Scalable Implementation of Efficient Locality Approximation

Xipeng Shen[1] and Jonathan Shaw[2]

[1] Computer Science Department
The College of William and Mary, Williamsburg, VA
[2] Shaw Technologies, Inc.
Tualatin, OR

Abstract. As memory hierarchy becomes deeper and shared by more processors, locality increasingly determines system performance. As a rigorous and precise locality model, reuse distance has been used in program optimizations, performance prediction, memory disambiguation, and locality phase prediction. However, the high cost of measurement has been severely impeding its uses in scenarios requiring high efficiency, such as product compilers, performance debugging, run-time optimizations.

We recently discovered the statistical connection between time and reuse distance, which led to an efficient way to approximate reuse distance using time. However, not exposed are some algorithmic and implementation techniques that are vital for the efficiency and scalability of the approximation model. This paper presents these techniques. It describes an algorithm that approximates reuse distance on arbitrary scales; it explains a portable scheme that employs memory controller to accelerate the measure of time distance; it uncovers the algorithm and proof of a trace generator that can facilitate various locality studies.

1 Introduction

In modern computers, memory hierarchy is becoming deeper and shared by more processors; system performance is increasingly determined by program data locality. Reuse distance, also called LRU stack distance, is a widely used model for locality analysis. Compared to other locality metrics such as cache miss rate, reuse distance is hardware independent, cross-input predictable, and more precise in characterization [7, 12, 20]. It has been used in system performance analysis [6,9,10], performance prediction [12,19], program analysis and optimizations [2, 8, 20].

On the other hand, reuse distance is also one of the most expensive locality models to build. Despite decades of enhancement (e.g., [7, 13]), the measurement still slows down a program's execution by hundreds of times [16]. The high cost impedes the practical uses of reuse distance: It would make offline performance debugging and locality analysis painfully slow or even infeasible for long-running applications, and prevent the uses in runtime locality optimizations. The high cost is inherent in the definition of **reuse distance**—the number of *distinct* data accessed between this and the previous access to the same data item [7]. The requirement of being "distinct" implies that the

measurement has to recognize and filter out all the repetitive accesses in an interval, which is often costly.

Recently Shen et al. discovered the strong statistical connection between reuse distance and **time distance**—the number of data accessed between this and the previous access to the same data item (e.g., the last "a" in "a b b c a" has time distance of 4). This discovery gives rise to an algorithm that approximates reuse distance histograms from time distance histograms. Given that time distance is much cheaper to measure, the algorithm speeds up reuse distance measurement significantly [16].

However, it remains un-exposed how to implement the algorithm efficiently. In particular, this paper focuses on the problems on three folds.

The first is a scale problem. The algorithm described previously requires the finest granularity of distance histograms [16]. Every bar in a histogram must have width of 1—that is, all references in a bar have to have the same reuse (or time) distance. Such a scale requires the recording of and computation on millions of distance bars for a typical execution, making the algorithm not applicable to long running programs. However, extending the algorithm to support histograms of arbitrary scales is challenging. A basic extension has a too high time complexity (shown in Section 2.2). In this paper (Section 2.3), we describe some algorithmic changes to enable the removal of redundant computations, lowering the time complexity by orders of magnitude.

The second problem addressed in this paper is the overhead in time distance measurement. Although time distance is less costly to measure than reuse distance, a straightforward implementation still slows down a program's execution significantly, forming the bottleneck in our extended scalable algorithm. This paper (Section 3) presents an optimization that reduces the measurement overhead by more than a factor of 3 with the help of memory management unit (MMU). The optimization differs from previous MMU-based techniques in that it avoids the direct access to system registers and therefore is more portable.

Finally, this paper (Section 4) presents a trace generator, which produces data reference traces that satisfy the given requirement of reuse distance histograms. We are not aware of any prior systems that offer such a function. This generator not only facilitates the comprehensive evaluation of the reuse-distance approximation algorithm, but also can serve as a tool for other locality studies.

