Predicting Whole-Program Locality through Reuse Distance Analysis

Chen Ding Yutao Zhong
Computer Science Department
University of Rochester
Rochester, New York
{cding,ytzhong}@cs.rochester.edu

ABSTRACT

Profiling can accurately analyze program behavior for select data inputs. We show that profiling can also predict program locality for inputs other than profiled ones. Here locality is defined by the distance of data reuse. Studying whole-program data reuse may reveal global patterns not apparent in short-distance reuses or local control flow. However, the analysis must meet two requirements to be useful. The first is efficiency. It needs to analyze all accesses to all data elements in full-size benchmarks and to measure distance of any length and in any required precision. The second is predication. Based on a few training runs, it needs to classify patterns as regular and irregular and, for regular ones, it should predict their (changing) behavior for other inputs. In this paper, we show that these goals are attainable through three techniques: approximate analysis of reuse distance (originally called LRU stack distance), pattern recognition, and distance-based sampling. When tested on 15 integer and floating-point programs from SPEC and other benchmark suites, our techniques predict with on average 94% accuracy for data inputs up to hundreds times larger than the training inputs. Based on these results, the paper discusses possible uses of this analysis.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*optimization, compilers*

General Terms

Algorithms, Measurement, Languages

Keywords

Program locality, reuse distance, stack distance, data locality, training, sampling, profiling, pattern recognition and prediction

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 depends on program locality or the pattern of data reuse. Many applications may have a consistent recurrence pattern at the whole-program level, for example, reusing a large amount of data across the time steps of an astronomical simulation, the optimization passes of a compiler, or the moves of a game-playing program. To exploit program locality, new cache designs are adding more cache levels and dynamic configuration control. As the memory hierarchy becomes deeper and more adaptive, its performance will increasingly depend on our ability to predict whole-program locality.

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 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.

In this paper, we predict locality in programs that have consistent reuse patterns. Since different runs of the same program may use different data and go through different control flow, our analysis is not based on program code nor its data but on a concept we call *reuse distance*. In a sequential execution, reuse distance is the number of *distinct* data elements accessed between two consecutive references to the same element. It measures the volume of the intervening data not the time between two accesses. While time distance is unbounded in a long-running program, reuse distance is always bounded by the size of physical data. In 1970, Mattson et al. studied stack algorithms in cache management and defined the concept of stack distance [30]. Reuse distance is the same as *LRU stack distance* or stack distance using LRU (Least Recently Used) replacement policy. In this paper, we use a different (and shorter) name to reflect our purpose in program analysis, not cache management. We later show that reuse distance is measured much faster using a tree instead of a stack.

Reuse distance is a powerful basis for pattern analysis for three reasons. First, reuse distance is at most a linear function of program data size. The search space is therefore much smaller for pattern recognition and prediction. Second, reuse distances reveal invariance in program behavior. Most control flow perturbs only short access sequences but not the cumulative distance over millions of data. Long reuse distances suggest important data and signal major phases of a program. For example, a distance equal to 50% of program data is likely to span a significant program phase. Finally and most importantly for this work, reuse distance allows direct comparison of data behavior in different program runs. Distance-based correlation does not require two executions to have the same data or execute the same function. Therefore, it can identify consistent patterns in the presence of dynamic data allocation and input-dependent control flow.

This paper presents three new techniques. The first is approximate reuse distance analysis, which bounds the relative error to arbitrarily close to zero. It takes $O(\log \log M)$ time per access and $O(\log M)$ total space, where M is the size of program data. The second is pattern recognition, which profiles a few training runs and extracts regular patterns as a function of program data size. The last one, distance-based sampling, predicts reuse pattern for an unknown data input at run time by sampling at the beginning of the execution when needed. Together these three techniques provide a general method that predicts locality patterns in whole or parts of a program or its data.

We should note two important limitations of this work. First, our goal 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. Program analysis, however, does not directly improve a program or a machine. In Section 4, we discuss current and future uses of this analysis in compiler, file system, and memory system design.

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. REUSE PATTERN ANALYSIS

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

2.1 Approximate reuse distance analysis

In a distance analysis, we view program execution as a sequence of accesses to data. Measuring reuse distance between two data accesses means counting the number of distinct data between them. In the worst case, the measurement needs to examine all preceding accesses for each access in the trace. So a naive algorithm would need $O(N^2)$ time and $O(N)$ space for a trace of length N. This cost is impractical for real programs, which have up to hundreds of billions of memory accesses.

The time and space costs can be improved, as shown by the 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 can store (and count) just the last access of each data. Part (c) shows that we can organize the last-access time of all data in a search tree. The counting can be done in a single tree search if we maintain the weight or the number of nodes in all sub-trees. For a balanced tree, reuse-distance measurement takes $O(\log M)$ time per access and $O(M)$ total space, where M is the size of program data. For a program with a large amount of data, however, the space requirement becomes a limiting factor. Each data element needs a tree node, which stores the last-access time, pointers to its children, and the weight of its sub-tree. Since the tree data at least quadruple program data, they easily overflow physical memory and even the 32-bit address space for a program with more than 100 million data.

To reduce the space cost, we introduce approximate analysis for long reuse distances. If the length of a distance is in the order of millions, the accuracy of the last couple of digits rarely matters. The main feature of our analysis is approximating a block of data in a tree node, as shown in Part (d) of Figure 1. The space requirement is reduced by a factor equal to the average block size. Using a large block size, the approximate analysis can make its tree data small enough to fit in not only the physical memory but also the processor cache.

