Locality Approximation Using Time

Xipeng Shen, Jonathan Shaw, Brian Meeker and Chen Ding

Computer Science Department, University of Rochester
{xshen, jshaw, bmeeker, cding}@cs.rochester.edu

Abstract

Locality, characterized by data reuses, determines caching performance. Reuse distance (i.e. LRU stack distance) precisely characterizes program locality and has been used in memory-related research since 1970s. However, the high cost of measuring it still urges a breakthrough before its practical uses in performance debugging, locality analysis and optimizations of long-running applications.

In this work, we improve the efficiency by exploring the connection between time and locality. We propose a statistical model converting cheap time distance to costly reuse distance. Compared to the state-of-the-art technique, this approach reduces measuring time by 17 times, and approximates cache line reuses with over 99% accuracy. Experiments demonstrate the effective uses of the approximated reuse distance in cache miss rate prediction. This work, for the first time, reveals the strong correlations between time and locality. It makes precise locality as easy to obtain as data access frequency, removes the obstacles to reuse distance's practical uses, and opens new opportunities for program optimizations.

1. Introduction

As the memory hierarchy becomes deeper and shared by more processors, cache performance increasingly determines system speed, cost and energy usage. The effect of caching depends on program locality or the pattern of data reuses. Reuse distance, also called LRU stack distance, characterizes program locality precisely. But it has been expensive to measure, which impedes its practical uses in performance debugging, locality analysis and optimizations of long-running applications.

Initially proposed as LRU stack distance by Mattson et al. [18] in 1970, **reuse distance** is the number of distinct data elements accessed between the current and the previous access to the same data element [9]. As an example, in data reference trace “a b c b d d a”, the reuse distance of the second access to data element “a” is 3 since “b”, “c” and “d” are the distinct data elements between the two accesses to “a”. Reuse distance precisely captures temporal locality and reflects spatial locality of a program's execution. A *Reuse distance histogram*, illustrated in Figure 1(a), summarizes the distribution of the reuse distances in an execution.

Researchers have used reuse distance for different purposes: to study the limit of register reuse [14] and cache reuse [8, 12, 27], to evaluate the effect of program transformations [2, 4, 8, 27, 30], to predict cross-architecture program performance [16, 17], to identify critical instructions [10, 11], and to detect locality phases [23].

Despite the many uses in research, the high cost of reuse distance measurement has severely hindered its uses. The state-of-the-art measurement technique slows down a program's execution up to hundreds of times: The measurement of a 1-minute execution takes more than 4 hours. That obstructs the application of reuse

distance in performance debugging during the development stage, eliminates the possible uses for efficient locality optimizations, and makes offline program analysis painful or even unfeasible for long-running applications.

The last 30 years have seen a steady stream of research on accelerating reuse distance measurement. In 1970, Mattson et al. published the first measurement algorithm [18] using a list-based stack. The later research—including that of Bennett and Kruskal in 1975 [3], Olken in 1981 [20], Kim et al. in 1991 [13], Sugumar and Abraham in 1993 [25], Almasi et al. in 2002 [2], and Ding and Zhong in 2003 [9]—has reduced the cost through various data structures and algorithms. However, the hundreds of times slowdown of the state-of-the-art technique is still urging a breakthrough.

In this work, we address the problem by exploring the connections between time distance and reuse distance. **Time distance** is defined as the number of data elements accessed between the current and the previous access to the same data element. (The time distance is 6 for the example given in the second paragraph.) The difference from reuse distance is not having the “distinct” requirement, which makes its measurement as light as just recording the last access time of each data element—a small portion of the cost of reuse distance measurement.

As what people commonly conceived, time distance cannot serve as an accurate locality model. An access with time distance much larger than the cache size could still be a cache hit as long as its reuse distance is less than the cache size (assuming fully-associative cache with each cache line having one data element.) For a particular access, suppose its time distance is d , its possible reuse distance could be any value from 1 to $d - 1$.

However, this work discovers that the two kinds of distances have surprisingly strong relations between them. We propose a novel statistical model which can derive an approximation of the reuse distance histogram of an execution, illustrated in Figure 1(a), from the time distance histogram of that execution, illustrated in Figure 1(b), with over 99% accuracy. The only other input to the model is the number of distinct data elements in the execution, which is just a byproduct of time distance measurement.

The statistical model includes three steps. For an execution, the model first uses the time distance histogram to calculate the probability for a randomly picked time-point to fall into a randomly picked data element's reuse interval of any given length. A *reuse interval* is an interval with accesses to the same data element at both ends and without accesses to that data element in the between. It then uses the result to calculate the probability for a randomly picked data element to appear in a time interval of any given length. By treating reuse distance estimation as a binomial process, the model then estimates the distribution of the number of distinct data in any time interval, thus the reuse distance histogram. The model

is general enough to allow the uses of histograms of any scale and to give the estimated reuses of different granularity.

We evaluate the model on both generated traces of different distributions and SPEC CPU2000 benchmarks. The model demonstrates over 94% accuracy for data element reuses and 99% for cache line reuses. Compared to the state-of-the-art reuse distance measurement, the approach reduces time overhead by 17 times. We apply the estimated reuse distance histograms to cache miss rate prediction across different cache sizes. The average error is less than 0.4%.

The rest of the paper is organized as follows. Section 2 gives a high level overview of the estimation problem and the statistical model. Section 3 lists the concepts and notations used in the model. Section 4 and 5 describes the statistical model and its implementation. Section 6 presents the experiment results. Section 7 discusses related work. The draft concludes with a summary.

2. Overview of the Problem and the Solution

Reuse distance is a precise model of program locality. Reuse distance histograms summarize the locality of an execution. Figure 1(a) gives an example, where the X-axis is reuse distance and the Y-axis is the percentage of memory references. In the graph, the seventh bar, for instance, shows that 25% of total memory accesses have reuse distance between 32 and 64. Accesses with reuse distance larger than a cache size would cause cache capacity misses. Researchers have used reuse distance histograms in performance prediction, program analysis and optimizations [2, 4, 5, 8, 10, 11, 12, 14, 16, 17, 23, 27, 30].

