Locality Approximation Using Time

Xipeng Shen
Computer Science Department
The College of William and Mary
xshen@cs.wm.edu

Jonathan Shaw
Shaw Technologies
jshaw@cs.rochester.edu

Brian Meeker Chen Ding
Computer Science Department
University of Rochester
{bmeeker,cding}@cs.rocheester.edu

Abstract

Reuse distance (i.e. LRU stack distance) precisely characterizes program locality and has been a basic tool for memory system research since the 1970s. However, the high cost of measuring has restricted 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 that converts cheaply obtained time distance to the more costly reuse distance. Compared to the state-of-the-art technique, this approach reduces measuring time by a factor of 17, and approximates cache line reuses with over 99% accuracy and the cache miss rate with less than 0.4% average error for 12 SPEC 2000 integer and floating-point benchmarks. By exploiting the strong correlations between time and locality, this work makes precise locality as easy to obtain as data access frequency, and opens new opportunities for program optimizations.

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

General Terms Algorithms, Measurement, Performance

Keywords Time Distance, Program Locality, Reuse Distance, Reference Affinity, Trace Generator, Performance Prediction

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.

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 [11]. As an example, in the 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 provides an architecture-independent locality metric, precisely capturing program temporal locality and reflecting memory reference affinity [26]. A *Reuse distance histogram*, illustrated in Figure 1, summarizes the distribution of the reuse distances in an execution. In the graph, the seventh bar, for instance, shows that 25% of total memory accesses have reuse distance in range [32, 64).

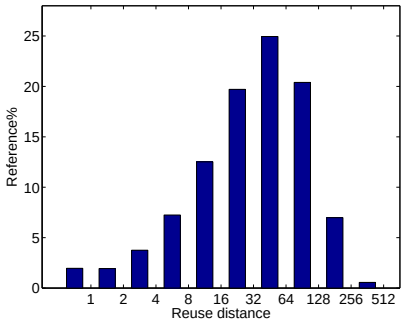


Figure 1. A reuse distance histogram on log scale.

Researchers have used reuse distance (mostly its histogram) for many purposes: to study the limit of register [15] and cache reuse [10, 13], to evaluate program transformations [1, 4, 25], to predict performance [17], to insert cache hints [5], to identify critical instructions [12], to model reference affinity [26], to detect locality phases [22], to manage superpages [7], and to model cache sharing between parallel processes [8].

Because of the importance, the last decades 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. Later studies—e.g. Bennett and Kruskal in 1975 [3], Olken in 1981 [19], Kim et al. in 1991 [14], Sugumar and Abraham in 1993 [24], Almasi et al. in 2002 [1], Ding and Zhong in 2003 [11]—have reduced the cost through various data structures and algorithms.

Despite those efforts, the state-of-the-art measurement technique still slows down a program's execution up to hundreds of times: The measurement of a 1-minute execution takes more than 4 hours. The high cost impedes the practical uses in performance debugging, locality analysis, and optimizations of long-running applications.

All previous algorithms have essentially implemented the definition of reuse distance—"counting" the number of distinct data accessed for each reuse. In this work, we address the problem from a different aspect: *Can we use some easily obtained program behavior to statistically approximate reuse distance?* The behavior we choose is **time distance**, which 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.

POPL'07 January 17–19, 2007, Nice, France.

As what people commonly conceived, time distance itself cannot serve as an accurate locality model. In access trace "*a b b b b a*", for example, the time distance of the second access to variable *a* is 5, which could correspond to 5 different reuse distances, from 0 to 4, if no other information is given. However, if we know the time distance histogram—among four reuses, one has time distance of 5 and three have time distance of 1, we can easily determine the trace given the number of variables and thus obtain the reuse distance. Although it's not always possible to derive a unique trace from time distances, this work discovers that a time distance histogram contains enough information to accurately approximate the reuse distance histogram.

