Instruction Based Memory Distance Analysis and its Application to Optimization*

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 as it provides a means to analyze complex program behavior that is not possible using traditional static analysis. Feedback-directed optimization offers the compiler opportunities to analyze and optimize the memory behavior of programs even when traditional array-based analysis is not applicable. As a result, both floating-point and integer programs can benefit from memory hierarchy optimization.

In this paper, we examine the notion of memory distance *as it is applied to the instruction space of a program and to feedback-directed optimization. Memory distance is defined as a dynamic quantifiable distance in terms of memory references between two accesses to the same memory location. We use memory distance to predict the miss rates of instructions in a program. Using the miss rates, we then identify the program's critical instructions – the set of high miss instructions whose cumulative misses account for 95% of the L2 cache misses in the program – in both integer and floating-point programs. Our experiments show that memory-distance analysis can effectively identify critical instructions in both integer and floating-point programs.*

Additionally, we apply memory-distance analysis to memory disambiguation in out-of-order issue processors, using those distances to determine when a load may be speculated ahead of a preceding store. Our experiments show that memory-distance-based disambiguation on average achieves within 5-10% of the performance gain of the store set technique which requires a hardware table.

1. Introduction

With the widening gap between processor and memory speeds, program performance relies heavily upon the effective use of a machine's memory hierarchy. In order to obtain good application performance on modern systems, the compiler and microarchitecture must address two important factors in memory system performance: (1) data locality and (2) load speculation. To improve locality in programs, compilers have traditionally used either static analysis of regular array references [12, 23] or profiling [1] to determine the locality of memory operations. Unfortunately, static analysis has limited applicability when index arrays or pointer operations are used in addressing and when determining locality across multiple loop nests. On the other hand, profiling-based techniques typically cannot adapt to program input changes. Similarly, numerous hardware techniques exist for determining when a load may be speculatively issued prior to the completion of a preceding store in order to improve superscalar performance [3, 14, 15], but compiler-based solutions typically do not yield good results across a wide spectrum of benchmarks.

Recently, reuse distance analysis [4, 5, 10, 24], has proven to be a good mechanism to predict the memory behavior of programs over varied input sets. The *reuse distance* of a memory reference is defined as the number of distinct memory locations accessed between two references to the same memory location. Both whole-program [4, 24] and instruction-based [5, 10] reuse distance have been predicted accurately across all program inputs using a few profiling runs. Reuse-distance analysis uses curve fitting to predict reuse distance 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 allows the prediction of reuse to be quite accurate over varied data sizes.

In this paper, we expand the concept of reuse distance to encompass other types of distances between memory references. We introduce the concept of *memory distance*, where the memory distance of a reference is a dynamic quantifiable distance in terms of memory references between two accesses to the same memory location. In our terminology, reuse distance is a form of memory distance. We present a new method for instruction-based memory distance analysis that handles some of the complexities exhibited in integer programs and use that analysis to predict both long and short memory distances accurately. We apply the improved memory distance analysis to the problem of identifying critical instructions – those instructions that cause 95% of the misses in a program – and to the problem of memory dependence prediction. Predicting miss rates and identifying critical instructions requires our analysis to predict large memory distance accurately. In contrast, determining when a particular load instruction may be issued ahead of a preceding store instruction requires us to predict short memory distance accurately.

Across a set of the SPEC2000 benchmark suite we are able to

*This work was partially supported by NSF grant CCR-0312892.

predict short and long memory distances accurately (above a 90% accuracy in most cases). In addition, our experiments show that we are able to predict L2 miss rates with an average 92% accuracy and identify an average of 92% and 89% of the critical instructions in a program using memory distance analysis for floating-point and integer programs, respectively. Furthermore, our experiments show that using memory distance prediction to disambiguate memory references yields performance competitive with well-known hardware memory disambiguation mechanisms, without requiring hardware to detect when a load may be issued ahead of a preceding store speculatively. The static schemes achieve performance within 5% of a 16K-entry store set implementation for floating point programs and within 10% for integer programs [3].

We begin the rest of this paper with a review of reuse distance analysis. Then, we present our memory-distance analysis and experiments examining instruction-based memory distance prediction, cache miss-rate prediction, critical instruction detection, and memory-distance based memory disambiguation. We conclude with a discussion of work related to locality analysis and memory disambiguation, and a discussion of future work.

2. Reuse-distance Analysis

In this section, we describe the *reuse distance* and whole-program locality analysis of Ding et al. [4]. Their work uses a histogram describing reuse distance distribution for the whole program. Each bar in the histogram consists of the portion of memory references whose reuse distance falls into the same range. 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.

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 histogram of a third data size is predictable in a selected set of benchmarks. The reuse-distance prediction step generates the histogram for the third input using the data size of that third input. The data size of the third input can be obtained via sampling. Typically, one can use this method to predict a locality histogram 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 ith 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.

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

Once 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.

Memory distance may be computed at any granularity in the memory hierarchy. For predicting miss rates and identifying critical instructions, we compute memory distance at the granularity of the cache line. For memory disambiguation, we compute memory distance at the granularity of a memory address.

Ding et al. compute reuse distances for each address referenced in the entire program without relating those distances to the instructions that cause the memory access. We observe that Ding et al.'s model can be extended to predict various input-related program behaviors, such as memory distance and execution frequency, at the instruction level. We examine mapping the memory distances to the instructions that cause the memory accesses and then compute the memory distances for each load instruction. In addition, we develop a scheme to group related memory distances that improves prediction accuracy. Sections 3 through 5 discuss our extensions for predicting memory distance at the instruction level and the application of memory distance to optimization.

3. Reuse Distance Prediction

Reuse distance is one form of memory distance that is applicable to analyzing the cache behavior of programs. Although previous work has shown that the reuse distance distribution of the whole program [4] and each instruction [5] is predictable for floating-point programs, it is unclear whether the reuse distances of an instruction show the same predictability for integer programs. Our focus is to predict the reuse distance distribution and miss rate of each instruction for a third input given the collected and analyzed reuse distances of each instruction in two training inputs of different size. When collecting reuse distance statistics, we simply map the reuse distances of an address to the instructions that access the address. Thus, the reuse distance for an instruction is the set of reuse distances of the addresses that the instruction references. In this section, we discuss our methods to predict instruction-based reuse distance, including an enhancement to improve predictability of integer programs. We use the predicted reuse distances to estimate cache misses on a per instruction basis in Section 4.