The direct measurement of reuse distance is expensive. A naive implementation is to store the access trace in a huge array and count the number of distinct elements between every two consecutive accesses to the same data. The complexity is $O(T^2)$, where T stands for the trace size. Starting from Mattson et al.’s algorithm in 1970, people have used stack, vector, AVL tree, splay tree and other data structures to improve the efficiency [2, 3, 13, 18, 20, 25]. In 2003, Ding and Zhong proposed an approximation algorithm, which uses a splay tree to organize data references with each node storing a group of references conducted in an interval. It gains efficiency by ignoring the access order of the references in a node and dynamically compressing and reshaping the tree. Their technique achieves the lowest time complexity as $O(T \log \log N)$, where N is the number of distinct data elements in a program. However, the actual overhead of the technique still slows down a program’s execution by hundreds of times, impeding many practical uses of reuse distance.

All previous algorithms have essentially implemented the definition of reuse distance—“counting” the number of distinct data accessed for each reuse. The technique to be presented tackles the problem from a different aspect: *Can we use some easily obtained program behavior to statistically approximate reuse distance?*

We choose time distance to play the role. Time distance is the number of data accessed between two consecutive accesses to the same data. Figure 1(b) gives a time distance histogram. Time distance has two properties. First, it is easy to measure: We can measure it in linear time with just a hash table recording the last access time of each data¹. Second, the time distance histogram of an execution has strong correlation with the reuse distance histogram. We take the following data access trace as our example:

a b b b b b b a

Each letter in the trace represents an access to a variable. For a particular data access, the two kinds of distance may seem unre-

lated: The time distance of the second access to variable a is 7, which could correspond to 7 different reuse distance, from 0 to 6. However, if we know the time distance histogram—among 6 reuses, one has time distance of 7 and five have time distance of 1, we can easily determine the trace given the number of variables and thus obtain the reuse distance. It’s not always possible or efficient to derive the whole trace from a time distance histogram.

Section 4 presents a statistical model directly converting a time distance histogram to the corresponding reuse distance histogram. From the time distance histogram, it calculates the probability for any given variable to appear in a time interval of any given length. It models the reuse distance estimation as a binomial process and estimates the distribution of the amount of distinct data in any time interval, which gives the reuse distance histogram.

3. Terminology

Before presenting the reuse distance estimation model, we first explain the concepts and notations used in the model.

A Reuse : is the access to a memory unit that has already been accessed before. In this paper, we only care two kinds of memory units: *data element*, i.e. the memory word accessed by a “load” or “store” instruction; *cache line*, i.e. a block of data mapping to the same cache line. The former reuse is called *element reuse* and the latter one is named *cache line reuse*. All the concepts below apply to both kinds of reuses. For simplicity, we will use “data” for the memory unit.

A reuse interval : is an interval with accesses to the same data at both ends and without accesses to that data in the between.

Reuse distance : also called stack distance, is defined as the number of distinct data accessed inside a reuse interval (excluding the data accessed at the two ends of the reuse interval.) The minimal value is 0.

Reuse distance histogram : also called reuse signature, is a histogram showing the distribution of reuse distances in an execution. The X-axis is reuse distance, and the Y-axis is the fraction of total references. Each bin in the histogram shows the fraction of total program references with certain reuse distances.

$P_R(k)$: the fraction of references having reuse distance of k . It is the Y-axis value of a point whose X-axis value is k on a reuse distance histogram.

Time distance : is defined as the number of data (no need to be “distinct”) accessed in a reuse interval, including the access at one end of the interval. The minimal value is 1.

Time distance histogram : is a histogram showing the distribution of time distances in an execution.

$P_T(k)$: the fraction of references having time distance of k . It is also the Y-axis value of a point on a time distance histogram with X-axis value as k .

$P(k, \Delta)$: the probability of having k distinct data in a Δ long interval.

T : the total time-points in an execution. We use logical time in this paper without explicitly saying so, i.e. the number of data accesses in a program execution. A time-point is a data access.

$T(v, n)$: the time when the n ’th access to data v is conducted. Note, we count time distance cyclically. $T(v, N+1) - T(v, N)$ is the time distance from the starting of the program to the first access of v plus that from the last access of v to the end of the execution. (More discussion on that is in Section 5.1.)

$T_{<t}(v)$: the time when data v ’s last access before time t is conducted.

¹ In this paper, without explicit noting, time is logical time, i.e. the number of data accesses from the beginning of program execution.

Figure 1. A reuse distance histogram and time distance histogram on log scale.

$T_{>t}(v)$: the time when data v 's first access after time t is conducted.

V : the set of distinct data accessed in a program's execution.

v_t : the data accessed in time t .

N : the total number of distinct data accessed in a program's execution.

$N(v)$: the number of accesses to data v in the execution.

$B(x)$: a binary function, returning 1 when x is true and 0 when x is false.

$\sum_v$: the abbreviation of $\sum_{v \in V}$.

$\sum_t$: the abbreviation of $\sum_{t=0}^{T-1}$.

$P_1(\delta)$: Given a random time-point t , if we pick a data v at random from those that are not accessed at time t , $P_1(\delta)$ is the probability that t is in one of v 's reuse intervals of length δ .

$P_2(\tau)$: Given a random time-point t , if we pick a data v at random from those that are not accessed at time t , $P_2(\tau)$ is the probability that v 's last access prior to t is at time $t - \tau$.

$P_3(\Delta)$: Given a random time-point t , if we pick a data v at random from those that are not accessed at time t , $P_3(\Delta)$ is the probability that v 's last access prior to t is after $t - \Delta - 1$.

