Reuse-distance-based Miss-rate Prediction on a Per Instruction Basis

Changpeng Fang
cfang@mtu.edu

Steve Carr
carr@mtu.edu

Soner Önder
soner@mtu.edu

Zhenlin Wang
zlwang@mtu.edu

Department of Computer Science
Michigan Technological University
Houghton MI 49931-1295 USA

ABSTRACT

Feedback-directed optimization has become an increasingly important tool in designing and building optimizing compilers. Recently, reuse-distance analysis has shown much promise in predicting the memory behavior of programs over a wide range of data sizes. Reuse-distance analysis predicts program locality by experimentally determining locality properties as a function of the data size of a program, allowing accurate locality analysis when the program's data size changes.

Prior work has established the effectiveness of reuse distance analysis in predicting whole-program locality and miss rates. In this paper, we show that reuse distance can also effectively predict locality and miss rates on a per instruction basis. Rather than predict locality by analyzing reuse distances for memory addresses alone, we relate those addresses to particular static memory operations and predict the locality of each instruction.

Our experiments show that using reuse distance without cache simulation to predict miss rates of instructions is superior to using cache simulations on a single representative data set to predict miss rates on various data sizes. In addition, our analysis allows us to identify the critical memory operations that are likely to produce a significant number of cache misses for a given data size. With this information, compilers can target cache optimization specifically to the instructions that can benefit from such optimizations most.

Categories and Subject Descriptors

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

General Terms

Algorithms, Measurement, Languages

Keywords

Data locality, reuse distance, profiling

1. INTRODUCTION

With the widening gap between processor and memory speeds, program performance has become increasingly dependent upon the effective use of a machine's memory hierarchy. To improve locality in programs, compilers have traditionally used either static analysis of regular array references [8, 13] or profiling [1] to determine the locality of memory operations. Static analysis has limited applicability when index arrays are used in addressing and when determining locality across multiple loop nests. Profiling can measure locality for a larger set of references, but the results may be specific to the profiled data set. This may lead to inaccurate analysis when the input set is changed.

Recently, Ding, et al. [3, 14], have proposed techniques to predict the reuse distance of memory references across all program inputs using a few profiling runs. They use curve fitting to predict reuse distance (the number of distinct memory locations accessed between two references to the same memory location) as a function of a program's data size. By quantifying reuse as a function of data size, the information obtained via a few profiled runs allow the prediction of reuse to be quite accurate over varied data sizes. Ding, et al., have used reuse-distance predictions to accurately predict whole program miss rates [14].

In order to use reuse distance in program optimization, it is beneficial to associate reuse distance prediction with the actual instructions that access those locations. By predicting the reuse distance for individual instructions, we can identify those instructions which may be most amenable to cache optimization. In this paper, we extend the work of Ding, et al., to apply reuse-distance prediction on a per instruction basis. We examine the reuse distance of the data referenced by each memory operation and use that information to determine the reuse distance of the operation as a function of the data size. We show experimentally that the reuse distance of static instructions is highly predictable. Furthermore, both the L1 and L2 miss rates for instructions can be predicted accurately using reuse-distance-based analysis.

Reuse-distance analysis opens up new possibilities for compiler and architectural optimization of memory operations. The dynamic analysis information provides the compiler with additional information that is impossible to compute at compile time, allowing the compiler to have more knowledge concerning the effects of optimization. Reuse-distance analysis may also be used in micro-architectural optimization via compiler hints to gain a more global view of the expected behavioral patterns of a program.

The most obvious application of reuse distance is to prefetching those memory operations that cause the most misses. Both hardware and software prefetching may issue many unnecessary prefetches. By predicting the miss rate of instructions, we can identify those instructions that are likely to provide the most benefit via prefetching and eliminate misses through dynamic or feedback-directed optimization. In addition, hardware could be constructed to use reuse-distance information to schedule prefetches dynamically for important instructions.

We begin the rest of this paper with a review of work related to static and dynamic locality analysis. Next, we describe our analysis techniques and algorithms for measuring instruction-based reuse distance. Then, we present our experiments examining the predictability of instruction-based reuse distance and finish with our intentions for application of this analysis.

2. RELATED WORK

The previous work most pertinent to this paper consists of the two recent papers by Ding, et al. [3, 14]. They present a technique to predict reuse patterns of the whole program based on a couple of training runs on small inputs [3]. A follow-up work uses the predicted reuse distances to estimate the capacity miss rates of a fully associative cache [14]. We describe their techniques in detail in Section 3.1. In contrast, our work focuses on the prediction of reuse patterns and miss rates for each instruction. We expect to use this prediction to capture *critical instructions*, which generate dominant misses for a program.

Typically we can rely on cache simulation or an analytical model to analyze program cache behavior. Cache simulation can supply accurate miss rates and even performance impact for a cache configuration. The simulation itself is costly, making it less adaptive to system changes and impossible to apply during dynamic optimization on the fly. Various techniques have been discussed to make cache simulation more effective. Mattson, et al., present a stack algorithm to measure cache misses for different cache sizes in one run [6]. Sugumar and Abraham [12] use Belady's algorithm to characterize capacity and conflict misses. They present three techniques for fast simulation of optimal cache replacement. We find using the reuse-distance pattern to predict cache misses for other inputs yields better accuracy than applying cache simulation on a single test input.

A simple analytical model uses dependence and locality analysis to detect data reuse and estimate cache misses [5, 10]. McKinley, et al., design such a model to group reuses and estimate cache miss costs for loop transformations [8]. McKinley and Temam quantify loop nest locality related to self or group, spatial or temporal reuses [9]. Wolf and Lam use an approach based upon uniformly generated sets to analyze locality. Their technique produces similar results to that of McKinley, but does not require the storage of input dependences [13]. Beyls and D'Hollander detect reuse patterns through profiling and generate hints for the Itanium processor [1]. It's unclear whether their profiling and experiments are on the same input or not. However, our work can be directly applied to generate their hints.