2 Algorithm Design for Scalability

This section reviews the statistical connection between time distance and reuse distance, identifies the scalability bottlenecks in a basic algorithm for reuse distance approximation, and then presents our changes to the algorithm to reduce the cost by orders of magnitude, making the algorithm scalable to long-running programs.

2.1 Review of the Connection between Time Distance and Reuse Distance

The key connection that bridges reuse distance and time distance is the following equation [16]:

$$p(\Delta) = \sum_{\tau=1}^{\Delta} \sum_{\delta=\tau+1}^{T} \frac{1}{N-1} p_{_T}(\delta), \tag{1}$$

where, $p(\Delta)$ is the probability for a randomly chosen data item (e.g., a variable) of a program to be referenced in an interval of length Δ, and $p_{_T}(\delta)$ is the probability for a randomly chosen data reference to have time distance of δ. This connection suggests that if we know all $p_{_T}(\delta)$s—that is, the time distance distribution of an execution—we will be able to compute all $p(\Delta)$s.

With $p(\Delta)$, the reuse distance can be computed easily. For a reuse interval of length Δ, the probability for its reuse distance to be k in an execution is

$$p(k, \Delta) = \binom{N}{k} p(\Delta)^k (1 - p(\Delta))^{N-k}, \tag{2}$$

where, N is the total number of distinct data items in a program. The intuition behind this equation is that $p(k, \Delta)$ is the probability for k distinct data items to appear in a Δ-long interval. This probability is like the probability to see k heads when N coins are tossed, with each coin's probability of showing heads to be $p(\Delta)$[3]. This probability obeys a binomial distribution.

It is easy to see that, with $p(k, \Delta)$, the probability for a data reference to have reuse distance of k is

$$p_{_R}(k) = \sum_{\Delta} p(k, \Delta) \cdot p_{_T}(\Delta). \tag{3}$$

2.2 Basic Algorithm for Approximating Reuse Distance Histograms

A program execution often conducts a large number of data references. A typical way to concisely characterize reference locality is to use reuse distance histograms instead of individual reuse distances [2, 7, 12, 20]. Figure 1 illustrates a reuse distance histogram. A bar in the graph shows the fraction of the memory references whose reuse distances are in a certain range. For the same reason, individual time distances are usually not affordable; time distance histograms are used.

Therefore, extending the equations in Section 2.1 to handle histograms is vital for practical uses. A straightforward way to do the extension is to consider that all the references in a bar have the same $p(\Delta)$, denoted by $P(b_i)$ ($i = 1, 2, \cdots, L_T$ for a time distance histogram that has L_T bars). So, the probability for k variables to be accessed in a reuse interval contained in bar i is

$$P(k, b_i) = \binom{N}{k} P(b_i)^k (1 - P(b_i))^{N-k}.$$

[3] This analogy assumes that two data items are independent in terms of the probability for them to be accessed in a given interval; this assumption has shown little influence to reuse-distance approximation [16].

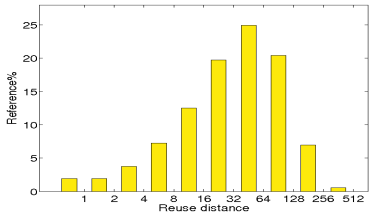


Fig. 1. A log-scale reuse distance histogram.

And the probability for a random data reference in an execution to have reuse distance of k is

$$P_R(k) = \sum_i P(k, b_i) \cdot P_T(b_i).$$

Therefore, the key step in the extension is to compute $P(b_i)$ from a time distance histogram. If we assume that reuse distances are in a uniform distribution inside a bar, $P(b_i)$ can be approximated by $p(\frac{\overleftarrow{b_i}+\overrightarrow{b_i}}{2})$ (where $\overleftarrow{b_i}$ and $\overrightarrow{b_i}$ are the lower and upper bounds of all the reuse distances in the ith bar)[4]. Under this assumption, $P(b_i)$ can be computed by Equation 1 as follows:

$$P(b_i) = \sum_{\tau=1}^{\frac{\overleftarrow{b_i}+\overrightarrow{b_i}}{2}} \sum_{\delta=\tau+1}^{T} \frac{1}{N-1} p_T(\delta).$$

Clearly, the sum of $p_T(\delta)$ can be converted to a sum of $P_T(b_i)$—the time distance histogram.