4. Approximation of Locality

This section describes the statistical model of using time distance to approximate program locality, represented by a data reuse distance histogram. Given a program's execution, the input to the model includes N , the number of distinct data accessed in the execution and the time distance histogram of the execution P_T ; the output of the model is the reuse distance histogram of that execution. The model is general in two aspects. First, the width of a bin in either histogram can be any value in any scale. Second, the model could be applied to data reuse as well as cache block reuse with any cache block size. These make the model flexible enough to estimate program locality in various granularity. But for easy explanation, in the following description, we assume that the size of bins in

both histograms is 1 and the histograms are of *element* reuses. Section 4.4 describes the extensions for other cases.

The algorithm includes three steps, described below in reverse order for the purpose of clarity.

4.1 Step 3: Estimate Reuse Distance

The key step, which is also the last step, is based on the following observation:

OBSERVATION 4.1. *Given a program's execution, we can approximate the reuse distance histogram if we know the probability for any given data to appear in any given time interval of length Δ , represented by $P_3(\Delta)$, and the time distance histogram.*

Note, the probability $P_3(\Delta)$ is a function of Δ only and is independent to the identity of the data.

The approximation is based on the model of Bernoulli processes. A *Bernoulli process* is a discrete-time stochastic process consisting of a sequence of independent random variables taking values over the set $0,1$. A typical example is coin tossing: there is a coin with probability p showing heads when being tossed. The probability for k heads in n tosses is in a *binomial distribution*, which is denoted as $f(k; n, p)$ and calculated as

$$f(k; n, p) = \binom{n}{k} p^k (1-p)^{n-k}.$$

Assuming the probability of any data to be accessed in an interval is independent from other data, the process of estimating the number of distinct data accessed in an interval becomes a Bernoulli process. Each distinct data is like one toss of an experimental coin with probability of $P_3(\Delta)$ showing heads. The number of tosses is equal to the total number of distinct data in the program. The number of times showing heads is the number of distinct data being accessed in an interval of length Δ .

We use $P(k, \Delta)$ to represent the probability of having k distinct data in a Δ long interval, which is equivalent to the probability of having k heads in N tosses of a coin with $P_3(\Delta)$ probability of showing heads each time. The percentage of references having reuse distance of k can therefore be calculated as follows:

$$P_R(k) = \sum_{\Delta} P(k, \Delta) \cdot P_T(\Delta)$$

where,

$$P(k, \Delta) = \binom{N}{k} P_3(\Delta)^k (1 - P_3(\Delta))^{N-k}$$

4.2 Step 2: Calculate $P_3(\Delta)$

This step is to calculate $P_3(\Delta)$, the probability of any given data to appear in a time interval of length Δ . Given any time-point t , the interval $[t - \Delta, t)$ includes Δ time points, represented by $t - \tau$, where $\tau = 1, 2, \dots, \Delta$. A given data being accessed in $[t - \Delta, t)$ is equivalent to that its last access time before t is at time $t - 1$, or $t - 2$, or, ..., or $t - \Delta$. Given the definition of $P_2(\tau)$, we can calculate $P_3(\Delta)$ as follows:

$$P_3(\Delta) = \sum_{\tau=1}^{\Delta} P_2(\tau)$$

In a real execution, the value of $P_2(\tau)$ may be different for different time-points and different data. But for statistical inference, we want to obtain the average probability, which is calculated as follows.

$$\begin{aligned} P_2(\tau) &= \frac{1}{T \cdot (N-1)} \sum_{t, v \neq v_t} B(t - T_{<t}(v) = \tau) \\ &= \frac{1}{T \cdot (N-1)} \sum_v \sum_{t(v_t \neq v)} B(t - T_{<t}(v) = \tau) \\ &= \frac{1}{T \cdot (N-1)} \sum_v \sum_{n=1}^{N(v)} \sum_{t=T(v,n)+1}^{T(v,n+1)-1} B(t - T(v, n) = \tau) \\ &= \frac{1}{T \cdot (N-1)} \sum_v \sum_{n=1}^{N(v)} B(T(v, n+1) - T(v, n) > \tau) \end{aligned}$$

Because

$$\sum_{t=T(v,n)+1}^{T(v,n+1)-1} \frac{1}{T(v, n+1) - T(v, n) - 1} = 1,$$

the value of $P_2(\tau)$ doesn't change after being multiplied by that term. Thus,

$$\begin{aligned} P_2(\tau) &= \frac{1}{T \cdot (N-1)} \sum_v \sum_{n=1}^{N(v)} B(T(v, n+1) - T(v, n) > \tau) \cdot \\ &\quad \sum_{t=T(v,n)+1}^{T(v,n+1)-1} \frac{1}{T(v, n+1) - T(v, n) - 1} \\ &= \frac{1}{T \cdot (N-1)} \cdot \\ &\quad \sum_v \sum_{n=1}^{N(v)} \sum_{t=T(v,n)+1}^{T(v,n+1)-1} \frac{1}{T(v, n+1) - T(v, n) - 1} \cdot \\ &\quad B(T(v, n+1) - T(v, n) > \tau) \\ &= \frac{1}{T \cdot (N-1)} \cdot \\ &\quad \sum_v \sum_{n=1}^{N(v)} \sum_{t=T(v,n)+1}^{T(v,n+1)-1} \frac{1}{T_{>t}(v) - T_{<t}(v) - 1} \cdot \\ &\quad B(T_{>t}(v) - T_{<t}(v) > \tau) \\ &= \frac{1}{T \cdot (N-1)} \sum_{v, t(v_t \neq v)} \frac{1}{T_{>t}(v) - T_{<t}(v) - 1} \cdot \\ &\quad B(T_{>t}(v) - T_{<t}(v) > \tau) \\ &= \frac{1}{T \cdot (N-1)} \sum_{v, t(v_t \neq v)} \sum_{\delta=\tau+1}^T \frac{1}{\delta - 1} \cdot \\ &\quad B(T_{>t}(v) - T_{<t}(v) = \delta) \\ &= \sum_{\delta=\tau+1}^T \frac{1}{(\delta - 1)} \cdot \\ &\quad \frac{1}{T \cdot (N-1)} \sum_{v, t(v_t \neq v)} B(T_{>t}(v) - T_{<t}(v) = \delta). \end{aligned}$$

$$\text{Let } P_1(\delta) = \frac{1}{T \cdot (N-1)} \cdot \sum_{v, t(v_t \neq v)} B(T_{>t}(v) - T_{<t}(v) = \delta),$$

thus

$$P_2(\tau) = \sum_{\delta=\tau+1}^T \frac{P_1(\delta)}{\delta - 1}$$

The problem now reduces to calculating $P_1(\delta)$, which the probability for a randomly chosen time-point t to fall into a δ -long reuse interval of a data, which is randomly chosen from the data that are not accessed at t .

