

Distance-Based Locality Analysis and Prediction

CHEN DING, YUTAO ZHONG, and XIPENG SHEN

University of Rochester

Profiling can accurately analyze program behavior for select data inputs. This article shows that profiling can also predict program locality for inputs other than profiled ones. Here locality is defined by the distance of data reuse. The article describes three distance-based techniques for whole-program locality analysis. The first is approximate measurement of reuse distance in near linear time. It can measure the reuse distance of all accesses to all data elements in full-size benchmarks with guaranteed precision. The second is pattern recognition. Based on a few training runs, it classifies patterns as regular and irregular and, for regular ones, it predicts their (changing) behavior for other inputs. It uses regression and multi-model analysis to reduce the prediction error, the space overhead, and the size of the training runs. The last technique is on-line prediction, which uses distance-based sampling at the beginning of an execution to estimate the locality of the whole execution. When tested on 15 integer and floating-point programs from SPEC and other benchmark suites, these techniques predict with on average 94% accuracy for data inputs up to hundreds times larger than the training inputs. Distance-based locality analysis has been used in measuring and improving the cache performance of a wide range of programs.

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, profiling analysis

1. INTRODUCTION

Caching is widely used in many computer programs and systems, and cache performance increasingly determines system speed, cost, and energy usage. The effect of caching is determined by the locality of the memory access of a program. As new cache designs are adding more cache levels and allowing dynamic reconfiguration, the cache performance increasingly depends on our ability to predict the program locality.

Many programs have predictable data-access patterns. Some patterns change from one input to another, for example, a finite-element analysis for different size terrains and a Fourier transformation for different length signals. Some patterns are constant, for example, a chess program looking ahead a finite number of moves and a compression tool operating over a constant-size window.

The past work provides mainly three ways of locality analysis: by a compiler, which analyzes loop nests but is not as effective for dynamic control flow and data indirection; by a profiler, which analyzes a program for select inputs but does not predict its behavior

The article combines, updates, and enhances the past work appeared in the 2002 Workshop on Languages, Compilers, and Run-time Systems, 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2003 Annual Symposium of Los Alamos Computer Science Institute.

Authors' address: Department of Computer Science, University of Rochester, Rochester, New York 14627; email: {cding,ytzhong,xshen}@cs.rochester.edu.

change in other inputs; or by run-time analysis, which cannot afford to analyze every access to every data. The inquiry continues for a prediction scheme that is efficient, accurate, and applicable to general-purpose programs.

The article presents a new method for locality prediction, using a concept we call the *reuse distance*. In a sequential execution, the reuse distance of a data access is the number of *distinct* data elements accessed between this and the previous access of the same data. Since it measures the volume of the intervening data access, it is always bounded, even for a long-running program. In 1970, Mattson et al. [1970] defined a collection of concepts called stack distances and laid a foundation for the research in virtual memory management in the following decades. The *LRU stack distance* is the stack distance using the least-recently-used replacement policy. The reuse distance of a data access is equal to the stack distance between this and the previous access of the same data. We use a different (and shorter) name to reflect our purpose in program analysis, not memory management. We later show that reuse distance is measured much faster using a tree instead of a stack.

Three properties of the reuse distance are critical for predicting program locality across different executions of a program. First, the reuse distance is at most a linear function of the program data size. The search space is much smaller for pattern recognition and prediction. Second, the reuse distance reveals invariance in program behavior. Most control flow perturbs only short access sequences but not the cumulative distance over a large amount of data. Long reuse distances suggest important data and signal major phases of a program. Finally, reuse distance allows direct comparison of data behavior in different program runs. Different executions of a program may allocate different data or allocate the same data at different locations. They may go through different paths. Distance-based correlation does not require two executions to have the same data or to execute the same function. Therefore, it can identify consistent patterns in the presence of dynamic data allocation and input-dependent control flows.

The article presents distance-based locality analysis and prediction. It has three new components. The first is approximate reuse-distance analysis, which bounds the relative error to arbitrarily close to zero. It takes $O(N \log \log M)$ time and $O(\log M)$ space, where N is the length of the trace and M is the size of the data. The second is pattern recognition, which profiles a few training runs, classifies patterns as regular and irregular, and, for regular ones, constructs a parameterized model. It uses regression and multi-model analysis to reduce the prediction error, the space overhead, and the size of the training runs. Using the locality pattern, the last technique, distance-based sampling, predicts the locality of an unknown execution by sampling at the beginning of the execution. Together these three techniques provide a general method for predicting the locality of a program across different data inputs.

We should note that the goal of this work is not cache analysis. Cache performance is not a direct measure of a program but a projection of a particular execution on a particular cache configuration. Our goal is program analysis. We find patterns consistent across all data inputs. We analyze the reuses of data elements instead of cache blocks. The element-level behavior is harder to analyze because it is not amortized by the size of cache blocks or memory pages (element miss rate is much higher than cache-block miss rate). We analyze the full distance, not its comparison with fixed cache sizes. Per-element, full-length analysis is most precise and demands highest efficiency and accuracy. The distance-based analysis has many uses in understanding and improving the performance of real cache systems, as discussed in Section 4.

We do not find all patterns in all programs. Not all programs have a consistent pattern, nor are all patterns predictable, let alone by our method. Our goal is to define common recurrence patterns and measure their presence in representative programs. As dependence analysis analyzes loops that can be analyzed, we predict patterns that are predictable. We now show that, in many cases, reuse distance can extend the scope of locality analysis to the whole program.

2. DISTANCE-BASED LOCALITY ANALYSIS AND PREDICTION

This section presents the three components: approximate reuse-distance analysis, distance-pattern recognition, and distance-based sampling.

2.1 Approximate reuse-distance measurement

We view a program execution by its data-access trace. To find the reuse distance, a naive algorithm would traverse the trace and for each access, search backwards to find the last access of the same data and count the number of different data in between. In the worst case, it needs to look back to the beginning of the trace, so the asymptotic complexity is $O(N^2)$ in time and $O(N)$ in space for a trace of N memory accesses. These costs are impractical for real programs, where N is often in hundreds of billions.

The time and space costs can be reduced by better measurement algorithms. We illustrate the past solutions and our new algorithm through an example in Figure 1. Part (a) shows that we need to count accesses to distinct data. Part (b) shows that instead of storing the whole trace, we store (and count) just the last access of each data element. Part (c) shows the most efficient counting in the past literature. By organizing the last-access times in a search tree, the counting is done in a single tree search. Assuming a balanced tree, the measurement takes $O(N \log M)$ time and $O(M)$ space, where M is the size of program data. For a program with a large amount of data, the space requirement becomes a limiting factor. The tree needs to store at least four attributes for each data element, as shown in Figure 1(d). Since the tree data is four times the program data, the space requirement of the analysis easily overflow the physical memory of a machine and even the 32-bit address space when a program uses more than 100 million data.

We describe an approximate analysis that further reduces the asymptotic costs of the measurement. Accurate analysis is costly but often unnecessary for long reuse distances. If the length of a distance is in the order of millions, we rarely care about the last couple of digits of the distance. By reducing the space cost, we can make the tree small enough to fit in not only the physical memory but also the processor cache.

In the following discussion, we do not consider the cost of finding the last access time. This requires a hashtable with one entry for each data. The time cost of the hash lookup is constant per access. Bennett and Kruskal [1975] showed that hashing can be done in a pre-pass, using blocked algorithms to reduce the memory requirement to arbitrarily low.

We present two approximation algorithms, with different guarantee on the accuracy of the measured distance, $d_{measured}$, compared to the actual distance, d_{actual} . Both guarantee $d_{measured} \leq d_{actual}$. The difference is whether the error is bounded by a relative rate or an absolute number.

1. bounded relative error e , $1 \geq e > 0$ and $\frac{d_{actual} - d_{measured}}{d_{actual}} \leq e$
2. bounded absolute error B , $B > 0$ and $d_{actual} - d_{measured} \leq B$

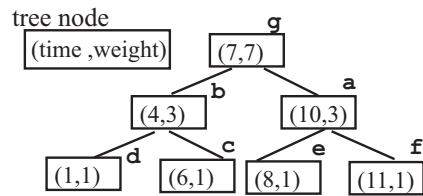
We show the main idea of the approximate analysis in Figure 1. As shown in Part (c) for the accurate analysis, the tree stores the last access of each element in a separate tree node.

time: 1 2 3 4 5 6 7 8 9 10 11 12
 access: **d a c b c c g e f a f b**
 distance: |← 5 distinct accesses →|

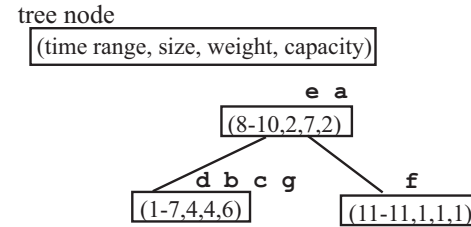
(a) An example access sequence. The reuse distance of the second access of *b* is five because five distinct elements are accessed since the last access of *b*.

time: 1 2 3 4 5 6 7 8 9 10 11 12
 access: **d ~~a~~ ~~c~~ b ~~c~~ c g e f a f b**
 distance: |← 5 last accesses →|

(b) All but the last access of each element in (a) are crossed out. The reuse distance is the number of the remaining accesses after the last access of *b* at time four.



(c) The simplified trace in (b) is organized as a search tree. Each tree node represents the last access of a data element. The first attribute is the search key, which is the last-access time. The weight attribute is the size of the sub-tree. To measure the reuse distance, we find the last access of *b* through a tree search. Then we count the number of tree nodes after its last access by a traversal of the search path using the sub-tree weight.



(d) Each node represents a time range given by the first attribute. The size attribute is the number of elements whose last access is within the time range. The weight is the total size of the sub-tree. The capacity bounds the size of a tree node to ensure a particular accuracy, which is 33% in this case. To measure the reuse distance, we find the range that includes the last access of *b* and add the size of the tree nodes since then. The approximate distance of two *b*'s is 3 or 60% of the actual distance.

4 Fig. 1. An example illustrating the reuse-distance measurement. Part (a) shows the definition of reuse distance. Parts (b) and (c) show two accurate measurements. Part (d) shows the approximate measurement with a bounded relative error (67%).

We can reduce the size of the tree by using a node to store a time range that includes the last access of multiple elements, as shown in Part (d). We define the size of a tree node as the number of last accesses contained in its time range. Compared to the accurate tree, the size of the approximation tree is smaller by a factor equal to the average node size. The smallest tree has one node, whose time range is the entire trace. It is also the least accurate. The approximation methods set the node size differently to obtain different efficiency and accuracy. The rest of this section describes them in detail.