Both the dependence based and uniformly generated set based locality analysis provide a high-level view of data reuse and data locality, but may generate poor accuracy, especially due to the lack of consideration of cache interference. For better cache interference analysis, Ghosh, et al. [4], suggest a set of miss equations for precisely analyzing cache misses for individual loop nests. Their framework enables compiler algorithms to find optimal solutions for transformations like blocking, loop fusion, and padding. Cache miss equations allow the compiler to reason about how different transformations work together. Chatterjee, et al. [2], set up a set of Presburger formulas to characterize and count cache misses. Chatterjee's model is powerful enough to handle imperfectly nested loops and various non-linear array layouts. Compared to cache simulation, these analytical models can capture high-level reuse patterns but may be hard to use for dynamic optimization due to lack of run-time knowledge such as loop bounds. Both the models of Ghosh and of Chatterjee sacrifice prediction efficiency for high accuracy.

3. REUSE-DISTANCE ANALYSIS

We begin this section with a description of *reuse distance* and whole-program locality analysis. Reuse distance provides the foundation for our analysis. We then describe per instruction reuse-distance and miss-rate analysis in detail.

3.1 Reuse Distance

The *reuse distance* of a reference is defined as the number of distinct memory references between itself and its reuse. In this paper, we ignore the impact of spatial locality by treating each cache line (block) as a unit, resulting in the measurement of reuse on a per cache line basis. *Backward* reuse distance is the gap between a particular memory reference and its previous access. Similarly, *forward* reuse distance quantifies the gap between a particular reference and the next access to the same location.

Previous work represents the whole program as a histogram describing reuse distance distribution, where each bar consists of the portion of memory references whose reuse distance falls into the same range [3]. Ding, et al., investigate dividing the consecutive ranges linearly, logarithmically, or simply by making the number of references in a range a fixed portion of total references. Our work uses logarithmic division where we put reuse distances between 2^k and 2^{k+1} into the same range. These bounds on the ranges are related to increments in cache size.

Ding, et al., define the *data size* of an input as the largest reuse distance. Given two histograms with different data sizes, they find the locality patterns of a third data size are predictable in a selected set of benchmarks. The pattern recognition step generates the histogram for the third input. Typically, one can use this method to predict locality patterns for a large data input of a program based on training runs of a pair of small inputs.

Let d_i^1 be the distance of the i^{th} bin in the first histogram and d_i^2 be that in the second histogram. Assuming that s_1 and s_2 are the data sizes of two training inputs, we can fit the reuse distances through two coefficients, c_i and e_i, and a function f_i as follows.

$$d_i^1 = c_i + e_i * f_i(s_1)$$
$$d_i^2 = c_i + e_i * f_i(s_2)$$

When the function f_i is fixed, c_i and e_i can be calculated and the equation can be applied to another data size to predict reuse distance distribution. Ding, et al., try several types of fitting functions, such as linear or square root, and choose the best fit.

3.2 Reuse Distance Prediction

Although we know that the reuse distance distribution of the whole program is predictable, it is unclear whether the reuse distances of an instruction show the same predictability. Moreover, the reuse pattern for each instruction can serve as better information for both static and dynamic optimizations. In this section, we discuss our methods to predict instruction-based reuse patterns.

To apply per instruction reuse distance and miss rate prediction on the fly, it is critical to represent the locality patterns of training runs as simply as possible without sacrificing much prediction accuracy. For the training runs, we collect the reuse distances of each instruction and store the number of instances for each of the power of 2 intervals. We also record the minimum, maximum, and mean distance within each interval. An interval is *active* if there exists an occurrence of reuse in the interval. Our results in Section 4 show that most instructions contain just one or two active intervals. We note that at most 6 words of information are needed for most instructions in order to track their reuse intervals.

The *ad hoc* use of power of 2 as bounds for intervals may split the locality pattern of certain reuses across the interval bounds. We solve this problem through an additional scan of original intervals and merge any pair of adjacent intervals i and $i+1$ if

$$min_{i+1} - max_i \leq max_i - min_i.$$

This inequality is true if the difference between the maximum distance in interval $i+1$ and the minimum distance in interval i is no greater than the length of interval i. The merging process continues by testing whether the newly created interval should be merged with the next interval. We observe that this additional pass reflects the locality patterns of each instruction and notably improves prediction accuracy.

Following the prediction model discussed in Section 3.1, the reuse distance patterns of each instruction for a third input can be predicted through two training runs. For each instruction, we predict its i^{th} interval by fitting the i^{th} interval in each of the training runs. The fitting function is then used to solve the minimum, maximum, and mean distances of the current interval. Note that this prediction is simple and fast, making it a good candidate for inclusion in adaptive compilation.

For reuse distance prediction, we compute both the prediction coverage and the prediction accuracy. Prediction coverage indicates the percentage of instructions whose reuse distance distribution can be predicted. An instruction's reuse distance distribution can be predicted if and only if it occurs in both of the training runs and all of its reuse distance patterns are regular. A pattern is said to be *regular* if the pattern occurs in both training runs and its reuse distance does not decrease in the larger input size. Irregular patterns rarely occur in our test programs. Prediction accuracy indicates the percentage of covered instructions whose reuse distance distribution is correctly predicted by our model.