4.3 Step 1: Calculate $P_1(\delta)$

We derive the probability from the time distance histogram as follows.

$$\begin{aligned}
P_1(\delta) &= \frac{1}{T \cdot (N-1)} \sum_{v, t(v_t \neq v)} B(T_{>t}(v) - T_{<t}(v) = \delta) \\
&= \frac{1}{T \cdot (N-1)} \sum_v \sum_{t(v_t \neq v)} B(T_{>t}(v) - T_{<t}(v) = \delta) \\
&= \frac{1}{T \cdot (N-1)} \sum_v \sum_{n=1}^{N(v)} \cdot \\
&\quad \sum_{t=T(v,n)+1}^{T(v,n+1)-1} B(T_{>t}(v) - T_{<t}(v) = \delta) \\
&= \frac{1}{T \cdot (N-1)} \sum_v \sum_{n=1}^{N(v)} (T(v, n+1) - T(v, n) - 1) \cdot \\
&\quad B(T(v, n+1) - T(v, n) = \delta) \\
&= \frac{1}{T \cdot (N-1)} (\delta - 1) \sum_v \sum_{n=1}^{N(v)} B(T(v, n+1) - T(v, n) = \delta) \\
&= \frac{1}{T \cdot (N-1)} (\delta - 1) \sum_t B(t - T_{<t}(v_t) = \delta) \\
&= \frac{\delta - 1}{N - 1} \frac{1}{T} \sum_t B(t - T_{<t}(v_t) = \delta)
\end{aligned}$$

According to the definition of the time distance histogram, we have $P_T(\delta) = \frac{1}{T} \sum_t B(t - T_{<t}(v_t) = \delta)$. Thus,

$$P_1(\delta) = \frac{\delta - 1}{N - 1} P_T(\delta)$$

This concludes the basic model for the approximation of a reuse distance histogram. In summary, given an execution, we use the time distance histogram to estimate the probability of any given data to be accessed within any time interval. By treating reuse distance estimation as a binomial process, we can estimate the distribution of the number of distinct data in any time interval, thus the reuse distance histogram.

5.4 Extensions

In the above description, we assumed that each bin's width in both time and reuse distance histograms is 1. With some extensions, our model is general enough to allow any width of a bin, and permit the width of the bins in a histogram on any scale (e.g. linear, log or random scale.) For lack of space, please see [21] for the extensions.

5. Implementation

This section describes the implementation of the statistical model and time distance collection.

5.1 Reuse Distance Estimator

We implement the statistical model with the extensions to allow histograms to have bins of different width. The complexity of the algorithm is $O(L * M)$, where L is the number of bins in the time distance histogram and M is that in the reuse distance histogram. The major computation is the calculation of binomial distributions.

We use the following optimizations to calculate binomial distributions efficiently. When n is large, a standard approximation of a binomial distribution $B(n, p)$ is the normal distribution $N(np, np(1 - p))$, whose probabilities can be easily derived from the standard normal distribution $N(0, 1)$. We offline calculate the probabilities of $N(0, 1)$ at one million uniformly distributed points

between $[-6, 6]$, and store them into a file. That simplifies the calculation of binomial distributions to a simple table look-up and a trivial derivation.

A boundary case is the treatment of the first access to a variable. We use its cyclic time distance, which is the time distance from the starting of the program to the first access plus that from the last access to that variable to the end of the execution. We tried other options like ignoring the first accesses or considering their time distances very large. The cyclic time distance scheme shows the best effects in reuse distance estimation.

5.2 Time Distance Measurement

We use a hash table to store the last access time of each distinct data. The algorithm is straightforward: On every data access, we obtain the time distance by subtracting the current time by the last access time to that data and store the distance to an accumulator.

To speedup the measurement, we use a batch-processing model: accesses are put into a fixed-size buffer and are processed together when the buffer is full. This accelerates the measurement but adds a branch after each data access to check whether the buffer is full or not. We use PIN as the binary instrumentor [15]. The branch prevents the inlining of the instrumented code, slowing down the measurement.

We remove the branch with the help of the operating system. The scheme is to close the permission of the last page of the buffer such that an access to the last page will trigger an interrupt, whose handler invokes the processing of the buffer as the buffer is almost full. This optimization reduces over half of the overhead of measuring time distance.

6. Evaluation

This section presents experiment results on both generated traces and the programs from SPEC CPU2000 suite. Generated traces make it easy for us to test the approximation model on histograms of different distributions. We use the SPEC programs to demonstrate the approximation efficiency and accuracy in real uses.

6.1 Methodology

All experiments were run on Intel(R) Xeon(TM) 2.00GHz Processors with 2 GB of memory running Fedora Core 3 Linux. We use PIN 3.4 for instrumentation with gcc version 3.4.4 as our compiler. All compilations have the highest level optimization with “-O3” turned on.

For testing on histograms of different distributions, we developed a trace generator, which automatically generates the corresponding data trace given a reuse distance histogram [21]. The reuse distance histogram of the generated trace matches the given histogram. The details of the generator are in our technical report [21].