To apply per instruction reuse distance and miss rate prediction on the fly, it is critical to represent the reuse distances of the 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 (*frequency*) for each bin. We also record the minimum, maximum, and mean distances within each bin. A bin is *active* if there exists an occurrence of reuse in the bin. We note that at most 8 words of information (min, max, mean and frequency) are needed for most instructions in order to track their reuse distances since most instructions need only two bins. Our work uses logarithmic division for distances less than 1K and uses 1K bins for distances greater than 1K.

Although we collect memory distance using fixed bin boundaries, those bins do not necessarily reflect the real distribution, particularly at the instruction level. For example, the set of related reuse distances may cross bin boundaries. We define a *locality pattern* as the set of nearby related reuse distances for an instruction. One instruction may have multiple locality patterns. To construct locality patterns, adjacent bins can be merged into a single pattern. Fang et al. [5] merge adjacent bins and assume a uniform distribution of distance frequency for the resulting locality pattern [5]. Assuming a uniform distribution works well for floating-point programs but, as we show in Section 4, performs poorly for integer programs, particularly for miss-rate prediction. Reuse distance often does not exhibit a uniform distribution in integer programs. In this section, we propose a new bin merging method that performs well on both integer and floating-point pro-

input: the set of memory-distance bins B
output: the set of locality patterns P

for each memory reference r {
  $P_r = \emptyset$; down = **false**; p = **null**;
  for (i = 0; i < numBins; i++)
    if (B_r^i.size > 0)
      if (p == **null** $\|(B_r^i.min - p.max > p.max - B_r^i.min)\|$
        $(down\&\&B_r^{i-1}.freq < B_r^i.freq))$ {
        p = new pattern; p.mean = B_r^i.mean;
        p.min = B_r^i.min; p.max = B_r^i.max;
        p.freq = B_r^i.freq; p.maxf = B_r^i.freq;
        $P_r = P_r \cup$ p; down = **false**;
      }
      else {
        p.max = B_r^i.max; p.freq += B_r^i.freq;
        if (B_r^i.freq > p.maxf) {
          p.mean = B_r^i.mean; p.maxf = B_r^i.maxf;
        }
        if (!down && $B_r^{i-1}.freq > B_r^i.freq$)
          down = **true**;
      }
    else
      p = **null**;
}

Figure 1. Pattern-formation Algorithm

grams. The new technique computes a linear distribution of reuse distances in a pattern using the minimum, maximum, mean and frequency of the reuse distance.

Once the reuse-distance data has been collected, we construct reuse-distance patterns for each instruction by merging bins using the algorithm in Figure 1. The algorithm scans the original bins from the smallest distance to largest distance and iteratively merges any pair of adjacent bins i and $i+1$ if

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

This inequality is true if the difference between the minimum distance in bin $i+1$ and the maximum distance in bin i is no greater than the length of bin i. The merging process stops when it reaches a minimum frequency and starts a new pattern for the next bin. *The set of merged bins for an instruction make up its locality patterns*. We observe that this additional merging pass reflects the locality patterns of each instruction and notably improves prediction accuracy since the patterns of reuse distance may cross the predefined bin bounds. As illustrated in Figure 2, the first four bins are merged as one pattern and the remaining two merged as the other. We represent the constructed locality patterns just as with the original bins using a mean, max, mean and frequency for the pattern. For a pattern, its mean is the mean of the bin with the maximum frequency and its frequency records the total frequency of all merged bins. Using min, max, mean, and frequency of each pattern, we indeed model up to two linear frequency distributions in each pattern split by its mean.

Following the prediction model discussed in Section 2, the reuse distance patterns of each instruction for a third input can be predicted through two training runs. For each instruction, we predict its ith pattern by fitting the ith pattern in each of the training runs. The fitting function is then used to find the minimum, maximum, and mean distance, and the frequency of the predicted

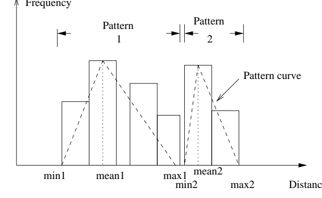


Figure 2. Pattern formation

pattern. 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. Prediction accuracy indicates the percentage of *covered* instructions whose reuse distance distribution is correctly predicted by our model. An instruction's reuse distance distribution can be predicted if and only if the instruction 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. Although irregular patterns do not occur often in all our experimental benchmarks (7-8% of the instructions on average), they occur more often in the integer programs.

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 patterns with the collected patterns through a real run. The prediction is said to be *correct* if the predicted pattern and the observed pattern fall into the same set of bins, or they 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

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

We have chosen an overlap factor of 90% because it yields the necessary accuracy for us to predict miss rates effectively. Since we use floating-point fitting functions to predict reuse distance some error must be tolerated. We note that, however, the effect on prediction accuracy varies by less than 1% if we require predicted patterns to have a 95% overlap with the actual patterns.

3.1 Experimental Methodology

To compute reuse distance, we instrument the program binaries using Atom [20] to collect the data addresses for all memory instructions. The Atom scripts incorporate Ding and Zhong's reuse-distance collection tool [4, 24] into our analyzer to obtain reuse distances. During profiling, our analysis records the cache-line based reuse distance distribution for each individual memory instruction using a cache-line size of 64 bytes.

We examine 11 programs from SPEC CFP2000 and 11 programs from SPEC CINT2000. Tables 1 and 2 list the programs that we use. The remaining four benchmarks in CFP2000 and CINT2000 are not included because we could not get them to compile correctly on our Alpha cluster. We use version 5.5 of the Compaq compilers using the -O3 optimization flag to compile the

programs. Since SPEC CPU2000 does not provide two train input sets for feedback-directed optimization with all benchmarks, we use the test and the train input sets. Using the reuse distances measured for the test and train input sets, we predict for the reference input sets. Even though Hsu et al. [9] show that the test input set does not represent the cache behavior of the program well due to its small size, we obtain good results since we can characterize the effects of a change in data size on the cache behavior using small inputs and translate those changes into the cache effects for a large input without using that large input set. We verify this claim in Section 4.

In the data reported throughout the rest of this paper, we report dynamic weighting of the results. The dynamic weighting weights each static instruction by the number of times it is executed. For instance, if a program contains two memory instructions, *A* and *B*, we correctly predict the result for instruction *A* and incorrectly predict the result for instruction *B*, and instruction *A* is executed 80 times and instruction *B* is executed 20 times, we have an 80% dynamic prediction accuracy.