An instruction's reuse distance distribution is said to be correctly predicted if and only if all of its patterns are correctly predicted. In the experiments, we cross-validate this prediction by comparing the predicted locality intervals with the collected intervals through a real run. The prediction is said to be *correct* if both the predicted and observed patterns fall in the same 2^k interval or if the predicted intervals and the observed intervals overlap by at least 90%. Given two patterns A and B such that $B.min < A.max \leq B.max$, we say that A and B overlap by at least 90% if

$$\frac{A.max - \max(A.min, B.min)}{\max(B.max - B.min, A.max - A.min)} \geq 0.9.$$

3.3 Miss Rate Prediction

This section describes miss rate prediction for each instruction. We first describe our general methodology for miss rate prediction given a single cache. Then, we apply our model to multi-level, set-associative caches.

Given the size of a fully associative cache, we predict a cache miss for the next reference to a particular cache line if its forward reuse distance is greater than the cache size. This model catches the compulsory and capacity misses, but neglects conflict misses, which have previously been shown to be relatively less critical in some programs [7].

If the minimum distance of an interval is greater than the cache size, all accesses in the interval are considered as misses. When the cache size falls in the middle of an interval, we estimate the miss rates based on relative position of the size in the interval. In our experiments, we used an effective cache size equal to one-half of the actual cache size for all set-associative cache configurations, where the associativity is small (≤ 4). For the fully associative configurations we used the actual size as the effective size. The prediction for L2 cache is identical to that for L1 cache with the predicted L1 cache hits filtered out.

In our analysis, an accurate miss-rate prediction for an instruction occurs if the predicted miss rate is within 5% of the actual miss rate gleaned through cache simulation using the same input. Although the predicted miss rate does not include conflict misses, the actual miss rate does. The results reported in the next section show that in spite of not incorporating conflict misses in the prediction, our prediction of miss rates is highly accurate.

4. EXPERIMENT

In this section, we report the results of our experiment evaluating the predictability of per instruction reuse distance. We begin with a discussion of our experimental methodology and then, we discuss the predictability of instruction reuse distance. Next, we report the predictability of per instruction cache miss rate for six different cache organizations. Finally, we show that our analysis can accurately predict the instructions that generate the most L2 misses.

4.1 Methodology

To compute reuse distance, we instrument the program binaries using Atom [11] to collect the data addresses for all memory instructions. The Atom scripts incorporate Ding and Zhong's reuse-distance collection tool [3, 14] into our analyzer to obtain reuse distances. During profiling, our analysis records the cache-line based forward reuse distance distribution for each individual memory instruction.

Our benchmark suite consists of 11 of the 14 programs from SPEC CFP2000. Table 1 lists the programs that we used. The remaining three benchmarks in SPEC CFP2000 are not included because we could not get them to compile correctly on our Alpha cluster. We used version 5.5 of the Compaq compilers using the -O3 optimization flag to compile the programs.

In the tables and figures reported throughout the rest of this section, we report both static and dynamic weightings of the results. The static weighting considers each instruction weighted identically, irrespective of the number of times it is executed. The dynamic weighting weights each static instruction by the number of times it executed. For instance, if a program contains two memory instructions, A and B, and we correctly predict the result for instruction A and incorrectly predict the result for instruction B, we have a 50% static prediction accuracy. If, however, instruction A is executed 80 times and instruction B is executed 20 times, we have an 80% dynamic prediction accuracy.

For miss-rate prediction measurements, we implemented a cache simulator and embedded it in our analysis routines to collect the number of L1 and L2 misses for each instruction. Table 2 shows the cache configurations that we used in our experiments. Each of the cache configurations uses 64-byte lines and an LRU replacement policy.

To compare the effectiveness of our miss-rate prediction, we implemented three miss-rate prediction schemes. The first scheme, called *predicted reuse distance* (PRD), uses the reuse-distance predicted by our analysis of the training runs to predict the miss rate for each instruction. We use the test and train input sets for the training runs. The second scheme, called *reference reuse distance* (RRD),

Benchmark	Patterns	Input Set	Data Size (cache blocks)	Static Weighting Coverage (%)	Static Weighting Accuracy (%)	Dynamic Weighting Coverage (%)	Dynamic Weighting Accuracy (%)

188.ammp	const(87.1%)
	linear(9.8%)
	others(3.9%)
		test*(1002 atoms)
		train(30002 atoms)
		ref(300002 atoms)
			7.2 K
			147.1 K
			366.7 K
				89.9	97.0	84.1	97.0

173.applu	const(87.3%)
	1third(6.8%)
	others(5.9%)
		test(12 × 12 × 12)
		train(24 × 24 × 24)
		ref(60 × 60 × 60)
			18.8 K
			175.1 K
			2.86 M
				87.7	97.8	95.6	98.2

301.apsi	const(86.9%)
	linear(8.7%)
	others(4.4%)
		test*(128 × 1 × 32)
		train(128 × 1 × 64)
		ref*(128 × 1 × 256)
			8.2 K
			15.1 K
			62.3 K
				96.0	98.6	92.1	96.6

179.art	const(92.4%)
	linear(3.9%)
	others(3.7%)
		test(1 object)
		train*(10 objects)
		ref*(30 objects)
			37.4 K
			57.4 K
			107.4 K
				82.8	98.4	99.9	99.9

183.equake	const(86.6%)
	1third(5.3%)
	others(7.9%)
		test*(512 nodes)
		train(7294 nodes)
		ref(30169 nodes)
			11.4 K
			158.8 K
			782.6 K
				96.5	99.2	97.4	99.2

189.lucas	const(81.7%)
	linear(16.7%)
	others(1.6%)
		test
		train
		ref
			782
			4.7 K
			2.3 M
				66.7	93.9	52.2	88.4

