear patterns are the lower and upper bound of the reuse distance changes. Between them are three sub-linear patterns. The pattern $s^{1/2}$ happens in two-dimensional problems such as matrix computations. The other two happen in three-dimensional problems such as ocean simulation. We could consider higher dimensional problems in the same way, although we did not find a need in our test programs.

3.3 Single-Model Prediction

In single-model prediction, each group has a single pattern. For a group of reuse distances, we calculate the ratio of their average distance in two executions, $d_i/\hat{d}_i$, and pick f_i to be the pattern function that is closest to $d_i/\hat{d}_i$. We take care not to mix sublinear patterns from a different number of dimensions. In our experiments, the dimensionality was given as an input to the analyzer. This can be automated by trying all choices and using the best fit.

Using more than two training inputs may produce a better prediction, because more data may reduce the noise from imprecise reuse distance measurement and histogram construction. We consider more inputs as follows. For each bin, instead of two linear equations, we have as many equations as the number of training runs. We use least square regression to determine the best values

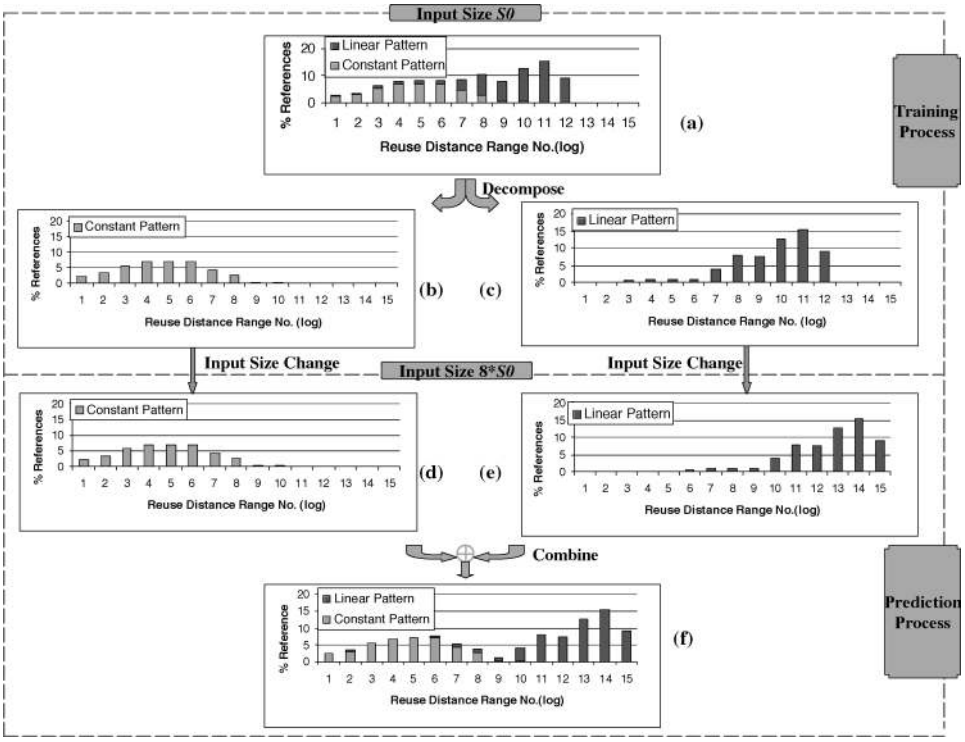


Fig. 6. An example of multimodel prediction. Part (a) is the standard histogram of input s_0. Part (b) and (c) show the composition of the constant and linear patterns in the standard histogram. Given a new input $8 * s_0$, the constant part remains unchanged, shown in (d). The distance of the linear part increases by a factor of eight, shown in (e). The prediction combines (d) and (e) to produce (f).

for the two unknowns. We will evaluate the relation between the number of training inputs and the prediction accuracy.

3.4 Multimodel Prediction

Modelmodel prediction allows a group of reuse distances to have mixed patterns. For example, some fraction of a group has one pattern, and the rest has a different pattern. In multimodel prediction, the size of the i^{th} group, $h_i(s)$, is as follows.

$$h_i(s) = \varphi_{m_1}(s, i) + \varphi_{m_2}(s, i) + \cdots + \varphi_{m_j}(s, i), \tag{3}$$

where s is the size of the input, and $\varphi_{m_1} \cdots \varphi_{m_j}$ are all possible pattern functions.

To ground the calculation on a single basis, we arbitrarily pick the result of one of the training runs as the *standard histogram*. In single-model prediction, one group in one histogram corresponds to one group in another histogram. In multimodel prediction, one group in one histogram may correspond to a piece in every group in another histogram.

We illustrate the process of multimodel prediction through an example in Figure 6. The standard histogram is shown in Part (a). The size of its input

is s_0. Other histograms are not shown, although they are used by the analysis to compute the mixing of patterns in the standard histogram. The standard histogram has 12 bins, and each bin has two models—the constant and the linear pattern. The two patterns are separated into two pieces shown in Part (b) and (c). Given another input size, $8 * s_0$, we predict the reuse distance according to the patterns. The constant pattern remains unchanged, shown in Part (d). The distance in the linear pattern is octupled by moving the bars right by 3 units along the x-axis, shown in Part (e). Finally, the prediction process combines the constant and linear pieces and produces the predicted histogram for input size $8 * s_0$ in Part (f).

We show how to derive the composition of patterns in the standard histogram, again through an example. Let s_0 be the size of the standard input, and $s_1 = 3s_0$ be the size of another training input. Let $\varphi(s, r)$ be the portion of reuse distances in range r at input s. For this example we again assume only two patterns and use φ_c for the constant pattern and φ_ℓ for the linear pattern. We use logarithmic scale ranges. The first four are $[0, 1\rangle, [1, 2\rangle, [2, 4\rangle, [4, 8\rangle$. The analysis assumes that reuse distances are distributed over a range continuously.

We compute the composition of the range $[4, 8\rangle$ in the histogram of $s_1 = 3s_0$ from the standard s_0 histogram as follows. The size of the bin $[4, 8\rangle$ in histogram s_1 consists of constant and linear parts. The size of the constant part is the same in s_1 as in s_0. The size of the linear part comes from the range $[\frac{4}{3}, \frac{8}{3}\rangle$ in s_0. The relations are shown in the next three equations.

$$\begin{aligned}
\varphi(s_1, [4, 8\rangle) &= \varphi_c(s_1, [4, 8\rangle) + \varphi_\ell(s_1, [4, 8\rangle) \\
\varphi_c(s_1, [4, 8\rangle) &= \varphi_c(s_0, [4, 8\rangle) \\
\varphi_\ell(s_1, [4, 8\rangle) &= \varphi_\ell\left(s_0, \left[\frac{4}{3}, \frac{8}{3}\right\rangle\right) = \varphi_\ell(s_0, \left[\frac{4}{3}, 2\right\rangle + \varphi_\ell\left(s_0, \left[2, \frac{8}{3}\right\rangle\right).
\end{aligned}$$

We assume the reuse distance has uniform distribution in each bin. Hence,

$$\varphi_\ell\left(s_0, \left[\frac{4}{3}, 2\right\rangle\right) = \left(\frac{2 - 4/3}{2 - 1}\right)\varphi_\ell(s_0, [1, 2\rangle) = \frac{2}{3}\varphi_\ell(s_0, [1, 2\rangle)$$

$$\varphi_\ell\left(s_0, \left[2, \frac{8}{3}\right\rangle\right) = \left(\frac{8/3 - 2}{4 - 2}\right)\varphi_\ell(s_0, [2, 4\rangle) = \frac{1}{3}\varphi_\ell(s_0, [2, 4\rangle).$$

Therefore,

$$\varphi(s_1, [4, 8\rangle) = \varphi_c(s_0, [4, 8\rangle) + \frac{2}{3}\varphi_\ell(s_0, [1, 2\rangle) + \frac{1}{3}\varphi_\ell(s_0, [2, 4\rangle).$$

After processing each bin of all training inputs in a similar manner, we obtain an equation group. The unknown variables are the size of the patterns in the standard histogram. Regression techniques are used to find the mixing that fits training results with the least error.

3.5 Distance-Based Sampling

Distance-based sampling is a heuristic for quickly estimating the input size by analyzing only the beginning of an execution. It takes samples of long reuse

distances and selects one to represent the input size. The rationale behind this scheme is the assumption that the change in input size is often proportional to the change in long reuse distances.

The sampling analysis uses a reuse-distance analyzer to monitor long-distance reuses. When a reuse distance is above a *qualification threshold*, the accessed memory location is taken as a data sample. Subsequent accesses to a data sample are recorded as access samples if the reuse distance is over a *temporal threshold*. To avoid picking too many data samples, it requires that a new data sample be at least a certain spatial distance away in memory from existing data samples. This is the *spatial threshold*. The sampling scheme requires certain manual effort to select the three thresholds for each program, although the threshold selection can be automated [Shen et al. 2007].