We describe a novel statistical model that takes a time distance histogram to estimate the reuse distance histogram in three steps by calculating the following probabilities: the probability for a time point to fall into a *reuse interval* (an interval with accesses to the same data at both ends and without accesses to that data element in between) of any given length, the probability for a data element to appear in a time interval of any given length, and the binomial distribution of the number of distinct data in a time interval.

The new model has two important advantages over previous precise methods. First, the model predicts the reuse distance histogram for bars of any width, which previous methods cannot do unless they store a histogram as large as the size of program data. The second is a drastic reduction of measuring costs with little loss of accuracy. Our current implementation produces over 99% accuracy for reuse distance histogram approximation, providing a factor of 17 speedup on average compared to the fastest precise method.

2. Approximation of Locality

The inputs to our model are the number of distinct data accessed in an execution and the time distance histogram of the execution; the output of the model is the reuse distance histogram which characterizes the locality of that execution. To ease the explanation, we assume that the size of bars in both histograms is 1 and the histograms are of data *element* reuses. Section 2.4 describes the extensions for histograms of any size bars and for any size cache blocks.

We use the following notations:

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

$M(v)$: the total number of accesses to data element v.

N : the total number of distinct data elements in an execution.

T : the length of an execution. Without explicitly saying so, we use logical time, i.e. the number of data accesses. Each point of time corresponds to a data access.

$T_n(v)$: the time of the n'th access to data v.

$T_{>t}(v)$, $T_{<t}(v)$: the time of v's first access after time t and the time of its last access before t respectively.

v_t : the data element accessed at time t.

The algorithm includes three steps, outlined below in reverse order for the purpose of clarity. Please see our technical report [20, 21] for more details.

2.1 Step 3: Estimate Reuse Distance

The last step is based on the following observation:

OBSERVATION 2.1. *Given a program's execution, we can approximate the reuse distance histogram if we know the time distance histogram of the execution and the probability for any given data to appear in any given time interval of length Δ, represented by $P_3(\Delta)$*[1].

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

The approximation of reuse distance through $P_3(\Delta)$ 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 a *binomial distribution*, denoted by $f(k; n, p)$

$$f(k; n, p) = \left(\begin{array}{c} n \\ k \end{array} \right) p^k (1-p)^{n-k}.$$

Assuming the probability of any data to be accessed in an interval is independent from other data, data accesses can be considered as 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 elements in the program. The number of times showing heads is the number of distinct data elements 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:

$$P_R(k) = \sum_{\Delta} P(k, \Delta) \cdot P_T(\Delta) \tag{1}$$

where,

$$P(k, \Delta) = \left(\begin{array}{c} N \\ k \end{array} \right) P_3(\Delta)^k (1 - P_3(\Delta))^{N-k}$$

$P_R(k)$ and $P_T(\Delta)$ respectively denote the Y-axis value of a bar in reuse distance histograms at position of k and time distance histograms at position of Δ.

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

This step is to calculate $P_3(\Delta)$, the probability of any given data element 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 τ = Δ, ..., 2, 1. A given data being accessed in time range $[t - \Delta, t)$ means that its last access time before t is at time $t - 1$, or $t - 2$, or, ..., or $t - \Delta$. Let $P_2(\tau)$ be the probability for the access to happen at time $t - \tau$. We calculate $P_3(\Delta)$ as follows:

$$P_3(\Delta) = \sum_{\tau=1}^{\Delta} P_2(\tau) \tag{2}$$

In a real execution, the value of $P_2(\tau)$ may differ at different times and for different data. But for statistical inference, we want the average probability, which is calculated as follows.

[1] The formal definition of $P_3(\Delta)$ is as follows: 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 $t - \Delta - 1$.

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

Mathematical inferences [21] produce the following result:

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

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

$$
P_2(\tau)=\sum_{\delta=\tau+1}^{T}\frac{P_1(\delta)}{\delta-1} \tag{3}
$$

The problem is now reduced to that of calculating $P_1(\delta)$.