In our discussion, we do not consider the cost of finding the last access time. This requires a hashtable with one entry for each data. The space cost is $O(M)$. However, Bennett and Kruskal showed that hashing can be done in a pre-pass, using blocked algorithms to reduce the memory requirement to arbitrarily low [6]. The time complexity of hashing is constant per access, a well-studied problem compared to distance measurement. In the rest of this paper, we will focus our attention only on reuse distance measurement.

We present two approximation algorithms, with different guarantee on the accuracy of the measured distance, $d_{measured}$, compared to the actual distance, d_{actual}, as shown below.

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 $d_{actual} - d_{measured} \leq B$

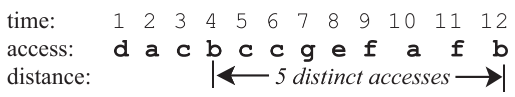
Both methods also guarantee $d_{measured} \leq d_{actual}$. The rest of this section describes them in more detail.

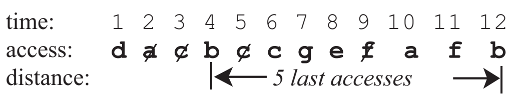
2.1.1 *Analysis with a bounded relative error*

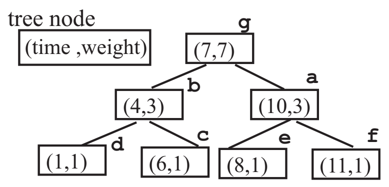
The analysis guarantees a bounded error rate that can be arbitrarily close to zero. Figure 2 shows the main algorithm. Given the current and last access time, the main routine uses *TreeSearchDelete* to search the block tree and calculate reuse distance using sub-tree weights. Once the node containing the last access time is found, the subroutine *TreeSearchDelete* updates the capacity of the node. The new capacity is $distance * \frac{e}{1-e}$. To simplify the notation, we use e' to represent $\frac{e}{1-e}$. The value of $distance$ is the number of distinct data accessed after this node. The subroutine uses $distance$ as the approximate distance. The approximation is never greater than the actual distance. The maximal relative error e happens when the actual distance is $distance * (1 + e')$. The formula of e' assumes $1 > e > 0$. The algorithm is not valid if $e = 0$, which means no approximation. It is trivial to approximate if $e = 1$: we simply report all reuse distance as 0.

After capacity update, the subroutine *TreeSearchDelete* deletes the last access and inserts the current 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 tree being used, which can be an AVL, red-black, splay, or B-tree.

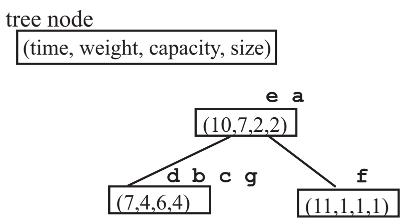
The most important part of the algorithm is dynamic tree compression by subroutine *TreeCompression*. It scans tree nodes in reverse time order, updates their capacity as in *TreeSearchDelete*, and merges adjacent tree nodes when possible. The size of the merged node must be no more than the smaller capacity of the two nodes; otherwise, the accuracy cannot be guaranteed. 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, as proved by the following proposition.


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 between two b's is 5.


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) Store and count only the last access of each data.



(c) Organize last access times as a tree. Each node represents a distinct element. Attribute *time* is its last access time, *weight* is the number of nodes in the subtree. The tree search for the first *b* finds the reuse distance, which is the number of nodes whose last access time is greater than 4.



(d) Use an approximation tree with 33% guranteed accuracy. Attributes *capacity* and *size* are the maximal and current number of distinct elements represented by a node, *time* is their last access time, and *weight* is the total *size* of subtree nodes. The approximate distance of two *b*'s is 3 or 60% of the actual distance.

Figure 1: Reuse distance example

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. Let n_0, n_1, ..., and n_t be the sequence of tree nodes in 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 $sum_{i-1} = \sum_{j=0,\ldots,i-1} size_j$. The new capacity of $n_{2i}.capacity$ is $\lfloor sum_{i-1} * e' \rfloor$. The combined size, $size_i$, must 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 call must reduce 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 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 precise measurement of distance shorter than C and approximate measurement of longer distances with a bounded error B. It keeps 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 with tree nodes having capacity 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 precise and approximate traces [44].

In this work, we generalize the previous algorithm. In addition to using B-Tree, 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 precise trace is bounded by a constant. Second, the minimal occupancy of the approximate tree is guaranteed. Invoking a transfer when the precise trace exceeds a pre-set size can satisfy the first requirement. For the second requirement, we dynamically merge a tree node with any of its neighbors when the combined size is no more than B. 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 [37] version of the algorithm in this work. 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 Comparison

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 1 show methods using a list, a vector, and a tree. In 1970, Mattson et al. published the first measurement algorithm [30]. They used a list-based stack. Bennett and Kruskal showed that a stack was too slow to measure long reuse distances. They used a vector and built an m-ary tree on it [6]. They also showed how to use blocked hashing in a pre-pass. In 1981, Olken implemented the first tree-based method using an AVL tree [34]. Olken also showed how to compress the trace vector in Bennett and Kruskal's method and improve the time and space efficiency to those of tree-based algorithms. In 1994, Sugumar and Abraham showed that a splay tree [37] has better memory performance [41]. Their analyzer, *Cheetah*, is widely