177.mesa	const(95.2%)
	linear(2.2%)
	others(5.6%)
		test*(200 × 200)
		train*(400 × 400)
		ref(800 × 800)
			5.2 K
			14.9 K
			44.3 K
				94.8	96.5	95.2	91.3

172.mgrid	const(82.8%)
	1third(7.8%)
	others(9.6%)
		test*(32 × 32 × 32)
		train(64 × 64 × 64)
		ref(128 × 128 × 128)
			16.6 K
			119.5 K
			907.4 K
				90.3	94.0	94.2	95.6

200.sixtrack		test train ref	391.8 K 391.8 K 391.8 K	99.9	99.7	99.7	99.9

171.swim	const(71.2%)
	sqrt(12.3%)
	others(16.5%)
		test*(128 × 128)
		train(512 × 512)
		ref(1334 × 1334)
			32.7 K
			476.4 K
			3.1 M
				86.3	95.1	93.7	99.8

168.wupwise	const(96.0%)
	1third(2.7%)
	others(1.3%)
		test*(16 × 5 × 5 × 5)
		train*(16 × 10 × 10 × 10)
		ref(16 × 20 × 20 × 20)
			45.1 K
			360.1 K
			2.9 M
				98.2	99.5	99.9	98.0

average				89.9	97.2	91.3	96.7

* – data size adjusted from SPEC distribution

Table 1: Forward reuse distance distribution prediction coverage and accuracy for CFP2000 benchmarks

config. no.	L1	L2	
1	32K, 2–way	1M	fully assoc.
2			4–way
3			2–way
4	16K, 2–way	512K	fully assoc.
5			4–way
6			2–way

Table 2: Cache configurations

uses the actual reuse distance computed by running the program on the reference input data set to predict the miss rates. RRD represents an upper bound on the effectiveness of using reuse distance to predict cache-miss rates. The third scheme, called *test cache simulation* (TCS), uses the miss rates collected from running the test data input set on a cache simulator to predict the miss rate of the same program run on the reference input data set. Table 3 gives a summary of each technique for easy reference.

4.2 Results

In this paper, we only report statistics involving forward reuse distance prediction. The results for backward reuse distance prediction are similar. We begin this section with data related to reuse distance distribution prediction accuracy and coverage. Then, we show how well reuse distance prediction predicts both L1 and L2 miss rates for six different cache configurations. Throughout the section, we focus more on the dynamically weighted results than the statically weighted results since the dynamic results factor in execution frequency.

4.2.1 Reuse-distance Prediction Accuracy and Coverage

Table 1 lists the coverage and accuracy of reuse-distance prediction on a per instruction basis. The statically weighted coverage is 89.9% while the dynamically weighted coverage is 91.3%, on average. For all programs except 188.ammp and 189.lucas the dynamic coverage is well over 90%. 188.ammp exhibits a larger than normal percentage of irregular patterns. In 189.lucas, approximately 31% of the memory operations do not appear in both training runs. If an instruction does not appear during execution for both the test and train data sets, we cannot predict its reuse distance.

Our model predicts reuse distance correctly for 97.2% of the covered instructions using a static weighting and 96.7% using a dynamic weighting, on average. Our analysis predicts the reuse distance accurately for over 95% of the covered instructions using dynamic weighting for all programs except 189.lucas and 177.mesa. In 189.lucas, the fact that all instructions do not appear in both the test and train runs throws off our computation of reuse distance. These extra instructions change the reuse distance because different memory locations are accessed. In 177.mesa, some instructions that exhibit a single pattern in the test and train runs, exhibit multiple patterns in the reference run.

In Table 1, the column labeled "Patterns" shows the major pat-

Scheme	Name	Description
PRD	predicted reuse distance	Use reuse distance predicted from data size
RRD	reference reuse distance	Use actual reference reuse distance
TCS	test cache simulation	Use miss rates from test run

Table 3: Miss-rate prediction schemes

terns identified in the programs by our analysis.[1] It can be seen that, for all programs, most patterns are constant patterns. However, a significant number of other patterns exist in some programs. For example, in 171.swim 12.3% of the patterns exhibit a square root (sqrt) distribution pattern and in 189.lucas 16.7% of the patterns exhibit a linear distribution. For 200.sixtrack, we do not report the patterns since all data sizes are identical. This makes it impossible to construct a fitting function.

In addition to measuring the prediction coverage and accuracy, we measured the number of locality intervals exhibited by each instruction. Table 4 below shows the percentage of instructions that exhibit 1, 2 or 3 or more intervals during execution.

Benchmark	Static Weighting			Dynamic Weighting		
	1	2	≥ 3	1	2	≥ 3
188.ammp	83.8	13.3	2.9	78.5	18.0	3.5
173.applu	73.8	23.3	2.9	62.7	32.5	4.8
301.apsi	87.2	10.0	2.8	60.3	31.3	8.4
179.art	92.8	6.4	0.8	91.1	8.8	0.1
183.equake	90.7	7.6	1.6	81.0	16.5	2.5
189.lucas	81.7	17.5	0.8	65.5	33.9	0.6
177.mesa	91.0	7.1	1.9	86.2	10.3	3.5
172.mgrid	70.4	17.1	12.5	60.4	25.0	14.6
200.sixtrack	84.2	23.1	2.7	64.8	22.1	13.1
171.swim	72.6	27.2	0.2	49.9	49.5	0.6
168.wupwise	91.5	6.7	1.8	66.2	22.1	11.7
average	83.6	14.5	2.8	69.7	24.5	5.8

Table 4: Number of locality intervals

For our benchmark suite, all but three programs have more than 90% of their instructions exhibiting only one or two reuse-distance patterns. Each of the other three programs have over 85% of their instructions having only one or two reuse-distance patterns. This information shows that most references have highly predictable reuse patterns.