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

$P_1(\delta)$ is the probability for a randomly chosen time t to fall into a δ-long reuse interval of a data element that is randomly chosen from the data elements that are not accessed at t. It has the following relation with time distance histogram $P_T(\delta)$ [21]:

$$
P_1(\delta)=\frac{\delta-1}{N-1}P_T(\delta) \tag{4}
$$

This concludes the basic model for the approximation of a reuse distance histogram. Putting equations (2, 3, 4) together produces the following formula for $P_3(\Delta)$:

$$
P_3(\Delta)=\sum_{\tau=1}^{\Delta}\sum_{\delta=\tau+1}^{T}\frac{1}{N-1}P_T(\delta)
$$

Although it seems simple, the formula is hard to interprete intuitively. When the bars are wider than 1, the corresponding formula becomes very complex [20]. The three steps above give not only an intuitive explanation of the model, but also a clear guide for implementation.

2.4 Extensions and Implementation

In the above description, we assumed that each bar's width in both time and reuse distance histograms is 1. With some extensions, our model is general enough to allow bars of any width, and hence histograms on any scale (e.g. linear, log or random scale.) For lack of space, please see [20] for the extensions.

In our implementation, we remove the redundant calculations by reordering some computations. We use a look-up table generated offline to minimize the computation of binomial probabilities. A boundary case is the treatment of the first access (i.e. the cold access) to a variable. We use the variable's circular time distance, which is the sum of the time distance from the starting of the program to its first access and the distance from its last access to the end of the execution. We tried other options such as ignoring the first accesses and assigning a large distance value to them. The circular time distance scheme shows the best effect.

3. Evaluation

This section presents experimental results on both generated traces from a trace generator [20] and 12 real programs in SPEC CPU2000 suite (Table 2). We use the generated traces to test the approximation model on histograms of different distributions. We use the SPEC programs to measure the efficiency and accuracy in real uses. 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 [16] for instrumentation with GCC version 3.4.4 as our compiler (with the "-O3" flag). We use Ding and Zhong's technique to measure the real reuse distance histograms [11]. It is asymptotically the fastest tool for measuring reuse distance at a guaranteed precision[2].

3.1 Results on Generated Traces

Using the trace generator, we generate traces of different reuse distributions. Figure 2(a)(b) show the reuse distance histograms of pulse-like and exponential distributions.

In the experiment, we first measure the time distance histogram (TDH) of the generated trace. We then apply the statistical model to TDH to approximate the reuse distance histogram ($\widehat{RDH}$) of the trace. After measuring the real reuse distance histogram (RDH) of the generated trace, we calculate the accuracy of the approximation as follows:

$$
accuracy = 1-\frac{\sum_i |B_i-\widehat{B_i}|}{2} \tag{5}
$$

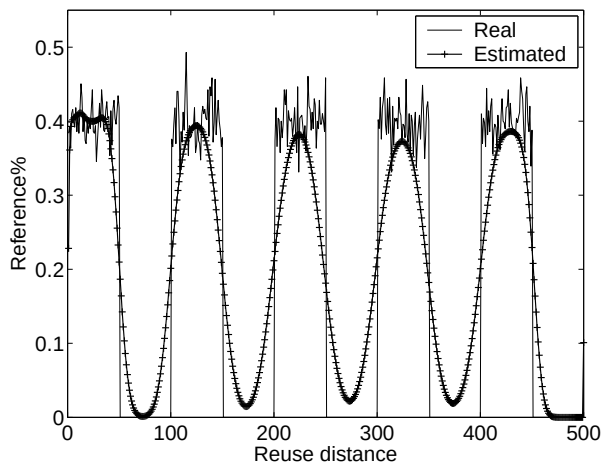
Where, B_i is the Y-axis value of the i'th bar in RDH and $\widehat{B_i}$ is the one in $\widehat{RDH}$. The division by 2 is to normalize the accuracy to [0,1]. In the experiments on generated traces, we make the bars of both histograms as wide as 1 so that we may observe errors that would otherwise be hidden by a larger bar size.