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

algorithm $ReuseDistance(last, current)$
    // inputs are the last and current access time
    1. $TreeSearchDelete(last, distance)$
    2. $new = TreeNode(current, 1, 1, 1, \bot, \bot, \bot)$
       $TreeInsert(new)$
    3. if ($tree\_size \geq 4 * \log_{1+e} root.weight + 4$)
           $TreeCompression(new)$
           Assert(compression more than halves the tree)
       end if
    4. return $distance$
end algorithm

subroutine $TraceSearchDelete(time, distance)$
    // time is last access time of the current data.
    // distance will be returned.
    $node$ = $root$; $distance$ = 0
    while true
        $node.weight = node.weight - 1$
        if ($time < node.time$ and
            $node.prev$ exists and $time \leq node.prev.time$)
            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
    $node.capacity = max(distance * \frac{e}{1-e}, 1)$
    $node.size = node.size - 1$
    return $distance$
end subroutine $TreeSearchDelete$

subroutine $TreeCompression(n)$
    // n is the last node in the trace
    $distance = 0$
    $n.capacity = 1$
    while ($n.prev$ exist)
        if ($n.prev.size + n.size \leq n.capacity$)
            // merge n.prev into n
            $n.size = n.size + n.prev.size$
            $n.prev = n.prev.prev$
            deallocate $n.prev$
        else
            $distance = distance + n.size$
            $n = n.prev$
            $n.capacity = max(distance * \frac{e}{1-e}, 1)$
        end if
    end while
    Build a balanced tree from the list and return the root
end subroutine $TreeCompression$
```

Figure 2: Approximate analysis with a bounded relative error

Analysis methods	Time	Space
trace as a stack (or list) [30]	$O(NM)$	$O(M)$
trace as a vector [6, 2]	$O(N \log N)$	$O(N)$
trace as a tree [34, 41, 2]	$O(N \log M)$	$O(M)$
list-based aggregation [25]	$O(NS)$	$O(M)$
block tree [44]	$O(N \log \frac{M}{B})$	$O(\frac{M}{B})$
dynamic tree compression	$O(N \log \log M)$	$O(\log M)$

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

Table 1: Asymptotic complexity of measuring full distance

available from the SimpleScalar tool set. Recently, Almasi et al. 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 [2].

Kim et al. gave the first imprecise (but accurate) analysis method in 1991 [25]. 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 reuse distance, where S and C need to be proportional to M. Unlike approximate analysis, this method is accurate in counting the reuse distance within a marked range.

In comparison, approximation methods, shown in the last two rows in Table 1, trade accuracy for efficiency especially space efficiency. They can analyze larger data and longer reuse distances. They are adjustable because the cost is proportional to accuracy. The analysis with bounded relative error has the lowest asymptotic space and time cost, for any error rate that is greater than (and can be arbitrarily close to) zero.

Reuse distance is no longer a favorable metric in low-level cache design because it cannot model the interaction between cache and CPU such as timing. However, at the high level, 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 [22, 30]. Section 4 will discuss the uses of reuse distance analysis in cache optimization.

2.2 Pattern recognition

Pattern recognition detects whether the recurrence pattern is predictable across different data inputs. Example recurrence patterns at the whole-program level include ocean simulation in a series of time steps, compilation of a collection of files, and computer chess-playing in a number of moves. Based on two or more training runs, pattern recognition constructs a parameterized reuse pattern. 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 program data size.

We define the reuse, recurrence or locality pattern as a histogram showing the percentage of memory accesses whose reuse distance falls inside consecutive ranges divided between 0 and the data size (maximal distance). We will use ranges of both logarithmic and linear sizes. Our approach is not specific to a particular accuracy of the histogram. We now describe the three steps of pattern recognition.

2.2.1 Collecting reference histograms

A reference histogram is a transpose of the reuse distance histogram. It sorts all memory accesses based on their reuse distance and shows the average distance of each k percent of memory references. For example, when k is 1, the reference histogram first gives the average distance for 1% shortest reuse distances, then the average for the next 1% shortest reuse distances, and so on.

We use the reference histogram for two purposes. First, we isolate 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, the size of the bin controls the granularity of prediction. A bin size of 1% means that we do not predict finer distribution of distances within 1% of memory references.

A reference histogram is computed from a reuse-distance histogram by traversing the latter and calculating the average distance for each k% 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 reuse-distance histogram using log-linear scale bins. The size of bins is a power of 2 up to 1024 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. The cost and accuracy of the approximation scheme can be adjusted by simply changing the size of bins in both types of histograms.