Benchmark	<1K	1–8K	8–32K	32–64K	≥64K
188.ammp	95.5	8.3	10.8	6.4	5.3
173.applu	98.2	1.7	2.2	1.9	7.4
301.apsi	95.9	4.6	6.1	3.6	0.0
179.art	86.3	0.3	16.6	6.2	6.2
183.equake	95.8	2.5	3.5	2.9	9.6
189.lucas	93.6	2.3	2.4	2.1	19.2
177.mesa	98.6	0.7	0.1	5.0	0.0
172.mgrid	97.9	14.8	13.1	6.6	17.4
200.sixtrack	95.8	3.1	4.4	3.1	2.6
171.swim	92.2	7.2	6.4	8.3	30.0
168.wupwise	96.6	0.4	0.2	0.4	5.1
average	95.1	4.2	6.0	4.2	9.3

Table 5: Percentage instructions per interval

Table 5 reports the distribution of the static instructions within the 2^k intervals for the reference data set using 64-byte cache lines. The table gives the percentage of instructions that have a reuse distance pattern that falls into each of the intervals. Note that instructions can fall into more than one interval. As can be seen from Table 5, most reuse patterns have a small reuse distance. However, there are a number of reuse patterns with a long reuse distance. These longer reuse distance instructions represent opportunities for cache optimizations as the likelihood of cache misses is higher.

To evaluate the effect of merging two intervals as discussed in Section 3.2, we report how often instructions whose reuse pattern crosses the 2^k boundaries are merged into a single pattern. Table 6 shows that in most programs less than 10% of the patterns are merged. Although not dominant, the merging still has a significant effect on our prediction mechanism.

Benchmark	% Intervals Merged
188.ammp	14.9
173.applu	14.2
301.apsi	8.0
179.art	8.2
183.equake	7.0
189.lucas	7.2
177.mesa	7.6
172.mgrid	19.2
200.sixtrack	10.3
171.swim	9.7
168.wupwise	7.6
average	**10.4**