The three graphs on the left column of Figure 2 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 2 (e) shows the log-scale reuse distance histograms with approximation accuracy of 98.4%. Figure 2(c) is the time distance curve, which has 5 smoothly fluctuating waves decreasing gradually to 0. What the graph does not show for lack of space is a thin peak at the end of the curve, distance 50000, with 0.001% references, which is due to the limited number of counters and the time distance of cold accesses (Section 2.4).

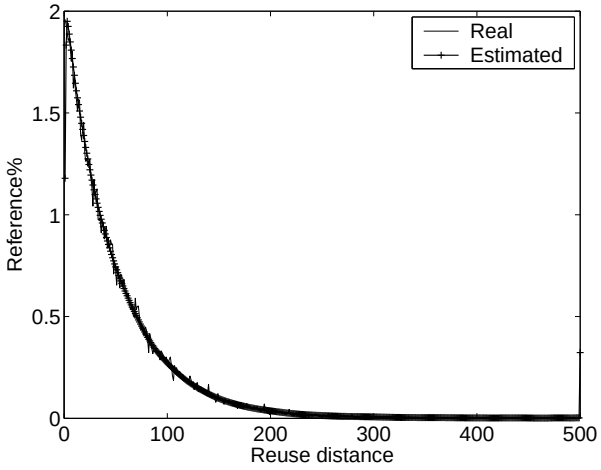
The right column of Figure 2 shows the results for a trace whose reuse distance histogram exhibits an exponential distribution. The estimated curve matches the real curve so well that they cannot be distinguished in Figure 2(b). Figure 2(d) shows the time distance curve, which is a long tail curve. Figure 2(f) gives the log-scale reuse distance histograms with accuracy 98.7%. We see similar results on random distribution histograms [21].

Note, in the generated traces, the data element to be accessed at a time point depends on all the prior accesses (including the accesses to other data elements.) Therefore, different data elements can have different reuse distributions. In our model, we give all data elements the same probability—the average probability, which may hurt the prediction of the reuses of a data element, but is effective for the reuses of a whole program.

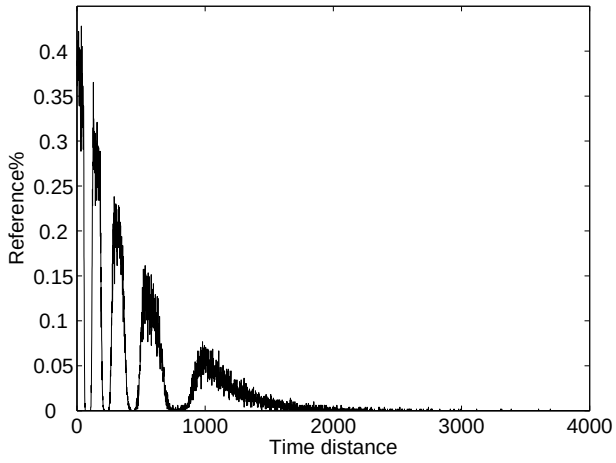
[2] We set the tool to guarantee 99.9% accuracy.



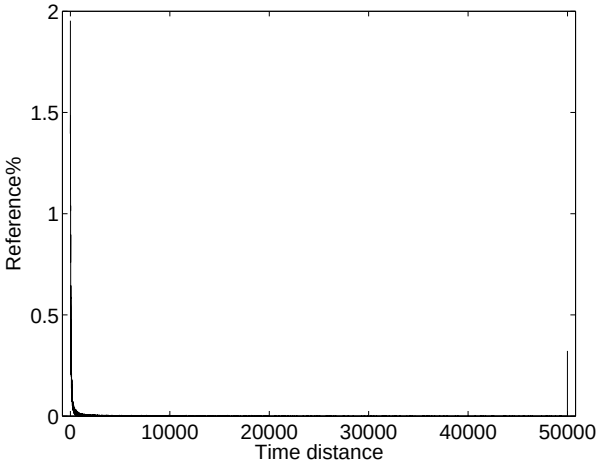
(a) Reuse distance histogram of a pulse distribution