2.2.2 Recognizing patterns

Given two reference histograms from two different data inputs (we call them training inputs), we construct a formula for each bin. Let d_{1i} be the distance of the ith bin in the first histogram, d_{2i} be the distance of the ith bin in the second histogram, s_1 be the data size of the first training input, and s_2 the data size of the second input. We use linear fitting to find the closest linear function that maps data size to reuse distance. Specifically, we find the two coefficients, c_i and e_i, that satisfy the following two equations.

$$d_{1i} = c_i + e_i * f_i(s_1)$$
$$d_{2i} = c_i + e_i * f_i(s_2)$$

Assuming the function f_i is known, the two coefficients uniquely determine the distance for any other data size. The formula therefore defines the reuse-distance pattern for memory accesses in the bin. The overall pattern is the aggregation of all bins. The pattern is more accurate if more training profiles are collected and used in linear fitting. 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, not a general polynomial function. In this work, we consider the following choices of f_i. The first is the function $p_{const}(x) = 0$. We call it a constant pattern because reuse distance does not change with data size. The second is $p_{linear}(x) = x$. We call it a linear pattern. Constant and linear are the lower and upper bound of the reuse distance changes. Between them are sub-linear patterns, for which we consider three: $p_{1/2}(x) = x^{1/2}$, $p_{1/3}(x) = x^{1/3}$, and $p_{2/3}(x) = x^{2/3}$. The first 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_{1i}/d_{2i}, and pick f_i to be the pattern function, p_t, such that $p_t(s_1)/p_t(s_2)$ is closest to d_{1i}/d_{2i}. Here t is one of the patterns described in the preceding paragraph. 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 Limitations

Although the analysis can handle any sequential program, the generality comes with several limitations. The profiling inputs should be large enough to factor out the effect of non-recurring accesses. The smallest input we use in our experiment has four million memory accesses. For linear and sub-linear patterns, our analysis needs inputs of different data sizes. The difference should be large enough to separate pattern functions from each other. 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. Finally, 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. If a distance is greater than a *threshold*, the accessed memory location is taken as a *data sample*. The sampler collects more data samples in the same way except that it requires data samples to have between each other a spatial distance of a fraction of the first above-threshold distance. The sampler records above-threshold reuse distances to all data samples. We call them *time samples*. Given the sequence of time 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 [19]. 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.

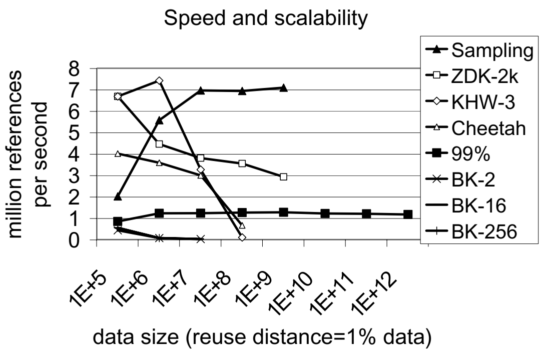
Finding the first few peaks of the first few data samplings is an unusual heuristic because it is not based on keeping track of a particular program instruction or a particular data item. The peaks found by sampling in different program executions do not have to be caused by the same memory access to 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 3 compares the speed and accuracy for eight analyzers, which we have described in Section 2.1. *BK-2*, *BK-16*, and *BK-256* are Bennett and Kruskal's k-ary tree analyzers with k equal to 2, 16, and 256 [6]. *KHW* is list-based aggregation with three markers at distance 32, 16K, and the size of analyzed data [25]. We reimplemented it since the original no longer exists. *Cheetah* uses a splay-tree, written by Sugumar [41]. *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 [44]. *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 of $M/100$. To measure only the cost of reuse-distance analysis, the hashing step was bypassed by pre-computing the last access time (except for *KHW*, where hashing was inherent). We did not separately measure the cost of hashing since we did not implement blocked hashing [6]. The timing was collected on a 1.7 GHz Pentium 4 PC with 800 MB of main memory. The programs were compiled with *gcc* with optimization flag *-O3*.

Compared to 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



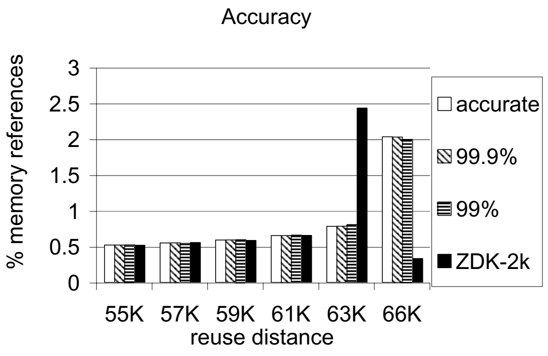


Figure 3: Comparison of analyzers

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, a satisfying moment considering that prediction is the spirit of this work. If we consider an analogy to physical distance, the precise methods measure the distance in miles of data, the approximation method measures light years.

The lower graph of Figure 3 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 third bar, 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 analysis accuracy is adjustable, so is the cost.

3.2 Pattern prediction