In the remainder of this paper, we present the data in both textual and tabular form. While the most important information is discussed in the text, the tables are provided for completeness and to give a summary view of the performance of our techniques.

3.2 Reuse Distance Prediction Results

This section reports statistics on reuse distance distribution, and our prediction accuracy and coverage. Tables 1 and 2 list reuse distance distribution, the prediction coverage and accuracy on a per instruction basis. For both floating point and integer programs, over 80% reuse distances remain constant with respect to the varied inputs and 5 to 7% of distances are linear to the data size, although both percentages for integer programs are significantly lower than those of floating-point programs. A significant number of other patterns exist in some programs. For example, in 183.equake, 13.6% of the patterns exhibit a square root (sqrt) distribution pattern. For 200.sixtrack and 186.crafty, we do not report the patterns since all data sizes are identical. Our model predicts all constant patterns.

For floating-point benchmarks, the dynamically weighted coverage is 93.0% on average, improving over the 91.3% average of Fang et al. [5]. In particular, the coverage of 188.ammp is improved from 84.7% to 96.7%. For all floating-point programs except 189.lucas, the dynamic coverage is well over 90%. In 189.lucas, approximately 31% of the static 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. The average prediction accuracy and coverage of integer programs are lower than those of floating-point programs but still over 90%. The low coverage of 164.gzip occurs because the reuse distance for the test run is greater than that for train. This occurs because of the change in alignment of structures in a cache line with the change in data size.

As mentioned previously, an instruction is not covered if one of the three following conditions is not satisfied: (1) the instruction does not occur in at least one training run, (2) the reuse distance of test is larger than that for train, or (3) the number of patterns for the instruction does not remain constant in both training runs. Overall, an average of 4.2% of the instructions in CFP2000 and 2.2% of the instruction in CINT2000 fall into the first category. Additionally, 0.3% and 4.4% of the instructions fall into the second category for CFP2000 and CINT2000, respectively. Finally, 2.5% of the CFP2000 instructions and 1.8% of the CINT2000 instructions fall into the third category.

Benchmark	Patterns		Coverage	Accuracy
	%constant	%linear	(%)	(%)
168.wupwise	95.4	3.9	97.4	99.7
171.swim	83.7	10.7	98.4	92.1
172.mgrid	85.8	3.8	93.9	97.8
173.applu	80.2	5.0	95.9	97.8
177.mesa	92.3	4.1	97.8	99.9
179.art	88.4	5.2	99.7	98.3
183.equake	76.9	8.1	97.0	97.4
188.ammp	85.2	10.3	96.7	97.0
189.lucas	81.3	13.5	52.0	98.3
200.sixtrack	N/A	N/A	99.9	99.9
301.apsi	82.0	12.0	94.0	95.6
average	85.1	7.7	93.0	97.6

Table 1. CFP2000 reuse distance prediction

Benchmark	Patterns		Coverage	Accuracy
	%constant	%linear	(%)	(%)
164.gzip	82.3	3.4	74.9	94.7
175.vpr	86.3	7.2	98.8	94.2
176.gcc	76.9	3.1	97.8	94.7
181.mcf	63.6	10.7	82.9	88.0
186.crafty	N/A	N/A	97.5	95.8
197.parser	77.4	5.9	94.8	96.6
252.eon	95.9	2.4	99.4	99.7
254.gap	79.8	3.5	85.2	93.3
255.vortex	89.9	1.1	97.3	92.0
256.bzip2	87.6	3.0	88.0	91.4
300.twolf	72.4	10.6	91.2	91.7
average	81.2	5.1	91.6	93.8

Table 2. CINT2000 reuse distance prediction

For floating-point benchmarks, our model predicts reuse distance correctly for 97.6% of the covered instructions on average, slightly improving the 96.7% obtained by Fang [5]. It predicts the reuse distance accurately for over 95% of the covered instructions for all programs except 171.swim which is the only benchmark on which we observe significant over-merging. For integer programs, our prediction accuracy for the covered instructions remains high with 93.8% on average and the lowest is 181.mcf which gives 88%. One major reason for the accuracy loss on 181.mcf is because several reuse patterns in the reference run would require super-linear pattern modeling which we do not use. The other major loss is from the cache-line alignment of a few instructions where we predict a positive distance which indeed is zero for the reference run.

In addition to measuring the prediction coverage and accuracy, we measured the number of locality patterns exhibited by each instruction. Table 3 below shows the average percentage of instructions that exhibit 1, 2, 3, 4, or more patterns during execution. On average, over 92% of the instructions in floating-point programs and over 83% in integer programs exhibit only one or two reuse-distance patterns. This information shows that most instructions have highly focused reuse patterns.

Benchmark	1	2	3	4	≥ 5
CFP2000	81.8	10.5	4.8	1.4	1.5
CINT2000	72.3	10.9	7.6	4.6	5.3

Table 3. Number of locality patterns

To evaluate the effect of merging bins as discussed in Section 3, we report how often instructions whose reuse pattern crosses the original 1K boundaries are merged into a single pattern. On average 14.1% and 30.8% of the original bins are merged for CFP2000 and CINT2000, respectively. This suggests that the distances in floating-point programs are more discrete while they are more continuous in integer programs. For both integer and floating-point programs, the merging significantly improves our reuse distance and miss rate prediction accuracy.

4. Miss Rate Prediction

Given the predicted reuse distance distribution, we can predict the miss rates of the instructions in a program. For a fully associative cache of a given size, we predict a cache miss for a reference to a particular cache line if the reuse distance to its previous access is greater than the cache size. For set associative caches, we predict the miss rate as if it were a fully associative cache. This model catches the compulsory and capacity misses, but neglects conflict misses.

If the minimum distance of a pattern is greater than the cache size, all accesses in the pattern are considered misses. When the cache size falls in the middle of a pattern, we estimate the miss rates by computing the percentage of the area under the pattern curve that falls to the right of the cache size.