In a sequence of access samples, we define a *peak* as a time sample whose value is greater than that of its preceding and succeeding time samples. The analysis records the first k peaks of the first m data samples. A user evaluates these peaks in locality prediction and chooses the best one to represent the input size. The choice is program dependent but identical for all executions of the same program.

For most programs we have tested, it is sufficient to take the first peak of either the first or the second data sample. In one program, *Apsi*, all executions initialize the same amount of data but use a different amount in computation. We use the second peak as the input size. In some other programs, early peaks do not show a consistent relation with the input size, or the best peak appears near the end of an execution. We identify these cases during training and instruct the predictor to predict only the constant pattern.

Distance-based sampling can enable online prediction for an unknown input as follows. It first builds the offline model parameterized by the input size. When the execution of the test input starts, the sampling tool creates a twin copy of the program to collect the reuse distances. The sampled version runs in parallel with the original version until it detects the input size. For sampling to work, it requires that the input of the program be replicated, and that the sampled version not produce side effects.

3.6 Limitations

Although the analysis can handle any sequential program, this generality comes with several limitations. For programs with high-dimensional data, current pattern prediction requires that the shape of the data be similar in training and prediction. It should be possible to combine the pattern analyzer with a compiler and incorporate the shape of the data as parameters in the locality model. Note that locality prediction is useful only if the program is too complex for compiler analysis; otherwise, compiler analysis should be used or combined with locality prediction (see Section 5 for a review of related techniques). An important assumption in locality prediction is that the percentage size of a group of reuse distances is the same in all executions of a program. For example, the group of the 1% shortest reuse distances in one execution corresponds to the group of the 1% shortest reuse distances in other executions of the same

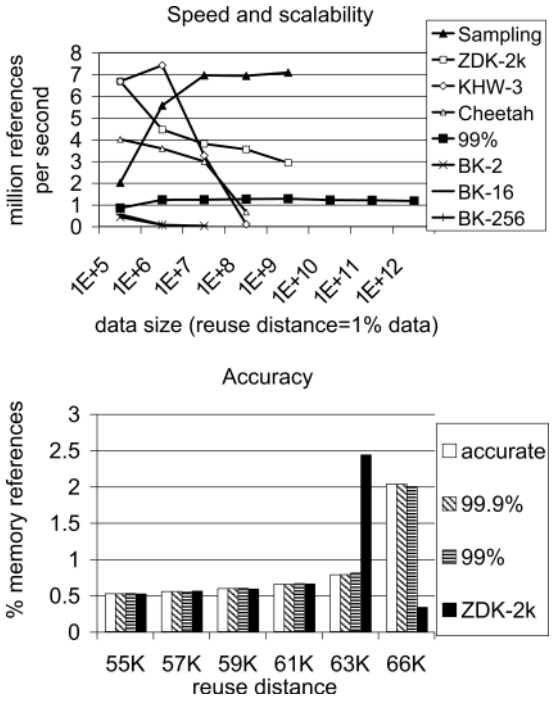


Fig. 7. A comparison of eight reuse-distance analyzers.

program. There is no logical reason that this relation has to hold in a program. We will use empirical validation by examining the accuracy of the prediction for a wide range of test programs. Finally, predicting locality does not mean predicting execution speed or execution time. The prediction gives the percentage of cache misses but not the effect on overall performance nor the total number of cache misses.

4. EVALUATION

For program analysis, we measure and predict reuse distance at the granularity of data elements. Analyzing data access at the finest granularity requires the highest efficiency and precision. The result shows the temporal locality independent of data layout. The same methods can be used to analyze temporal and spatial locality at larger data granularity such as cache blocks and memory pages (see Section 6 for a review of such studies).

4.1 Reuse Distance Measurement

Figure 7 compares the speed and accuracy of eight analyzers based on the algorithms described in Section 2. *Cheetah* [Sugumar and Abraham 1993] implements the Olken algorithm using a splay-tree. *BK-2*, *BK-16*, and *BK-256* are the bit-vector algorithm by Bennett and Kruskal [1975], implemented using k-ary trees with $k = 2, 16, 256$. These four measure reuse distance accurately. *KHW* is our implementation of Kim et al. [1991] with three markers at distances of 32, 16K, and the size of analyzed data. It classifies each reuse distance in three

bins. We test three approximate analyzers. *99%* is the relative-precision approximation with 99% accuracy. *Sampling* and *ZDK-2k* are absolute-precision approximations with maximal error $b = 2048$. *Sampling* uses a splay tree and only an approximate trace. *ZDK-2k* uses a B-tree and a mixed trace [Zhong et al. 2002]. The test program traverses N data elements twice with reuse distance equal to $N/100$. To measure only the cost of reuse-distance analysis, the hashing step is bypassed by pre-computing the last access time in all analyzers (except for *KHW*, which does not need the access time). The programs are compiled using *gcc* with full optimization (flag *-O3*) and tested on a 1.7 GHz Pentium 4 PC with 800 MB main memory.

Among the five accurate analyzers, the bit-vector methods are the slowest, *Cheetah* achieves an initial speed of 4 million memory references per second, and *KHW* with three markers is fastest (7.4 million memory references per second) for small data sizes. The accurate analyzers start to run out of the physical memory at 100 million data elements, so the three approximate analyzers become the fastest, with *Sampling* at 7 million references per second, *ZDK-2k* over 3 million references per second, and *99%* over 1 million references per second. *Sampling* and *ZDK-2k* do not analyze beyond 4 billion data elements since their implementation uses 32-bit integers.

Among the eight, the 99% precise approximate analyzer shows the most scalable performance. We use 64-bit integers in the program and test it for up to 1 trillion data elements. The asymptotic cost, $O(\log^2 N)$ per access, should be effectively linear in practice. We tested data sizes up to the 1 trillion because it is in the order of the length of a light year measured in miles. In the experiment, the analyzer ran at a near constant speed of 1.2 million references per second from 100 thousand data elements to 1 trillion data elements. The consistent high speed is remarkable considering that the data size and reuse distance differ by eight orders of magnitude. The speed was so stable that we could predict how much time our tests would take.

The lower graph of Figure 7 compares the accuracy of the approximation on a partial histogram of *FFT*. The y-axis shows the percentage of memory references, and the x-axis shows the distance on a linear scale between 55K and 66K with an increment of 2048. The *99.9%* and *99%* analyzers produce histograms that closely match the accurate histogram. The overall error is about 0.2% and 2% respectively. The analyzer with the constant error bound 2048, shown by the histogram marked ZDK-2k, misclassifies under 4% of the memory references at the far end of the histogram. If we compare the space overhead, accurate analyzers need 67 thousand tree or list nodes, *ZDK-2k* needs 2080 tree nodes (of which 32 nodes are in the approximate tree), *99.9%* needs 5869, and *99%* needs 823. The results show that approximate analyzers can greatly reduce the space cost without a significant loss of precision, and the cost and the accuracy are adjustable.

4.2 Locality Prediction

We begin by testing program locality prediction using reference histograms with 1000 bins, first for one benchmark program and then for all 15 programs.

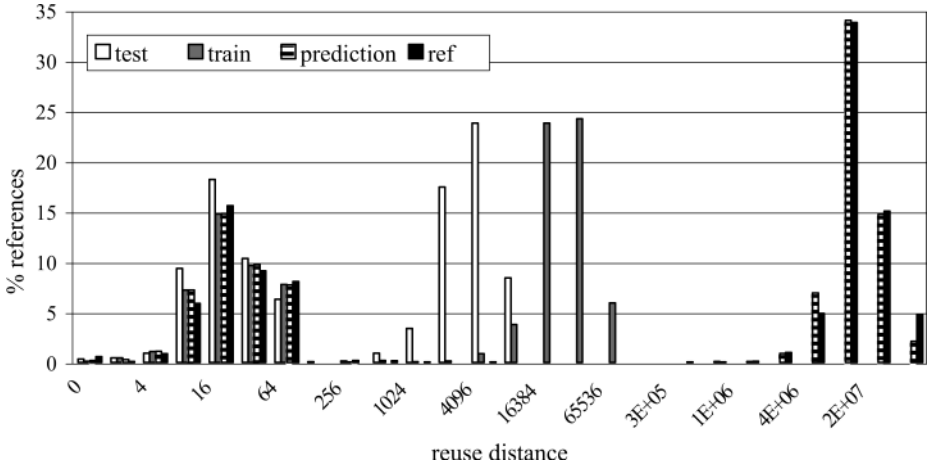


Fig. 8. Program locality prediction for *Spec2K/Lucas*.

Then we compare our full set of prediction methods and finally discuss a few notable features of whole-program locality analysis.

4.2.1 *Single-Model Prediction Based on Reference Histograms.* An illustrative example is the program *Lucas* from the SPEC 2000 benchmark suite. Based on the Lucas-Lehmer lemma, it tests the primality of very large numbers—numbers up to 2^{1000}. The program performs many large-number multiplications through specialized fast Fourier transforms coded using the C language. The program is difficult for a compiler to analyze.