Figure 4 shows the result of pattern prediction for *Lucas* from Spec2K, which is representative in our test suite. The graph has four groups of bars. The first two are for two training inputs. Their data access traces are feed into our analyzer for pattern recognition. The third input is the target of prediction. The analyzer samples 0.4% of its execution, finds the data size, and predicts the reuse distance histogram shown by the third bar. The prediction matches closely with the measured histogram shown by the fourth bar. The two histograms overlap by 95%. The accuracy is remarkable considering that the target execution has 500 times more data and 300 times more data accesses than the training runs. 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 2 shows 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 DEC compiler with the default compiler optimization (*-O3*). Different compiler optimization levels may change the reuse pattern but not the accuracy of our prediction. We use Atom [39] 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 Figure 2 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 second to the last column of the table. Let x_i and y_i be the size of ith 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* is 95%, shown in Figure 4.

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 middle input—and for *Li*—because only the constant pattern is considered by prediction. Among all prediction results, the highest accuracy is 98.7% for the train input of *Gcc*, the lowest is 81.8% for the train input of *Lucas*. The average accuracy is 93.7%.

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 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 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 98.8%.

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 sampled version of a program runs much slower than the program without sampling. Our fastest analyzer causes a slowdown by factors ranging from 20 to 100. For a slowdown of 100, we need coverage of at least 99% to finish prediction before the end of the execution! Fortunately, the low coverage happens only in *Compress* and the train input of *Swim*. Without them, the average coverage is 99.88%, indicating a time coverage over 88%. 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 at all.

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 larger test input. Finally, *Apsi* uses different-shape inputs of high-dimensional data, which our current predictor cannot accurately predict. We change the shape of its largest input.

3.2.1 Comparisons

Most profiling methods use the result from training runs as the prediction for other runs. An early study by Wood measured the accuracy of this scheme in finding the most frequently accessed variables and executed control structures in a set of dynamic programs [43]. 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 57% on average. The largest is 84% in *Twolf*, and the smallest 28% in *Apsi*. The accuracy can be higher if the lin-

Benchmarks	Description	Patterns	Inputs	Data elements	Avg. reuses per element	$\frac{\text{Avg. dist.}}{\text{data size}}$	Accuracy(%)	Coverage(%)
Lucas	Lucas-Lehmer test	const	ref	20.8M	621	2.49E-1	**95.1**	**99.6**
(Spec2K)	for primality	linear	train	41.5K	971	2.66E-1	**81.8**	**100**
			test	6.47K	619	2.17E-1		
Applu	solution of five	const	ref(60^3)	22.8M	185	1.37E-1	**83.6**	**99.6**
(Spec2K)	coupled nonlinear	3rd roots	train(24^3)	1.29M	178	1.37E-1	**94.7**	**99.4**
	PDE's	linear	test(12^3)	128K	176	1.33E-1		
Swim	finite difference	const	ref(512^2)	3.68M	46.9	2.47E-1	**99.0**	**99.9**
(Spec95)	approximations for	2nd root	400^2	2.26M	46.6	2.47E-1	**99.3**	**88.1**
	shallow water equation	linear	200^2	572K	46.2	2.47E-1		
SP	computational fluid	const	50^3	4.80M	132	1.05E-1	**90.3**	**99.9**
(NAS)	dynamics (CFD)	3rd roots	32^3	1.26M	124	1.01E-1	**95.8**	**99.9**
	simulation	linear	28^3	850K	125	9.78E-2		
Tomcatv	vectorized mesh	const	ref(513^2)	1.83M	208	1.71E-1	**92.4**	**99.5**
(Spec95)	generation	2nd root	400^2	1.12M	104	1.67E-1	**99.2**	**99.3**
		linear	train(257^2)	460K	104	1.67E-1		
Hydro2d	hydrodynamical		ref	1.10M	13.4K	2.23E-1	**98.5**	**100**
(Spec95)	equations computing	const	train	1.10M	1.35K	2.23E-1	**98.4**	**100**
	galactical jets		test	1.10M	139	2.20E-1		
FFT	fast Fourier	const	512^2	1.05M	72.9	6.41E-2	**84.3**	**99.7**
	transformation	2nd root	256^2	265K	69.0	6.76E-2	**94.0**	**99.6**
		linear	128^2	66.8K	61.4	7.60E-2		
Mgrid	multi-grid solver	const	ref(64^3)	956K	35.6K	6.81E-2	**96.4**	**100**
(Spec95)	in 3D potential	3rd roots	test(64^3)	956K	1.42K	6.76E-2	**96.5**	**99.3**
	field	linear	train(32^3)	132K	32.4K	7.15E-2		
Apsi	pollutant distribution	const	128x1x128	25.0M	6.35	1.60E-3	**91.6**	**97.8**
(Spec2K)	for weather prediction	3rd roots	train(128x1x64)	25.0M	146	2.86E-4	**92.5**	**99.1**
		linear	test(128x1x32)	25.0M	73.6	1.65E-4		
Compress	an in-memory version	const	ref	36.1M	628	4.06E-2	**85.9**	**92.2**
(Spec95)	of the common UNIX	linear	train	279K	314	6.31E-2	**92.3**	**86.9**
	compression utility		test	142K	147	9.73E-2		
Twolf	circuit placement and	const	ref(1888-cell)	734K	177K	2.08E-2	**94.2**	**100**
(Spec2K)	global routing, using	linear	train(752-cell)	402K	111K	1.82E-2	**96.6**	**100**
	simulated annealing		370-cell	227K	8.41K	1.87E-2		
Vortex	an object oriented		ref	7.78M	4.60K	4.31E-4	**95.1**	**100**
(Spec95)	database	const	test	2.58M	530	3.25E-4	**97.2**	**100**
(Spec2K)			train	501K	71.3K	4.51E-4		