In our analysis, miss-rate prediction accuracy is calculated as

$$1 - \frac{|\, actual - predicted \,|}{\max(actual, predicted)}.$$

We glean the actual rates through cache simulation using the same input. Although the predicted miss rate does not include conflict misses, the actual miss rate does. While cache conflicts may affect miss rates significantly in some circumstances, reuse distance alone will not capture conflicts since we assume a fully associative cache. For the SPEC2000 benchmarks that we analyzed, in spite of not incorporating conflict misses in the prediction, our prediction of miss rates is highly accurate. Note that the prediction for L2 cache is identical to that for L1 cache with the predicted L1 cache hits filtered out.

The miss rates reported include all instructions, whether or not they are covered by our prediction mechanism. If the instruction's reuse distance is predictable, then we use the predicted reuse distance distribution to determine the miss rate. If the instruction appears in at least one training run and its reuse distance is not predictable, we use the reuse distance of the larger of the training runs to predict the miss rate. If the instruction does not appear in either training run, we predict a miss rate of 0%.

4.1 Experimental Methodology

For miss-rate prediction measurements, we have implemented a cache simulator and embedded it in our analysis routines to collect the number of L1 and L2 misses for each instruction. We use a 32K, 2-way set associative L1 cache and a 1MB, 4-way set associative L2 cache. Each of the cache configurations uses 64-byte lines and an LRU replacement policy.

To compare the effectiveness of our miss-rate prediction, we have 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 and verify our miss rate prediction using the reference input sets. The second scheme, called *reference reuse distance* (RRD), 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. For comparison, we report L2 miss rate and critical instruction prediction using Fang's approach that assumes a uniform distribution of reuse distances in a pattern (U-PRD) [5].

4.2 Miss-rate Prediction Accuracy

Table 4 reports our miss-rate prediction accuracy for an L1 cache. Examining the table reveals that our prediction method (PRD) predicts the L1 miss rate of instructions with an average 97.5% and 94.4% accuracy for floating-point and integer programs, respectively. On average PRD more accurately predicts the miss rate than TCS, but is slightly less accurate than RRD. Even though TCS can consider conflict misses, PRD still outperforms it on average. Conflict misses tend to be more pronounced in the integer benchmarks, yielding a lower improvement of PRD over TCS on integer codes. In general, PRD does better when the data size increases significantly since PRD can capture the effects of the larger data sets. TCS does better when the data sizes between test, train and reference are similar since TCS includes conflict misses.

Suite	PRD	RRD	TCS
CFP2000	97.5	98.4	95.1
CINT2000	94.4	96.7	93.9

Table 4. L1 miss rate prediction accuracy

Table 5 presents our prediction accuracies for our L2 cache configuration for floating-point and integer programs, respectively. Table 6 provides a summary of the results for three other L2 associativities. As can be seen, these results show that PRD is effective in predicting L2 misses for a range of associativities. We will limit our detailed discussion to the 4-way set associative cache. On average, smaller associativity sees slightly worse results.

PRD has a 92.1% and 92.4% miss-rate prediction accuracy for floating-point and integer programs, respectively. PRD outperforms TCS on all programs in CFP2000 except 189.lucas and 200.sixtrack. In general, the larger reuse distances are handled much better with PRD than TCS, giving the larger increase in prediction accuracy compared to the L1 cache. For 200.sixtrack, the data size does not change, so TCS outperforms both PRD and RRD. For 189.lucas, a significant number of misses occur for instructions that do not appear in either training run.

CFP2000	U-PRD	PRD	RRD	TCS	CINT2000	U-PRD	PRD	RRD	TCS
168.wupwise	97.7	98.2	98.9	95.2	164.gzip	98.0	99.3	99.9	99.9
171.swim	91.7	92.8	98.0	86.0	175.vpr	90.1	95.1	96.0	90.0
172.mgrid	97.3	97.6	99.3	90.3	176.gcc	88.8	92.0	95.5	89.9
173.applu	96.6	97.3	99.0	91.1	181.mcf	59.4	67.3	93.8	46.8
177.mesa	92.8	97.2	97.2	95.8	186.crafty	99.9	99.9	99.9	99.9
179.art	82.6	81.5	81.6	78.7	197.parser	79.2	91.4	96.6	88.7
183.equake	93.1	94.3	95.0	85.9	252.eon	99.9	99.9	99.9	99.9
188.ammp	82.6	82.7	84.4	81.5	254.gap	76.9	86.6	94.3	86.0
189.lucas	82.7	83.4	92.1	90.6	255.vortex	90.8	97.6	99.6	97.7
200.sixtrack	95.9	95.9	95.9	98.1	256.bzip2	93.7	95.4	98.6	94.9
301.apsi	92.3	92.6	93.6	88.9	300.twolf	91.8	92.4	95.7	88.9
average	91.4	92.1	94.1	89.3	average	88.0	92.4	97.3	89.3

Table 5. 4-way L2 miss rate prediction accuracy

Suite	2-way			8-way			FA		
	PRD	RRD	TCS	PRD	RRD	TCS	PRD	RRD	TCS
CFP2000	91.0	93.0	87.1	92.4	94.4	88.4	96.8	99.9	91.2
CINT2000	90.6	94.7	87.5	92.6	97.5	89.7	93.6	99.9	89.1

Table 6. Effect of associativity on L2 miss rate prediction accuracy

For CINT2000, PRD outperforms TCS on all programs except 164.gzip where the gain of TCS is negligible. For 164.gzip, the L2 miss rate is quite low (0.02%). In addition, the coverage is low because the reuse distance for the test dataset for some instructions is larger than the reuse distance for train due to a change in alignment in the cache line. As a result, TCS is better able to predict the miss rate since PRD will overestimate the miss rate.

PRD outperforms U-PRD for all programs except 179.art. For this program, U-PRD predicts a larger miss rate, but due to conflict misses, the miss rate is realized. The difference between PRD and U-PRD is more pronounced for integer programs than floating-point programs. This shows that assuming a uniform distribution of reuse distances in a pattern leads to less desirable results. This difference in effectiveness becomes more pronounced when identifying critical instructions as shown in the next section.

In general, PRD is much more effective than TCS for large reuse distances. This is extremely important since identifying L2 misses is significantly more important than L1 misses because of the miss latency difference. In the next section, we show that TCS is inadequate for identifying the most important L2 misses and that PRD is quite effective.

4.3 Identifying Critical Instructions

For static or dynamic optimizations, we are interested in the *critical* instructions which generate a large fraction (95%) of the cumulative L2 misses. In this section, we show that we can predict most of the critical instructions accurately. We also observe that the locality patterns of the critical instructions tend to be more diverse than non-critical instructions and tend to exhibit fewer constant patterns.