This basic algorithm has high cost, mainly due to the calculation of $P(b_i)$ and $P(k, b_i)$. The time complexity to compute all of $P(b_i)$s is $O(L_T^3)$ (recall that L_T is the number of bars in a time distance histogram), and the complexity for all $P(k, b_i)$s is a factorial of N. We have to lower the complexity before the algorithm can be applied to real programs.

2.3 Scalable Algorithm

The basic algorithm contains some repetitive computations, especially in the computation of $P(b_i)$. We make two changes to the algorithm to remove the redundant computations and lower the complexity.

The first change is based on a well known fact that a binomial distribution can be approximated by a Normal distribution. For further speedup, we use an offline generated table to store the standard Normal distribution. The computation of $P(k, b_i)$ is reduced to a table-lookup operation with $O(1)$ complexity.

[4] This assumption is also used in the scalable algorithm; its influence to accuracy is evaluated by the experiments in Section 5.

The second change is to decompose the computation of $P(b_i)$ into 3 sub-equations with repetitive computations removed. The 3 sub-equations are as follows:

$$P(b_i) = P_2(b_i)/2 + \sum_{j=0}^{i-1} P_2(b_j) \tag{4}$$

$$P_2(b_i) = [\sum_{j=i+1}^{L_T} P_1(b_j) \frac{\overrightarrow{b_i} - \overleftarrow{b_i}}{\overrightarrow{b_j} - \overleftarrow{b_j}} \sum_{\tau=\overleftarrow{b_j}}^{\overrightarrow{b_j}-1} \frac{1}{\tau - 1}]$$
$$+ P_1(b_i) \frac{1}{\overrightarrow{b_i} - \overleftarrow{b_i}} \sum_{\tau=\overleftarrow{b_i}+1}^{\overrightarrow{b_i}-1} \frac{\tau - \overleftarrow{b_i}}{\tau - 1} \tag{5}$$

$$P_1(b_i) = \frac{\overleftarrow{b_i} + \overrightarrow{b_i} - 3}{2(N-1)} P_T(b_i). \tag{6}$$

The detailed derivation of these sub-equations is too complex to be included in this paper. We give a brief explanation and refer the reader to our technical report [15]. Recall that $P(b_i)$ can be approximated by $p(\frac{\overleftarrow{b_i}+\overrightarrow{b_i}}{2})$—that is, the probability for a variable to be accessed in an interval whose length is $\frac{\overleftarrow{b_i}+\overrightarrow{b_i}}{2}$. This probability can be viewed as the probability for the variable's last access prior to a random time point t to be after $t - \frac{\overleftarrow{b_i}+\overrightarrow{b_i}}{2}$, i.e., to be in the dark segment illustrated in Figure 2. This dark segment can be regarded as a sequence of sub-segments[5], $[t - \frac{\overleftarrow{b_i}+\overrightarrow{b_i}}{2}, t - \overleftarrow{b_i})$, $[t - \overleftarrow{b_i}, t - \overleftarrow{b_{i-1}})$, $\cdots$, $[t - \overleftarrow{b_1}, t)$. We use $P_2(b_j)$ to denote the probability that the variable's last access prior to t occurs in a sub-segment, $[t - \overleftarrow{b_{j-1}}, t - \overleftarrow{b_j})$, which leads to Equation 4.

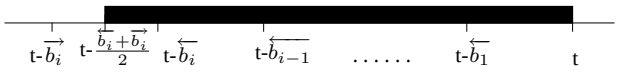


Fig. 2. Illustration of $P(b_i)$.

The probability $P_1(b_i)$ in Equations 5 and 6 is the probability that for a randomly chosen variable v, a random time-point t is in one of v's reuse intervals whose time distance is in range $[\overleftarrow{b_i}, \overrightarrow{b_i})$. Equations 5 and 6 come from a complex mathematical deduction, which examines the different sections of a reuse interval and the different relations between reuse distance and time distance histograms [15].