2.1.1 Analysis with a bounded relative error. The analysis guarantees a bounded error rate that can be arbitrarily close to zero. Figure 2 shows the data structure and the main algorithm. Each node represents a time range in the trace. Its size is the number of the last accesses in the time range. Given the current and the last access time, the main routine uses *TreeSearchDelete* to find the reuse distance, *TreeInsert* to insert the current access as the last access, and *TreeCompression* to reduce the tree size when the size is above a threshold. The algorithms for *TreeSearchDelete* and *TreeCompression* are shown in Figure 3. The first subroutine searches the tree, calculates the reuse distance, and updates the capacity of the node once it is found. Then it deletes the tree node because the current access will be added as the last access. The tree insertion and deletion will rebalance the tree and update sub-tree weights. These two steps are not shown because they depend on the type of the balanced tree being used, which can be an AVL, red-black, splay, or B-tree.

To ensure the relative accuracy, the algorithm sets the capacity of a tree node n to be at most $distance * \frac{e}{1-e}$, where $distance$ is the total size of the *later* tree nodes, which are the nodes whose time range is greater than the time range of n . In other words, the $distance$ gives the number of distinct data accessed after n . If the last access time falls in the time range of n , the algorithm use $distance$ as the approximate reuse distance. The approximation is no greater than the actual distance. Since the actual distance can be at most $distance$ plus the node capacity, the accuracy is at least $distance$ divided by $distance + distance * \frac{e}{1-e}$, which is $1 - e$. The formula of $\frac{e}{1-e}$ is not valid if $e = 0$ or $e = 1$. The former means accurate analysis. The latter means that the error can be any fraction of the actual distance. We can simply return 0 as the measured distance.

The size of the tree determines the efficiency of the algorithm. The most important part of the algorithm is tree compression. The subroutine *TreeCompression* scans the tree nodes in the reverse time order and updates the capacity to be $distance * \frac{e}{1-e}$. It merges adjacent tree nodes if the size of the merged node is no more than the smaller capacity of the two nodes. Tree compression is triggered when the tree size exceeds $4 * \log_{1+e'} M + 4$, where M is the number of accessed data. It guarantees that the tree size is cut by at least a half. The following proposition proves this property and gives the time and space complexity.

PROPOSITION 2.1. *For a trace of N accesses to M data elements, the approximate analysis with a bounded relative error e ($1 > e > 0$) takes $O(N \log \log M)$ time and $O(\log M)$ space, assuming it uses a balanced tree.*

PROOF. The maximal tree size cannot exceed $4 * \log_{1+e'} M + 4$, or, $O(\log M)$, because of tree compression. Here $e' = \frac{e}{1-e}$. We now show that *TreeCompression* is guaranteed to reduce the tree size by at least a half every time it is invoked. Let n_0, n_1, \dots , and n_t be the sequence of tree nodes in the reverse time order. Consider each pair of nodes after compression, n_{2i} and n_{2i+1} . Let $size_i$ be the combined size of the two nodes. Let sum_{i-1} be the total size of nodes before n_{2i} , that is $sum_{i-1} = \sum_{j=0, \dots, i-1} size_j$. The new capacity of the node, $n_{2i}.capacity$, is $\lfloor sum_{i-1} * e' \rfloor$. The combined size, $size_i$, must

data declarations

TreeNode = **structure**(*time, weight, capacity, size, left, right, prev*)

root: the root of the tree

e': the bound on the error rate

algorithm *ReuseDistance*(*last, current*)

// *inputs are the last and current access time*

TreeSearchDelete(*last, distance*)

new = *TreeNode*(*current, 1, 1, 1, ⊥, ⊥, ⊥*)

TreeInsert(*new*)

if (*tree_size* $\geq 4 * \log_{1+e'} \text{root.weight} + 4$)

TreeCompression(*new*)

Assert(*compression more than halves the tree*)

end if

return *distance*

end algorithm

Fig. 2. Approximate analysis with a bounded relative error. Part I.

be at least $n_{2i}.capacity + 1$ and consequently no smaller than $sum_{i-1} * e'$; otherwise the two nodes should have been compressed. We have $size_0 \geq 1$ and $size_i \geq sum_{i-1} * e'$. By induction, we have $sum_i \geq (1 + e')^i$ or $i \leq \log_{1+e'} sum_i$. For a tree holding M data in $T_{compressed}$ tree nodes after compression, we have $i = \lfloor T_{compressed}/2 \rfloor$ and $sum_i = M$. Therefore, $T_{compressed} \leq 2 * \log_{1+e'} M + 2$. In other words, each compression must reduce the tree size by at least a half.

Now we consider the time cost. Assume that the tree is balanced and its size is T . The time for the tree search, deletion, and insertion is $O(\log T)$ per access. Tree compression happens periodically after a tree growth of at least $2 * \log_{1+e'} M + 2$ or $T/2$ tree nodes. Since at most one tree node is added for each access, the number of accesses between successive tree compressions is at least $T/2$ accesses. Each compression takes $O(T)$ time because it examines each node in a constant time, and the tree construction from an ordered list takes $O(T)$. Hence the amortized compression cost is $O(1)$ for each access. The total time is therefore $O(\log T + 1)$, or $O(\log \log M)$ per access. \square

2.1.2 Analysis with a bounded absolute error. For a cut-off distance C and a constant error bound B , the second approximation algorithm guarantees the precise measurement of distances shorter than C and an approximate measurement of longer distances with a bounded error B . It divides the access trace in two parts. The *precise trace* keeps the last accessed C elements. The *approximate trace* stores the remaining data in a tree where the capacity of each tree node is B . Periodically, the algorithm transfers data from the precise trace to the approximate trace. Our earlier paper describes a detailed algorithm and its implementation using a B-Tree in both the precise and approximate trace [Zhong et al. 2002].

We now generalize algorithm. The precise trace can use a list, a vector, or any type of trees, and the approximate trace can use any type of trees, as long as two minimal requirements are met. First, the size of the precise trace is bounded by a constant. Second, the minimal occupancy of the approximate tree is a constant fraction. To satisfy the first requirement, we need to transfer data from the precise trace to the approximate trace when

```

subroutine TraceSearchDelete(time, distance)
  // time is the last access time
  node = root; distance = 0
  while true
    node.weight = node.weight - 1
    if (time < node.time and node.prev exists and time ≤ 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
    else if (time > node.time)
      if (node.right not exists) break
      node = node.right
    else break
  end if
end while
  node.capacity =  $\max(\text{distance} * \frac{\epsilon}{1-\epsilon}, 1)$ 
  node.size = node.size - 1
  return distance
end subroutine TreeSearchDelete

subroutine TreeCompression(n)
  // n is the latest node in the tree
  distance = 0
  n.capacity = 1
  while (n.prev exist)
    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 =  $\max(\text{distance} * \frac{\epsilon}{1-\epsilon}, 1)$ 
    end if
  end while
  Build a balanced tree from the list and return the root
end subroutine TreeCompression

```

Fig. 3. Approximate analysis with a bounded relative error. Part II.

Table I. The asymptotic complexity of the reuse-distance measurement algorithms

Measurement Algorithms	Time	Space
trace as a stack (or list) [Mattson et al. 1970]	$O(NM)$	$O(M)$
trace as a vector-based interval tree [Bennett and Kruskal 1975; Almasi et al. 2002]	$O(N \log N)$	$O(N)$
trace as a search tree [Olken 1981] [Sugumar and Abraham 1993; Almasi et al. 2002]	$O(N \log M)$	$O(M)$
list-based aggregation [Kim et al. 1991]	$O(NS)$	$O(M)$
approx. w/ bounded absolute error	$O(N \log \frac{M}{B})$	$O(\frac{M}{B})$
approx. w/ bounded relative error	$O(N \log \log M)$	$O(\log M)$

N is the length of execution, M is the size of program data

the size of the former exceeds a threshold. To ensure a minimal occupancy of the approximate tree, we can simply merge two consecutive tree nodes if the combined size is no more than their capacity. The merge operation guarantees at least half utilization of the tree capacity. Therefore, the maximal size of the approximate tree is $\frac{2M}{B}$.

We implemented a splay tree [Sleator and Tarjan 1985] version of the algorithm. We will use only the approximate trace (the size of precise trace is set to 0) in distance-based sampling because it runs fastest among all analyzers, as shown in Section 3.

2.1.3 Comparisons with Previous Algorithms. The past 30 years have seen a steady stream of work in measuring reuse distance. We categorize previous methods by their organization of the data access trace. The first three rows of Table I show methods using a list, a vector, and a tree. In 1970, Mattson et al. [1970] published the first measurement algorithm. They used a list-based stack. Bennett and Kruskal [1975] showed that a stack was too slow to measure long reuse distances in database traces. They used a vector and built an m -ary interval tree on it. They also showed how to use blocked hashing in a pre-pass. In 1981, Olken [1981] implemented the first tree-based method using an AVL tree. He also showed how to compress the trace vector in Bennett and Kruskal’s method and improve the time and space efficiency to those of the tree-based algorithms. In 1994, [Sugumar and Abraham 1993] showed that a splay tree [Sleator and Tarjan 1985] has better memory performance. Their analyzer, *Cheetah*, is widely distributed with the SimpleScalar tool set. Recently, [Almasi et al. 2002] gave an algorithm that records the empty regions instead of non-empty cells in the trace. Although the asymptotic complexity remains the same, the actual cost of trace maintenance is reduced by 20% to 40% in vector and tree based traces. They found that the modified Bennett and Kruskal method was much faster than methods using AVL and red-black trees.

Kim et al. [1991] gave the first imprecise (but accurate) analysis method in 1991. Their method stores program data in a list, marks S ranges in the list, and counts the number of distances fell inside each range. The time cost per access is proportional to the number of markers smaller than the reuse distance. The space cost is $O(C)$, where C is the furthest marker. The method is efficient if S and C are bounded and not too large. It is not suitable for measuring the full length of the reuse distance, where S and C need to be proportional to M . Unlike approximate analysis, this method gives an accurate count of the the reuse distances fallen within a marked range.

In comparison, the two approximation methods, shown in the last two rows in Table I,

trade accuracy for efficiency especially the space efficiency. They can analyze traces with a larger amount of data and longer reuse distances. The methods are adjustable, and the cost is proportional to the accuracy. The analysis with a bounded relative error has the lowest asymptotic complexity in space and time, for an error rate that can be arbitrarily close to zero. It for the first time cuts the space complexity from linear to logarithmic. The time cost per access is $O(\log \log M)$, which is effectively constant for any practical M .

Reuse distance is no longer a favorable metric in low-level cache design because it cannot model the exact interaction between the cache and CPU, for example, the timing. However, for program traces, reuse distance determines the number of capacity misses for all cache sizes. Earlier work has also extended it to analyze interference in various types of set-associative cache [Hill 1987; Mattson et al. 1970]. Section 4 will discuss the uses of reuse distance analysis in performance analysis and optimization.