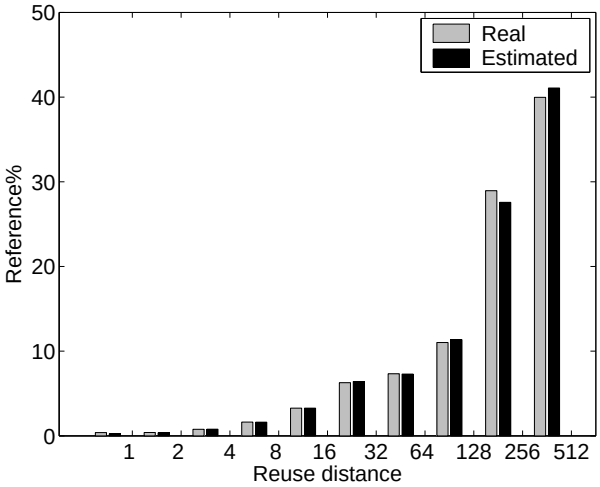
(b) Reuse distance histogram of an exponential distribution



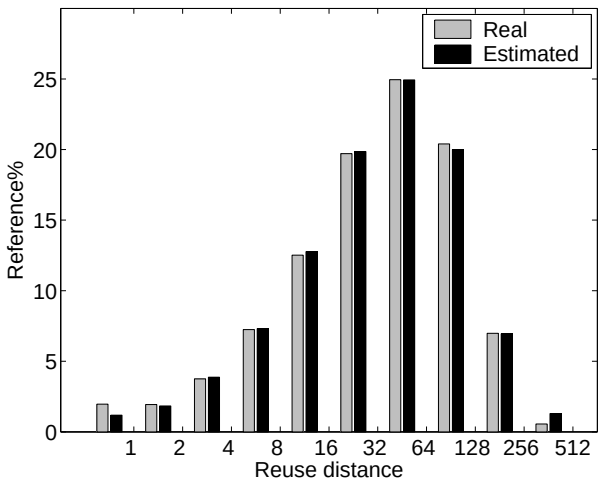
(c) Time distance histogram of a pulse distribution



(d) Time distance histogram of an exponential distribution



(e) Log-scale reuse distance histogram of a pulse distribution



(f) Log-scale reuse distance hist. of an exp. distribution

Figure 2. Comparisons of the actual reuse distance histograms and the histograms estimated from the time distance histograms. Different distributions and bar sizes are shown.

3.2 Results on SPEC CPU2000

We apply the statistical model to 12 SPEC CPU2000 benchmarks. In all the experiments, the approximation uses 1K-wide bars in both the measured time distance histograms and the approximated reuse distance histograms. The log-scale histograms are derived from the linear-scale ones.

3.2.1 Comparison of Time Cost

Table 1 shows the time cost of the statistical model compared with the previous fastest method [11] for 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, including 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. In order to avoid the effects of different instrumentors, we subtract the instrumentation overhead from both kinds of measured time 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 make 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 3 shows the reuse distance histograms of element reuse and cache line reuse. The distance range shrinks from 16M to 512K.

3.2.2 Approximation Accuracy

Table 2 shows the accuracy of the reuse distance approximation on both element and cache line level for the test and train runs of the SPEC CPU2000 benchmarks. A bar in 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 5.

The approximation accuracy for cache line reuses is 99.3% and 99.4% for linear and log scale respectively. The lowest accuracy is 96.5% and 96.6% on benchmark *ammp*. 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. Figure 3 shows the histograms of *mcf* reuses. The largest error of element reuse approximation happens 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 of the statistical model: we assume that the probability for a variable to appear in an interval is independent of the other variables. However, a larger granularity removes the error almost completely, as shown in the result of cache line reuses in Figure 3 (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 17 times speedup, over 99% accuracy for cache line reuses, and 94% for element reuses.

Uses for Cache Miss Rate Prediction Previous work has shown the uses of reuse distance histograms in cache miss rate prediction [25]. In this experiment, we compare the predicted cache miss rates from the actual and the approximated reuse distance histograms to test the effectiveness of the statistical model in real uses.