For the second part of the evaluation, we use a dozen of the SPEC CPU2000 benchmarks, listed in Table 2. All six integer benchmarks (CINT) are written in C; half of the floating-point benchmarks (CFP) are in Fortran 77 and others are in C. We were unable to use Fortran 90 benchmarks because PIN 3.4 has some problem in supporting the signal handler used in time distance measurement for Fortran 90 programs. We choose the 12 benchmarks from the rest to test on.

In the experiments, we use Ding and Zhong's implementation of their technique (using PIN) to measure the real reuse distance histograms [9]. To our knowledge, that is the fastest tool for reuse distance collection.

6.2 Results on Generated Traces

Using the trace generator, we generate traces of three reuse distance histograms of different reuse distributions: random, pulse-

like and exponential distributions as shown in Figure 2(a) and Figure 3(a)(b). The fluctuations in the three histograms vary from plenty to zero.

We measure the time distance histogram of the generated trace, represented by TDH , from which we use the statistical model to approximate the reuse distance histogram of the trace, represented by $\widehat{RDH}$. We then measure the real reuse distance histogram of the generated trace, denoted by RDH . The accuracy of $\widehat{RDH}$ is calculated as follows:

$$accuracy = 1 - \frac{\sum_i |B_i - \widehat{B}_i|}{2} \quad (1)$$

Where, B_i is the Y-axis value of the i 'th bin in RDH and $\widehat{B}_i$ is the one in $\widehat{RDH}$. The division by 2 is to normalize the accuracy to [0,1]. In the experiments on generated traces, we make the width of the bins of both histograms to be 1 in order to reveal what happens within a wider bin.

6.2.1 Random Distribution

In this experiment, the generated trace has a reuse distance histogram in which each bin's value is randomly chosen as shown by the light lines in the top two graphs of Figure 2 (for easy reading, we connect the points into lines.) There are 500 variables in the generated trace, therefore the largest reuse distance is 500. The trace generator generates 50000 accesses to those variables. The measured time distance is shown in the middle two graphs of Figure 2, which is a long tail curve. Visually the time distance histogram is dramatically different from the reuse distance histogram. However, the reuse distance histogram it derives through our statistical model matches the real histogram very well. In Figure 2(a) and (b), although the estimated curve cannot match every peak or valley of the real curve, it fits the flow of the curve very well. Because of that, the bar graphs of the histograms match each other well: the accuracy is 97.8% for linear scale and 99.1% for log scale, shown by the bottom two graphs of Figure 2.

6.3 Other Distributions

Figure 3 shows the approximation results on two other distributions. The three graphs on the left column are for the trace whose reuse distance histogram is in a pulse-like shape. The estimated histogram matches the pulses with a smoother shape, which causes some deviations at the pulse boundaries. However, because the estimated curve fits the flow of the real curve well, the local deviations tend to cancel each other in a bar graph. Figure 3 (e) shows the log-scale reuse distance histograms with approximation accuracy of 98.4%. Figure 3(c) is the time distance curve, which has 5 smoothly fluctuating waves decreasing gradually to 0. What the graph doesn't show for lack of space is a thin peak at the end of the curve, distance 50000, with 0.001% references.

The right column of Figure 3 shows the results for a trace whose reuse distance histogram is in an exponential distribution. The estimated curve matches the real curve so well that they cannot be distinguished in Figure 3(b). Figure 3(d) shows the time distance curve, which, like that of random distribution (Figure 2(b)), is also a long tail curve. The difference is that it has a thin peak at the end of the tail and the head part is thinner and higher. Figure 3(f) gives the log-scale reuse distance histograms with accuracy 98.7%.

Summary This section gives the result of reuse distance estimation on three different distributions. The statistical model generates histograms well matching the real ones with smoother shapes. There could be some local deviations, which however usually cancel each other in a bar graph thanks to the good flow match. The next section demonstrates the effectiveness of the model on real programs.

6.4 Results on SPEC CPU2000

We apply the statistical model to 12 SPEC CPU2000 benchmarks. The experiment demonstrates over 17 times speedup over the past technique, shows 94% accuracy for element reuses, and 99.5% for cache line reuses on log scale.

6.4.1 Comparison of Time Cost

Table 1 shows the time cost of the statistical model compared with the fastest reuse distance measurement in the past developed by Ding and Zhong [9]. The results are of the runs of 12 SPEC CPU2000 benchmarks with the train inputs. The first column shows the benchmark names. The second column gives the basic overhead of the instrumentation, T_{inst} . It is the running time of the benchmarks after being inserted an invocation of an empty function at each memory access. Much of that overhead could be saved with a more efficient instrumentor, e.g. a source-code level instrumentor through a compiler. The left half of the rest of the table shows the result of element reuses, and the right half gives that of cache line reuses (cache line is 128B wide.) Within each part, the first column is the time of Ding and Zhong's technique, T_{RD} ; the sum of the next two columns is the time of our technique, which includes the time to measure time distance, T_{TD} , and the time to convert time distance histograms to reuse distance histograms, T_{conv} . The next column shows the speedup of our technique compared to Ding and Zhong's. In order to avoid the effects of different instrumentors, we subtract the instrumentation overhead from both kinds of measure time when calculating the speedup as follows:

$$Speedup = (T_{RD} - T_{inst}) / (T_{TD} + T_{conv} - T_{inst})$$

The table demonstrates 19.4 times speedup for data element reuses. The smallest speedup is 8 times on program *mcf*, which has the largest number of data elements, 10.1 million, among integer benchmarks. The program has a large range of reuse distances and time distances, which makes its conversion time much longer than others.

For cache line reuses, the speedup is from 10 to 21 times, with the average as 17.2. Program *mcf* shows 19.2 times speedup. The significantly greater speedup compared to data element reuse is due to the decrease of the number of counting units and the range of distances. Figure 4 shows the reuse distance histograms of element reuse and cache line reuse. The distance range shrinks from 16M to 32K.