Table 6: Percentage of merged intervals

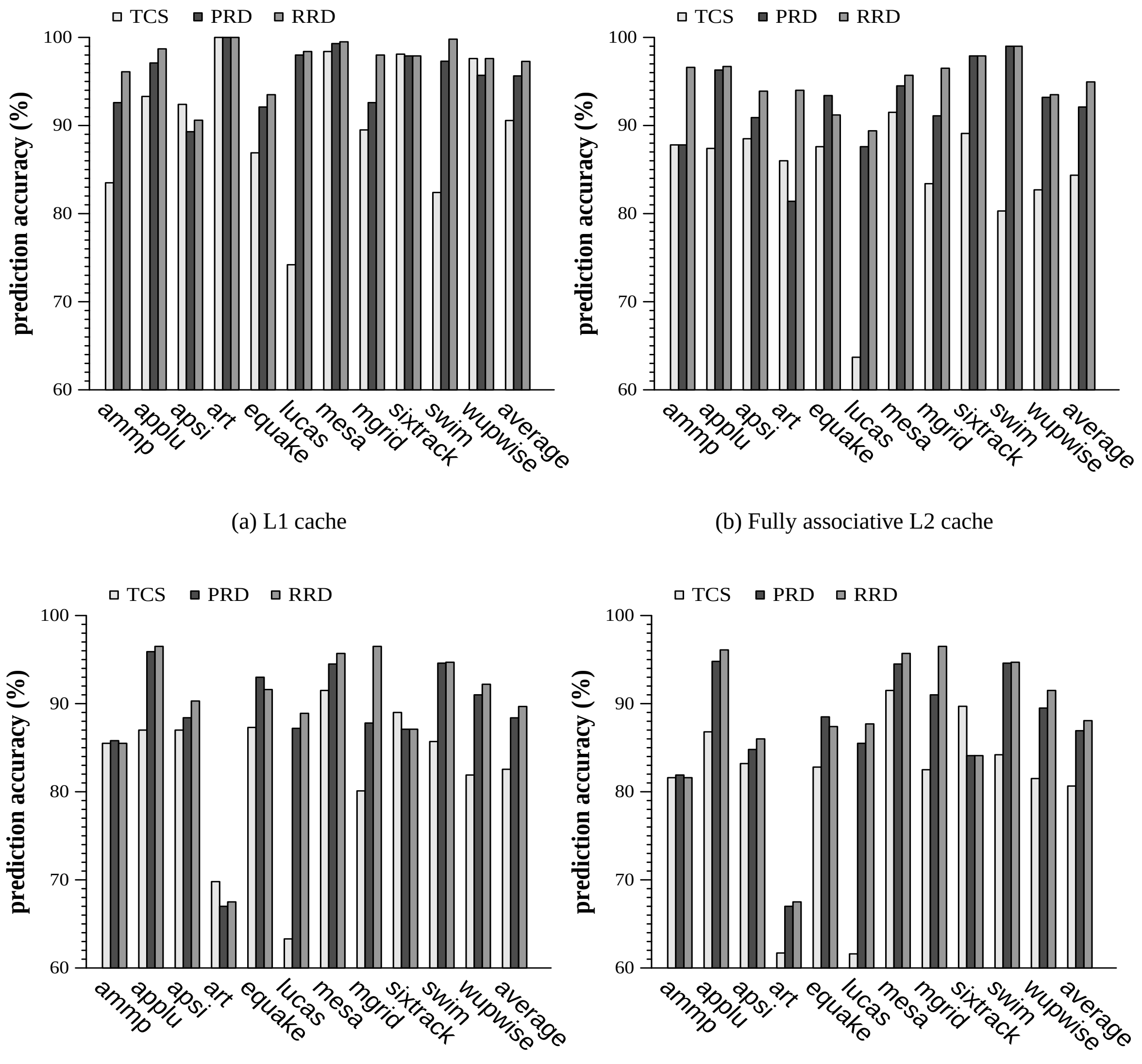
4.2.2 Miss-rate Prediction Accuracy

Figures 1 and 2 report our miss-rate prediction accuracy using dynamic and static weighting, respectively, for cache configurations 1, 2 and 3 reported in Table 2. Table 7 summarizes the average miss rate prediction accuracy from those figures and the remaining cache configurations. These figures and table report the percentage of instructions whose miss rate is predicted within 5% of the actual miss rate. Examining Table 7 reveals that our prediction method (PRD) predicts the L1 miss rate of instructions with at least a 93.1% accuracy and the L2 miss rate with at least a 87.4% accuracy using the dynamic weighting. On average PRD more accurately predicts the miss rate than TCS, but is slightly less accurate than RRD.

Since TCS simply considers the test run's miss rate as the reference miss rate, TCS is only effective when an instruction has only constant patterns or the data sizes of the test and reference are very close (reuse distance change is minimal). PRD takes the data size change into account, and thus its prediction accuracy is not likely to be negatively influenced by a large change in input data size. For 189.lucas and 171.swim, the difference between the data sizes of test and reference is large, and both programs have a significant number of non-constant patterns. TCS makes a highly inaccurate prediction compared to PRD, as shown in each part of Figures 1 and 2.

When the data size difference between test and reference runs is small (e.g. 200.sixtrack and 179.art), TCS can predict the miss rate quite accurately. TCS outperforms PRD for 200.sixtrack on

[1]Note that the rows labeled "1third" indicate $f_i(s) = s^{\frac{1}{3}}$.



(a) L1 cache

(b) Fully associative L2 cache

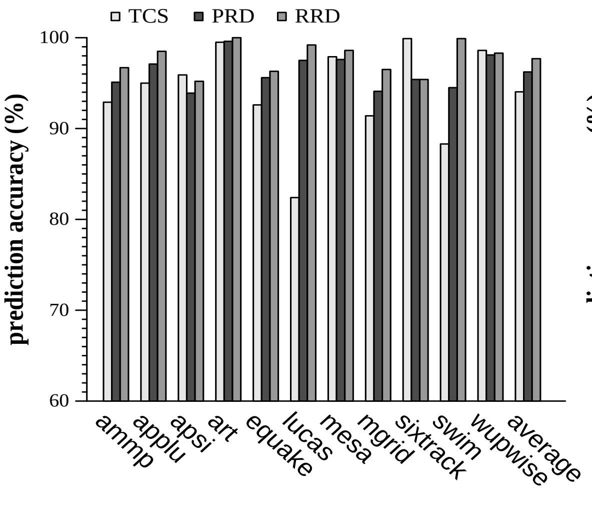
(c) 4–way set associative L2 cache

(d) 2-way set associative L2 cache

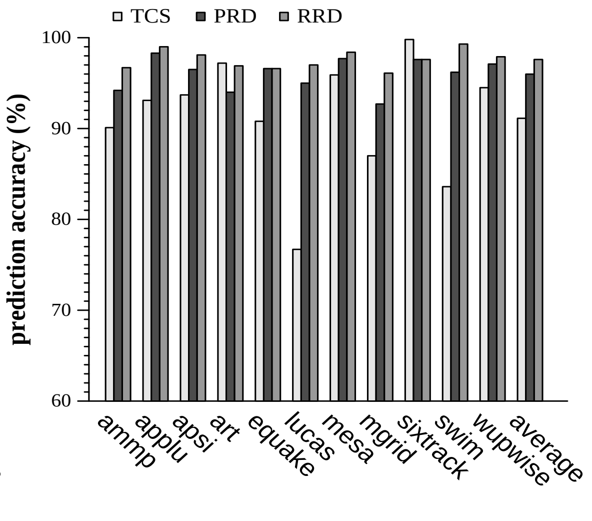
Figure 1: Dynamic miss-rate prediction accuracy

Cache Config. No.	L1						L2					
	Dynamic Weighting			Static Weighting			Dynamic Weighting			Static Weighting		
	TCS	PRD	RRD	TCS	PRD	RRD	TCS	PRD	RRD	TCS	PRD	RRD
1	90.6	95.6	97.3	94.0	96.2	97.7	84.0	92.5	94.8	91.2	96.2	97.7
2							82.6	88.4	89.7	90.7	94.9	96.3
3							80.6	87.4	88.7	90.4	94.5	95.9
4	90.3	93.1	94.5	93.8	94.8	96.2	85.9	92.4	93.8	92.0	96.0	97.4
5							84.0	89.9	90.3	90.4	94.9	96.0
6							82.1	87.7	88.0	90.9	94.3	95.3

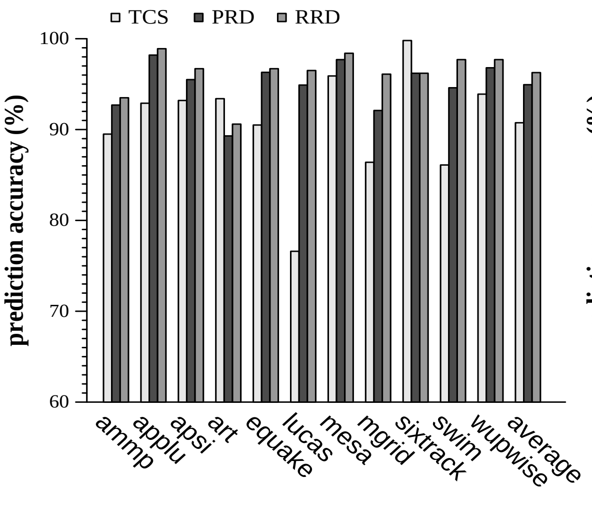
Table 7: Average miss-rate prediction accuracy