This extended algorithm enables two simplifications. First, through the following commonly used mathematical approximation (m_1, m_2 and i are positive integers):

$$\sum_{i=m_1}^{m_2} \frac{1}{i} \simeq ln \frac{m_2 + 0.5}{m_1 - 0.5},$$

[5] This work uses logical time: the number of memory references since program starts.

Equation 5 can be simplified to the following form:

$$P_2(b_i) \simeq [\sum_{j=i+1}^{L_T} P_1(b_j) \frac{\overrightarrow{b_i} - \overleftarrow{b_i}}{\overrightarrow{b_j} - \overleftarrow{b_j}} ln \frac{\overrightarrow{b_j} - 1.5}{\overleftarrow{b_j} - 1.5}] + \frac{P_1(b_i)}{\overrightarrow{b_i} - \overleftarrow{b_i}} [\overrightarrow{b_i} - \overleftarrow{b_i} - 2 - (\overleftarrow{b_i} - 1) ln \frac{\overrightarrow{b_i} - 1.5}{\overleftarrow{b_i} - 0.5}]$$

Second, Equation 4 reveals the relation between $P(b_i)$ and $P(b_{i-1})$ as follows:

$$P(b_i) = P(b_{i-1}) + \frac{P_2(b_{i-1}) + P_2(b_i)}{2}.$$

So, given all $P_2(b_i)$s, the time complexity of obtaining all $P(b_i)$s is $O(L_T)$. Apparently, the time complexity of computing all $P_2(b_i)$s is $O({L_T}^2)$. Therefore, the complexity for computing all of $P(b_i)$ is $O({L_T}^2)$. With the simplification to $P(k, b_i)$ by the first change, the time complexity of this extended algorithm is $O({L_T}^2)$, orders of magnitude lower than that of the basic algorithm.

3 Measurement Acceleration for Efficiency

To efficiently implement the approximation model, we use a portable scheme to take advantage of MMU. This scheme accelerates time distance measurement—the bottleneck in the scalable algorithm—by more than a factor of 3.

The measurement of time distance requires detailed recording of memory references. We use a binary instrumentation tool, PIN [11], to insert a function call for recording the memory address after each memory reference. Algorithm 1 shows the basic implementation of the recording procedure. It first stores the accessed memory address into a buffer and then checks if the buffer is full. When it is full, a procedure, *ProcessBuffer()*, is invoked to calculate the time distances in the buffer and record them into a histogram.

Algorithm 1 Basic measurement of time distance

```
Procedure RecordMemAcc (address)
  buffer [index++] ⇐ address;
  if index == BUFFERSIZE then
    ProcessBuffer ( ); // Calculate and record reuse distances and reset index to 0
  end if
end
```

The basic implementation slows down a program's execution significantly. To reduce the overhead, we optimize the procedure *RecordMemAcc( )* by removing the branch and function call in it with the help of MMU. When the instrumented program starts, through MMU, the last page of the array *buffer* is set to be non-writable. When

Algorithm 2 Portable MMU-based optimization for time distance measurement 1

```
Procedure RecordMemAcc (address)
    buffer [index++] ⇐ address;
end

// Procedure to handle memory access fault
// Initially, the last page of buffer is locked
Procedure sig_SEGV_H (siginfo)
    faultAddr ⇐ GetFaultAddress (siginfo);
    if IsInLastPage(faultAddr, buffer) then
        ProcessBuffer ( ); // Calculate and record reuse distances
        OpenLastPage (buffer); // Open the access permission of the last page
        Close2ndToLastPage (buffer); // Close the access permission of the second to last
        page
    else if IsIn2ndToLastPage(faultAddr, buffer) then
        ProcessBuffer ( );
        Open2ndToLastPage (buffer);
        CloseLastPage (buffer);
    else
        ...
    end if
    index = -1; // RecordMemAcc() will make it zero right after the current instruction
    finishes
end
```

the program tries to write an address to that page, a page access violation signal is triggered. In the customized signal handler, the procedure *ProcessBuffer()* is invoked and the buffer index is reset. This optimization removes the necessity for buffer boundary check, reducing procedure *RecordMemAcc( )* to just one statement as shown at the top of Algorithm 2.