The SPEC 2000 benchmark suite provides three inputs for the program. The smaller two are "test" and "train" inputs. Respectively they make 5 million reuses of 6 thousand data elements and 40 million reuses of 41 thousand data elements. The first two sets of bars in Figure 8 show their locality signatures in logarithmic scale distance histograms. The bars in the left half of their signatures show a similar distribution of short reuse distances. The bars in the right half show much longer reuse distances in "train" than in "test."

Single-model locality prediction measures the log-linear distance histogram for the two inputs, partitions the reuse distances into 1000 reference partitions, identifies constant and linear patterns, and builds a locality model parameterized by the input size measured using distance-based sampling. To test on-line prediction, we ran the third "ref" input. The execution has 644 billion accesses to 21 million data elements. After 0.4% of the execution time, distance-based sampling detected the input size. Substituting this in the model, we predicted the locality signature shown by the third group of bars in Figure 8. To compare, we measured the locality signature for the entire "ref" run, shown by the fourth set of bars.

Comparing the last two sets of bars in Figure 8, we see that the prediction largely agrees with the measurement. The two signatures match. The right half of the "ref" signature has no overlap with "test" and "train" signatures, yet the predicted signature is correct in shape and height, demonstrating the

ability by our method to predict large-scale behavior changes across the input of this program. The "ref" execution is four orders of magnitude longer and uses three orders of magnitude more data than "train" and "test" combined. The accurate prediction shows that the model is successful in characterizing the locality property at the program level, not just in a few executions.

We define prediction *accuracy* as follows. Let x_i and y_i be the size of ith bar in predicted and measured histograms. The accuracy is

$$accuracy = 1 - \frac{\sum_i |y_i - x_i|}{2}.$$

It measures the overlap between the two signatures, which ranges from 0% for no match to 100% for a complete match. For example, the prediction of *Lucas* in Figure 8 is 95% accurate.

Tables II and III list all the programs in our test set and summarize the accuracy and coverage of the single-model prediction. The test set consists of 15 benchmarks, including 9 floating-point programs and 6 integer programs. All programs came from SPEC 1995 and SPEC 2000 benchmark suites except for *SP* from the NAS benchmark suite and a textbook version of a two-dimensional *FFT* kernel. In experiments, we reduced the number of iterations in a program if it did not affect the overall pattern. Most experiments used DEC Alpha systems. We compiled the test programs with the DEC compiler using the default optimization (*-O3*). We used Atom [Srivastava and Eustace 1994] to instrument the binary code to collect the addresses of all loads and stores and fed them to our analyzer. The tool treated each distinct memory address as a data element.

The two tables have the same format, reporting each program in one row. The first two columns give the name and a short description of the program. The next column lists its reuse distance patterns, which can be constant, linear, or sublinear. Floating-point programs generally have more patterns than integer programs do. The fourth column shows the inputs used. They are all different as shown in the next three columns in terms of the number of distinct data elements, the number of data reuses per element, and the average reuse distance. The programs are listed in decreasing order of the average reuse distance.

Most inputs we used were standard test, train, and reference inputs from SPEC, with the following exceptions. For *GCC*, we picked the largest and two random inputs from the 50 files in its "ref" directory. *Tomcatv* and *Swim* had only two different data sizes. We added more inputs. The test input of *Twolf* had 26 cells and was too small. We randomly removed half of the cells in its train data set to produce a test input of 300 cells. *Applu* had a long execution time, so we replaced the reference input with a smaller one. Finally, the inputs of *Apsi* used high-dimensional data of different shapes, for which our predictor could not make an accurate prediction. We changed the shape of its largest input. We should also mention that all inputs of *Hydro2d* had a similar data size, but we did not make any change. *SP* and *FFT* did not come from SPEC, so we randomly picked their input sizes.

Columns 5 to 7 of the two tables show a range of data sizes from 14 thousand to 36 million data elements, average reuse frequency from 6 to over 300 thousand reuses per element, and average reuse distance from 15 to over

Table II. Prediction Accuracy and Coverage for Nine Floating-Point Programs

Benchmark	Description	Patterns	Inputs	Num. data elem.	Avg. reuses per elem.	Avg. dist. per elem.	Accuracy w/ data size (%)	Accuracy w/ sample size (%)	Cover-age (%)
Lucas (Spec2K)	Lucas-Lehmer test for primality	const	ref	20.8M	621	2.49E-1	**85.0**	**95.1**	**99.6**
		linear	train	41.5K	971	2.66E-1	**85.9**	**81.8**	**100**
			test	6.47K	619	2.17E-1			
Applu (Spec2K)	solution of five coupled nonlinear PDE's	const	45^3	9.33M	153	1.62E-1	**91.9**	**92.1**	**99.4**
		3rd roots	train(24^3)	1.28M	150	1.62E-1	**94.1**	**94.1**	**99.4**
		linear	test(12^3)	127K	146	1.57E-1			
Swim (Spec95)	finite difference approximations for shallow water equation	const	ref(512^2)	3.68M	33.1	4.00E-1	**94.0**	**94.0**	**99.8**
		2nd root	400^2	2.26M	33.0	4.00E-1	**98.7**	**98.7**	**99.8**
		linear	200^2	568K	32.8	3.99E-1			
SP (NAS)	computational fluid dynamics (CFD) simulation	const	50^3	4.80M	132	1.05E-1	**90.3**	**90.3**	**99.9**
		3rd roots	32^3	1.26M	124	1.01E-1	**95.8**	**95.8**	**99.9**
		linear	28^3	850K	125	9.78E-2			
Tomcatv (Spec95)	vectorized mesh generation	const	ref(513^2)	1.83M	208	1.71E-1	**92.4**	**92.4**	**99.5**
		2nd root	400^2	1.12M	104	1.67E-1	**77.3**	**99.2**	**99.3**
		linear	train(257^2)	460K	104	1.67E-1			
Hydro2d (Spec95)	hydrodynamic equations computing galactic jets	const	ref	1.10M	13.4K	2.23E-1	**98.5**	**98.5**	**100**
			train	1.10M	1.35K	2.23E-1	**98.5**	**98.4**	**100**
			test	1.10M	139	2.20E-1			
FFT	fast Fourier transformation	const	512^2	1.05M	63.7	7.34E-2	**72.6**	**72.8**	**99.6**
		2nd root	256^2	263K	57.5	8.13E-2	**95.5**	**95.5**	**99.5**
		linear	128^2	65.8K	51.4	9.04E-2			
Mgrid (Spec95)	multigrid solver in 3D potential field	const	ref(64^3)	956K	35.6K	6.81E-2	**96.4**	**96.4**	**100**
		3rd roots	test(64^3)	956K	1.42K	6.76E-2	**96.5**	**96.5**	**99.3**
		linear	train(32^3)	132K	32.4K	7.15E-2			
Apsi (Spec2K)	pollutant distribution for weather predication	const	128x1x128	25.0M	6.35	1.60E-3	**27.2**	**91.6**	**97.8**
		3rd roots	train(128x1x64)	25.0M	146	2.86E-4	**27.8**	**92.5**	**99.1**
		linear	test(128x1x32)	25.0M	73.6	1.65E-4			

Table III. Prediction Accuracy and Coverage for Six Integer Programs

Benchmark	Description	Patterns	Inputs	Num. data elem.	Avg. reuses per elem.	Avg. dist. per elem.	Accuracy w/ data size (%)	Accuracy w/ sample size (%)	Cover-age (%)
Compress (Spec95)	an in-memory version of the common UNIX compression utility	const	ref	36.1M	628	4.06E-2	**86.1**	**85.9**	**92.2**
		linear	train	279K	314	6.31E-2	**92.3**	**92.3**	**86.9**
			test	142K	147	9.73E-2			
Twolf (Spec2K)	circuit placement and global routing, using simulated annealing	const	ref(1888-cell)	734K	177K	2.08E-2	**92.6**	**94.2**	**100**
		linear	train(752-cell)	402K	111K	1.82E-2	**96.2**	**96.6**	**100**
			370-cell	227K	8.41K	1.87E-2			
Vortex (Spec95) (Spec2K)	an object oriented database	const	ref	7.78M	4.60K	4.31E-4	**95.1**	**95.1**	**100**
			test	2.58M	530	3.25E-4	**97.2**	**97.2**	**100**
			train	501K	71.3K	4.51E-4			
GCC (Spec95)	based on the GNU C compiler version 2.5.3	const	expr	711K	137	2.75E-3	**98.2**	**98.2**	**100**
			cp-decl	705K	190	2.65E-3	**98.6**	**98.6**	**100**
			explow	321K	68.3	3.69E-3	**96.1**	**96.1**	**100**
			train(amptjp)	467K	221	3.08E-3	**98.7**	**98.7**	**100**
			test(cccp)	456K	233	3.25E-3			
Li (Spec95)	Xlisp interpreter	const	ref	87.9K	328K	2.19E-2	**85.6**	**82.7**	**100**
		linear	train	44.2K	1.86K	3.11E-2	**85.8**	**86.0**	**100**
			test	14.5K	37.0K	2.56E-2			
Go (Spec95)	an internationally ranked go-playing program	const	ref	109K	124K	3.78E-3	**96.5**	**96.5**	**100**
			test	104K	64.6K	3.78E-3	**96.9**	**96.9**	**100**
			train	86.1K	2.68K	2.02E-3			
						average	**88.6**	**93.5**	**99.1**

5 million. The longest trace is generated by the third input of *Twolf* and has over 130 billion memory references. No two inputs are similar in data size or execution length. The maximal reuse distance is very close to the data size in all programs.