Gcc	based on the		expr	711K	137	2.75E-3	**98.2**	**100**
(Spec95)	GNU C compiler		cp-decl	705K	190	2.65E-3	**98.6**	**100**
	version 2.5.3	const	explow	321K	68.3	3.69E-3	**96.1**	**100**
			train(amptjp)	467K	221	3.08E-3	**98.7**	**100**
			test(cccp)	456K	233	3.25E-3		
Li		const	ref	87.9K	328K	2.19E-2	**82.7**	**100**
(Spec95)	Xlisp interpreter	linear	train	44.2K	1.86K	3.11E-2	**86.0**	**100**
			test	14.5K	37.0K	2.56E-2		
Go	an internationally		ref	109K	124K	3.78E-3	**96.5**	**100**
(Spec95)	ranked go-playing	const	test	104K	64.6K	3.78E-3	**96.9**	**100**
	program		train	86.1K	2.68K	2.02E-3		
						average	**93.7**	**98.8**

Table 2: Prediction accuracy and coverage for 15 programs

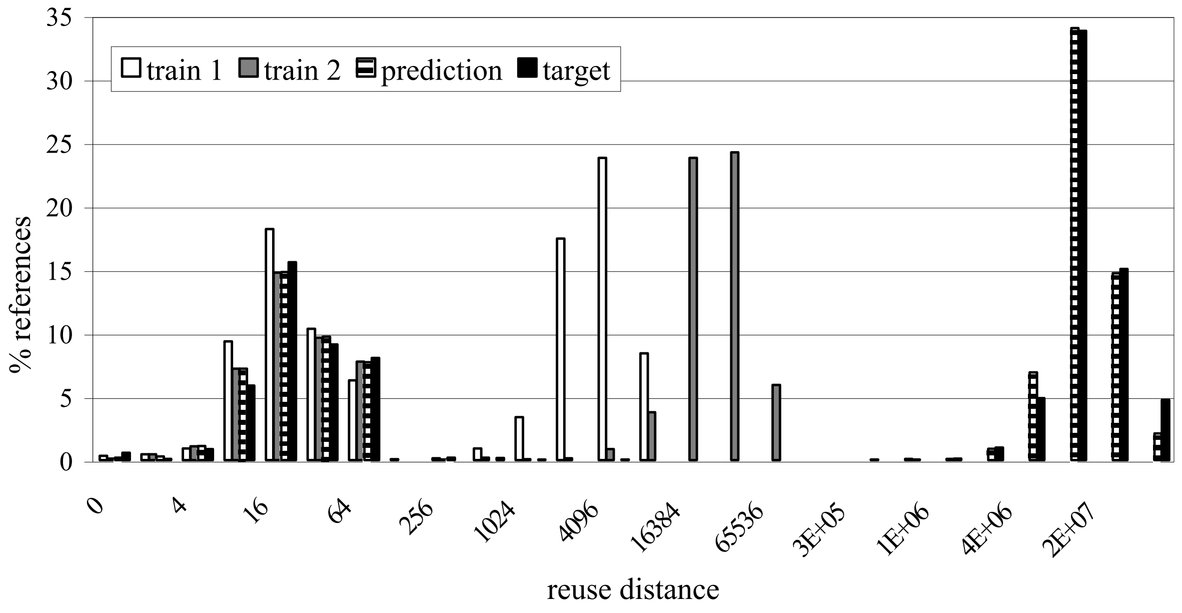


Figure 4: Pattern prediction for Spec2K/Lucas

ear 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 [1]. For example in *Swim*, the average distance is 25% 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 25%. 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 25% 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 4, n is 26 and the probability of a random prediction achieving over 95% accuracy is 0.05^{25} or statistically impossible.

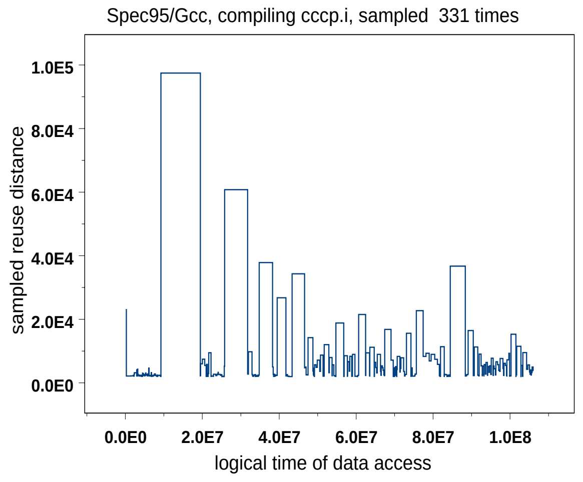
[1]The observation came from two anonymous reviewers of PLDI'03

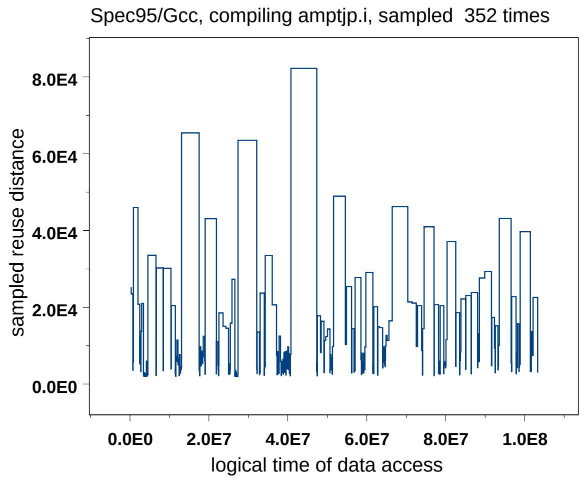
3.2.2 *A case study*

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 `cccp.i` and `amptjp.i` and *Spec2K/Gcc* compiling `166.i`. The three graphs in Figure 5 show the time samples of one data sample. Other data samples produce similar graphs. The upper two graphs, `cccp.i` and `amptjp.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 `cccp.i` and 129 functions in the 7088-line `amptjp.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 2). 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 5. 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.





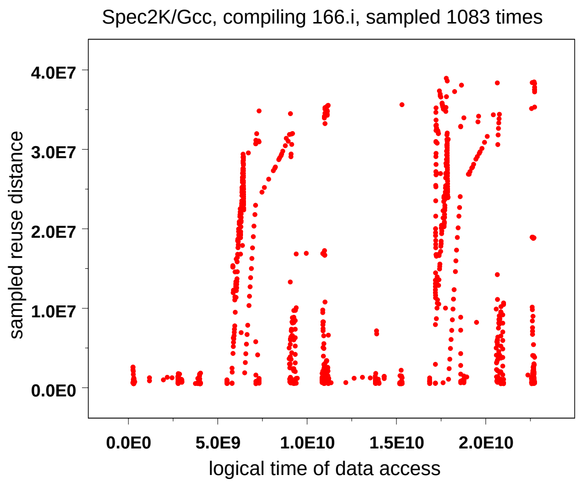


Figure 5: Sampling results of *Gcc* for three inputs

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 PATTERN INFORMATION

A program has different portions of accesses with different reuse distances, as shown graphically by the peaks in the histogram of *Lucas* in Figure 4. The peaks differ in number, position, shape, and size among different programs and among different inputs to the same program. Since our method predicts these peaks, it helps to better model program locality and consequently improves program and machine optimization as well as program-machine co-design.

Compiler design Reuse distance provides much richer information about a program than a cache miss rate does. For this reason, at least four compiler groups have used reuse distance for different purposes: to study the limit of register reuse [29] and cache reuse [17, 44], to evaluate the effect of program transformations [2, 7, 17, 44], and to annotate programs with cache hints to a processor [8]. In the last work, Beyls and D'Hollander used reuse distance profiles to generate hints in SPEC95 FP benchmarks and improved performance by 7% on an Itanium processor [8]. The techniques in this paper will allow compiler writers to analyze larger programs faster and with adjustable accuracy and to predict analysis results on data inputs other than analyzed ones. Another potential use is to find related data in a program based on their usage pattern, for example, arrays or structure fields that can be grouped to improve cache performance.

Reconfigurable memory system A recent trend in memory system design is adaptive caching based on the usage pattern of a running program. Balasubramonian et al. described a system that can dynamically change the size, associativity, and the number of levels of on-chip cache to improve cache speed and save energy [4]. They used an on-line method that tries different choices and searches for an appropriate cache configuration. Since our pattern analysis directly determines the best cache size for capacity misses, it should reduce the search space (and overhead) of run-time adaptation. For FPGA-based systems, So et al. showed that a best design can be found by examining only 0.3% of design space with the help of program information [38], including the balance between computation and memory transfer as defined by Callahan et al [9]. 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 [10]), it can measure memory balance and support hardware adaptation for general programs.

File caching Two recent studies by Zhou et al. [45] and by Jiang and Zhang [24] have used reuse distance in file caching. The common approach is to partition cache space into multiple buffers, each holding data of different reuse distances. Both studies showed that reuse-distance based methods well adapt to the access pattern in server and database traces and therefore significantly outperform single-buffer LRU and frequency-based multi-buffer schemes. Zhou et al. used run-time statistics to estimate the peak distance [45]. Our work will help in two ways. The first is faster analysis, which reduces 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 use of reuse distance. This section compares our work with program analysis techniques. We focus on data reuse analysis.