2.2 Distance-pattern prediction

Pattern recognition detects whether the recurrence pattern is predictable across different data inputs. We define the reuse, recurrence or locality pattern as the distribution of the reuse distance in a program, represented by a histogram and called the *reuse signature*. Based on the reuse signature of two or more training runs, distance-based pattern recognition constructs a parameterized pattern that predicts the reuse signature for other inputs of the program. The main parameter is the size of data involved in program recurrences. This is not the same as the size of data touched by a program. The next section will show how to obtain an estimate of this number through distance-based sampling. In this section, we assume it exists and refer to it indistinctively as the program data size.

Three factors strongly affect the prediction accuracy: the number of training inputs, the precision of the histogram collection, and the complexity of patterns. The number of training inputs needs to be at least two, although using more inputs may allow more precise recognition of common patterns. The precision of data collection is determined by the number of histogram bins. Using more bins leads to more precise distribution of the reuse distance but lower speed in data collection and pattern prediction. The third factor is the complexity of patterns. We now describe the collection of histograms and the recognition of their patterns.

2.2.1 Collecting distance and reference histograms. We use two types of histograms. In a *reuse-distance histogram* (*distance histogram* in short), the x -axis is reuse-distance ranges, and the y -axis is the percentage of data accesses in each range or the *size* of each bin. The range of the distance can be in a *linear scale*, e.g. $[0k, 1k)$, $[1k, 2k)$, $[2k, 3k)$, \dots , or a *log scale*, e.g. $[0, 1)$, $[1, 2)$, $[2, 4)$, $[4, 8)$, \dots , or a *log-linear scale* where the ranges below 2048 are in a log scale and those above 2048 in a linear scale. Figure 4(a) shows the reuse-distance histogram of a fluid dynamics simulation program, *SP*, in the log scale.

A *reference histogram* is a transpose of the reuse-distance histogram. The x -axis is groups of data accesses, sorted by the reuse distance. The y -axis is the average reuse distance of each bin. All bins have the same size. Figure 4(b) is the reference histogram of *SP* in 100 bins. It first gives the average distance for the 1% shortest reuse distances, then the average for the next 1% shortest reuse distances, and so on.

The reference histogram complements the distance histogram. A distance histogram controls the range of the distance of a bin but not its size. A reference histogram ensures equal size but the range of distance may not be uniform. Controlling the size of a bin

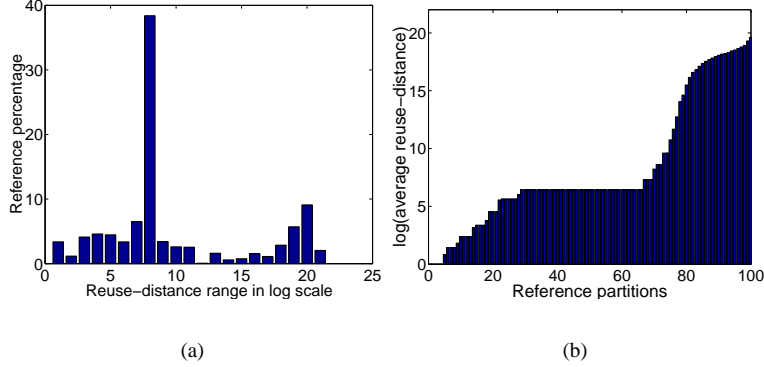


Fig. 4. The histograms of SP with the input size 28^3 . (a) the reuse-distance histogram (b) the reference histogram

serves two purposes. First, it isolates the effect of non-recurrent parts of the program. Some instructions are executed per execution; some are repeated per program data. When the data size becomes sufficiently large, the effect of the former group diminishes into at most a single bin of the histogram. Second, it offers a trade-off between information loss and computation/space efficiency. For dense regions in the distance histogram, where a large portion of memory accesses have similar reuse distances, the reference histogram uses short ranges to increase accuracy. For sparse regions in the distance histogram, the reference histogram uses large ranges to reduce the total number of bins. The size of the bin determines the granularity and the cost of prediction. A bin size of 1% means that we need to analyze only 100 bins. At the same time, we do not predict the distribution of distances within each 1% of memory references.

We first collect the distance histogram through profiling. We then compute the reference histogram by traversing the distance histogram and calculating the average distance for each fraction of memory references. Getting a precise histogram incurs a high space cost. We again use approximation since we do not measure precise distances anyway. In the experiment, we collect the distance histogram using a log-linear scale. The size of bins is a power of 2 up to 2048 and then it is 2048 for each bin. To improve precision, we calculate the average distance within each bin and use the average distance as the distance of all references in the bin when converting it to the reference histogram. Each bin in our reference histogram holds 0.1% of the total data accesses. The cost and accuracy of the histogram collection can be adjusted by simply changing the size of bins in both types of histograms.

2.2.2 Distance patterns. Given two reference histograms from two training runs, we construct a formula to represent the distance value of each bin. We denote the bins in the two histograms as $\langle g_1, g_2, \dots, g_B \rangle$ and $\langle \hat{g}_1, \hat{g}_2, \dots, \hat{g}_B \rangle$ and denote the average reuse distances of g_i and \hat{g}_i by d_i and \hat{d}_i respectively, $i = 1, 2, \dots, B$, where B is the number of bins. Let s and \hat{s} be the data size of the two training runs. We can use linear fitting to find the closest linear function that maps the data size to the reuse distance. Specifically,

we find the two coefficients, c_i and e_i , that satisfy the following two equations.

$$d_i = c_i + e_i * f_i(s) \quad (1)$$

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

where d_i and \hat{d}_i is the average reuse distance of i^{th} reference group when the input size is s and \hat{s} , c_i and e_i are two parameters to be determined by the prediction method, and f_i is a function. Assuming the function f_i is known, the two coefficients uniquely determine the distance for any other data size. The formula therefore defines the pattern for each bin, parameterized by the data size. The program pattern is the aggregation of all bins. The pattern is more accurate if more training runs are used, as shown later. The minimal number of training inputs is two.

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

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

The first is 0. We call it a constant pattern because reuse distance does not change with the data size. A bin i has a constant pattern if its average reuse distance stays the same in the two runs, i.e. $d_i = \hat{d}_i$. The second is s . We call it a linear pattern. A bin i has a linear pattern if the average distance changes linearly with the change in program input size, i.e. $\frac{d_i}{\hat{d}_i} = c + e \frac{s}{\hat{s}}$, where c and k are constants. Constant and linear are the lower and upper bound of the reuse distance changes. Between them are the three sub-linear patterns. The pattern $s^{1/2}$ happens in two-dimensional problems such as matrix computation. 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.

For each bin of the two reference histograms, we calculate the ratio of their average distance, 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 sub-linear patterns from a different number of dimensions. In our experiments, the dimension of the problems was given as an input to the analyzer. This can be automated by trying all dimension choices and using the best overall fit.

2.2.3 Regression-based prediction. Using more than two training inputs may produce a better prediction, because it reduces the noise from imprecise reuse distance measurement and reference histogram construction. According to the regression theory, more data can reduce the effect of noises and reveal a pattern closer to the real pattern [Rawlings 1988]. Accordingly, we apply a regression method on more than two training inputs.

The extension is straightforward. For each input, we have an equation as shown in Equation 1. For each bin, instead of two linear equations for two unknowns, we have as many equations the number of training runs. We use the *Least square regression* [Rawlings 1988] to determine the best values for the two unknowns. We use 3 to 6 training inputs in our experiment. Although more training data can lead to better results, they also lengthen the profiling process. We will show that a small number of training inputs is sufficient to gain high prediction accuracy.

2.2.4 Multi-model prediction. An important source of imprecision comes from the limited granularity because the above methods assume that all accesses in the same bin

have the same pattern. We call them single-model prediction. We now describe multi-model prediction, which allows multiple patterns inside each bin of a histogram. We will use the terms pattern and model interchangeably.

In multi-model prediction, the reuse distance function of a bin is as follows.

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

where, s is the size of input data, $h_i(s)$ is the y -axis value of the i^{th} bin for input of size s , and $\varphi_{m_1} \dots \varphi_{m_j}$ are the functions corresponding to all possible patterns or models.

Each $h_i(s)$ is a linear combination of all the possible models of the standard histogram:

$$\varphi_{m_1}(s_0, 1), \varphi_{m_1}(s_0, 2), \dots, \varphi_{m_1}(s_0, B), \varphi_{m_2}(s_0, 1), \varphi_{m_2}(s_0, 2), \dots, \varphi_{m_2}(s_0, B), \dots, \varphi_{m_j}(s_0, 1), \varphi_{m_j}(s_0, 2), \dots, \varphi_{m_j}(s_0, B)$$

where, B is number of bins in the standard histogram.

Figure 5 shows an example of the multi-model prediction. We arbitrarily pick one of the training inputs as the *standard input*. In this example, s_0 is the size of the standard input (the other training inputs are not showed in the figure.) Its reuse distance histogram, called *standard histogram*, has 12 bins, and each bin has two models—the constant and the linear pattern. Using multi-model analysis, the standard histogram in Figure 3(a) is decomposed into the constant and the linear pattern in Figure 3(b) and 3(c). Given another program data size, e.g. $8 * s_0$, we predict the reuse distance accordingly for the two patterns, Figure 5(d) for the constant pattern and 5(e) for the linear pattern. The constant pattern remains unchanged, and the distance of the accesses in the linear pattern is lengthened by a factor of 8. The x -axis is in log scale, so the bar in the linear pattern moves right by 3 points. The final prediction is the combination of the predicted constant and linear parts, shown in Figure 3(f).