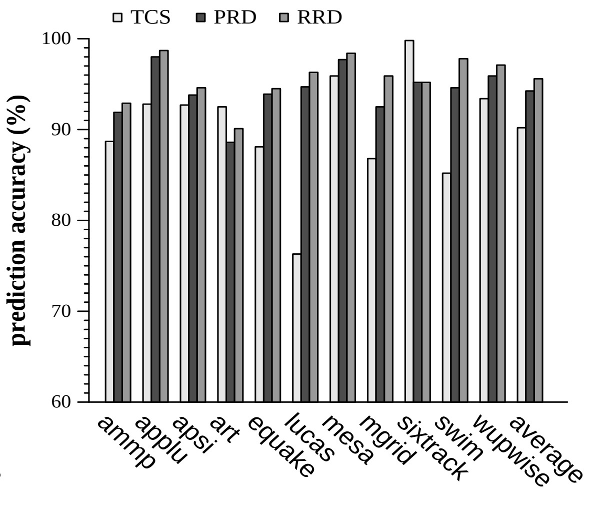
(a) L1 cache



(b) Fully associative L2 cache



(c) 4–way set associative L2 cache



(d) 2-way set associative L2 cache

Figure 2: Static miss-rate prediction accuracy

the L1 miss rate slightly, and both set-associative L2 miss rates, but not the fully associative L2 miss rate. TCS even outperforms RRD in predicting miss rate on those same configurations. This is because the data size is not a factor and TCS can take into account conflict misses through simulation, whereas reuse distance cannot. 179.art exhibits some of the same patterns, but since the data size does change, PRD and RRD outperform TCS in some instances.

TCS also does well in L1 miss rate prediction for 301.apsi and 168.wupwise. For 168.wupwise, we believe this is due to the large percentage of constant reuse distance patterns, making data-set size change less pronounced. Note that for 168.wupwise PRD outperforms TCS on the L2 miss rates indicating that the constant patterns are more likely handled in the L1 cache and the changes in data size have more of an effect on the larger reuse distances. For 301.apsi, a large percentage of L1 miss are conflict misses. For small constant reuse patterns, TCS can predict those misses better than both PRD and RRD.

Figures 1(b) through 1(d) and Figures 2(b) through 2(d) illustrate our prediction accuracies for L2 misses for 1M L2 caches with different associativities. These results show that PRD is effective in predicting L2 misses for a range of associativities. For fully-associative L2 cache, as shown in Figure 1(b) and 2(b), PRD predicts the miss rate correctly for 92.5% of the dynamically weighted instructions, and 96.2% of the statically weighted instructions correctly, while TCS predicts the miss rate with accuracies of only 84.0% and 91.2%, respectively. Since there are no conflict misses for a fully-associative L2, PRD shows a significant improvement in the prediction over TCS for all programs except 179.art. In this instance, RRD outperforms TCS, while PRD does not. Since PRD is a heuristic, it does not always perform as expected. However, our results show that in most instances, PRD is quite effective.

Table 7 shows that across all cache configurations, on average PRD performs almost as well as RRD and better than TCS. These results indicate that PRD is not highly sensitive to cache size. As-

Cache	2%			5%			8%		
	TCS	PRD	RRD	TCS	PRD	RRD	TCS	PRD	RRD
2-way, 32K L1	90.4	95.0	97.2	90.6	95.6	97.3	90.7	96.0	97.3
4-way, 1M L2	82.1	88.0	89.4	82.6	88.4	89.7	82.9	89.0	90.2

Table 8: Prediction accuracy measures

sociativity does have an effect on PRD's prediction accuracy. Because lower associativity introduces more conflict misses, the accuracy will decrease. However, even for a 2-way set-associative L2 cache, on average, our model's prediction can still achieve 87.4% accuracy for dynamic weighting, and 94.5% for static weighting, leading us to believe that PRD shows much promise in future applications to cache optimization.

To examine the effects of our accuracy measure, we examine the miss-rate prediction accuracy if we considered a predicted miss rate to be accurate if that rate was within 2%, 5% and 8% of the acutal miss rate. Table 8 reports the dynamically weighted average miss rate prediction accuracy for the cache configuration 2. As can be seen from Table 8, the accuracy changes only slightly as we modify the prediction accuracy measure. Note that there are no individual programs where the changes are more dramatic than reported in the averages.

4.2.3 Critical Instructions

For static or dynamic optimizations, we are interested in the *critical* instructions which generate the most L2 misses. In this section, we show that we can predict most of the critical instructions. We also observe that the locality intervals of the critical instructions tend to be more diverse than non-critical instructions and tend to exhibit non-constant reuse patterns more frequently.

We perform cache simulation on the reference input using cache configuration 2 to identify the critical instructions which generate the most L2 misses: those static instructions whose cumulative L2 misses account for 95% of the total L2 misses in the program. To predict critical instructions, we use the execution frequency in one training run to estimate the relative contribution of the number of misses for each instruction given the total miss rate. We then compare the predicted critical instructions with the real ones and show the prediction accuracy weighted by the absolute number of misses in Table 9. We also list the number of critical instructions for each benchmark in the column labeled "# Critical" and the total number of static memory operations for each benchmark in the column labeled "Total".