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 by Allen and Kennedy [1]. Cascaval gave a compiler algorithm that measures reuse distance directly [11]. 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.

Bala et al. used training sets in performance prediction on a parallel machine [5]. 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 carefully measured various types of reference locality within and between loop nests [31]. Mellor-Crummey et al. measured fine-grained reuse and program balance through their HPCView tool [32]. Reuse distance can be included in these or binary-level tools to recognize reuse-distance pattern in smaller code or data units. Since our current analyzer can analyze all data in complete executions, it can definitely handle program fragments or data subsets.

Data profiling Access frequency has been used since the early days of computing. In early 80s, Thabit measured how often two data elements were used together [42]. Chilimbi recently used grammar compression to find longer sequences of repetition called *hot data streams* [12]. To measure both CPU and cache behavior, many studies have tried to identify representative segments in an execution trace. The most recent (and very accurate) is reported by Lafage and Seznec [28] and by Sherwood et al. [36]. The two techniques cut instruction traces into fixed size windows (10 and 100 million instructions) and find representative windows through hierarchical and k-means clustering respectively.

While previous studies find repeating sequences by measuring frequency and individual similarity, we find recurrence patterns by measuring distance and overall accumulation. Reuse distance 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. Lafage and Seznec found that *Spec95/Gcc* was so irregular that they needed to sample 33% of program trace [28], while Sherwood et al. found that *Spec2K/Gcc* (compiling 166.i) consisted of two identical phases with mainly four repeating patterns [36]. Being a completely different approach than cycle-accurate CPU simulation, data sampling shown in Figure 5 confirms both of their observations and suggests that the different recurrence pattern is the reason for this seemingly contradiction. On the other hand, clustering analysis like theirs provides a broader framework than ours does. They can include reuse distance as one of the clustering parameters. Lafage and Seznec mentioned this possibility but chose not to use reuse distance because it was too time consuming to measure.

Phalke and Gopinath used a Markov model to predict the time distance of data reuses inside the same trace [35]. Our focus is to predict behavior changes in other inputs. After a few training inputs, it predicts locality pattern in other inputs, including those that are too large to run, let alone to simulate.

Correlation among data inputs Early analysis of execution frequency included sample- and counter-based profiling by Knuth [27] and static probability analysis by Cocke and Kennedy [15]. Most dynamic profiling work considered only a single data input. Wall presented an early study of execution frequency across multiple runs [43]. Recently, Chilimbi examined the consistency of hot streams [13]. 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. compared frequency and path profiles in different runs [23]. Eeckhout et al. studied correlation in 79 inputs of 9 programs using principal components analysis followed by hierarchical clustering [20]. They considered data properties including access frequency of global variables and the cache miss rate. All these techniques measure rather than predict correlation.

Instead of using program data or code like most previous work, we correlate program data by their reference histogram. The direct correlation of data recurrence allows us to predict the *changing behavior* in other program inputs, a feature that we do not know any previous work has attempted.

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 [16]. Similar strategies were used to improve dynamic locality, including studies by Ding and Kennedy [18], Han and Tseng [21], Mellor-Crummey et al. [33], and Strout et al [40]. Knobe and Sarkar included run-time data analysis in array static-single assignment (SSA) form [26]. To reduce the overhead of run-time analysis, Arnold and Ryder described a general framework for dynamic sampling [3], which Chilimbi and Hirzel extended to discover hot data streams to aid data prefetching [14]. 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 used compiler and language support to mark and monitor important arrays [18]. Ding and Zhong extended it to selectively monitor structure and pointer data [19]. 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.

7. ACKNOWLEDGEMENTS

Mark Hill pointed us to several related work in stack distance analysis. The original 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. The D System group at Rice University provided access to their Alpha clusters at the critical time before the 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, especially the fourth reviewer of PLDI'03 for his or her excellent five-page review.

8. REFERENCES

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, 2001.