As an example of the actual calculation, take a program that has both constant and linear patterns. For easy description, we assume:

$$\text{range 0: } [0,1); \text{ range 1: } [1,2); \text{ range 2: } [2,4); \text{ range 3: } [4,8), \dots$$

For another input of size $s_1 = 3 * s_0$, we calculate the y -axis value of range $[4, 8)$ as follows:

$$h_3(s_1) = \varphi_0(s_0, 3) + \varphi_1(s_0, r)$$

where, r is the range $[\frac{4}{3}, \frac{8}{3})$. We calculate $\varphi_1(s_0, r)$ as

$$\varphi_1(s_0, r) = \varphi_1(s_0, r_1) + \varphi_1(s_0, r_2)$$

where, $r_1 = [\frac{4}{3}, 2)$ and $r_2 = [2, \frac{8}{3})$. We assume the reuse distance has uniform distribution in each bin. Hence,

$$\begin{aligned} \varphi_1(s_0, r_1) &= \left(\frac{2-4/3}{2-1}\right)\varphi_1(s_0, 1) = \frac{2}{3}\varphi_1(s_0, 1) \\ \varphi_1(s_0, r_2) &= \left(\frac{8/3-2}{4-2}\right)\varphi_1(s_0, 2) = \frac{2}{3}\varphi_1(s_0, 2) \end{aligned}$$

Finally, we calculate $h_3(s_1)$ as

$$h_3(s_1) = \varphi_0(s_0, 3) + \frac{2}{3}\varphi_1(s_0, 1) + \frac{2}{3}\varphi_1(s_0, 2)$$

After processing each $h_i(s)$ of all training inputs in a similar manner, we obtain an equation group. The unknown variables are the models in the standard histogram. Regression

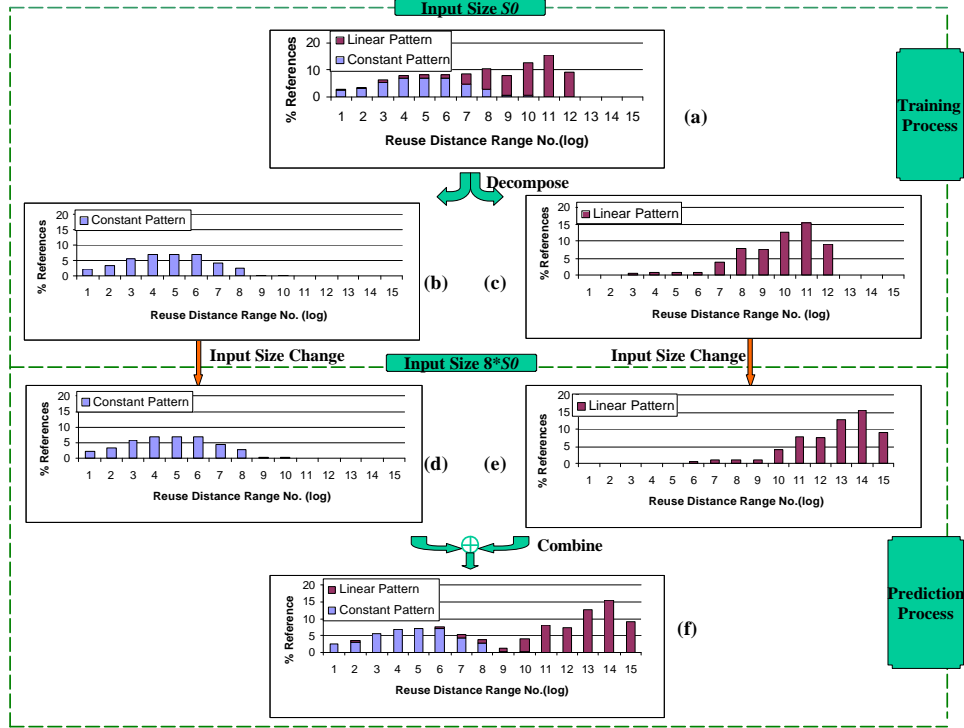


Fig. 5. An example of multi-model prediction. Part (a) is the reuse-distance histogram of the input s_0 . After regression analysis on this and other training inputs, the histogram is decomposed into two parts—the constant and the linear parts in (b) and (c). Given a new input $8 * s_0$, the constant part keeps unchanged, shown in (d), and the distance of the linear part multiplies by 8 times, shown in (e). The x -axis is in the log-2 scale. The reuse-distance histogram of the new input is the combination of (d) and (e), showed in Figure (f).

techniques are used to find the models that fit all training histograms with the least error. An important assumption is that the percentage of memory accesses in each model remains unchanged for different inputs. We will show later that the assumption is valid for a wide range of programs.

A multi-model method does not depend on the type of histograms. It can use distance histograms with log or linear scales. It can also use reference histograms. The equations are constructed and solved in the same manner.

2.2.5 Space and time complexity. The space and time cost of the prediction methods are $O(B)$, where B is the size of the histogram. The size is $O(M)$ for a linear-scale distance histogram, $O(\log M)$ for a log-scale distance histogram, and $O(1)$ for a reference histogram. Although most costly, the linear-scale histogram has higher precision, which

can produce better results especially when using small-size training runs. The log scale is needed to separate reference groups with short reuse distances. The reference histogram has a constant cost, but it raises a problem: how to choose the best number of bins. The range of choices is large. The maximal number of groups, is the number of memory references in the smallest training run. We will see in the evaluation section that the prediction accuracy depends heavily on the choice of the number of bins.

2.2.6 Limitations. Although the analysis can handle any sequential program, the generality comes with several limitations. For high-dimensional data, pattern prediction requires that different inputs have a similar shape, in other words, their size needs to be proportional or close to proportional in all dimensions. Otherwise, a user has to train the analyzer for each shape. In our future work, we will combine the pattern analyzer with a compiler to predict for all shapes. All high-dimensional data we have seen come from scientific programs, for which a compiler can collect high-level information. In addition, predicting reuse pattern does not mean predicting execution time. The prediction gives the percentage distribution but not the total number of memory accesses, just as loop analysis can know the dependence but not the total number of loop iterations.

Once the pattern is recognized from training inputs, we can predict constant patterns in another input statically. For other patterns, we need the data size of the other input, for which we use distance-based sampling.

2.3 Distance-based sampling

The purpose of data sampling is to estimate data size in a program execution. For on-line pattern prediction, the sampler creates a twin copy of the program and instruments it to generate data access trace. When the program starts to execute, the sampling version starts to run in parallel until it finds an estimate of data size. Independent sampling requires that the input of the program be replicated, and that the sampling run do not produce side effects.

The sampling is *distance-based*. It uses the reuse distance analyzer and monitors each measured distance. It records only long-distance reuses because they reveal global patterns. When the reuse distance is above a threshold (the *qualification threshold*), the accessed memory location is taken as a data sample. A later access to a data sample is recorded as an access sample if the reuse distance is over a second threshold (the *temporal threshold*). To avoid picking too many data samples, it requires that a new data sample to be at least a certain space distance away (the *spatial threshold*) in memory from existing data samples. Given the sequence of access samples of a data sample, the sampler finds *peaks*, which are time samples whose height (reuse distance) is greater than that of its preceding and succeeding time samples.

The sampler runs until seeing the first k peaks of at least m data samples. It then takes the appropriate peak as the data size. The peak does not have to be the actual data size. It just needs to be proportional to the data size in different inputs. We use the same sampling scheme to determine data size in both training and prediction runs. For most programs we tested, it is sufficient to take the first peak of the first two data samples. An exception is *Apsi*. All its runs initialize the same amount of data as required by the largest input size, but smaller inputs use only a fraction of the data in the computation. We then use the second peak as the program data size. More complex cases happen when early peaks do not show a consistent relation with data size, or the highest peak appears at the end of a program.

We identify these cases during pattern recognition and instruct the predictor to predict only the constant pattern.

The sampling can be improved by more intelligent peak finding. For example, we require the peak and the trough differ by a certain factor, or use a moving average to remove noises. The literature on statistics and time series is a rich resource for sample analysis. For pattern prediction, however, we do not find a need for sophisticated methods yet because the (data-size) peak is either readily recognizable at the beginning or it is not well defined at all.

The cost of distance-based sampling is significant since it needs to measure reuse distance of every memory reference until peaks are found. The analysis does not slow the program down since it uses a separate copy. It only lengthens the time taken to make a prediction. For minimal delay, it uses the fastest approximation analyzer. It can also use selective instrumentation and monitor only distinct memory references to global and dynamic data [Ding and Zhong 2002]. For long-running programs, this one-time cost is insignificant. In addition, many programs have majority of memory references reused in constant patterns, which we predict without run-time sampling.

Another use of distance-based sampling is to detect phases in a program. For this purpose, we continue sampling through the entire execution. Time segments between consecutive peaks are phases. A temporal graph of time samples shows recurrent accesses in time order and the length and shape of each recurrence. The evaluation section will use phase graphs to understand the results of pattern prediction.

Finding the first few peaks of the first few data samplings is an unusual heuristic because it is not based on keeping track of a particular program instruction or a particular data item. The peaks found by sampling in different program executions do not have to be caused by the same memory access to the same data. Very likely they are not. In programs with input-dependent control flow, one cannot guarantee the execution of a function or the existence of a dynamic data item. Distance-based sampling allows correlation across data inputs without relying on any pre-assumed knowledge about program code or its data.

3. EVALUATION

3.1 Reuse distance measurement

Figure 6 compares the speed and accuracy for eight analyzers, which we have described in Section 2.1. *BK-2*, *BK-16*, and *BK-256* are vector-based k -ary tree analyzers with k equal to 2, 16, and 256 [Bennett and Kruskal 1975]. *KHW* is list-based aggregation with three markers at distance 32, 16K, and the size of analyzed data [Kim et al. 1991]. We re-implemented it since the original no longer exists. *Cheetah* [Sugumar and Abraham 1993] uses a splay-tree. *ZDK-2k* and *Sampling* are approximate analysis with the error bound $B = 2048$, as described in Section 2.1.2. *ZDK-2k* uses a B-tree and a mixed trace [Zhong et al. 2002]. *Sampling* uses a splay tree and only the approximate trace. 99% is the analysis with the bounded relative error $e = 1\%$. The input program traverses M data twice with a reuse distance equal to $M/100$. To measure only the cost of reuse-distance analysis, the hashing step was bypassed by pre-computing the last access time (except for *KHW*, which does not need the access time). The timing was collected on a 1.7 GHz Pentium 4 PC with 800 MB of main memory. The programs were compiled using *gcc* with the optimization flag *-O3*.

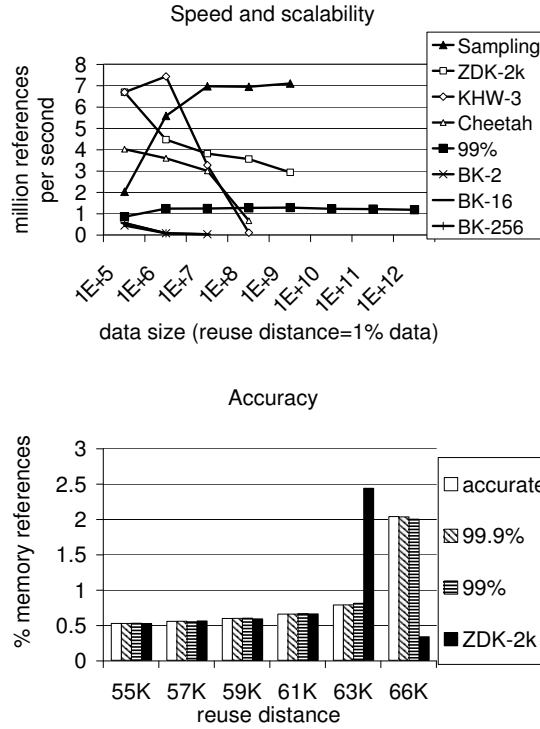


Fig. 6. Comparison of analyzers

Compared to the accurate methods, approximate analysis is faster and more scalable with data size and distance length. The vector-based methods have the lowest speed. *KHW* with three markers is fastest (7.4 million memory references per second) for small and medium distances but is not suited for measuring very long reuse distances. *Cheetah* achieves an initial speed of 4 million memory references per second. All accurate analyses run out of physical memory at 100 million data. *Sampling* has the highest speed, around 7 million memory references per second, for large data sizes. *ZDK-2k* runs at a speed from 6.7 million references per second for 100 thousand data to 2.9 million references per second for 1 billion data. *Sampling* and *ZDK-2k* do not analyze beyond 4 billion data since they use 32-bit integers.