We use three different input sizes for all programs except for *GCC*. Based on the two smaller inputs, we predict the largest input. We call this forward prediction. The prediction also works backwards: based on the smallest and the largest inputs, we predict the middle one. Locality in all executions can be thus predicted by extrapolation and interpolation. The prediction accuracy is shown by the 8^{th} and 9^{th} columns. The former, marked "Accuracy w/ data size," gives the prediction accuracy when using the number of distinct data elements as the input size. The latter, marked "Accuracy w/ sample size," gives the accuracy when using distance-based sampling.

For most benchmarks, the two columns give comparable results, which indicates a proportional relation between the input size and the data size. One exception is *Apsi* in Table II. For different input parameters, the program initializes the same amount of data but uses different portions of the data in computation. The prediction accuracy is only 27% using the data size but over 91% using distance-based sampling. In general, prediction based on sampling yields a higher accuracy.

Both forward and backward predictions are fairly accurate. Backward prediction is generally better except for *Lucas*—because the largest input is three orders of magnitude larger than the medium-size input—and for *Li*—because only the constant pattern is considered by the prediction. Among all prediction results, the highest accuracy is 99.2% for the medium-size input of *Tomcatv*, and the lowest is 72.8% for the large-size input of *FFT*. The average accuracy is 93.5%.

The last column shows the prediction coverage. The coverage is 100% for programs with only constant patterns because they need no sampling. For the others, the coverage starts after the input size is found in the execution trace. Let T be the length of the execution trace, and P be the logical time of the discovery; the coverage is $1 - P/T$. For programs using a reduced number of iterations, T is scaled up to the length of the full execution. To be consistent with other SPEC programs, we let programs *SP* and *FFT* have the same number of iterations as *Tomcatv*. Data sampling uses the first peak of the first two data samples for all programs with non-constant patterns except for *Compress* and *Li*. *Compress* needs 12 data samples. It is predictable in this test because it repeats compression multiple times. The results from program phase analysis show that *Gzip*, which uses the same algorithm as *Compress*, has the same locality when compressing files of different sizes and content [Shen et al. 2007]. *Li* has random peaks that cannot be consistently sampled. We predict *Li* based only on the constant pattern. The average coverage across all programs is 99.1%.

The actual coverage is smaller because the instrumented program (for sampling) runs slower than the original program. Our fastest analyzer causes a slowdown of 20 to 100 times. In the worst case, we need a coverage of at least 99% to finish prediction before the end of the execution. Fortunately, the low coverage happens only in *Compress*. Without *Compress*, the average coverage

Table IV. Five Methods of Locality Prediction

Models	*Single*	*Single*	*Multiple*	*Multiple*	*Multiple*
histogram	reference	reference	distance	distance	reference
histogram x-axis	log-linear	log-linear	logarithmic	log-linear	log-linear
num. inputs	2	3+	3+	3+	3+
num. patterns per bin	1	1	2+	2+	2+

Table V. Comparison of the Accuracy of Five Prediction Methods

	Single Model		Multi-model, 3+ Inputs			
	ref. hist.	ref. hist.	dist. hist.			Total
Bench-mark	2 inputs	3+ inputs	logarithmic	log-linear	ref. hist.	Inputs
Applu	92.06	97.40	93.65	93.90	90.83	6
Swim	94.02	94.05	84.67	92.20	72.84	5
SP	90.34	96.69	94.20	94.37	90.02	5
FFT	72.82	93.30	93.22	93.34	95.26	3
Tomcatv	92.36	94.38	94.70	96.69	88.89	5
GCC	98.61	97.95	98.83	98.91	93.34	4
Avg.	**90.04**	**95.63**	**93.21**	**94.90**	**88.53**	**4.7**

is 99.73%, suggesting 73% time coverage in online prediction on average. Even without a fast sampler, the prediction is still useful for long running programs and programs with mainly constant patterns. Six programs (or 40% of the test suite) do not need sampling.

4.2.2 *Multimodel Prediction.* Multimodel prediction allows us to examine three aspects of locality prediction in more depth: the use of multimodel prediction and all three types of histograms, the effect of using more than two training inputs, and prediction using very small inputs.

We compare five prediction methods listed in Table IV. The first two are single-model prediction using two training inputs and more than two inputs. Both use reference histograms computed from log-linear distance histograms. The next three are multimodel prediction using all three types of histograms: the reference histogram and the distance histogram in logarithmic and log-linear scale. All methods use sampling to measure the input size.

We restrict our attention to six test programs in Table V, which have multiple inputs and have some significant errors in single-model prediction. The second and third columns show that regression analysis on multiple inputs improves prediction accuracy. The largest improvement is from 73% to 93% in the case of *FFT*. Across the six programs, the average accuracy is raised from 90.0% to 95.6%.

The multimodel prediction using the logarithmic histogram is the most efficient among all methods, using merely 20 or fewer bins. The average accuracy, 93%, is comparable to the accuracy obtained from much larger histograms. However, low accuracy may result because the logarithmic scale produces large ranges that may hide important details. In one execution of *Swim*, 12% of the reuse distances occupy a narrow range between 260 and 280. Our prediction method assumes a uniform distribution in each range, so the accuracy is only 85% with logarithmic ranges, compared with 92% with log-linear ranges.

Table VI. Accuracy for SP with Small-Size Inputs

Largest Training	Testing Size	Single-Model 2 Inputs	Single-model 3+ Inputs	Multi-model Log Scale	Multi-model Log-linear Scale
8^3	10^3	79.61	79.61	85.92	89.50
	12^3	79.72	75.93	79.35	82.84
	14^3	69.62	71.12	74.12	85.14
	28^3	64.38	68.03	76.46	80.30
10^3	12^3	91.25	87.09	84.58	90.44
	14^3	81.91	83.20	78.52	87.23
	16^3	77.28	77.64	76.01	84.61
16^3	28^3	75.93	74.11	77.86	83.50

Multimodel prediction with reference histograms has the lowest average accuracy among the five methods, although it makes the best prediction for one program, *FFT*.

4.2.3 *Prediction Using Small Inputs.* An important assumption in all our prediction methods is that the composition of patterns remains constant across all executions. The assumption appears operable as shown by the accurate prediction we have observed so far. If we decrease the input size, the effect of nonrecurrent parts in a program, for example the initialization before a loop, becomes significant. Next we pick one program, *SP*, artificially reduce the input size, and evaluate our four best performing predictors.

Table VI shows the prediction accuracy when the size of the largest training run is reduced to 1.6%, 3%, and 13% of the size used previously in Table II. The two multimodel methods are up to 16% more accurate than the two single-model methods. Multimodel prediction using log-linear distance histograms is the most accurate, with accuracy ranging from 80% to 90%. The average accuracy is 7% higher than the next best method. Multimodel prediction using the logarithmic scale histogram does not perform as well for small inputs.

The ability of multimodel prediction to use very small inputs is useful in two cases. First, the training time is proportional to the size of the training runs, so very small training runs lead to very fast locality analysis. Second, it is often unclear how to determine when an input size is large enough, so the prediction is more reliable if it can maintain good accuracy across most input sizes.

4.3 Understanding Whole-Program Locality

Locality is considered a fundamental concept in computing because to understand a computation we must understand its use of data. The predictive models described in this article provide a new definition of locality that is quantitative, verifiable, whole-program based and applicable to any sequential system. It opens new ways to examine the active data usage in complex computations. We discuss several unique features of this work that have significant implications in how a programmer can better understand this important yet often elusive concept.

4.3.1 *Quantifying Locality as Patterns of Change.* Profiling has long been used to find invariant program properties. Examples include most used variables and functions [Wall 1991] and recurring program path [Hsu et al. 2002]

for feedback-guided optimization, "hot" memory instruction streams for runtime optimization [Chilimbi 2001b], and function-level [KleinOsowski and Lilja 2002] and general statistics [Eeckhout et al. 2002] for workload characterization. The constant pattern in this article represents invariant locality. In 15 programs, 4 have only the constant pattern. Its presence in the other 11 programs ranges from 28% in *Apsi* to 84% in *Twolf*. The average is 55%. For locality analysis, however, the constant pattern may be the least important because reuse distances in a constant pattern are usually short and do not cause misses in a large cache.

To fully characterize locality, locality prediction extends the use of profiling analysis to capture behavioral variations between executions. The analysis is aided by the property that reuse distance measures the recurrence independent of the instruction and data addresses involved in the data access.