To identify the actual critical instructions, we perform cache simulation on the reference input. 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. Table 7 presents the percentage of critical instructions identified using all four prediction mechanisms for our cache configuration. Additionally, the table reports the percentage of loads predicted as critical (%pred) by PRD and the percentage of actual critical loads (%act).

The prediction accuracy for critical instructions is 92.2% and 89.2% on average for floating-point and integer programs, respectively. 189.lucas shows a very low accuracy because of low prediction coverage. The unpredictable instructions in 189.lucas contribute a significant number of misses. The critical instruction accuracy for 181.mcf is lower than average because two critical instructions are not predictable. In the train run for 181.mcf, the instructions exhibit a reuse distance of 0. However, in the test run, the reuse distance is very large. This is due to the fact that the instructions reference data contained within a cache line in the train run and data that appear in different cache lines in the test run due to the data alignment of the memory allocator. In 256.bzip2, a number of the critical instructions only appear in the train data set. For this data set, these instructions do not generate L2 misses and are, therefore, not critical. Since we use the train reuse distance to predict misses in this case, our mechanism is unable to identify these instructions as critical. For 300.twolf, a number of the critical instructions have unpredictable patterns. This makes predicting the reference reuse distance difficult and prevents PRD from recognizing these instructions as critical. Note that we do not report statistics for 252.eon because the L2 miss rate is nearly 0%.

Comparing the accuracy of TCS in identifying critical instructions, we see that TCS is considerably worse when compared with its relative miss-rate prediction accuracy. This is because TCS mis-predicts the miss rate more often for the longer reuse distance instructions (more likely critical) since its prediction is not sensitive to data size. U-PRD performs significantly worse than PRD, on average, for CINT2000. This is because the enhanced pattern formation presented in Section 3 is able to characterize the reuse distance patterns better in integer programs. For 181.mcf and 254.gap, U-PRD identifies more of the actual critical loads, but it also identifies a higher percentage of loads as critical that are not critical. In general, U-PRD identifies 1.6 times as many false

CFP2000	U-PRD	PRD	RRD	TCS	%pred	%act	CINT2000	U-PRD	PRD	RRD	TCS	%pred	%act
168.wupwise	99.9	99.9	99.9	88.3	0.77	0.77	164.gzip	1.2	92.9	99.9	0.0	0.59	0.80
171.swim	99.9	99.9	99.9	99.9	3.61	3.09	175.vpr	67.8	89.9	94.4	0.0	0.30	0.45
172.mgrid	99.7	99.9	99.9	55.9	2.61	2.11	176.gcc	78.5	96.5	99.6	87.3	1.22	1.27
173.applu	98.0	98.5	99.9	85.5	2.23	1.78	181.mcf	80.1	73.3	99.9	28.1	2.18	1.50
177.mesa	99.9	99.9	99.9	99.9	0.06	0.06	186.crafty	97.1	97.1	97.2	99.9	0.4	0.49
179.art	99.9	99.9	99.9	96.4	1.82	0.83	197.parser	81.7	96.6	98.9	67.3	1.16	1.14
183.equake	91.4	95.9	99.6	0.0	2.35	2.52	252.eon	–	–	–	–	–	–
188.ammp	90.2	90.9	96.3	10.9	0.41	0.41	254.gap	96.9	93.2	99.7	56.5	0.22	0.17
189.lucas	24.1	35.2	99.9	5.0	1.77	4.54	255.vortex	59.1	98.1	98.9	97.8	0.32	0.15
200.sixtrack	98.7	98.7	91.5	21.6	1.05	0.60	256.bzip2	65.9	82.5	99.9	84.2	1.07	1.65
301.apsi	89.9	95.9	94.5	0.0	1.51	1.56	300.twolf	69.8	72.0	96.0	6.1	0.99	1.12
average	90.2	92.2	98.3	51.2	1.66	1.67	average	63.5	89.2	98.4	52.7	0.94	0.97

Table 7. 4-way set-associative L2 critical instruction prediction comparison

critical instructions compared to PRD, even though the absolute number is quite low on average for both techniques.

We tested critical instruction prediction on the other three associativities listed in Table 6 and, on average, the associativity of the cache does not affect the accuracy of our prediction for critical instructions significantly. The only noticeable difference occurred on the 2-way set associative cache for 301.apsi, 175.vpr and 186.crafty. For this cache configuration, conflict misses play a larger role for these three applications, resulting in a lower critical instruction prediction accuracy.

Finally, Table 7 shows 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 8 reports the distribution of the number of locality patterns for critical instructions using dynamic weighting. We find that the distribution is more diverse than that shown in Table 3. Although less than 20% of the instructions on average have more than 2 patterns, the average goes up to over 40% when considering only critical instructions.

Benchmark	1	2	3	4	≥ 5
CFP2000	22.1	38.4	20.0	12.8	6.7
CINT2000	18.7	14.5	25.5	22.5	18.0

Table 8. Critical instruction locality patterns

Critical instructions also tend to exhibit a higher percentage of non-constant patterns than non-critical instructions. Critical instructions in CFP2000 have an average of 12.7% all constant patterns and an average of 10.8% in CINT2000. Since this data reveals that critical instructions are more sensitive to data size, it is important to predict reuse distance accurately in order to apply optimization to the most important memory operations.

5. Memory Disambiguation

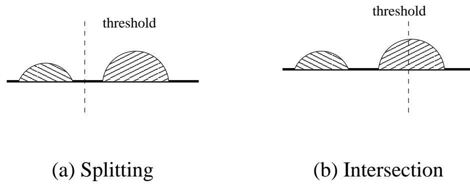
Mis-speculation of memory operations can counteract the performance advantage of speculative execution. When a mis-speculation occurs, the speculative load and dependent instructions need to be re-executed to restore the state. Therefore, a good memory disambiguation strategy is critical for the performance of speculative execution. This section describes a novel profile-based memory disambiguation technique based on the instruction-based memory distance prediction model discussed in Section 3 with a few extensions. In this section, we introduce two new forms of memory distance – *access distance* and *value distance* – and explore the potential of using them to determine which loads in a program may be speculated. The access distance of a memory reference is the number of memory instructions between a store to and a load from the same address. The value distance of a reference is defined as the access distance of a load to the first store in a sequence of stores of the same value. Differing from cache miss prediction which is sensitive to relatively large distances, we focus on shorter access and value distances that may cause memory order violations.