6.4.2 Approximation Accuracy

Table 2 shows the accuracy of the reuse distance approximation on both the element and cache line level for the test and train runs of the SPEC CPU2000 benchmarks. We measure the accuracy on both linear and log scale: A bar in a linear-scale histogram covers the reuse distance of 1K; the bars in a log-scale histogram have the range as [0 1K), [1K, 2K), [2K, 4K), [4K, 8K), The accuracy calculation is through Equation 1. We use Ding and Zhong's technique to measure the real reuse distance histogram².

The approximation accuracy for cache line reuses is 99.3% and 99.4% for linear and log scale respectively. The worst accuracy is on benchmark *ammp*, but still as accurate as 96.5% and 96.6%. Nine out of the 12 benchmarks have over 99% accuracy.

The accuracy for element reuses is 91.8% and 94.1% on average. Benchmark *mcf* and *equake* give low accuracy. In Figure 4, we draw the histograms of *mcf* reuses on both element and cache line level. The largest error of element reuse approximation hap-

²Ding and Zhong's technique allows the indication of either relative or absolute error bound of the measured reuse distance. We used the version with 0.1% relative error limit.

(a) Reuse distance histogram with bin width as 1

(b) Enlarged reuse distance histogram

(c) Time distance histogram with bin width as 1

(d) Enlarged time distance histogram

(e) Reuse distance histogram bin graph on linear scale

(f) Reuse distance histogram bin graph on log scale

Figure 2. Time distance histogram and the real and approximated reuse distance histogram of a trace with random data accesses.

(a) Reuse distance histogram of pulse distribution

(b) Reuse distance histogram of exponential distribution

(c) Time distance histogram of pulse distribution

(d) Time distance histogram of exponential distribution

(e) Reuse distance bins of pulse distribution

(f) Reuse distance bins of exponential distribution

Figure 3. The time distance and estimated reuse distance histograms on traces generated from a pulse-shape reuse distance histogram and an exponentially distributed reuse distance histogram.

pens in bars of [128K, 256K) and [256K, 512K). The estimated bar in the range [256K, 512K) matches well with the real bar in the range [128K, 256K). A possible reason for that mismatch is the independence assumption made in the statistical model: we assume that each variable has a probability to appear in an interval and that probability is independent of the other variables. Although that assumption holds in most cases, it could result in longer reuse distances as shown in *mcf* case. However, a larger granularity removes the error almost completely, as shown in the result of cache line reuses in Figure 4 (b). Program *equake* has the similar phenomenon. A common use of reuse distance is to study cache behavior through cache line reuse histograms, for which the occasionally low accuracy of element reuses does not matter.

Overall, the statistical model shows over 99% accuracy for cache line reuses and 94% for element reuses. Its 17 times speedup over the past technique removes the obstacles blocking the practical uses of precise locality models in program analysis and optimizations.

6.5 Uses for Cache Miss Rate Prediction

A cache-line reuse histogram is useful to estimate the cache miss rates of the execution of a program given different cache sizes. For a fully-associative cache, if we draw a vertical line at the cache size on a reuse distance histogram, all accesses to the left of the line will definitely be hit in the cache and all accesses to the right will surely be missed. This technique has been used for program analysis across inputs [28] and performance prediction across architectures [16]. In this experiment, we compare the approximated cache miss rates from the real and estimated reuse distance histograms in order to test the effect of the statistical model in real uses.

In the experiment, we assume a fully-associative cache with a cache line of 128 bytes. (Smith showed how to calculate analytically the effect of a set-associative cache from a fully-associative one [24].) The cache size range is from 128K to 8M.

Figure 5 shows the cache miss rate curves of two benchmarks. The solid curves show the miss rates obtained from the real reuse distance histograms and the dotted curves show those from the estimated ones. Table 2 shows low estimation accuracy for element reuses but high accuracy for cache line reuses of benchmark *mcf*. The latter determines the miss rate prediction precision: Figure 5(a) shows the good match of the two curves given the big changing range of miss rates from 13.4% to 0.4%. Figure 5(b) demonstrates the similar result of a floating-point benchmark, *ammp*, whose miss rates vary from 2.4% to 0. Table 3 gives the largest and the average errors of the approximated cache miss rates of all benchmarks. Here we use absolute error. The largest errors are 2.5% for *twolf* and 1.4% for *equake*, which is consistent to Table 2, where, the two benchmarks have the lowest accuracy of cache line reuse estimation. Note although it has the worst element reuse estimation, benchmark *mcf* has very accurate cache miss rate estimation. It is because of its excellent cache line reuse estimation, which is the base for cache performance prediction. On average of all benchmarks, the approximated miss rates are less than 0.42% from the estimation from the real reuse distance histograms.

The experiments above estimate cache misses for different cache sizes. Another dimension is for different runs of a program on different inputs. Ding et al. demonstrate the predictability of reuse distance histograms across program runs and use them to estimate cache misses of a program across different runs [9, 22, 29]. In the same way, we can use the approximated reuse distance histograms from the statistical model to serve that estimation. It is worth noting that the cross-input prediction of reuse distances makes it possible to use short runs on small inputs to estimate long runs on large inputs. However, as Ding and Zhong showed, not all programs can be predicted in that way [9], and furthermore, the training runs

Benchmark		largest error	average error
CINT	crafty	0.0003	0.0001
	gcc	0.0011	0.0006
	gzip	0.0041	0.0021
	mcf	0.0063	0.0034
	twolf	0.0250	0.0176
	vortex	0.0002	0.0001
CFP	ammp	0.0046	0.0023
	applu	0.0065	0.0058
	equake	0.0137	0.0131
	mesa	0.0001	0.0000
	mgrid	0.0016	0.0015
	wupwise	0.0041	0.0036
average		0.0056	0.0042

Table 3. The absolute error of the cache miss rates approximated from the estimated cache line reuse distance histograms. The cache size changes from 128K to 8M. The largest error is the error among all cache sizes.

to build the prediction model have to be long enough to separate different reuse patterns, which hurts the efficiency benefits.