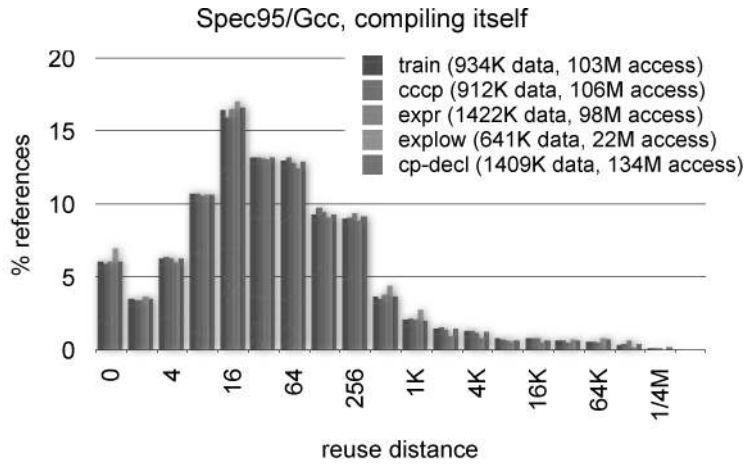
A careful reader may have noticed that for a number of programs, Tables II and III display a near identical number in the "average distance per element" column for different inputs. For example the number is 0.4 for all three inputs of *Swim*, which means that the average distance is 40% of the data size. The reason is that when the input size is sufficiently large, the total distance is dominated by reuse distances in the linear pattern. The same average distance is the result of the constant size of the linear pattern.

4.3.2 *Relation between a Program and Its Input.* The locality of the GNU C Compiler, *GCC*, is predicted with 96% to 99% accuracy, which is higher than people would normally expect. The program is large and complex—this version has 222,182 lines of source code in 120 files. More importantly, the data usage of the compiler should depend largely on the input. However, when measured using reuse distance, *GCC* manifests surprisingly regular locality. Figure 9 shows the locality signature of five executions of *GCC* when compiling some of its largest program files. The five signatures overlap by over 98%, which makes prediction accurate and trivial.

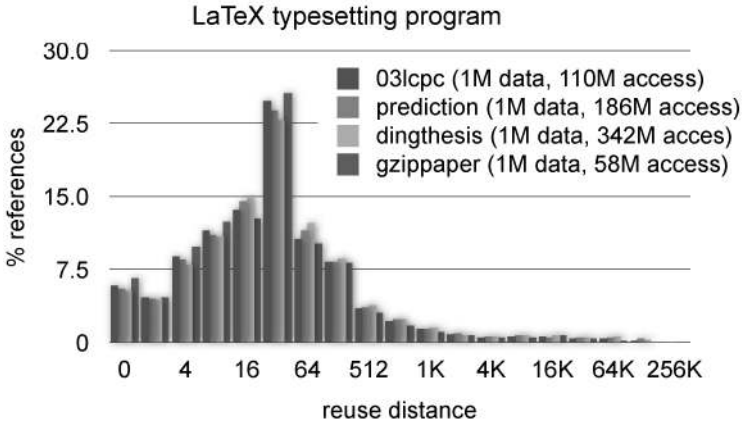
A compiler belongs to the general class of service-oriented programs. It processes input requests as a sequence of tasks, which in this case are the functions to be compiled. Shen et al. [2007] found that the compilation tasks go through the same sequence of phases in each task even though the length of the task is input dependent and unpredictable.

The input files we used for *GCC* have hundreds of functions. The signature might be showing the locality of compiling an "average" function. The consistency across inputs might be due to consistency in programmers' coding style, for example, the distribution of function sizes. To understand this more, we tested an extreme input, `166.i`, provided by the benchmark set. It contains two functions each with over one thousand lines of code. Distance-based sampling shows that long reuse distances are two orders of magnitude greater in *166.i* than in other inputs. The locality of compiling `166.i` is around 70% identical to the locality signatures shown in Figure 9(a), which suggests that the locality in *GCC* is 30% due to the input and 70% due to the program code.[1] This example

[1]Since the Alpha cluster has been replaced by a PC cluster at the time, we used the x86 binary instead of the Alpha binary.



(a) The locality signature of five executions of *GCC* compiling its largest program files



(b) The locality signature of four executions of the *LaTeX* typesetting program

Fig. 9. Locality signatures of *GCC* and *Latex*. The average locality is predictable even though the execution is largely input driven. The two programs have very similar signatures, which indicates common data usage patterns in compilation and type setting.

also suggests a general method—by testing a program on extreme inputs and measuring the deviance from the average—for exploring the range of variability in program locality.

4.3.3 *Relation between Programs.* With the whole-program locality model, we can now conduct comprehensive comparisons of the locality in different programs. In most cases, the locality signature is consistent in the same program but differs from one program to another. One interesting exception is the pair of programs *GCC* and *Latex*. *Latex* is a typesetting program commonly used in scientific publishing (including this article). Four locality signatures of

Latex, measured by Cheng [Cheng and Ding 2005] and shown in Figure 9(b), are strikingly similar to the locality signatures of *GCC* shown in the same figure. Intuitively, the result is expected since *GCC* and *Latex* are both language processors, one for the C programming language and the other for a type-setting language. However, it is significant that an automatic method can identify and quantify this similarity using an objective metric.

The two programs were developed independently, *Latex* by Knuth, Lamport, and other people mostly in universities, and *GCC* by a large group of open-source developers. In the introduction we mentioned that reuse distance is independent of localized transformations of a program or its execution. Here is an demonstration that the metric captures inherent similarity between two programs that share no code and have completely different data structures and program organization.

There may be practical uses in comparing program locality. A customer who is serious about performance may use locality as a metric in evaluating software from different vendors. A software company may be interested in maintaining performance (not just correctness) in new versions its products. The comparison can reveal changes in locality in different software versions and confirm or repudiate locality as a factor in complex performance problems. We have tested two versions of *GCC*, one in SPEC 1995 and the other in SPEC 2000 benchmark set, and found that their locality signatures overlap by 89%.

4.4 Summary

Compared to precise algorithms, approximate algorithms have faster performance and the performance scales better with the size of data. Relative-precision approximation provides unprecedented scalability, maintaining a constant speed regardless of the size of data and the length of reuse distance. At 99% accuracy, it measures the length of reuse distance in trillions at the same speed as exact solutions measure the length in millions. The scale tree it uses is orders of magnitude more space efficient than precise representations. A user can further improve the measurement speed by specifying a lower precision.

Using two training inputs, single-model prediction is 94% accurate on average for 15 benchmark programs, and its accuracy can be improved by 6% using more training data. Distance-based sampling can detect the input size after seeing less than 1% of an execution. The log-linear histograms always provide the most precise information for locality prediction, either in single-model and multimodel prediction.

Multimodel prediction can use logarithmic distance histograms and consequently be an order of magnitude more space efficient. Space efficiency is necessary for fine-grained analysis such as analyzing individual program instruction or data. In addition, multimodel prediction is 90% accurate with one fiftieth of the training data. However, the two efficiency boosts—logarithmic histograms and small inputs—should not be used at the same time.

Whole-program locality shows aggregate data usage in general sequential code. Our results show that a high degree of regularity in locality is a common phenomenon of collective behavior in complex code. The measurement and

prediction techniques are useful in observing and understanding these emergent effects, for example in understanding the relationships both between program code and input and between different programs.

5. RELATED WORK

This section discusses related work in program locality analysis. See Section 2.3 for related work in reuse distance measurement.

Training-Based Analysis. Independent of our work, Marin and Mellor-Crummey [2004, 2005] solved the same problem in the context of building a performance modeling toolkit. It uses compiler analysis to identify groups of related memory references for reuse-distance profiling. Given two histograms, their method first finds leading bins that have identical reuse distances and classifies them as constant patterns. Then it recursively divides the remaining group by its average reuse distance until the two halves show the same pattern of change in the two histograms, measured by the ratio between the average reuse distances. It uses quadratic programming to determine the best pattern parameters (because they model both locality and computation). A pattern function is a linear combination of base functions, which can include user supplied formula.

Recursive partitioning produces the minimal number of patterns with no loss of accuracy. By analyzing one reference group at a time, it can identify individual patterns that are difficult to separate in whole-program analysis. Recursive partitioning, however, is costly in data collection because it needs to store all reuse distances. In comparison, histograms with fixed-size bins are much more efficient to store, but they lose information about the distribution of reuse distances inside a bin. Multimodel prediction alleviates the problem by statistically estimating the mixing of patterns in the same bin.

Fang et al. [2005] solved the problem of measuring and analyzing the locality of every memory operation in a program. They used log-linear bins in data collection and experimented with different assumptions about the distribution within a bin. They found that a linear distribution worked well for both floating-point and integer programs while a uniform distribution (which we use in this work) worked well only for floating-point code. To improve efficiency, their method merges bins that have a similar distribution of reuse distances. The adaptive merging has a similar effect to Marin and Mellor-Crummey's recursive partitioning and makes the model compact without sacrificing its precision.

A common assumption in locality prediction is that a fixed fraction of reuse distances belong to each pattern in every locality signature. Marin and Mellor-Crummey [2004] tested heuristics not limited to this assumption but did not find them as stable and accurate. Our results in Section 4.2.2 showed that the assumption is mostly valid even for very small training inputs.