Zhao et al. used memory management unit to remove similar branches in detailed execution profile [18]. Our scheme is different in how to resume the execution in the signal handler. The previous work resets the register that contains the buffer index to zero so that after returning from the signal handler, the program can write to the first element of the buffer. The shortcoming of the scheme is the portability problem: The registers that contain the buffer index may differ on different platforms.

We address that problem using a portable scheme, shown by procedure *sig_SEGV_H* in Algorithm 2. Initially we close the permission to access the last page of *buffer*. The signal handler opens the permission of the *last* page and closes the *second to last* page. (The variable for the buffer index is reset.) The execution continues until trying to access the second to last page of *buffer*. The signal handler then opens the second to last page and closes the last page again. By using the last two pages alternatively to signal the end of *buffer*, this scheme removes the necessity for modifying specific register values.

The optimization accelerates time distance measurement by 3.3 times as shown in Section 5. The significant benefits come from three sources. First, it directly reduces the number of instructions and branch miss predictions in the recording procedure. Second, it reduces the overhead of the dynamic instrumentor. The instrumentor, PIN, uses a

virtual machine equipped with a just-in-time compiler for dynamic instrumentation. Fewer instructions in the analysis code leads to fewer computations in the virtual machine. Third, the optimization enables the inlining of the recording procedure. PIN is strict in inlining since the instrumented procedures usually have a large number of call sites. The control flow in the basic implementation prevents PIN from inlining the procedure, causing high calling overhead. Function calls in PIN are especially expensive: At an function call, PIN calls a bridge routine that saves all caller-saved registers, sets up analysis routine arguments, and finally calls the target function [11].

This scheme provides an architecture-independent way to employ MMU for code optimization. It suggests the potential of serving as a general optimization technique for a compiler to apply.

4 Trace Generator for Evaluation

Although this work is on measuring locality from data accesses, this section discusses the reverse problem—how to generate data traces from locality. The initial motivation is for the evaluation of our locality approximation model: Although the model demonstrates high accuracy for SPEC CPU2000 benchmarks, the reuse distance histograms of those programs fall into several categories, covering only a small portion of the entire histogram space. The capability to generate data access traces that satisfy given locality requirements is desirable: We can freely use various traces to evaluate the locality model comprehensively. Although this capability provides conveniences for many locality studies, we are not aware of any prior explorations on it.

This work constructs a stochastic trace generator. Its inputs include a reuse distance histogram P, and the length T and the number of distinct variables N of the trace to be generated; its output is a data reference sequence satisfying the input requirements.

4.1 Algorithm

Algorithm 3 contains the pseudo code of the trace generator. For the purpose of clarity, the following explanation assumes that the bars in the histograms are of unit width.

(a) Reuse distance histogram (with bar width as 1) (b) Cumulative distribution of the histogram in (a)

Fig. 3. A histogram and the accumulative distribution.

The first step is to construct the cumulative distribution of the given reuse distance histogram. The probability for a data access to have reuse distance no longer than i is calculated as $C_i = \sum_{j=0}^{i} P_j$, where, P_j is the probability for a data access to have reuse distance of j—that is, the Y-axis value at reuse distance j in the reuse distance histogram.

The second step is to fill a variable into each position in a T-element trace. The first N positions are simply filled by all the variables. (The order does not matter.) To explain the process of filling the remaining positions, we use the example shown in Figure 3. In the example, there are 8 possible reuse distances 0 to 7. Their cumulative probabilities C_i (i=0,1,. . .,7) separate the range [0, 100%] into 8 segments, $[0, C_0]$, $(C_0, C_1)]$, . . ., $(C_6, C_7]$ (apparently $C_7 = 100\%$), shown on the Y-axis of the cumulative histogram.