Among all 12 benchmarks, the largest errors are 2.5% for *twolf* and 1.4% for *equake*, which is consistent with 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 only 0.34% miss rate estimation error. That is because of its excellent cache line reuse estimation, the basis for cache performance prediction. On average for all benchmarks, the miss rates predicted from the actual and the approximated histograms have less than 0.42% differences [21].

4. Related Work

This section discusses some related work that are not mentioned in Section 1. In 1976, Smith gave a probability method for predicting the miss rate of set-associative cache [23]. To predict the effect of cache sharing, Chandra et al. proposed models to predict inter-thread cache contention from isolated cache stack distance [8]. Our method complements these techniques—by combining them one can predict the (shared) performance of set-associative cache solely based on time distance histograms.

An orthogonal way of reducing the measuring cost is using a sampled trace rather than the full trace [2, 6, 9]. The combination of sampling and this work has the potential to make reuse distance applicable to run-time optimizations.

5. Conclusions

In this work, we demonstrate the strong connection between time and locality with a novel statistical model to approximate program locality from easily-obtained time distance histograms. The 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 new levels of efficiency and generality open up opportunities for performance prediction, debugging, and optimizations.

6. Acknowledgments

We thank Hans Boehm for his insightful comments and valuable suggestions on this paper. Yutao Zhong, Kristof Beyls, and the anonymous referees also helped improve the presentation. This research is supported in part by grants from The College of William and Mary and NSF grants CNS-0509270 and CCR-0238176. Any opinion, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] 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.

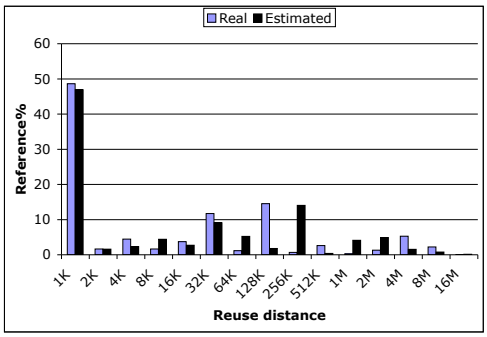
[2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.

Benchmark		Instrument overhead	Data element level				Cache line level			
			Reuse dist. measure	Time dist. measure	Time to approximate	Speedup times	Reuse dist. measure	Time dist. measure	Time to approximate	Speedup times
		(sec.)	(sec.)	(sec.)	(sec.)		(sec.)	(sec.)	(sec.)	
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**

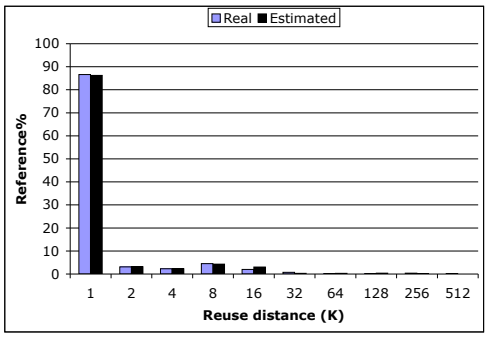
Table 1. Comparison of the time of reuse distance approximation and measurement

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

[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. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 51(4):223–250, 2005.

[6] K. Beyls and E. D'Hollander. Discovery of locality-improving refactoring by reuse path analysis. In *Proceedings of HPCC. Springer. Lecture Notes in Computer Science Vol. 4208*, pages 220–229, 2006.

[7] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski. Multiple page size modeling and optimization. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, St. Louis, MO, 2005.

[8] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2005.

[9] 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, June 2002.

[10] C. Ding and M. Orlovich. The potential of computation regrouping for improving locality. In *Proceedings of SC2004 High Performance Computing, Networking, and Storage Conference*, Pittsburgh, PA, November 2004.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[20] 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.

[21] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. Technical Report TR901, Computer Science Department, University of Rochester, 2006.

[22] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the Eleventh International Conference on Architect ural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Boston, MA, 2004.

[23] 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.

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

[25] 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*.

[26] 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.