Static Analysis. Cascaval and Padua [2003] extended dependence analysis to estimate the reuse distances and the locality signature in scientific programs. Beyls and D'Hollander [2005] developed *reuse distance equations*, which

precisely compute the reuse distance in polyhedral loops using the Omega library [Kelly et al. 1996] as a fast symbolic but worst-case exponential-time solver. There are a number of compiler techniques that estimate the miss rate of a program [Porterfield 1989; Ferrante et al. 1991; Ghosh et al. 1999; Chatterjee et al. 2001; Xue and Vera 2004]. Compared to the miss rate, reuse distance is machine independent. Furthermore, it can be used to derive the miss rate. Beyls and D'Hollander [2005] compared compiler analysis with profiling by testing the effect of their use in cache-hint insertion.

A basic task of dependence checking is to analyze repeated accesses to data [Allen and Kennedy 2001; Wolfe 1996; Banerjee 1988]. Large-scale data usage can be summarized by various types of array section analysis [Havlak and Kennedy 1991], including linearization for high-dimensional arrays [Burke and Cytron 1986], linear inequalities for convex sections [Triolet et al. 1986], regular array sections [Callahan et al. 1988a], and reference lists [Li et al. 1990]. Other locality analyses include the matrix model [Wolf and Lam 1991; Kandemir 2005], memory ordering [McKinley et al. 1996], a number of later studies using high-dimensional discrete optimization [Cierniak and Li 1995; Kodukula et al. 1997], transitive closures [Song and Li 1999; Wonnacott 2002; Yi et al. 2000], and integer equations [Adve and Mellor-Crummey 1998].

Locality affects the fundamental balance between computation and memory transfer. Callahan et al. [1988b] defined the concept of program balance and machine balance. Techniques for matching the two balances have benefits from improving memory performance on conventional systems [Carr and Kennedy 1994; Ding and Kennedy 2004] to accelerating the design-space exploration in hardware-software co-design [So et al. 2002]. Whole-program locality can be used to estimate program balance in general-purpose applications.

Pure program analysis has a limited effect on general-purpose code because of the difficulty in analyzing complex control flow and indirect data accesses and characterizing aggregate program behavior. However, for regular loop nests with linearly indexed array references, static analysis can precisely model the iteration space and the data space. It can analyze locality in high-dimensional data, which is difficult for training-based analysis. In addition, compiler analysis is sound in that the result can be used to reorder program execution, for example, to change the program balance. Locality prediction is probabilistic. It measures common behavior but not all behavior. It cannot observe program behavior that does not occur in training runs. One solution is to combine compiler and profiling analysis, as demonstrated by Marin and Mellor-Crummey [2005]. Another solution is speculative program optimization. For example, Kelsey et al. [2009] proposed a software system that creates a FastTrack, which is a copy of a program optimized for the common behavior. The original code is run in parallel as a fallback in case the FastTrack code produces incorrect results.

Reuse Frequency and Data Streams. Access frequency gives the first model of data usage [Knuth 1971; Cocke and Kennedy 1974]. Later refinements include the lifetime of single objects [Seidl and Zorn 1998] or the affinity between data pairs [Thabit 1981; Calder et al. 1998; Chilimbi et al. 1999]. Chilimbi and

his colleagues extended the notion of affinity using *hot data streams*, which are repeated sequences of data accesses up to 100 elements long [Chilimbi 2001a, 2001b]. Streams and distances are complementary concepts. Hot streams show frequent repetitions, while the locality signature shows common recurrences. A stream contains order information that can be used for data prefetching. The locality signature provides a way to relate data by their aggregate locality (as used in reference affinity analysis described in Section 6).

Runtime Analysis. Saltz and his colleagues pioneered dynamic parallelization with the approach known as inspector-executor, where the inspector examines and partitions the data and computation at run time [Das et al. 1994]. Similar strategies were used to improve dynamic program locality [Ding and Kennedy 1999; Mellor-Crummey et al. 2001; Strout et al. 2003; Han and Tseng 2006]. Knobe and Sarkar [1998] included runtime data analysis in array static-single assignment (SSA) form. To reduce the overhead of runtime analysis, Arnold and Ryder [2001] developed a software framework for periodic sampling, which Chilimbi and Hirzel [2002] extended to discover hot data streams for data prefetching. Liu et al. [2004] developed a dynamic optimization system by leveraging hardware monitoring support for very low-cost sampling. In addition to sampling based on program code and hardware events, Ding and Kennedy [1999] sampled accesses to a subset of data in an array. Ding and Zhong [2002] extended the scheme for use on dynamic data. Distance-based sampling is a form of data-based sampling as it uses the reuse distance to select data samples.

While runtime analysis can identify patterns unique to the current execution, it is not as thorough as off-line training analysis. On the other hand, offline models and online analysis can be combined to help each other, as this article has shown in combining training analysis and distance-based sampling.

6. FIVE DIMENSIONS OF LOCALITY

Reuse distance has uses in numerous studies. As an imprecise and incomplete count, a keyword search in the ACM Digital Library shows 91 publications since 2003 that contain the phrase “reuse distance” in addition to the words “locality” and “cache” in conferences and journals in the area of programming systems, computer architecture, operating systems, and embedded systems. In this section, we present a taxonomy that classifies representative problems, solutions and uses of locality analysis in five mostly orthogonal dimensions: input, code, data, time, and execution environment.

Whole-Program Locality. Locality affected by the program input. The simple-model prediction described in this article has been used to predict the capacity miss rate for a set of programs across cache sizes, program inputs [Zhong et al. 2007], and program phases [Shen et al. 2004b]. Fang et al. [2005] extended whole-program prediction to predict the locality of each program instruction as a function of the input size. They defined a general concept called memory distance to include reuse, access, and value distance. Marin and Mellor-Crummey [2004] considered cache associativity and computational

characteristics and predicted performance across machine platforms. On parallel systems, the scalability of a program depends on the ratio of computation to communication. Locality determines the amount of communication. Rothberg et al. [1993] used simulation and curve fitting to derive the program locality for a SPLASH benchmark program, *Barnes-Hut*, which was too difficult for symbolic analysis.

Locality in Program Code. Beyls and D'Hollander [2002, 2005] used the locality signature of each instruction to generate *cache hints*, which guide cache replacement decisions in hardware so that the data loaded by low-locality instructions do not evict the data loaded by high-locality instructions. They reported performance improvements for both integer and floating-point benchmarks on Intel Itanium, demonstrating the first distance-based technique to directly improve performance. In a program, the locality of statements, loops, and functions can be analyzed using training analysis [Fang et al. 2005], compiler analysis (for scientific code) [Cascaval and Padua 2003; Beyls and D'Hollander 2005], or their combination [Marin and Mellor-Crummey 2004]. Beyls and D'Hollander [2005] compared profiling analysis and compiler analysis (called reuse distance equations) in generating cache hints.

Beyls and D'Hollander [2006a, 2006b] developed a program tuning tool *SLO*, which identifies the cause of long distance reuses and gives improvement suggestions for restructing the code. Using the tool, they were able to double the average speed of five SPEC 2000 benchmarks on four machine platforms. Furthermore, they used sampling analysis to reduce the profiling overhead from a 1000 times slowdown to a 5 times slowdown.

Locality in Program Data. As an optimization problem, data placement is theoretically intractable in general [Petrank and Rawitz 2002]. In practice, a useful metric is reference affinity, which identifies data that are used together. Zhong et al. [2004] defined reference affinity using reuse distance and showed its use in array regrouping and structure splitting. Zhang et al. [2006] showed formally that reference affinity uncovers the hierarchical locality in data from the access pattern in computation. Shen et al. [2005] developed a static analysis of reference affinity and tested its use in the IBM compiler. Zhao et al. [2007] included affinity in the Forma framework for automatic array reshaping in the IBM compiler.

Spatial locality measures the quality of a data layout. Three studies have defined spatial locality as the change in locality when the granularity of data increases from data elements to cache blocks [Berg and Hagersten 2005; Gu et al. 2009] or from data elements to memory pages [Bunt and Murphy 1984]. Gu et al. [2009] ranked program functions by (the lack of) spatial locality to aid program tuning.

Locality over Time. Batson and Madison [1976] defined a phase as a period of execution accessing a subset of program data. Denning [1980] stated that a proper model must account for the tranquility of phases as well as disruptive transitions. Shen et al. [2004a, 2007] built a model by effectively converting an execution to a signal, that is, a sequence of reuse distances, and applying

wavelet analysis to separate gradual changes from disruptive ones. Many subsequent studies considered wavelets and locality phases in modeling temporal behavior in a system.

For cache and memory management, a basic problem is predicting the time of future data access. Increasing evidence shows that the last reuse distance is an effective predictor. It has been used in memory management [Smaragdakis et al. 2003; Chen et al. 2005; Zhou et al. 2004] and file caching [Zhou et al. 2001; Jiang and Zhang 2002]. Kelly et al. [2004] found reuse distance to be a powerful predictor of response time in server systems. Almeida et al. [1996] showed that the locality signature of web reference streams follows log-normal distributions, in which the logarithm is normally distributed.