Every time, a random number α is generated whose value is between 0 and 1. If α falls into segment $(C_{i-1}, C_i]$, the trace generator uses i as the desired reuse distance of the current position and identifies the corresponding variable and put it into the current position. For example, if the current data trace is ". . . a b c b c" and α is 0.05, the segment that α falls into is $(C_1, C_2]$. Therefore, the reuse distance of the current position should be 2. Because "a" satisfies the distance requirement, an instance of "a" is put into the current position.

Algorithm 3 Algorithm to generate a data trace from a reuse distance histogram

```
Procedure GenTrace (RDH[], N, T, trace[])
  // RDH[]: the given reuse distance histogram;
  // N: the number of distinct data;
  // T: the length of the trace to be generated;
  // trace[]: the array to contain the generated trace;
  BuildCH (RDH, CH); // build cumulative histogram from RDH

  // fill the first N positions
  for i=0 to N-1 do
    trace[i] = var[i];
  end for

  // fill the rest
  for i=N to T do
    a = random();
    s = findSegment (a, CH); // find the segment in CH that contains a
    r = RDH[s];
    trace[i] = findData (trace, r); // find the data having reuse distance of r at the end
    of current trace
  end for
end
```

4.2 Proof of Correctness

This section outlines the proof that the trace generated by Algorithm 3 satisfies the input requirements.

Theorem 1 *In a trace generated by Algorithm 3, the statistical expectation of the number of accesses that have reuse distance of i is ($T * P_i$) (i=0,1,. . .,N-1)—that is, the statistical expectation of the reuse distance histogram of the generated trace is equal to the given histogram. (T: the number of total accesses; P_i: the value of the ith bar in the given reuse distance histogram.)*

To see the correctness of the theorem, notice that the probability for α to fall into the ith segment in the cumulative reuse distance histogram is equal to P_i. This is because α is uniformly distributed between 0 and 1, and the length of the ith segment is P_i. So, if the trace generator follows Algorithm 3, the probability for a generated data to have reuse distance of i equals P_i; the conclusion thus follows. (The proof assumes $T >> N$, which holds for most program executions.)

Finding the data with reuse distance of i takes $O(logN)$ time when we organize the last access to each data in a binary tree. The time complexity of the whole trace generation is therefore $O(TlogN)$.

5 Evaluation

Our evaluation platform is an Intel Xeon 2GHz processor running Fedora Core 3 Linux. We use PIN 3.4 for instrumentation and GCC 3.4.4 (with "-O3") as the compiler. We employ Performance Application Programming Interface (PAPI) [3] to read hardware performance counters.

This section first presents the benefits from the portable MMU scheme in accelerating the measurement of time distance, then reports the efficiency and accuracy of the scalable algorithm on real programs, and finally shows the accuracy of the trace generator, along with its uses in the evaluation of the reuse distance approximation algorithm.

5.1 Time Distance Measurement

Table 1. Optimization benefits for time distance measurement

Prog	Instr. reduct (%)	Branch miss pred reduct (%)	Speedup (%)
gcc	59.7	74.3	225.7
gzip	66.4	2.6	308.0
mcf	66.5	20.9	99.2
twolf	66.8	30.8	267.0
ammp	69.2	6.5	384.1
applu	66.6	15.7	389.4
equake	69.4	3.9	354.9
mesa	67.8	4.7	522.3
mgrid	67.3	2.2	409.6
Average	**66.7**	**18.0**	**328.9**

Table 1 shows the effect of the portable MMU-based optimization on time distance measurement. The 9 benchmarks are randomly chosen from SPEC CPU2000 suite. We use their *train* runs for measurement. The second column in the table contains the reduction of the total instructions executed by the instrumented code. The optimization reduces 60–69% instructions with an average of 66.7%. The Third column shows that the optimization reduces branch miss prediction by 2.2–74.3% with an average of 18%. Together, the two kinds of reduction accelerate the time distance measurement by 99–522% with an average of 329%. We apply this MMU-based optimization to direct measurement of reuse distances (based on Ding and Zhong's method [7], the fastest tool we know), but only see 34.4% average speedup. This relatively modest speedup is because the major bottleneck in reuse distance measurement is in the computation of reuse distances rather than memory monitoring, whereas, the computation of a time distance is trivial—only a single reduction operation.