7. Related Work

As a precise locality model, reuse distance has been used to study the limit of register reuse [14] and cache reuse [12, 8, 27], and to evaluate the effect of program transformations [8, 4, 2, 27]. For cache performance prediction, Marin and Mellor-Crummey applied distance-based analysis to memory blocks and reported accurate miss-rate prediction across different program inputs and cache sizes [17, 16]. Fang et al. [11, 10] examined the reuse pattern per instruction and predicted the miss rate of 90% of instructions with a 97% accuracy. They used the prediction tool to identify *critical* instructions that generate the most cache misses and extended the distance model to memory disambiguation. Beyls and D’Hollander [5] used reuse distance profiles to generate cache hints, which tell the hardware if and at which level to place or replace a loaded memory block in cache. Zhong et al. recognize data with good affinity from their reuse distance patterns and reorganize program data accordingly [30]. Shen et al. identify long, recurring locality phases based on reuse distance patterns, which show significant effectiveness in cache resizing [23].

Because of the importance of reuse distance, its measurement has been studied for several decades. In 1970, Mattson et al. published the first measurement algorithm [18] using a list-based stack. The time complexity is $O(TN)$, where T is the execution length and N is the size of program data. Space cost is $O(N)$. Bennett and Kruskal used a vector and built an m -ary tree on it [3] to speedup the measurement. That reduces time complexity to $O(T \log T)$ but increases space complexity to $O(T)$. In 1981, Olken implemented the first tree-based method using an AVL tree [20] with time complexity $O(T \log N)$ and space overhead $O(N)$. Sugumar and Abraham in 1993 showed that a splay tree has better memory performance [25] and developed the widely used cache simulator, *Cheetah*. In 2002, Almasi et al. gave an algorithm that records the empty regions instead of non-empty cells in the trace and reduced 20% to 40% cost [2].

In 1991, Kim et al. proposed the first imprecise analysis method [13]. Their method marks S ranges in the list that store program data and counts the number of distances that fall inside each range. Its time complexity is $O(TS)$ and space complexity $O(C)$, where C is the furthest marker. Unlike approximate analysis, their method is accurate in counting the reuse distance within a marked range but not suitable for measuring the full length of reuse

Benchmark		Instrument overhead (sec.)	Data element level				Cache line level			
			Reuse dist. measure (sec.)	Time dist. measure (sec.)	Time to approximate (sec.)	Speedup times	Reuse dist. measure (sec.)	Time dist. measure (sec.)	Time to approximate (sec.)	Speedup times
CINT	crafty	423	15654	1162	5	20.5 X	9946	1030	2	15.6 X
	gcc	135	1333	223	7	12.6 X	926	212	1	10.1 X
	gzip	12	254	28	1	14.2 X	159	23	1	12.3 X
	mcf	131	3856	534	61	8.0 X	2585	257	2	19.2 X
	twolf	138	4262	403	2	15.4 X	2859	292	2	17.4 X
	vortex	236	8142	601	6	21.3 X	5142	548	2	15.6 X
CFP	ammp	354	20175	1333	4	20.2 X	12923	997	2	19.5 X
	applu	170	9806	534	9	25.8 X	5718	447	2	19.9 X
	equake	127	13182	766	3	20.3 X	7773	489	1	21.1 X
	mesa	1363	45316	3131	5	24.8 X	31191	2955	2	18.7 X
	mgrid	206	17336	823	2	27.7 X	10358	677	2	21.5 X
	wupwise	406	21145	1374	1	21.4 X	10876	1075	2	15.6 X
Average					19.4 X					17.2 X

Table 1. Comparison of the time of reuse distance approximation and measurement

Benchmark		Lang.	Description	Input	Number of data elements	Number of mem. accesses	Accuracy (%)			
							Element		Cache line	
							linear	log	linear	log
CINT	crafty	C	Game playing: chess	test	484K	2.8B	93.2	93.2	100	100
				train	484K	17.8B	93.4	93.4	100	100
	gcc	C	GNU C programming language compiler	test	916K	675M	99.3	99.4	99.9	99.9
				train	3.46M	1.5B	99.2	99.4	99.9	99.9
	gzip	C	GNU compression using Lempel-Ziv coding	test	69.8K	94.1M	98.9	99.8	99.5	99.5
				train	79.6K	273M	98.5	99.3	99.6	99.6
	mcf	C	Combinatorial optimization for vehicle scheduling	test	349K	53.5M	82.7	89.0	99.0	99.1
				train	10.1M	3.4B	67.9	72.4	97.7	98.4
	twolf	C	Place and route simulator	test	29.1K	109M	99.0	99.0	100	100
				train	435K	4.8B	89.5	90.2	97.5	97.5
vortex	C	Object-oriented Database	test	2.92M	5.3B	93.0	93.1	100	100	
			train	2.76M	9.6B	93.2	93.2	100	100	
CFP	ammp	C	Computational chemistry: Modeling molecule system	test	3.16M	2.5B	89.2	96.5	96.5	96.6
				train	3.16M	22.8B	95.9	96.2	99.2	99.5
	applu	F77	Parabolic/Elliptic Partial Differential Equations	test	212K	207M	94.0	99.5	99.8	99.9
				train	2.00M	10.5B	92.5	99.9	99.3	99.4
	equake	C	Seismic wave propagation simulation	test	2.30M	478M	91.9	92.8	99.5	99.5
				train	2.30M	13.0B	75.7	77.9	98.5	98.5
	mesa	C	3-D graphics library	test	1.63K	57.6B	95.4	95.4	99.9	99.9
				train	1.63K	62.1B	96.8	98.6	99.9	100
	mgrid	F77	Multi-grid solver: 3-D potential field	test	10.0M	14.0B	89.8	97.1	99.6	100
				train	1.39M	17.9B	90.5	97.6	99.7	99.8
wupwise	F77	Physics/Quantum Chromodynamics	test	38.4M	5.09B	92.8	93.2	99.7	99.7	
			train	38.4M	24.9B	91.2	91.6	99.6	99.6	
						Average	91.8	94.1	99.3	99.4

Table 2. Approximation accuracy of data element and cache line reuse distance histograms

(a) *mcf* element reuse histogram

(b) *mcf* cache line reuse histogram

Figure 4. The real and estimated reuse distance histograms of *mcf*, the benchmark with the largest approximation error. The X-axes are on log scale.