Interaction between Programs. When multiple running programs share a cache, the performance of one program is influenced by the locality of others. The effect can be modeled by inflating the reuse distance of the program with the footprint of its peers [Suh et al. 2001; Chandra et al. 2005]. Jiang et al. [2008] formulated the problem of optimal coscheduling on shared cache. For memory sharing, Yang et al. [2006] studied cooperative interaction between heap management in the Java virtual machine and memory management in the operating system. The key metric is the LRU reference histogram, which is equivalent to the locality signature in this article.

7. CONCLUSIONS

Locality has become increasingly important in the design of algorithms, compilers, operating systems, and computer architectures. In this article we have presented training-based whole-program locality analysis, which consists of two approximate algorithms for measuring reuse distance and five prediction methods for modeling whole-program locality. The approximate algorithms are faster and more scalable than exact solutions while guaranteeing an absolute or relative precision. The precision and cost are adjustable. The asymptotic cost of the relative-precision algorithm is effectively linear in the length of the trace. The five prediction methods decompose reuse distances using either reference histograms or distance histograms in logarithmic or log-linear scales. Each locality component can have a single pattern or multiple patterns. For 15 floating-point and integer benchmark applications, single-model prediction using two inputs shows 94% accuracy and 99% coverage. The accuracy can be improved by using more inputs and multimodel prediction. The efficiency can be improved by using compact histograms or very small inputs.

Locality is a fundamental aspect of computation. The new locality models in this article are quantitative yet they are not tied to any specific machine and are unaffected by irrelevant aspects of program construction. The results show that through them locality can be quantified for complex applications. The decomposition of locality as done in this work is orthogonal to the traditional decomposition of program code and data and hence provides a new dimension in program analysis. It provides a systematic model of application data behavior and a quantitative basis for understanding and managing dynamic data usage at different levels of a computing system.

ACKNOWLEDGMENTS

The authors wish to thank Grant Farmer, Wei Jiang, Roland Cheng, and Matthew Nettleton for their help in implementation and testing; Trishul Chilimbi, Sandhya Dwarkadas, Steve Dropsho, Xiaoming Gu, Matthew Hertz, Mark Hill, Bryan Jacobs, Gabriel Marin, John Mellor-Crummey, Joseph Modayil, Mitsu Ogihara, Michael Scott, Zhenlin Wang, Xiaoya Xiang, Mihalis Yannakaki, the anonymous reviewers of this journal, the LCR'02 workshop, PLDI'03 and LACSI'03 conferences for their comments, corrections and suggestions on the work and its presentation. Our experiments mainly used machines at Rochester purchased by several NSF Infrastructure grants and equipment grants from DEC/Compaq/HP and Intel. John Mellor-Crummey and Rob Fowler provided access to Rice Alpha clusters.

REFERENCES

ADVE, V. AND MELLOR-CRUMMEY, J. 1998. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

ALLEN, R. AND KENNEDY, K. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers.

ALMASI, G., CASCAVAL, C., AND PADUA, D. 2002. Calculating stack distances efficiently. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance*.

ALMEIDA, V., BESTAVROS, A., CROVELLA, M., AND DE OLIVEIRA, A. 1996. Characterizing reference locality in the WWW. In *Proceedings of the International Conference on Parallel and Distributed Information Systems (PDIS)*. 92–103.

ALON, N., MATIAS, Y., AND SZEGEDY, M. 1996. The space complexity of approximating the frequency moments. In *Proceedings of the ACM Symposium on Theory of Computing*.

ARNOLD, M. AND RYDER, B. G. 2001. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

BANERJEE, U. 1988. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, MA.

BATSON, A. P. AND MADISON, A. W. 1976. Measurements of major locality phases in symbolic reference strings. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*.

BENNETT, B. T. AND KRUSKAL, V. J. 1975. LRU stack processing. *IBM J. Resear. Devel.* 353–357.

BERG, E. AND HAGERSTEN, E. 2004. Statcache: A probabilistic approach to efficient and accurate data locality analysis. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 20–27.

BERG, E. AND HAGERSTEN, E. 2005. Fast data-locality profiling of native execution. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*. 169–180.

BEYLS, K. AND D'HOLLANDER, E. 2002. Reuse distance-based cache hint selection. In *Proceedings of the 8th International Euro-Par Conference*. Paderborn, Germany.

BEYLS, K. AND D'HOLLANDER, E. 2005. Generating cache hints for improved program efficiency. *J. Syst. Archit. 51*, 4, 223–250.

BEYLS, K. AND D'HOLLANDER, E. 2006a. Discovery of locality-improving refactoring by reuse path analysis. In *Proceedings of the High-Performance Computing and Communications Council*. Springer. Lecture Notes in Computer Science, vol. 4208. 220–229.

BEYLS, K. AND D'HOLLANDER, E. 2006b. Intermediately executed code is the key to find refactorings that improve temporal data locality. In *Proceedings of the ACM Conference on Computing Frontiers*.

BUNT, R. B. AND MURPHY, J. M. 1984. Measurement of locality and the behaviour of programs. *Comput. J. 27*, 3, 238–245.

BURKE, M. AND CYTRON, R. 1986. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*.

CALDER, B., KRINTZ, C., JOHN, S., AND AUSTIN, T. 1998. Cache-conscious data placement. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*.

CALLAHAN, D., COCKE, J., AND KENNEDY, K. 1988a. Analysis of interprocedural side effects in a parallel programming environment. *J. Paral. Distrib. Comput. 5*, 5, 517–550.

CALLAHAN, D., COCKE, J., AND KENNEDY, K. 1988b. Estimating interlock and improving balance for pipelined machines. *J. Paral. Distrib. Comput. 5*, 4, 334–358.

CARR, S. AND KENNEDY, K. 1994. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. Program. Lang. Syst. 16*, 6, 1768–1810.

CASCAVAL, C. AND PADUA, D. A. 2003. Estimating cache misses and locality using stack distances. In *Proceedings of the International Conference on Supercomputing*. San Francisco, CA.

CHANDRA, D., GUO, F., KIM, S., AND SOLIHIN, Y. 2005. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 340–351.

CHATTERJEE, S., PARKER, E., HANLON, P. J., AND LEBECK, A. R. 2001. Exact analysis of the cache behavior of nested loops. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

CHEN, F., JIANG, S., AND ZHANG, X. 2005. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *Proceedings of the USENIX Annual Technical Conference*.

CHENG, R. AND DING, C. 2005. Measuring temporal locality variation across program inputs. Tech. rep. TR 875, Department of Computer Science, University of Rochester.

CHILIMBI, T. M. 2001a. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

CHILIMBI, T. M. 2001b. On the stability of temporal data reference profiles. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*.

CHILIMBI, T. M., HILL, M. D., AND LARUS, J. R. 1999. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

CHILIMBI, T. M. AND HIRZEL, M. 2002. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

CIERNIAK, M. AND LI, W. 1995. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*.

COCKE, J. AND KENNEDY, K. 1974. Profitability computations on program flow graphs. Tech. rep. RC 5123, IBM.

DAS, R., UYSAL, M., SALTZ, J., AND HWANG, Y.-S. 1994. Communication optimizations for irregular scientific computations on distributed memory architectures. *J. Paral. Distrib. Comput. 22*, 3, 462–479.

DATAR, M., GIONIS, A., INDYK, P., AND MOTWANI, R. 2002. Maintaining stream statistics over sliding windows. *SIAM J. Comput. 31*, 6, 1794–1813.

DENNING, P. 1980. Working sets past and present. *IEEE Trans. Softw. Engin. 6*, 1.

DING, C. AND KENNEDY, K. 1999. Improving cache performance in dynamic applications through data and computation reorganization at runtime. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*.

DING, C. AND KENNEDY, K. 2004. Improving effective bandwidth through compiler enhancement of global cache reuse. *J. Paral. Distrib. Comput. 64*, 1, 108–134.

DING, C. AND ZHONG, Y. 2002. Compiler-directed runtime monitoring of program data access. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance*.

EECKHOUT, L., VANDIERENDONCK, H., AND BOSSCHERE, K. D. 2002. Workload design: Selecting representative program-input pairs. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*.

FANG, C., CARR, S., ONDER, S., AND WANG, Z. 2005. Instruction-based memory distance analysis and its application to optimization. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*.

FERRANTE, J., SARKAR, V., AND THRASH, W. 1991. On estimating and enhancing cache effectiveness. In *Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Springer-Verlag.

FLAJOLET, P. AND MARTIN, G. 1983. Probabilistic counting. In *Proceedings of the Symposium on Foundations of Computer Science*.

GHOSH, S., MARTONOSI, M., AND MALIK, S. 1999. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst. 21*, 4.

GU, X., CHRISTOPHER, I., BAI, T., ZHANG, C., AND DING, C. 2009. A component model of spatial locality. In *Proceedings of the International Symposium on Memory Management*.

HAN, H. AND TSENG, C.-W. 2006. Exploiting locality for irregular scientific codes. *IEEE Trans. Paral. Distrib. Syst. 17*, 7, 606–618.

HAVLAK, P. AND KENNEDY, K. 1991. An implementation of interprocedural bounded regular section analysis. *IEEE Trans. Paral. Distrib. Syst. 2*, 3, 350–360.