5.2 Locality Approximation on SPEC *ref* Runs

We use the *ref* runs of the 9 SPEC programs to evaluate the effectiveness of the scalable algorithm for reuse distance histogram approximation. Compared to the *test* and *train* runs used in our previous work [16], the *ref* runs are over 8 times longer, including a wider range of reuse distances, which pose more challenges to measurement efficiency.

We measure the accuracy of the reuse distance approximation on both element and cache-line levels. On the element level, each address is a data item; on the cache-line level, a block of consecutive memory addresses are treated as a single data item (the block size is the width of a cache line).

The accuracy is measured on a linear scale: The width—that is, the range of reuse distance—of each bar in the histograms is 1000. The formula to calculate the accuracy is $(1-\sum_i |R_i - \widehat{R_i}|/2)$, where, R_i is the Y-axis value of the ith bar in a real histogram, and $\widehat{R_i}$ is for the approximated histogram. The division by 2 normalizes the accuracy to [0,1].

As shown in Table 2, the accuracy for element reuse is 82.8% on average. Benchmark *mcf* has the lowest accuracy, 42.6%, due to the unusual thin peaks on its reuse distance histogram. The accuracy on the cache-line level is much higher, 94% for *mcf*, 98.6% on average. This higher accuracy is because a data item on this level becomes larger, and results in a smaller range of reuse distances. Most local peaks in the histograms are therefore smoothed out. Because most uses of reuse distance requires cache-line level information (e.g., for cache performance analysis), the occasional poor accuracy on element reuses has little effect. Compared to the latest reuse distance measurement [7] (with the MMU-based optimization), the approximation technique improves the speed by a factor of 17.

5.3 Trace Generator

We evaluate the trace generator on reuse distance histograms of some Normal and exponential distributions, two kinds of distributions that are close to typical data reuses. In the exponential distribution, the height of a bar at reuse distance of k is proportional

Table 2. Accuracy and speedup of reuse distance approximation

Prog	Element		Cache line	
	acc. (%)	speedup	acc. (%)	speedup
gcc	89.0	21.2X	99.4	16.7X
gzip	99.0	19.0X	99.5	17.0X
mcf	42.6	8.3X	94.0	18.2X
twolf	88.2	5.9X	98.1	20.2X
ammp	95.8	14.3X	99.2	21.5X
applu	86.1	19.0X	99.2	21.4X
equake	57.6	23.7X	98.5	15.1X
mesa	97.3	26.3X	100	14.0X
mgrid	89.7	20.6X	99.6	21.5X
Average	**82.8**	**17.6X**	**98.6**	**18.4X**

to $e^{-0.02*k}$. In the Normal distributions, we change the variance from 20 to 200; the histograms change from a shape with thin high peaks to a flatter shape as illustrated by Figure 4. The figure also shows the histograms of the generated traces, whose curves fluctuate around the given histograms. In our experiments, each trace contains 50,000 references to 500 variables. To prevent histogram bins from hiding inaccuracy, we let each bin in the histograms have width of 1.

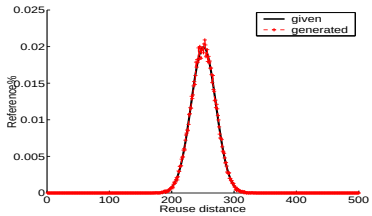
The second column in Table 3 reports the accuracy of the trace generator, reflecting the difference between the generated and the given histograms. All accuracies are greater than 96%, demonstrating the effectiveness of the trace generator in meeting the input requirements. The last column of the table contains the accuracy of the histograms that are approximated by the scalable algorithm presented in Section 2.3. On average, the accuracy is 95.5%, demonstrating the effectiveness of the algorithm in approximating reuse distance histograms of different distributions.