(a) *mcf* cache miss rate curves

(b) *ammp* cache miss rate curves

Figure 5. The cache miss rates obtained from the real and estimated reuse distance histograms given different cache size of fully associativity.

distance. In 2002, Zhong et al. proposed an approximation algorithm using block tree [27]. Ding and Zhong further extended the algorithm through dynamic tree compression in 2003 [9]. The approximation algorithms trade accuracy for efficiency by increasing the measurement granularity. A tree node contains a group of data accessed closely and assigns a single time stamp to all of them. The time complexity becomes $O(T \log \log N)$ and the space overhead is $O(\log N)$.

This work tackles the problem by exploring the relations between time distance and reuse distance, which provides 17 times speedup than Ding and Zhong’s technique. The complexity for measuring time distance is $O(T)$.

The key component of the technique is a statistical model. In 1976, Smith gave us an excellent example of applying statistical models to memory behavior characterization. He described a model that successfully approximated miss rates of set associative caches given the miss ratio for fully associative caches [24].

Compiler analysis has been successful in understanding and improving locality in basic blocks and loop nests. McKinley and Temam carefully studied various types of locality within and be-

tween loop nests [19]. Cascaval presented a compiler algorithm that measures reuse distance directly [6]. Allen and Kennedy discussed the subject comprehensively in their book [1]. Thabit identified data often used together based on their access frequency [26]. Chilimbi used grammar compression to find hot data streams and reorganized data accordingly [7].

8. Conclusions

In this work, we demonstrates the strong connections between time and locality. We propose a novel statistical model to approximate program locality from easily-obtained time distance histograms. Experiments show 17 times speedup over the state-of-the-art locality measurement. The approximation accuracy is over 99% for cache block reuse and over 94% for element reuse. The model is general enough to allow reuse distance histograms of any scale and data reuse of different granularity to be approximated. The technique eliminates the obstacles blocking efficient uses of reuse distance and opens up opportunities for various efficient performance debugging, performance prediction and program optimizations.

Acknowledgments

This material is based upon work supported by a grant from The National Science Foundation grant numbers EIA-0080124, EIA-0205061 (subcontract, Keck Graduate Inst), NSF EF-0328363 and NSF CNS-0509270. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of above named organizations.

We thank Yutao Zhong and Arrvinth Shriraman for the invaluable comments on the presentation.

References

- [1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, October 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, June 2002.
- [3] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, pages 353–357, 1975.
- [4] 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*, August 2001.
- [5] K. Beyls and E. D'Hollander. Reuse distance-based cache hint selection. In *Proceedings of the 8th International Euro-Par Conference*, Paderborn, Germany, August 2002.
- [6] G. C. Cascaval. *Compile-time Performance Prediction of Scientific Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 2000.
- [7] 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, June 2001.
- [8] C. Ding. *Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse*. PhD thesis, Dept. of Computer Science, Rice University, January 2000.
- [9] C. Ding and Y. Zhong. Predicting whole-program locality with reuse distance analysis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.
- [10] C. Fang, S. Carr, S. Onder, and Z. Wang. 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, June 2004.
- [11] C. Fang, S. Carr, S. Onder, and Z. Wang. Instruction based memory distance analysis and its application to optimization. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, St. Louis, MO, 2005.
- [12] S. A. Huang and J. P. Shen. 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, October 1996.
- [13] 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.
- [14] 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 conjunction with the HPCA-2*, San Jose, California, February 1996.
- [15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 2005.
- [16] G. Marin and J. Mellor-Crummey. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of Joint International Conference on Measurement and Modeling of Computer Systems*, New York City, NY, June 2004.
- [17] G. Marin and J. Mellor-Crummey. Scalable cross-architecture predictions of memory hierarchy response for scientific applications. In *Proceedings of the Symposium of the Las Alamos Computer Science Institute*, Sante Fe, New Mexico, 2005.
- [18] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.
- [19] 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, November 1999.
- [20] F. Olken. Efficient methods for calculating the success function of fixed space replacement policies. Technical Report LBL-12370, Lawrence Berkeley Laboratory, 1981.
- [21] X. Shen, J. Shaw, and B. Meeker. Accurate approximation of locality from time distance histograms. Technical report, Computer Science Department, University of Rochester, 2006.
- [22] X. Shen, Y. Zhong, and C. Ding. Regression-based multi-model prediction of data reuse signature. In *Proceedings of the 4th Annual Symposium of the Las Alamos Computer Science Institute*, Sante Fe, New Mexico, November 2003.
- [23] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Boston, MA, 2004.
- [24] A. J. Smith. On the effectiveness of set associative page mapping and its applications in main memory management. In *Proceedings of the 2nd International Conference on Software Engineering*, 1976.
- [25] R. A. Sugumar and S. G. Abraham. Multi-configuration simulation algorithms for the evaluation of computer architecture designs. Technical report, University of Michigan, 1993.
- [26] K. O. Thabit. *Cache Management by the Compiler*. PhD thesis, Dept. of Computer Science, Rice University, 1981.
- [27] 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.
- [28] Y. Zhong, S. G. Dropsho, and C. Ding. Miss rate prediction across all program inputs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, Louisiana, September 2003.
- [29] Y. Zhong, S. G. Dropsho, X. Shen, A. Studer, and C. Ding. Miss rate prediction across program inputs and cache configurations. *IEEE Transactions on Computers*, to appear.
- [30] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.