5.1 Access Distance and Speculation

For speculative execution, if a load is sufficiently far away from the previous store to the same address, the load will be a good speculative candidate. Otherwise, it will likely cause a mis-speculation and introduce penalties. The possibility of a mis-speculation depends on the distance between the store and the load as well as the instruction window size, the load/store queue size, and machine state. Taking all these factors into account, we examine the effectiveness of access distance in characterizing memory dependences. Although it is also advisable to consider *instruction distance* (the number of instructions between two references to the same address) with respect to instruction window size, we observe that instruction distance typically correlates well to access distance and using access distance only is sufficient.

When we know ahead of real execution the backward access distance of a load, we can mark the load *speculative* if the distance is greater than a threshold. We mark the load as *non-speculative*, otherwise. During execution, only marked speculative loads are allowed for speculative scheduling. In Section 5.4, our experimental results show that a threshold value of 10 for access distance yields the best performance for our system configuration.

The access distance prediction is essentially the same as the reuse distance prediction. Instead of collecting reuse distances in the training runs, we need to track access distances. A difficulty here is that we need to mark speculative loads before the real execution using the real inputs. Reuse distance prediction in Section 3 uses sampling at the beginning of the program execution to detect the data-set size and then applies prediction to the rest of the execution. For a system supporting adaptive compilation, the compiler may mark loads after the input data size is known and adaptively apply access distance analysis. In our method, we do



(a) Splitting (b) Intersection

Figure 3. PMSF Illustration

not require knowledge of the data size ahead of the real execution and thus do not require either sampling or adaptive compilation. Instead, we base our access-distance prediction solely on two training runs.

Our method collects the access distances for two training runs and then predicts the access distance pattern for each load instruction for a presumably larger input set of unknown size. Two facts suggested by Tables 1 and 2 make this prediction plausible: most access distances are constant across inputs and a larger input typically increases the non-constant distances. Since a constant pattern does not change with respect to the data size, the access distance is predictable without data-size sampling. We also predict a lower bound for a non-constant access distance assuming that the new input size is larger than the training runs. Since the fitting functions are monotonically increasing, we take the lower bound of the access distance pattern for the larger training set as the lower bound on the access distance. If the predicted lower bound is greater than the speculation threshold, we mark the load as *speculative*.

We define the *predicted mis-speculation frequency* (PMSF) of a load as the frequency of occurrences of access distances less than the threshold. We mark a load as *speculative* when its PMSF is less than 5%. The PMSF of a load is the ratio of the frequencies of the patterns on the left of the threshold over the total frequencies. When the patterns are all greater or all less than the threshold, it is straightforward to mark the instruction as speculative or non-speculative, respectively. For the cases illustrated by Figures 3(a) and 3(b), the threshold sits between patterns or intersects one of the patterns. We presume that the occurrences of distances less than the threshold will more likely cause mis-speculations but the occurrences greater than the threshold can still bring performance gains. When the threshold does not intersect any of the access distance patterns, the PMSF of a load is the total frequencies of the patterns less than the threshold divided by the total frequency of all patterns. When the threshold value falls into a pattern, we calculate the mis-speculation frequency of that pattern as

$$\frac{(threshold - min)}{(max - min)} * frequency\ of\ the\ pattern.$$

5.2 Value Distance and Speculation

Önder and Gupta [17] have shown that when multiple successive stores to the same address write the same value, a subsequent load to that address may be safely moved prior to all of those stores except the first as long as the memory order violation detection hardware examines the values of loads and stores. Given the following sequence of memory operations,

1: store a_1, v_1
2: store a_2, v_2
3: store a_3, v_3
4: load a_4, v_4

where a_1 through a_4 are memory addresses and v_1 through v_4 are the values associated with those addresses. If $a_1 = a_2 = a_3 = a_4$, $v_2 = v_3$ and $v_1 \neq v_2$, then the load may be moved ahead of the third store, but not the second using a value-based approach.

We call the access distance of a load to the first store in a sequence of stores of the same value the *value distance* of that load. To compute the value distance of a load, we modify our access distance tool to ignore subsequent stores to the same memory location with the same value. In this way, we only keep track of the stores that change the value of the memory location.

Similar to access distance prediction, we can predict value distance distribution for each instruction. Note that the value distance of an instance of a load is no smaller than the access distance. By using value distances and the supporting hardware, we can mark more instructions as *speculative*.

5.3 Experimental Design

To examine the performance of memory distance based memory disambiguation, we use the FAST micro-architectural simulator based upon the MIPS instruction set [16]. The simulated architecture is an out-of-order superscalar pipeline which can fetch, dispatch and issue 8 operations per cycle. A 128 instruction central window, and a load store queue of 128 elements are simulated. Two memory pipelines allow simultaneous issuing of two memory operations per cycle, and a perfect data cache is assumed. The assumption of perfect cache eliminates ill effects of data cache misses which would affect scheduling decisions as they may alter the order of memory operations. We believe the effectiveness of any memory dependence predictor should be evaluated upon whether or not the predictor can correctly identify the times that load instructions should be held and the times that the load instructions should be allowed to execute speculatively. However, for completeness we also examine the performance of the benchmark suite when using a 32KB direct-mapped non-blocking L1 cache with a latency of 2 cycles and a 1 MB 2-way set associative LRU L2 cache with a latency of 10 cycles. Both caches have a line size of 64 bytes.

For our test suite, we use a subset of the C and Fortran 77 benchmarks in the SPEC CPU2000 benchmark suite. The programs missing from SPEC CPU2000 include all Fortran 90 and C++ programs, for which we have no compiler, and five programs (254.gap, 255.vortex, 256.bzip2, 200.sixtrack and 168.wupwise) which could not be compiled and run correctly with our simulator. For compilation, we use gcc-2.7.2 with the -O3 optimization flag. Again, we use the test and train input sets for training and generating hints, and then test the performance using the reference inputs.

Since we perform our analysis on MIPS binaries, we cannot use ATOM as is done in Section 3. Therefore, we add the same instrumentation to our micro-architectural simulator to gather memory distance statistics. To compute which loads should be speculated we augment the MIPS instruction set with an additional opcode to indicate a load that may be speculated.