The most scalable performance is obtained by the analyzer with 99% accuracy ($e = 1\%$), shown by the line marked 99%. We use 64-bit integers in the program and test it for up to 1 trillion data. The asymptotic cost is $O(\log \log M)$ per access. In the experiment, the analyzer runs at an almost constant speed of 1.2 million references per second from 100 thousand to 1 trillion data. The consistent high speed is remarkable considering that the data size and reuse distance differs by eight orders of magnitude. The speed is so predictable that when we first ran 1 trillion data test, we estimated that it would finish in 19.5 days: It finished half a day later, which was a satisfying moment considering that prediction is the spirit of this work. If we consider an analogy to physical distance, the

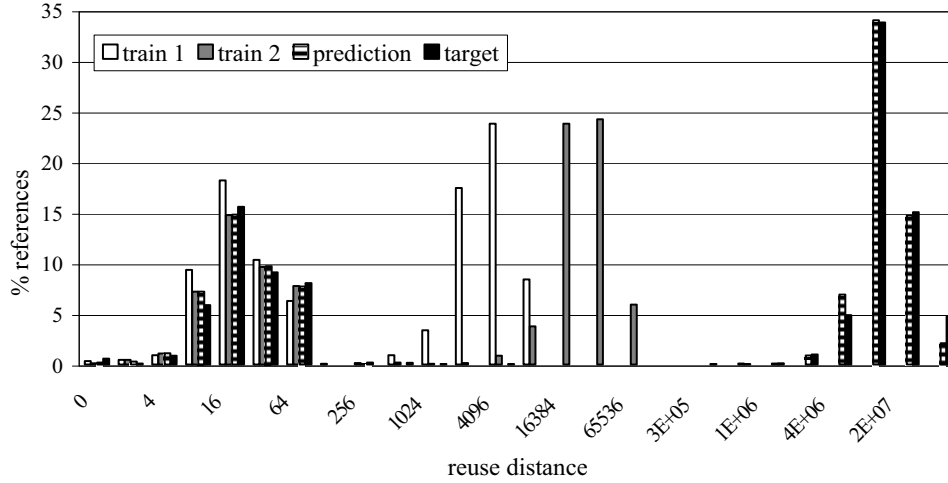


Fig. 7. Pattern prediction for Spec2K/Lucas

precise methods measure the distance in miles, the approximation method measures light years with the same speed.

The lower graph of Figure 6 compares the accuracy of approximation on a partial histogram of *FFT*. The y -axis shows the percentage of memory references, and the x -axis shows the distance in a linear scale between 55 thousand and 66 thousand with an increment of 2048. 99.9% and 99% approximation ($e = 0.1\%$ and $e = 1\%$ respectively), shown by the second and the third bars in each group, closely match the accurate distance. Their overall error is about 0.2% and 2% respectively. The bounded absolute error with a bound 2048, shown by the last bar, has a large misclassification near the end, although the error is no more than 4% of the actual distance. In terms of the space overhead, accurate analyzers need 67 thousand tree or list nodes, *ZDK-2k* needs 2080 tree nodes, 99% needs 823, and 99.9% needs 5869. The last two results show that the cost and the accuracy of the approximate analysis are adjustable.

3.2 Pattern prediction

Figure 7 shows the result of pattern prediction for *Lucas* from Spec2K, which is representative in our test suite. The graph shows four reuse signatures. The first two are for the two training inputs. They are used by the analyzer to recognize distance patterns and to make the locality prediction for a third input. The analyzer runs the program with the third input, samples 0.4% of its execution, finds the data size, and predicts the reuse signature shown by the third bar in each group. The prediction matches closely with the measured signature shown by the fourth bar of each group. The two histograms overlap by 95%. The accuracy is remarkable considering that the target execution has 480 times more data and 320 times longer trace than the larger one of the two training runs has and 3650 times more data and 3220 times longer traces than the smaller one has. The correct prediction of the peaks on the far side of the histograms is especially telling because they differ from the peaks of the training inputs not only in position but also in shape and height.

Table II. Prediction accuracy and coverage for floating-point programs

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

Table III. Prediction accuracy and coverage for integer programs

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

Table II and III show the effect of pattern prediction on 15 benchmarks, including 7 floating-point programs and 6 integer programs from SPEC95 and SPEC2K benchmark suites, and 2 additional programs, *SP* from NASA and a two-dimensional *FFT* kernel. We reduce the number of iterations in a program if it does not affect the overall pattern. We compile the tested programs with the DEC compiler using the default optimization (*-O3*). Different compiler optimization levels may change the reuse pattern but not the accuracy of our prediction. We use Atom [Srivastava and Eustace 1994] to instrument the binary code to collect the address of all loads and stores and feed them to our analyzer, which treats each distinct memory address as a data element.

Column 1 and 2 of the table in Table II give the name and a short description of the test programs. The programs are listed in the decreasing order of the average reuse distance. Their data inputs are listed by the decreasing order of the data size. For each input, Column 5 shows the data size or the number of distinct data, and Column 6 and 7 give the number of data reuses and average reuse distance normalized by the data size. The programs have up to 36 million data, 130 billion memory references, and 5 million average reuse distance. The table shows that these are a diverse set of programs: no two programs are similar in data size or execution length. Although not shown in the table, the programs have different reuse distance histograms (even though the average distance is a similar fraction of the data size in a few programs). In addition, the maximal reuse distance is very close to the data size in each program run.

The third column lists the patterns in benchmark programs, which can be constant, linear, or sub-linear. Sub-linear patterns include *2nd root* ($x^{1/2}$) and *3rd roots* ($x^{1/3}$ and $x^{2/3}$). Floating-point programs generally have more patterns than integer programs.

The prediction accuracy is shown by the Column 8 and 9 of the table. Let x_i and y_i be the size of i th bar in predicted and measured histograms. The cumulative difference, E , is the sum of $|y_i - x_i|$ for all i . In the worst case, E is 200%. We use $1 - E/2$ as the accuracy. It measures the overlap between the two histograms, ranging from 0% or no match to 100% or complete match. The accuracy of *Lucas* with sampled size is 95%, shown in Figure 7.

As we discussed in Section 2.2, reuse signature pattern is parameterized by the size of data involved in program recurrences. This is not the same as the size of data touched by a program and we use distance-based sampling described in Section 2.3 to estimate the number. The accuracy of prediction based on this sampling estimation is given by Column 9. As a comparison, Column 8 lists the prediction accuracy based on program data size. For many benchmarks, the two columns give comparable results, which indicates a proportional relation between the size of data involved in program recurrences and the size of data visited in whole program execution. But this does not hold for all programs. An obvious example is *Apsi*. For different input parameters, the program initializes the same amount of data, but only uses part of it in computation. Therefore reuse signature pattern built on program data size can not capture the locality behavior for *Apsi* and the prediction accuracy is only 27%. In general, prediction based on sample size has a higher average accuracy.

We use three different input sizes for all programs except for *Gcc*. Based on 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. In fact, the prediction works for any data input over a reasonable size. The table shows that both forward and backward predictions are very accurate. Backward prediction is

generally better except for *Lucas*—because the largest input is about 500 times 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*, 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 others, the coverage starts after the data-size peak is found in the execution trace. Let N be the length of the execution trace, P be the logical time of the peak, then the coverage is $1 - P/N$. For programs using a reduced number of iterations, N is scaled up to be the length of the full execution. To be consistent with other SPEC programs, we let *SP* and *FFT* to 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 only because it repeats compression multiple times, an unlikely case in real uses. *Li* has random peaks that cannot be consistently sampled. We predict *Li* based on only the constant pattern. The average coverage is 99.1%.

The reported coverage is for predicting simulation results. Instead of measuring reuse distance for the whole program, we can predict it by sampling on average 1.2% of the execution. To predict a running program, the coverage is smaller because the instrumented program (for sampling) runs much slower than the original program. Our fastest analyzer causes a slowdown by factors ranging from 20 to 100. For a slowdown of 100, 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 them, the average coverage is 99.73%, suggesting 73% time coverage 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 our test suite do not need sampling.

Most inputs are test, train, and reference inputs from SPEC. For *GCC*, we pick the largest and two random ones from the 50 input files in its *ref* directory. *SP* and *FFT* do not come from SPEC, so we randomly pick their input sizes (*FFT* needs a power of two matrix). We change a few inputs for SPEC programs, shown in Column 4. *Tomcatv* and *Swim* has only two different data sizes. We add in more inputs. All inputs of *Hydro2d* have a similar data size, but we do not make any change. The test input of *Twolf* has 26 cells and is too small. We randomly remove half of the cells in its train data set to produce a test input of 300 cells. *Applu* is a benchmark with long-time execution, so we replace reference input with a smaller one to save data collection time. Finally, *Apsi* uses different-shape inputs of high-dimensional data, for which our current predictor cannot make an accurate prediction. We change the shape of its largest input.

3.3 Regression-based multi-model prediction

We now turn our attention to a sub-set of the test programs that have multiple inputs so that we can test regression-based and multi-model prediction. The first column of Table IV gives results obtained by Ding and Zhong’s original method. Other columns show the accuracy of the new methods. All methods are based on histograms given by the same reuse-distance analyzer and the input sizes given by the same distance-based sampler.

The results in Table IV show that for most programs, all regression-based predictions produce better results than the method using two training inputs. Therefore, regression on multiple inputs indeed improves prediction accuracy. Except for *SWIM*, multi-model

logarithmic scale method is comparable to the best predictors. However, it is the most efficient among all methods because the storage space of a log-scale histogram is 95% less than other histograms.

SWIM shows inconsistent results. The multi-model logarithmic scale has poor result for *SWIM*, but multi-model log-linear scale and single-model methods give very accurate predictions. Figure 8 shows the distance histogram of *SWIM*. Note it has a high peak in a very small reuse distance range. Multi-model logarithmic scale uses *log* scale ranges. It assumes that the reuse distance is evenly distributed in each range, which brings significant noise in a histogram using a log scale.