HILL, M. D. AND SMITH, A. J. 1989. Evaluating associativity in CPU caches. *IEEE Trans. Comput. 38*, 12, 1612–1630.

HSU, W., CHEN, H., YEW, P. C., AND CHEN, D. 2002. On the predictability of program behavior using different input data sets. In *Proceedings of the 6th Workshop on Interaction Between Compilers and Computer Architectures (INTERACT)*.

JIANG, S. AND ZHANG, X. 2002. LIRS: An efficient low inter-reference recency set replacement to improve buffer cache performance. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*.

JIANG, Y., SHEN, X., CHEN, J., AND TRIPATHI, R. 2008. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. 220–229.

KANDEMIR, M. T. 2005. Improving whole-program locality using intra-procedural and inter-procedural transformations. *J. Paral. Distrib. Comput. 65*, 5, 564–582.

KELLY, T., COHEN, I., GOLDSZMIDT, M., AND KEETON, K. 2004. Inducing models of black-box storage arrays. Tech. rep. HPL-2004-108, HP Laboratories Palo Alto, CA.

KELLY, W., MASLOV, V., PUGH, W., ROSSER, E., SHPEISMAN, T., AND WONNACOTT, D. 1996. The Omega Library Interface Guide. Tech. rep., Department of Computer Science, University of Maryland, College Park.

KELSEY, K., BAI, T., AND DING, C. 2009. Fast track: A software system for speculative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*.

KIM, Y. H., HILL, M. D., AND WOOD, D. A. 1991. Implementing stack simulation for highly-associative memories. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. 212–213.

KLEINOSOWSKI, A. AND LILJA, D. J. 2002. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Comput. Archit. Lett. 1*.

KNOBE, K. AND SARKAR, V. 1998. Array SSA form and its use in parallelization. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

KNUTH, D. 1971. An empirical study of FORTRAN programs. *Softw. Pract. Exper. 1*, 105–133.

KODUKULA, I., AHMED, N., AND PINGALI, K. 1997. Data-centric multi-level blocking. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*.

LI, Z., YEW, P., AND ZHU, C. 1990. An efficient data dependence analysis for parallelizing compilers. *IEEE Trans. Paral. Distrib. Syst. 1*, 1, 26–34.

LIU, J., CHEN, H., YEW, P.-C., AND HSU, W.-C. 2004. Design and implementation of a lightweight dynamic optimization system. *J. Instruct.-Level Paral. 6*.

MARIN, G. AND MELLOR-CRUMMEY, J. 2004. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*.

MARIN, G. AND MELLOR-CRUMMEY, J. 2005. Scalable cross-architecture predictions of memory hierarchy response for scientific applications. In *Proceedings of the Symposium of the Las Alamos Computer Science Institute*.

MATTSON, R. L., GECSEI, J., SLUTZ, D., AND TRAIGER, I. L. 1970. Evaluation techniques for storage hierarchies. *IBM Syst. J. 9*, 2, 78–117.

MCKINLEY, K. S., CARR, S., AND TSENG, C.-W. 1996. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst. 18*, 4, 424–453.

MELLOR-CRUMMEY, J., WHALLEY, D., AND KENNEDY, K. 2001. Improving memory hierarchy performance for irregular applications. *Int. J. Paral. Program. 29*, 3.

OLKEN, F. 1981. Efficient methods for calculating the success function of fixed space replacement policies. Tech. rep. LBL-12370, Lawrence Berkeley Laboratory.

PETRANK, E. AND RAWITZ, D. 2002. The hardness of cache conscious data placement. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

PORTERFIELD, A. 1989. Software methods for improvement of cache performance. Ph.D. thesis, Department of Computer Science, Rice University.

RAWLINGS, J. O. 1988. *Applied Regression Analysis: A Research Tool*. Wadsworth and Brooks.

ROTHBERG, E., SINGH, J. P., AND GUPTA, A. 1993. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*. 14–25.

SEIDL, M. L. AND ZORN, B. G. 1998. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*.

SHEN, X., GAO, Y., DING, C., AND ARCHAMBAULT, R. 2005. Lightweight reference affinity analysis. In *Proceedings of the 19th ACM International Conference on Super-Computing*. 131–140.

SHEN, X., SHAW, J., MEEKER, B., AND DING, C. 2007. Locality approximation using time. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 55–61.

SHEN, X., ZHANG, C., DING, C., SCOTT, M., DWARKADAS, S., AND OGIHARA, M. 2007. Analysis of input-dependent program behavior using active profiling. In *Proceedings of The 1st Workshop on Experimental Computer Science*.

SHEN, X., ZHONG, Y., AND DING, C. 2004a. Locality phase prediction. In *Proceedings of the International Conference on Architectual Support for Programming Languages and Operating Systems*. 165–176.

SHEN, X., ZHONG, Y., AND DING, C. 2004b. Phase-based miss rate prediction. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*.

SHEN, X., ZHONG, Y., AND DING, C. 2007. Predicting locality phases for dynamic memory optimization. *J. Paral. Distrib. Comput. 67*, 7, 783–796.

SLEATOR, D. D. AND TARJAN, R. E. 1985. Self adjusting binary search trees. *J. ACM 32*, 3.

SMARAGDAKIS, Y., KAPLAN, S., AND WILSON, P. 2003. The EELRU adaptive replacement algorithm. *Perform. Eval. 53*, 2, 93–123.

SMITH, A. J. 1976. 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*.

SO, B., HALL, M. W., AND DINIZ, P. C. 2002. A compiler approach to fast hardware design space exploration in FPGA-based systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

SONG, Y. AND LI, Z. 1999. New tiling techniques to improve cache temporal locality. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

SRIVASTAVA, A. AND EUSTACE, A. 1994. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

STROUT, M. M., CARTER, L., AND FERRANTE, J. 2003. Compile-time composition of runtime data and iteration reorderings. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 245–257.

SUGUMAR, R. A. AND ABRAHAM, S. G. 1993. Multi-configuration simulation algorithms for the evaluation of computer architecture designs. Tech. rep., University of Michigan.

SUH, G. E., DEVADAS, S., AND RUDOLPH, L. 2001. Analytical cache models with applications to cache partitioning. In *Proceedings of the International Conference on Super-Computing*. 1–12.

THABIT, K. O. 1981. Cache management by the compiler. Ph.D. thesis, Department of Computer Science, Rice University.

THOMPSON, J. G. AND SMITH, A. J. 1989. Efficient (stack) algorithms for analysis of write-back and sector memories. *ACM Trans. Comput. Syst. 7*, 1, 78–117.

TRIOLET, R., IRIGOIN, F., AND FEAUTRIER, P. 1986. Direct parallelization of CALL statements. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*.

WALL, D. W. 1991. Predicting program behavior using real or estimated profiles. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

WANG, W. AND BAER, J.-L. 1991. Efficient trace-driven simulation methods for cache performance analysis. *ACM Trans. Comput. Syst. 9*, 3.

WOLF, M. E. AND LAM, M. 1991. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*.

WOLFE, M. J. 1996. *High-Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA.

WONNACOTT, D. 2002. Achieving scalable locality with time skewing. *Int. J. Paral. Program. 30*, 3.

XUE, J. AND VERA, X. 2004. Efficient and accurate analytical modeling of whole-program data cache behavior. *IEEE Trans. Comput. 53*, 5.

YANG, T., BERGER, E. D., KAPLAN, S. F., AND MOSS, J. E. B. 2006. Cramm: Virtual memory support for garbage-collected applications. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. 103–116.

YI, Q., ADVE, V., AND KENNEDY, K. 2000. Transforming loops to recursion for multi-level memory hierarchies. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

ZHANG, C., DING, C., OGIHARA, M., ZHONG, Y., AND WU, Y. 2006. A hierarchical model of data locality. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

ZHAO, P., CUI, S., GAO, Y., SILVERA, R., AND AMARAL, J. N. 2007. Forma: A framework for safe automatic array reshaping. *ACM Trans. Program. Lang. Syst. 30*, 1, 2.

ZHONG, Y. AND CHANG, W. 2008. Sampling-based program locality approximation. In *Proceedings of the International Symposium on Memory Management*. 91–100.

ZHONG, Y., DING, C., AND KENNEDY, K. 2002. Reuse distance analysis for scientific programs. In *Proceedings of Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*.

ZHONG, Y., DROPSHO, S. G., SHEN, X., STUDER, A., AND DING, C. 2007. Miss rate prediction across program inputs and cache configurations. *IEEE Trans. Comput. 56*, 3, 328–343.

ZHONG, Y., ORLOVICH, M., SHEN, X., AND DING, C. 2004. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

ZHOU, P., PANDEY, V., SUNDARESAN, J., RAGHURAMAN, A., ZHOU, Y., AND KUMAR, S. 2004. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

ZHOU, Y., CHEN, P. M., AND LI, K. 2001. The multi-queue replacement algorithm for second-level buffer caches. In *Proceedings of the USENIX Technical Conference*.

Received September 2004; revised September 2005; accepted May 2007