5.4 Results

In this section, we report the results of our experiment using access distance for memory disambiguation. Note that we do not report access and value distance prediction accuracy since the re-

sults are similar to those for reuse distance prediction. Given this, we report the raw IPC data using a number of speculation schemes.

5.4.1 IPC with Address-Based Exception Checking

We have run our benchmark suite using five different memory disambiguation schemes: access distance, no speculation, blind speculation, perfect disambiguation and store sets using varied table sizes [3]. The no-speculation scheme always assumes a load and store are dependent and the blind-speculation scheme always assumes that a load and store are independent. Perfect memory disambiguation never mis-speculates with the assumption that it always knows ahead the addresses accessed by a load and store operation. The store set schemes use a hardware table to record the set of stores with which a load has experienced memory-order violations in the past. Figures 4 and 5 report the raw IPC data for each scheme where only address-based exception checking is performed.

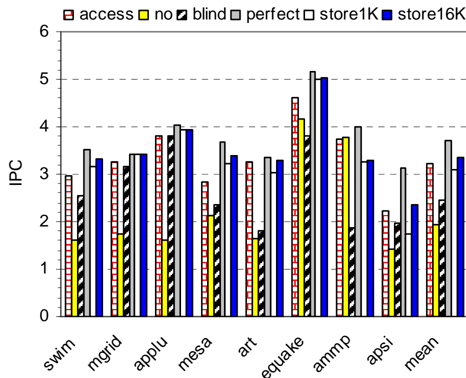


Figure 4. CFP2000 address-based IPC

As can be seen in Figure 4, on the floating-point programs, the access-distance-based memory disambiguation scheme achieves a harmonic mean performance that is between 1K-entry and 16K-entry store set techniques. It reduces the 34% performance gap for blind speculation to 13% with respect to the perfect memory disambiguation. It also performs within 5% of the 16K-entry store set. This 5% performance gap is largely from 171.swim, 177.mesa, and 183.equake, where the 16K store set outperforms our profile-based scheme by at least 8%. For these three benchmarks, we observe that the access-distance-based scheme suffers over a 1% miss speculation rate. A special case is 188.ammp, for which all speculation schemes degrade the performance. The 16K store set degrades performance by 13%. The access-distance-based scheme lowers this performance degradation to less than 1%. 188.ammp has an excessive number of short distance loads. The access-distance-based technique blocks speculations for these loads. Although the store set scheme does not show a substantially higher number of speculated loads, we suspect that its performance loss stems from some pathological mis-speculations where the penalty is high.

Figure 5 reports performance for the integer benchmarks. The average gap between blind speculation and the perfect scheme is 23%, compared to an average 34% performance gap for CFP2000, suggesting a smaller improvement space. The blind scheme is marginally better than no speculation. This negligible improvement is due to high mis-speculation rates and fewer opportunities for speculation. The access-distance-based scheme reduces the 23% performance gap of blind speculation with respect to perfect disambiguation to 13%. Access distance performs close to a 1K-entry store set scheme and within 10% of the 16K-entry scheme. Three benchmarks, 164.gzip, 176.gcc, and 300.twolf, contribute most of this performance disparity. These three benchmarks show the highest mis-speculation rates for the access-distance scheme.

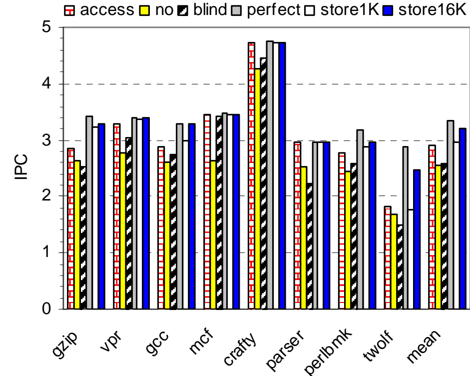


Figure 5. CINT2000 address-based IPC

The mis-speculation rates for the memory-distance schemes are generally higher than those of store set, but much lower than those of blind speculation. The relative high mis-speculation rate of the profile-based schemes are mostly because they cannot adjust to dynamic program behaviors. Our memory-distance schemes mark a load as non-speculative when 95% of its predicted memory distances are greater than a threshold. This could cause up to 5% mis-speculation of an instruction. The mis-speculation rate and performance are sensitive to the threshold values. We examined thresholds of 4, 8, 10, 12, 16, 20 and 24. On average, a threshold value of 10 is the best. However, other thresholds yield good results for some individual benchmarks. For instance, 177.mesa favors a threshold of 12.

Table 9 gives the harmonic mean IPC of our benchmark suite using address-based exception checking with the cache model instead of a perfect memory hierarchy. As can be seen by the results, the relative performance of our technique remains similar for CFP2000, but improves for CINT2000. The performance improves because cache misses hide the effects of the reduced prediction accuracy obtained by our access distance model.

Bench	no	access	store set	
			1KB	16KB
CFP2000	0.91	1.55	1.45	1.61
CINT200	1.13	1.53	1.43	1.60

Table 9. Address-based IPC with Cache

5.4.2 IPC with Value-Based Exception Checking

Value-distance-based speculation, the store set technique, and blind speculation can all take advantage of value-based exception checking in order to reduce memory order violations. Figures 6 and 7 show the performance of these three schemes where the value-based exception checking is used. Table 10 reports the

harmonic mean IPC achieved using the cache model instead of the perfect memory hierarchy. For all schemes, on average, the value-based exception checking improves performance over the corresponding address-based schemes since some of the address conflicts can be ignored due to value redundancy.

For floating-point benchmarks, blind speculation gains over 12% because of a significant reduction in the mis-speculation rate. On average, the value-distance-based scheme and store set improve 3 to 5%. Although the value-distance scheme still performs below the store set technique, value-distance prediction is still needed when using value-based exception checking.

For integer programs, the improvement obtained by using value-based exception checking is notably smaller than that for floating-point programs. The value-distance scheme shows an improvement of 3% while the store set techniques all improve less than 2.5%. We attribute this to fewer value redundancies in integer benchmarks and the smaller performance gap between blind speculation and perfect memory disambiguation.