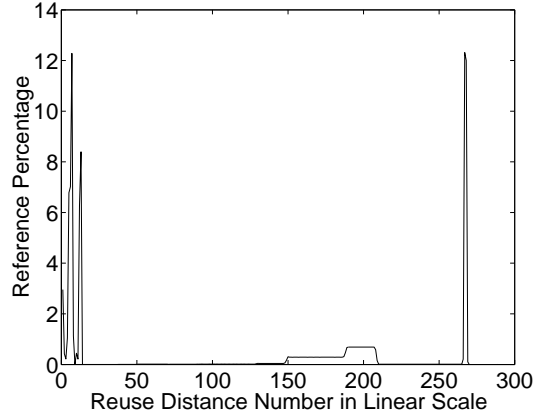
The performance of multi-model log-linear scale method is slightly better than multi-model logarithmic scale method for the first four benchmarks and much better for *SWIM*. However, log-linear scale costs more than 20 times in space and computations than logarithmic scale for most programs. The multi-model method based on reference histograms outperforms single-model two-input method for two out of six programs. It gives the highest accuracy for *FFT*. As we explained in Section 2.2.4, this approach is very flexible and its performance depends heavily on the number of groups. In our experiment, we tried 7 different numbers of groups for each benchmark and presented the highest accuracy, but finding the maximal accuracy requires trying thousands of choices. The result for *FFT* shows the potential of this method, but the overhead of finding the best result is prohibitively high.

Table IV. The prediction accuracy of the five methods

Bench- mark	Single Model		Multi-model			Max. Num. Inputs
	RF-Hist. 2 inputs	RF-Hist. 3+ inputs	Log RD-Hist.	Log-linear RD-Hist.	RF-Hist.	
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
Tomcatv	92.36	94.38	94.70	96.69	88.89	5
FFT	72.82	93.30	93.22	93.34	95.26	3
GCC	98.61	97.95	98.83	98.91	93.34	4
Average	90.04	95.63	93.21	94.90	88.53	4.7

So far the prediction uses a relatively large input, so that different patterns are separated from each other. It is important for the two single-model methods that different patterns do not overlap, because the methods assume that only one model exists in each range. In addition, the composition of patterns in a bin is likely constant when the input size is large. This is required by both single-model and multi-model based methods.

When the training uses small input sizes, single-model methods are not expected to perform well, but multi-model methods should work as well as in large input sizes. Table V shows the performance of the four methods on small size inputs of *SP* benchmark. We do not show the results of the multi-model method using reference histograms because it is difficult to tune. The results show that multi-model log-linear scale method is significantly more accurate than other methods. The good accuracy shows that the percentage of each model remains unchanged even for small inputs. The performance of multi-model logarithmic scale method is worse than the log-linear scale method because of the low precision in

Fig. 8. The reuse-distance histogram of *SWIM*

logarithmic scale histograms. Although multi-model log-linear scale method needs more computation and more space than the logarithmic scale method, this cost is not an issue for small-size inputs.

Table V. 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	89.5
	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	80.3
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

We compare the five methods in Table VI. Methods A, B, and E use reference histograms while Methods C and D use reuse distance histograms. The latter group needs not to transform between the two histograms. Using log-scale distance histograms, Method C saves 20 times in space and computation compared to other methods. The last method can also save cost because it can freely select the number of bins, but it is hard to pick the right number. While efficient, Method C loses information because it assumes a uniform distribution in large ranges. It could not accurately predict the locality of *SWIM*, where a large number of reuse distances stay in a narrow range. In that case, other methods produce much better results because they use shorter, linear-scale ranges. Among them, multi-model prediction methods predict with a higher accuracy than the single-model methods do if multiple patterns overlap in the same bin. Overlapping often happens for inputs of a small size.

Table VI. Features of Various Reuse Distance Prediction Methods

Methods	A	B	C	D	E
Models	<i>single</i>	<i>single</i>	<i>multiple</i>	<i>multiple</i>	<i>multiple</i>
Histogram	Reference	Reference	Distance	Distance	Reference
Histogram x -axis	log-linear	log-linear	log	log-linear	log-linear
Number of inputs	2	3+	3+	3+	3+
Number of models per bin	1	1	2+	2+	2+

In summary, regression analysis significantly improves the accuracy of reuse distance prediction, even with only a few training inputs. The multi-model method using logarithmic histograms can save 95% space and computations and still keep the best accuracy in most programs, although it is not as consistent as those methods using log-linear histograms. Space efficiency is necessary for fine-grained analysis of the patterns in individual program data. It is a good choice when efficiency is important. The single-model multi-input method has the highest accuracy, but it cannot accurately model small-size inputs. It is the best choice when one can tolerate a high profiling cost. The multi-model method using log-linear scale histograms is the best for small input sizes, where different models tend to overlap each other. It is also efficient because the input size is small.

3.3.1 *Comparisons with Profiling Analysis.* Most profiling methods use the result from training runs as the prediction for other runs. An early study measured the accuracy of this scheme in finding the most frequently accessed variables and executed control structures in a set of dynamic programs [Wall 1991]. We call this scheme *constant prediction*, which in our case uses the reuse-distance histogram of a training run as the prediction for other runs. For programs with only constant patterns, constant prediction is the same as our method. For the other 11 programs, the worst-case accuracy is the size of the constant pattern, which is 55% on average. The largest is 84% in *Twolf*, and the smallest 28% in *Apsi*. The accuracy can be higher if the linear and sub-linear patterns overlap in training and target runs. It is also possible that the linear pattern of a training run overlaps with a sub-linear pattern of the target run. However, the latter two cases are not guaranteed; in fact, they are guaranteed not to happen for certain target runs. The last case is a faulty match since those accesses have different locality.

For several programs, the average reuse distance is of a similar fraction of the data size. For example in *Swim*, the average distance is 40% of the data size in all three runs. This suggests that we can predict the average reuse distance of other runs by the data size times 40%. This prediction scheme is in fact quite accurate for programs with a linear pattern (although not for other programs). When the data size is sufficiently large, the total distance will be dominated by the contribution from the linear pattern. The average distance is basically the size of the linear pattern, which in *Swim* is 40% of all references. This scheme, however, cannot predict the overall distribution of reuse distance. It also needs to know the input data size from distance-based sampling. The total data size is not always appropriate. For example, *Apsi* touches the same amount of data regardless of the size of the data input.

As a reality check, we compare with the accuracy of random prediction. If a random distribution matches the target distribution equally well, our method would not be very good.

A distribution is an n -element vector, where each element is a non-negative real number and they sum to 1. Assuming any distribution is equally likely, the probability of a random prediction has an error α or less is equal to the number of distributions that are within α error to the target distribution divided by the total number of possible distributions. We calculate this probability using n -dimensional geometry. The total number of such vectors is equal to the surface volume on a corner cut of an n -dimensional unit-size hypercube. The number of distributions that differ by α with a given distribution equals to the surface volume of a perpendicular cut through 2^{n-1} corner cuts of an α -size hypercube. The probability that a random prediction yields at least $1 - \alpha$ accuracy is α^{n-1} , the ratio of the latter volume to the former. For the program *Lucas* shown in Figure 7, n is 26 and the probability of a random prediction achieving over 95% accuracy is 0.05^{25} or statistically impossible.

3.3.2 *A case study.* The program *Gcc* compiles C functions from an input file. It has dynamic data allocation and input-dependent control flow. A closer look at *Gcc* helps to understand the strength and limitation of our approach. We sample the entire execution of three inputs, *Spec95/Gcc* compiling `ccccp.i` and `ampt.jp.i` and *Spec2K/Gcc* compiling `166.i`. The three graphs in Figure 9 show the time samples of one data sample. Other data samples produce similar graphs. The upper two graphs, `ccccp.i` and `ampt.jp.i`, link time samples in vertical steps, where the starting point of each horizontal line is a time sample. The time samples for `166.i` are shown directly in the bottom graph.

The two upper graphs show many peaks, related to 100 functions in the 6383-line `ccccp.i` and 129 functions in the 7088-line `ampt.jp.i`. Although the size and location of each peak appear random, their overall distribution is 96% to 98% identical between them and to three other input files (shown previously in Table II). The consistent pattern seems to come from the consistency in programmers' coding, for example, the distribution of function sizes. Our analyzer is able to detect such consistency in logically unrelated recurrences. On the other hand, our prediction is incorrect if the input is unusual. For example for `166.i`, *Gcc* spends most of its time on two functions consisting of thousands lines of code. They dominate the recurrence pattern, as shown by the lower graph in Figure 9. Note the two orders of magnitude difference in the range of x - and y -axes. Our method cannot predict such unusual pattern.

Our analyzer is also able to detect the similarity between different programs. For example, based on the training runs of *Spec95/Gcc*, we can predict the reuse pattern of *Spec2K/Gcc* on its test input (the same as the test input in *Spec95*) with 89% accuracy.

While our initial goal was to predict programs with regular recurrence patterns, *Gcc* and other programs such as *Li* and *Vortex* took us by surprise. They showed that our method also captured the cumulative pattern despite the inherent randomness in these programs. High degree of consistency was not uncommon in applications including program compilation, interpretation, and databases. In addition, *Gcc* showed that our method could predict the behavior of a later version of software by profiling its earlier version.

4. USES OF DISTANCE-BASED LOCALITY PATTERNS

Distance-based locality analysis is unique because it is generally applicable as program profiling is, yet it describes the program behavior under all inputs as compiler analysis does. This section briefly surveys the current and potential uses of the distance-based locality information.