The prediction accuracy for critical instructions is 86% on average. 189.lucas shows a very low accuracy because of low prediction coverage. The unpredictable instructions in 189.lucas contribute a significant number of misses. Notice that the number of critical instructions in most programs is very small. These results show that reuse distance can be used to allow compilers to target the most important instructions for optimization effectively.

Critical instructions tend to have more diverse locality patterns than non-critical instructions. Table 10 reports the distribution of the number of locality intervals for critical instructions using dynamic weighting. We find that the distribution is more diverse than that shown in Table 4. Although less than 3% of the instructions on average have more than 2 intervals, the average goes up to over 23% when considering only critcial instructions. However, only a small number of critical instructions have more than three intervals and no instructions exhibit over six intervals.

Critical instructions also tend to exhibit a higher percentage of non-constant patterns than non-critical instructions. Table 11 re-

Benchmark	Accuracy	# Critical	Total
188.ammp	85.2	16	5180
173.applu	93.0	224	11352
301.apsi	84.7	194	11567
179.art	99.5	14	1820
183.equake	87.7	76	3018
189.lucas	57.3	201	2915
177.mesa	100.0	3	5736
172.mgrid	76.0	84	2185
200.sixtrack	93.2	85	19688
171.swim	99.4	263	2028
168.wupwise	72.7	15	1667
average	86.2	107	6105

Table 9: Prediction accuracy for critical instructions

Benchmark	Interval Distribution (%)					
	1	2	3	4	5	6
188.ammp	96.3	0.0	3.7	0.0	0.0	0.0
173.applu	6.0	77.9	16.1	0.0	0.0	0.0
301.apsi	7.0	58.5	30.7	3.7	0.0	0.0
179.art	100.0	0.0	0.0	0.0	0.0	0.0
183.equake	18.8	80.0	1.2	0.0	0.0	0.0
189.lucas	7.9	92.1	0.0	0.0	0.0	0.0
177.mesa	0.0	100.0	0.0	0.0	0.0	0.0
172.mgrid	0.0	67.3	28.1	3.0	0.6	1.0
200.sixtrack	0.3	18.8	79.4	1.4	0.0	0.0
171.swim	0.0	98.8	1.2	0.0	0.0	0.0
168.wupwise	0.4	10.1	68.4	5.4	15.7	0.0
average	21.5	54.9	20.8	1.2	1.5	0.1

Table 10: Distribution of the number of locality intervals for critical instructions

ports the percentage of critical instructions that have all constant patterns. As can be seen, critical instructions are more sensitive to data size. This fact makes it important to predict accurately reuse distance in order to apply optimization to the most important memory operations. Note that we report no data for 200.sixtrack because all of its data sizes are identical.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have demonstrated that reuse distance is predictable on a per instruction basis for floating-point programs. On average, over 90% of all memory operations executed in a program are predictable with an almost 97% accuracy. In addition, the predictable reuse distances translate to predictable miss rates for the instructions. For a 32KB 2-way set associative cache, our miss-rate prediction accuracy is almost 96%, and for a 1MB fully associative L2 cache, our miss-rate prediction accuracy is over 92%. Our analysis also identifies the critical instructions in a program that contribute to 95% of the program's L2 misses. On average, our method predicts the critical instructions with an 86% accuracy.

The next step in our research is to apply reuse-distance and miss-rate prediction to cache optimization. We are currently developing a hardware-based prefetching mechanism that will utilize reuse distance for those instructions likely to cause a significant number of

Benchmark	% Constant
188.ammp	25.0
173.applu	13.8
301.apsi	7.7
179.art	50.0
183.equake	11.8
189.lucas	1.5
177.mesa	0.0
172.mgrid	35.7
200.sixtrack	-
171.swim	0.0
168.wupwise	33.3
average	**17.9**

Table 11: Percentage of critical instructions having all constant patterns

cache misses. In addition, we are beginning work on phase detection of reuse distance. If an instruction has multiple reuse-distance intervals, we would like to be able to determine if those reuse distance intervals occur in phases. Detecting phases will allow us to further target cache optimization and possibly use source-level transformations to isolate the regions in the code where misses are likely to occur. Additionally, we plan to investigate the use of reuse distance in optimizing pointer-based applications.

In order for significant gains to be made in improving program performance, compilers must improve the performance of the memory hierarchy. Our work is a step in opening up new avenues of research through the use of feedback-directed and dynamic optimization in improving program locality through the use of reuse distance.

6. ACKNOWLEDGMENTS

This work was partially supported by NSF grant CCR-0312892. We would like to thank Chen Ding and Yutao Zhang for providing their reuse distance tools. Additionally, we would like to thank the anonymous referees for their suggestions to improve this paper.

7. REFERENCES

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

[2] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behaviour of nested loops. In *Proceedings of the SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 286–297, Snowbird, Utah, June 2001.

[3] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–257, San Diego, California, June 2003.

[4] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, San Jose, CA, Oct. 1998.

[5] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 15–29, Toronto, Canada, June 1991.

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

[7] K. S. McKinley. A compiler optimization algorithm for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 9(8):769–787, Aug. 1998.

[8] K. S. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[9] 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, Nov. 1999.

[10] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, Aug. 1992.

[11] A. Srivastava and E. A. Eustace. Atom: A system for building customized program analysis tools. In *Proceeding of ACM SIGPLAN Conference on Programming Language Design and Inplementation*, June 1994.

[12] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 24–35, Santa Clara, CA, May 1993.

[13] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Canada, June 1991.

[14] Y. Zhong, S. Dropsho, and C. Ding. Miss rate prediction across all program inputs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 91–101, New Orleans, LA, September 2003.