Table 3. Accuracy of trace generation and reuse distance approximation

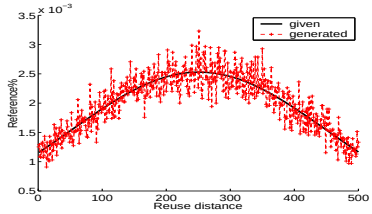
Distr.	Gen. acc. (%)	Approx. acc. (%)
Normal (var=20)	98.2	92.8
Normal (var=100)	96.4	96.3
Normal (var=200)	96.1	95.8
Exponential	96.0	96.9
Average	**96.7**	**95.5**

6 Related Work

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 between loop nests [14]. Cascaval presented a compiler algorithm that



(a) mean=250, var=20



(b) mean=250, var=200

Fig. 4. Reuse distance histograms of two Normal distributions, along with those of the generated traces.

measures reuse distance directly [4]. Allen and Kennedy discussed the subject comprehensively in their book [1]. Thabit identified data often used together based on their access frequency [17]. Chilimbi used grammar compression to find hot data streams and reorganized data accordingly [5].

As a locality model, reuse distance has been studied for several decades since Mattson et al.'s first measurement algorithm [13]. A more recent work is from Ding and Zhong, who proposed an approximation algorithm that used dynamic tree compression [7] to reduce time complexity to $O(TloglogN)$. Shen et al. discovered the statistical connection between time distance and reuse distance, and proposed an algorithm to approximate reuse distance from time distance [16]. This current paper exposes the extensions to the algorithm to make it more scalable, meanwhile presenting a portable scheme for resolving the bottleneck in the implementation of the algorithm, and describing a trace generator to facilitate the evaluation.

7 Conclusions

This paper presents a set of techniques to efficiently approximate reuse distance histograms on arbitrary scales. It describes an extended algorithm that significantly reduces the time complexity of the basic algorithm. It exposes a portable MMU-based optimization that accelerates time distance measurement by more than a factor of 3. Finally, it presents a trace generator that facilitates the comprehensive evaluation of the approximation algorithm. The output from this work will enhance the applicability of reuse distance in practical uses, opening new opportunities for program analysis and optimizations.

Acknowledgment. We thank the anonymous reviewers for all the helpful comments. Chen Ding suggested the lock switching scheme in the memory controller component; Brian Meeker collected some preliminary data in the early stage of this research. This material is based upon work supported by the National Science Foundation under Grant No. 0720499. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, October 2001.
2. K. Beyls and E.H. D'Hollander. Reuse distance-based cache hint selection. In *Proceedings of the 8th International Euro-Par Conference*, Paderborn, Germany, August 2002.
3. S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of Supercomputing*, 2000.
4. G. C. Cascaval. *Compile-time Performance Prediction of Scientific Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 2000.
5. 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.
6. C. Ding. *Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse*. PhD thesis, Dept. of Computer Science, Rice University, January 2000.
7. 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.
8. 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.
9. 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.
10. 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.
11. C-K Luk et al. 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.
12. 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.
13. R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.
14. 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.

15. X. Shen, J. Shaw, and B. Meeker. Accurate approximation of locality from time distance histograms. Technical Report TR902, Computer Science Department, University of Rochester, 2006.
16. Xipeng Shen, Jonathan Shaw, Brian Meeker, and Chen Ding. Locality approximation using time. In *Proceedings of the ACM SIGPLAN Conference on Priciples of Programming Languages*, 2007. (short paper, 7 pages).
17. K. O. Thabit. *Cache Management by the Compiler*. PhD thesis, Dept. of Computer Science, Rice University, 1981.
18. Qin Zhao, Joon Edward Sim, Weng-Fai Wong, and Larry Rudolph. DEP: detailed execution profile. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2006.
19. Y Zhong, S. G. Dropsho, X. Shen, A. Studer, and C. Ding. Miss rate prediction across program inputs and cache configurations. *IEEE Transactions on Computers*, 56(3), 2007.
20. 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.