[2] G. Almasi, C. Cascaval, and D. Padua. Calculating stack distances efficiently. In *Proceedings of the first ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, 2002.

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

[4] R. Balasubramonian, D. Albonesi, A. Buyuktos, and S. Dwarkadas. Dynamic memory hierarchy performance and energy optimization. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[5] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. 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, Apr. 1991.

[6] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, pages 353–357, 1975.

[7] K. Beyls and E. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, 2001.

[8] K. Beyls and E. D'Hollander. Reuse distance-based cache hint selection. In *Proceedings of the 8th International Euro-Par Conference*, Paderborn, Germany, 2002.

[9] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5(4):334–358, Aug. 1988.

[10] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.

[11] G. C. Cascaval. *Compile-time Performance Prediction of Scientific Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 2000.

[12] T. M. Chilimbi. 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, 2001.

[13] T. M. Chilimbi. On the stability of temporal data reference profiles. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, Spain, 2001.

[14] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, 2002.

[15] J. Cocke and K. Kennedy. Profitability computations on program flow graphs. Technical Report RC 5123, IBM, 1974.

[16] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, Sept. 1994.

[17] C. Ding. *Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse*. PhD thesis, Dept. of Computer Science, Rice University, January 2000.

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

[19] C. Ding and Y. Zhong. Compiler-directed run-time monitoring of program data access. In *Proceedings of the first ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, 2002.

[20] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Workload design: selecting representative program-input pairs. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, Virginia, 2002.

[21] H. Han and C. W. Tseng. Locality optimizations for adaptive irregular scientific codes. Technical report, Department of Computer Science, University of Maryland, College Park, 2000.

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

[23] W. Hsu, H. Chen, P. C. Yew, and D. Chen. 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)*, 2002.

[24] S. Jiang and X. Zhang. 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, 2002.

[25] Y. H. Kim, M. D. Hill, and D. A. Wood. Implementing stack simulation for highly-associative memories. In *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 212–213, May 1991.

[26] K. Knobe and V. Sarkar. Array SSA form and its use in

parallelization. In *Proceedings of Symposium on Principles of Programming Languages*, San Diego, CA, January 1998.

[27] D. Knuth. An empirical study of FORTRAN programs. *Software—Practice and Experience*, 1:105–133, 1971.

[28] T. Lafage and A. Seznec. 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*, 2000.

[29] Z. Li, J. Gu, and G. Lee. An evaluation of the potential benefits of register allocation for array references. In *Workshop on Interaction between Compilers and Computer Architectures in conjuction with the HPCA-2*, San Jose, California, February 1996.

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

[31] K. S. McKinley and O. Temam. Quantifying loop nest locality using SPEC'95 and the perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4):288–336, 1999.

[32] J. Mellor-Crummey, R. Fowler, and D. B. Whalley. Tools for application-oriented performance tuning. In *Proceedings of the 15th ACM International Conference on Supercomputing*, Sorrento, Italy, 2001.

[33] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. *International Journal of Parallel Programming*, 29(3), June 2001.

[34] F. Olken. Efficient methods for calculating the success function of fixed space replacement policies. Technical Report LBL-12370, Lawrence Berkeley Laboratory, 1981.

[35] V. Phalke and B. Gopinath. 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, 1995.

[36] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, 2002.

[37] D. D. Sleator and R. E. Tarjan. Self adjusting binary search trees. *Journal of the ACM*, 32(3), 1985.

[38] B. So, M. W. Hall, and P. C. Diniz. 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, 2002.

[39] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, Florida, June 1994.

[40] M. M. Strout, L. Carter, and J. Ferrante. 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, 2003.

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

[42] K. O. Thabit. *Cache Management by the Compiler*. PhD thesis, Dept. of Computer Science, Rice University, 1981.

[43] D. W. Wall. Predicting program behavior using real or estimated profiles. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, Canada, 1991.

[44] Y. Zhong, C. Ding, and K. Kennedy. Reuse distance analysis for scientific programs. In *Proceedings of Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Washington DC, March 2002.

[45] Y. Zhou, P. M. Chen, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of USENIX Technical Conference*, 2001.