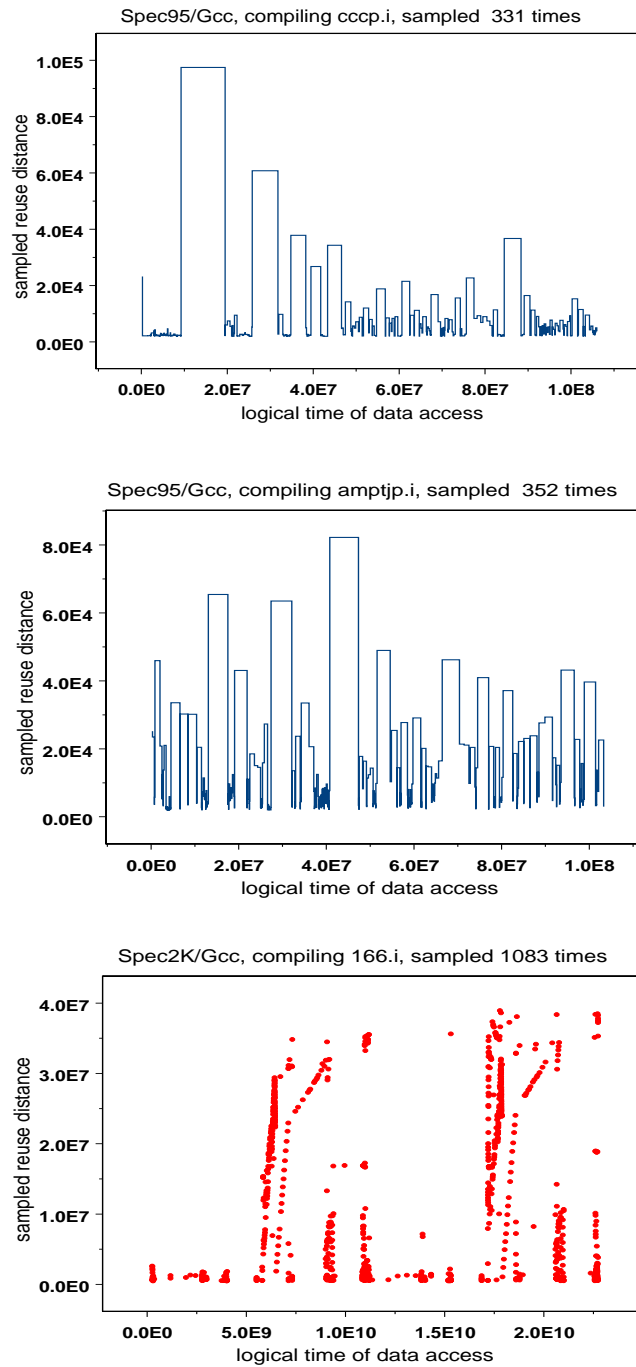


Fig. 9. Sampling results of *Gcc* for three inputs

Performance modeling Reuse distance gives richer information about a program than a cache miss rate does. At least four compiler groups have used reuse distance for different purposes: to study the limit of register reuse [Li et al. 1996] and cache reuse [Huang and Shen 1996; Ding 2000; Zhong et al. 2002], and to evaluate the effect of program transformations [Ding 2000; Beyls and D’Hollander 2001; Almasi et al. 2002; Zhong et al. 2002]. Recently, Zhong et al. [2003] and [Marin and Mellor-Crummey 2004] applied distance-based analysis to memory blocks and reported accurate miss-rate prediction across different program inputs and cache sizes. Using the predictor described in this article, Fang et al. [2004] examined the reuse pattern of each program instruction of 11 SPEC2K CFP benchmark programs and predicted the miss rate of 90% of instructions with a 97% accuracy. They used to prediction tool to identify *critical* instructions that generate the most cache misses.

Program transformation Beyls and D’Hollander [2002] is the first to show real performance improvement using the reuse distance information. They used reuse distance profiles to generate cache hints, which tell the hardware whether and which level to place or replace a loaded memory block in cache. Their method improved the performance of SPEC95 CFP benchmarks by an average 7% on an Itanium processor. Our recent work sub-divides the whole-program distance pattern in the space of its data. The spatial analysis identifies locality relations among program data. Programs often have a large number of homogeneous data objects such as molecules in a simulated space or nodes in a search tree. Each object has a set of attributes. In Fortran 77 programs, attributes of an object are stored separately in arrays. In C programs, the attributes are stored together in a structure. Neither scheme is sensitive to the access pattern of a program. A better way is to group attributes based on the locality of their access. For arrays, the transformation is array re-grouping. For structures, it is structure splitting. We grouped arrays and structure fields that have a similar reuse signature. The new data layout consistently outperformed array and structure layouts given by the programmer, compiler analysis, frequency profiling, and statistical clustering on machines from all major vendors [Zhong et al. 2004].

Memory adaptation A recent trend in memory system design is adaptive caching based on the usage pattern of a running program. Balasubramonian et al. [2000] described a system that can dynamically change the size, associativity, and the number of levels of on-chip cache to improve speed and save energy. To enable phase-based adaptation, our recent work divides the distance pattern in time to identify abrupt reuse-distance changes as phase boundaries. The new technique is shown more effective at identifying long, recurring phases than previous methods based on program code, execution intervals, and manual analysis [Shen et al. 2004]. For FPGA-based systems, So et al. [2002] showed that a best design can be found by examining only 0.3% of design space with the help of program information, including the balance between computation and memory transfer as defined by Callahan et al [1988b]. So et al. used a compiler to adjust program balance in loop nests and to enable software and hardware co-design. While our analysis cannot change a program to have a particular balance (as techniques such as unroll-and-jam do [Carr and Kennedy 1994]), it can be used to measure memory balance and support hardware adaptation for programs that are not amenable to loop-nest analysis.

File caching For software managed cache, Jiang and Zhang [2002] developed an efficient buffer cache replacement policy, *LIRS*, based on the assumption that the reuse distance of

cache blocks is stable over a certain time period. Zhou et al. [2001] divided the second-level server cache into multiple buffers dedicated to blocks of different reuse intervals. The common approach is partition cache space into multiple buffers, each holding data of different reuse distances. Both studies showed that reuse distance based management outperforms single LRU cache and other frequency-based schemes. Our work will help in two ways. The first is faster analysis, which reduces the management cost for large buffers (such as server cache), handles larger traces, and provides faster run-time feedbacks. The second is predication, which gives not only the changing pattern but also a quantitative measure of the regularity within and between different types of workloads.

5. RELATED WORK

The preceding sections have discussed related work in the measurement and the use of reuse distance. This section compares our work with program analysis techniques. We focus on data reuse analysis. Data reuse analysis can be performed mainly in three ways: by a compiler, by profiling or by run-time sampling.

Compiler analysis Compiler analysis has achieved great success in understanding and improving locality in basic blocks and loop nests. A basic tool is dependence analysis. Dependence summarizes not only the location but also the distance of data reuse. We refer the reader to a recent, comprehensive book on this subject [Allen and Kennedy 2001]. Various types of array sections can measure data locality in loops and procedures. Such analysis includes 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]. Havlak and Kennedy [1991] studied the effect of array section analysis on a wide range of programs. Cascaval and Padua [2003] extended dependence analysis to estimate the distance of data reuses. Other locality analysis includes the matrix model [Wolf and Lam 1991], the memory ordering [McKinley et al. 1996], and a number of later studies using high-dimensional discrete optimization [Cierniak and Li 1995; Kodukula et al. 1997], transitive dependence analysis [Song and Li 1999; Wonnacott 2002; Yi et al. 2000], and integer sets and equations [Chatterjee et al. 2001; Ghosh et al. 1999].

Because dependence analysis is static, it cannot accurately analyze input-dependent control flow and dynamic data indirection, for which we need profiling or run-time analysis. However, dynamic analysis cannot replace compiler analysis, especially for understanding high-dimensional computation.

Balasundaram et al. [1991] used training sets in performance prediction on a parallel machine. They ran test programs to measure the cost of primitive operations and used the result to calibrate the performance predictor. While their method trains for different machines, our scheme trains for different program inputs. Compiler analysis can differentiate fine-grain locality patterns. Recent source-level tools use a combination of program instrumentation and profiling analysis. McKinley and Temam [1999] carefully measured various types of reference locality within and between loop nests. Mellor-Crummey et al. [2001] measured fine-grained reuse and program balance through their HPCView tool. Reuse distance has recently been used in source- and binary-level tools [Zhong et al. 2004; Marin and Mellor-Crummey 2004]. Since our current analyzer can analyze all data in complete executions, it can definitely handle program fragments or data subsets.

Data Access Frequency Access frequency has been used since the early days of comput-

ing. Early analysis included sample- and counter-based profiling [Knuth 1971] and static probability estimation [Cocke and Kennedy 1974]. Most profiling work considered only a single data input. Thabit [1981] measured how often two data elements were used together. Chilimbi later used grammar compression to find *hot data streams*, which are sequences of repeated data accesses up to 100 elements long [2001a; 2001b].

While previous studies find repeating sequences by measuring frequency and individual similarity, we find recurrence patterns by measuring the distance between data reuses. Phalke and Gopinath [1995] used a Markov model to predict the time distance between data reuses inside the same trace. Distance-based analysis does not construct frequent sub-sequences as other techniques do. On the other hand, it discovers the overall pattern without relying on identical sequences or fixed-size trace windows. Repetition and recurrence are orthogonal and complementary aspects of program behavior. Recurrence helps to explain the presence or absence of repetition. For example in architectural simulation, one study found that *Spec95/Gcc* was so irregular that they needed to sample 33% of program trace [Lafage and Seznec 2000], while another study showed that *Spec2K/Gcc* (compiling 166.i) consisted of two identical phases with mainly four repeating patterns [Sherwood et al. 2002]. Being a completely different approach than cycle-accurate CPU simulation, distance-based sampling shown in Figure 9 confirms both of their observations and suggests that the different recurrence pattern is the reason for this seemingly contradiction. Lafage and Seznec [2000] mentioned that although reuse distance might help their analysis, it was too time consuming to measure. Using the efficient analysis described in this paper, our later work has successfully identified repeating patterns in an execution [Shen et al. 2004].

Correlation among data inputs Wall [1991] presented an early study of execution frequency across multiple runs. Chilimbi [2001b] examined the consistency of hot streams. Since data may be different from one input to another, Chilimbi used the instruction PC instead of the identity of data and found that hot streams include similar sets of instructions if not the same sequence. The maximal stream length he showed was 100. Hsu et al. [2002] compared frequency and path profiles in different runs. Eeckhout et al. [2002] studied correlation in 79 inputs of 9 programs using principal components analysis followed by hierarchical clustering. They considered data properties including access frequency of global variables and the cache miss rate. All these techniques measure rather than predict correlation.

The past profiling analysis is limited to using a single input or discovering invariance among a few inputs. Most used program data or code. The focus of this work is to analyze the program behavior in terms of its reuse distances and to predict the *changing behavior* in other program inputs, including those that are too large to run, let alone to simulate.

Run-time data analysis Saltz and his colleagues pioneered dynamic parallelization with an approach known as inspector-executor, where the inspector examines and partitions data (and computation) at run time [Das et al. 1994]. Similar strategies were used to improve dynamic locality [Ding and Kennedy 1999; Han and Tseng 2000; Mellor-Crummey et al. 2001; Strout et al. 2003]. Knobe and Sarkar [1998] included run-time data analysis in array static-single assignment (SSA) form. To reduce the overhead of run-time analysis, Arnold and Ryder [2001] described a general framework for dynamic sampling, which Chilimbi and Hirzel [2002] extended to discover hot data streams to aid data prefetching. Their run-time sampling was based on program code, while our run-time sampling is based on data

(selected using reuse distance). The two schemes are orthogonal and complementary. Ding and Kennedy [1999] used compiler and language support to mark and monitor important arrays. Ding and Zhong [2002] extended it to selectively monitor structure and pointer data. Run-time analysis can identify patterns that are unique to a program input, while training-based prediction cannot. On the other hand, profiling analysis like ours is more thorough because it analyzes all accesses to all data.

6. CONCLUSIONS

The paper has presented a general method for predicting program locality. It makes three contributions. First, it builds on the 30-year long series of work on stack distance measurement. By using approximate analysis with arbitrarily high precision, it for the first time reduces the space cost from linear to logarithmic. The new analyzer achieves a consistently high speed for practically any large data and long distance. Second, it extends profiling to provide predication for data inputs other than profiled ones. It defines common locality patterns including the constant, linear, and a few sub-linear patterns. Finally, it enables correlation among different executions with distance-based histogram and sampling, which overcomes the limitation of traditional code or data based techniques. When tested on an extensive set of benchmarks, the new method achieves 94% accuracy and 99% coverage, suggesting that pattern prediction is practical for use by locality optimizations in compilers, architecture, and operating systems.

ACKNOWLEDGMENT

Mark Hill pointed us to several related work in stack distance analysis. The original top-down splay-tree implementation was provided by Daniel Sleator. Grant Farmer extended it to maintain sub-tree weights. We also thank our colleagues in and outside Rochester for their valuable inputs. Our experiments were mainly based on local machines purchased by several NSF Infrastructure grants and equipment grants from Compaq/HP and Intel. John Mellor-Crummey and Rob Fowler at Rice University provided access to their Alpha clusters at the critical time before the PLDI submission deadline. The authors were supported by a DoE Young Investigator grant as well as funding from DARPA and NSF. Finally, we wish to thank the anonymous reviewers of the LCR'02 workshop and PLDI'03 conference for their comments and suggestions.

REFERENCES

- 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 first ACM SIGPLAN Workshop on Memory System Performance*. Berlin, Germany.
- ARNOLD, M. AND RYDER, B. G. 2001. A framework for reducing the cost of instrumented code. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. Snowbird, Utah.
- BALASUBRAMONIAN, R., ALBONESI, D., BUYUKTOS, A., AND DWARKADAS, S. 2000. Dynamic memory hierarchy performance and energy optimization. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*.
- BALASUNDARAM, V., FOX, G., KENNEDY, K., AND KREMER, U. 1991. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Williamsburg, VA.
- BENNETT, B. T. AND KRUSKAL, V. J. 1975. LRU stack processing. *IBM Journal of Research and Development*, 353–357.

- BEYLS, K. AND D'HOLLANDER, E. 2001. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*.
- BEYLS, K. AND D'HOLLANDER, E. 2002. Reuse distance-based cache hint selection. In *Proceedings of the 8th International Euro-Par Conference*. Paderborn, Germany.
- BURKE, M. AND CYTRON, R. 1986. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*. Palo Alto, CA.
- CALLAHAN, D., COCKE, J., AND KENNEDY, K. 1988a. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing* 5, 5 (Oct.), 517–550.
- CALLAHAN, D., COCKE, J., AND KENNEDY, K. 1988b. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing* 5, 4 (Aug.), 334–358.
- CARR, S. AND KENNEDY, K. 1994. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems* 16, 6, 1768–1810.
- CASCAVAL, C. AND PADUA, D. A. 2003. Estimating cache misses and locality using stack distances. In *Proceedings of International Conference on Supercomputing*. San Francisco, CA.
- CHATTERJEE, S., PARKER, E., HANLON, P. J., AND LEBECK, A. R. 2001. Exact analysis of the cache behavior of nested loops. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. Snowbird, UT.
- CHILIMBI, T. M. 2001a. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. Snowbird, Utah.
- CHILIMBI, T. M. 2001b. On the stability of temporal data reference profiles. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*. Barcelona, Spain.
- CHILIMBI, T. M. AND HIRZEL, M. 2002. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. Berlin, Germany.
- CIERNIAK, M. AND LI, W. 1995. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*. La Jolla, California.
- 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. *Journal of Parallel and Distributed Computing* 22, 3 (Sept.), 462–479.
- DING, C. 2000. Improving effective bandwidth through compiler enhancement of global and dynamic cache reuse. Ph.D. thesis, Dept. of Computer Science, Rice University.
- DING, C. AND KENNEDY, K. 1999. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*. Atlanta, GA.
- DING, C. AND ZHONG, Y. 2002. Compiler-directed run-time monitoring of program data access. In *Proceedings of the first ACM SIGPLAN Workshop on Memory System Performance*. Berlin, Germany.
- 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*. Charlottesville, Virginia.
- FANG, C., CARR, S., ONDER, S., AND WANG, Z. 2004. Reuse-distance-based miss-rate prediction on a per instruction basis. In *Proceedings of the first ACM SIGPLAN Workshop on Memory System Performance*. Washington DC.
- GHOSH, S., MARTONOSI, M., AND MALIK, S. 1999. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems* 21, 4.
- HAN, H. AND TSENG, C. W. 2000. Locality optimizations for adaptive irregular scientific codes. Tech. rep., Department of Computer Science, University of Maryland, College Park.
- HAVLAK, P. AND KENNEDY, K. 1991. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems* 2, 3 (July), 350–360.
- HILL, M. D. 1987. Aspects of cache memory and instruction buffer performance. Ph.D. thesis, University of California, Berkeley.

- 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 Sixth Workshop on Interaction Between Compilers and Computer Architectures (INTERACT)*.
- HUANG, S. A. AND SHEN, J. P. 1996. The intrinsic bandwidth requirements of ordinary programs. In *Proceedings of the 7th International Conferences on Architectural Support for Programming Languages and Operating Systems*. Cambridge, MA.
- JIANG, S. AND ZHANG, X. 2002. LIRS: an efficient low inter-reference recency set replacement to improve buffer cache performance. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. Marina Del Rey, California.
- KIM, Y. H., HILL, M. D., AND WOOD, D. A. 1991. Implementing stack simulation for highly-associative memories. In *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. 212–213.
- KNOBE, K. AND SARKAR, V. 1998. Array SSA form and its use in parallelization. In *Proceedings of Symposium on Principles of Programming Languages*. San Diego, CA.
- KNUTH, D. 1971. An empirical study of FORTRAN programs. *Software—Practice and Experience* 1, 105–133.
- KODUKULA, I., AHMED, N., AND PINGALI, K. 1997. Data-centric multi-level blocking. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*. Las Vegas, NV.
- LAFAGE, T. AND SEZNEC, A. 2000. Choosing representative slices of program execution for microarchitecture simulations: a preliminary application to the data stream. In *Workload Characterization of Emerging Applications*, Kluwer Academic Publishers.
- LI, Z., GU, J., AND LEE, G. 1996. An evaluation of the potential benefits of register allocation for array references. In *Workshop on Interaction between Compilers and Computer Architectures in conjunction with the HPCA-2*. San Jose, California.
- LI, Z., YEW, P., AND ZHU, C. 1990. An efficient data dependence analysis for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems* 1, 1 (Jan.), 26–34.
- MARIN, G. AND MELLOR-CRUMMEY, J. 2004. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of Joint International Conference on Measurement and Modeling of Computer Systems*. New York City, NY.
- MATTSON, R. L., GECSEI, J., SLUTZ, D., AND TRAIGER, I. L. 1970. Evaluation techniques for storage hierarchies. *IBM System Journal* 9, 2, 78–117.
- MCKINLEY, K. S., CARR, S., AND TSENG, C.-W. 1996. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems* 18, 4 (July), 424–453.
- MCKINLEY, K. S. AND TEMAM, O. 1999. Quantifying loop nest locality using SPEC'95 and the perfect benchmarks. *ACM Transactions on Computer Systems* 17, 4, 288–336.
- MELLOR-CRUMMEY, J., FOWLER, R., AND WHALLEY, D. B. 2001. Tools for application-oriented performance tuning. In *Proceedings of the 15th ACM International Conference on Supercomputing*. Sorrento, Italy.
- MELLOR-CRUMMEY, J., WHALLEY, D., AND KENNEDY, K. 2001. Improving memory hierarchy performance for irregular applications. *International Journal of Parallel Programming* 29, 3 (June).
- OLKEN, F. 1981. Efficient methods for calculating the success function of fixed space replacement policies. Tech. Rep. LBL-12370, Lawrence Berkeley Laboratory.
- PHALKE, V. AND GOPINATH, B. 1995. An inter-reference gap model for temporal locality in program behavior. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. Ottawa, Ontario, Canada.
- RAWLINGS, J. O. 1988. *Applied Regression Analysis: A Research Tool*. Wadsworth and Brooks.
- SHEN, X., ZHONG, Y., AND DING, C. 2004. Locality phase prediction. In *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. Boston, MA. (To appear).
- SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*. San Jose, CA.
- SLEATOR, D. D. AND TARJAN, R. E. 1985. Self adjusting binary search trees. *Journal of the ACM* 32, 3.
- 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 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Berlin, Germany.

- SONG, Y. AND LI, Z. 1999. New tiling techniques to improve cache temporal locality. In *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*. Atlanta, Georgia.
- SRIVASTAVA, A. AND EUSTACE, A. 1994. ATOM: A system for building customized program analysis tools. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. Orlando, Florida.
- STROUT, M. M., CARTER, L., AND FERRANTE, J. 2003. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. San Diego, CA.
- SUGUMAR, R. A. AND ABRAHAM, S. G. 1993. Multi-configuration simulation algorithms for the evaluation of computer architecture designs. Tech. rep., University of Michigan.
- THABIT, K. O. 1981. Cache management by the compiler. Ph.D. thesis, Dept. of Computer Science, Rice University.
- TRIOLET, R., IRIGOIN, F., AND FEAUTRIER, P. 1986. Direct parallelization of CALL statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*. Palo Alto, CA.
- WALL, D. W. 1991. Predicting program behavior using real or estimated profiles. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. Toronto, Canada.
- WOLF, M. E. AND LAM, M. 1991. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*. Toronto, Canada.
- WONNACOTT, D. 2002. Achieving scalable locality with time skewing. *International Journal of Parallel Programming* 30, 3 (June).
- YI, Q., ADVE, V., AND KENNEDY, K. 2000. Transforming loops to recursion for multi-level memory hierarchies. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. Vancouver, Canada.
- ZHONG, Y., DING, C., AND KENNEDY, K. 2002. Reuse distance analysis for scientific programs. In *Proceedings of Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*. Washington DC.
- ZHONG, Y., DROPSHO, S. G., AND DING, C. 2003. Miss rate prediction across all program inputs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*. New Orleans, Louisiana.
- ZHONG, Y., ORLOVICH, M., SHEN, X., AND DING, C. 2004. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- ZHOU, Y., CHEN, P. M., AND LI, K. 2001. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of USENIX Technical Conference*.