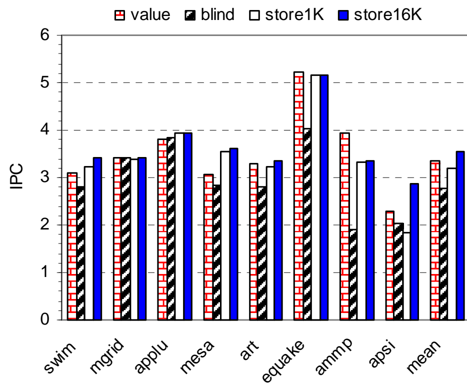


Figure 6. CFP2000 value-based IPC

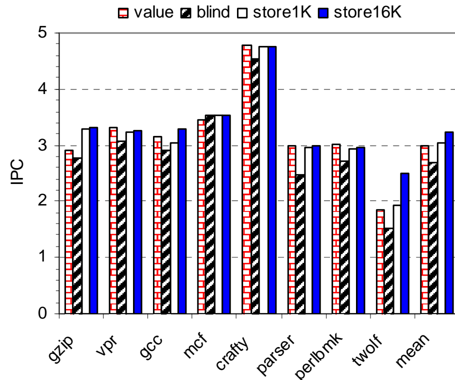


Figure 7. CINT2000 value-based IPC

			store set	
Bench	**no**	**value**	1KB	16KB
CFP2000	0.91	1.59	1.52	1.63
CINT200	1.13	1.55	1.48	1.65

Table 10. Value-based IPC with Cache

6. Related Work

In addition to the work discussed in Section 3, Ding et al. predict reuse distances to estimate the capacity miss rates of a fully associative cache [24], to perform data transformations [25] and to predict the locality phases of a program [19]. Beyls and D'Hollander detect reuse distance 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 used to generate their hints. Marin and Mellor-Crummey [10] use instruction-based reuse distance in the prediction of application performance. Their analysis may require significantly more space than ours. Pinait, et al. [18], statically identify critical instructions by analyzing the address arithmetic for load operations.

Cache simulation can supply accurate miss rates and even performance impact for a cache configuration; however, the simulation itself is costly and impossible to apply during dynamic optimization on the fly. Mattson, et al., present a stack algorithm to measure cache misses for different cache sizes in one run [11]. Sugumar and Abraham [22] use Belady's algorithm to characterize capacity and conflict misses. They present three techniques for fast simulation of optimal cache replacement.

Many static models of locality exist and may be utilized by the compiler to predict cache misses [2, 6, 12, 13, 23]. Each of these models is restricted in the types of array subscript and loop forms that can be handled. Furthermore, program inputs, which determine, for instance, symbolic bounds of loops, remain a problem for all aforementioned static analyses.

Work in the area of dynamic memory disambiguation has yielded increasingly better results [3, 7, 14]. Moshovos and Sohi have studied memory disambiguation and the communication through memory extensively [14]. The predictors they have designed aim at precisely identifying the load/store pairs involved in the communication. Various patents [21, 7] also exist which identify those loads and stores that cause memory order violations and synchronizing them when they are encountered.

Chrysos and Emer [3] introduce the store set concept which allows using direct mapped structures without explicitly aiming to identify the load/store pairs precisely. Önder [15] has proposed a light-weight memory dependence predictor which uses multiple speculation levels in the hardware to direct load speculation. Önder and Gupta [17] have shown that the restriction of issuing store instructions in-order can be removed and store instructions can be allowed to execute out-of-order if the memory order violation detection mechanism is modified appropriately. Furthermore, they have shown that memory order violation detection can be based on values, instead of addresses. Our work in this paper uses this memory order violation detection algorithm.

7. Conclusions and Future Work

In this paper, we have demonstrated that memory distance is predictable on a per instruction basis for both integer and floating-point programs. On average, over 90% of all memory operations executed in a program are predictable with a 97% accuracy for floating-point programs and a 93% accuracy for integer programs. In addition, the predictable reuse distances translate to predictable miss rates for the instructions. For a 32KB 2-way set associative

L1 cache, our miss-rate prediction accuracy is 96% for floating-point programs and 89% for integer programs, and for a 1MB 4-way set associative L2 cache, our miss-rate prediction accuracy is over 92% for floating-point and integer programs. Most importantly, our analysis accurately 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 a 92% accuracy for floating-point programs and a 89% accuracy for integer programs for a 1MB 4-way set associative L2 cache. In addition to predicting large memory distances accurately for critical instruction detection, we have shown that our analysis can effectively predict small reuse distances. Our experiments show that without a dynamic memory disambiguator we can disambiguate memory references using access and value distance and achieve performance within 5-10% of a store-set predictor.

The next step in our research will apply critical instruction detection to cache optimization. We are currently developing a mechanism based upon informing memory operations [8] to overlap both cache misses and branch misprediction recovery. We also believe that our work in memory disambiguation has significant potential for EPIC architectures where the compiler is completely responsible for identifying and scheduling loads for speculative execution. We are currently applying memory-distance-based memory disambiguation to speculative load scheduling for the Intel IA-64. We expect that significant performance improvement will be possible with our technique.

In order for significant gains to be made in improving program performance, compilers must improve the performance of the memory subsystem. 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 and memory disambiguation through the use of memory distance.

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] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th International Conference on Computer Architecture*, pages 142–153, June 1998.

[4] 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.

[5] C. Fang, S. Carr, S. Önder, and Z. Wang. Reuse-distance-based miss-rate prediction on a per instruction basis. In *Proceedings of the 2nd ACM Workshop on Memory System Performance*, pages 60–68, Washington, D.C., June 2004.

[6] 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.

[7] J. Hesson, J. LeBlanc, and S. Ciavaglia. Apparatus to dynamically control the Out-Of-Order execution of Load-Store instructions. *US. Patent* 5,615,350, Filed Dec. 1995, Issued Mar. 1997.

[8] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing memory operations: memory performance feedback mechanisms and their applications. *ACM Trans. Comput. Syst.*, 16(2):170–205, 1998.

[9] W.-C. Hsu, H. Chen, P.-C. Yew, and D.-Y. Chen. On the predictability of program behavior using different input data sets. In *Proceedings of the Sixth Annual Workshop on Interaction between Compilers Computer Architectures*, 2002.

[10] G. Marin and J. Mellor-Crummey. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, New York, NY, June 2004.

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

[12] 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.

[13] 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.

[14] A. I. Moshovos. *Memory Dependence Prediction*. PhD thesis, University of Wisconsin - Madison, 1998.

[15] S. Önder. Cost effective memory dependence prediction using speculation levels and color sets. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 232–241, Charlottesville, Virginia, September 2002.

[16] S. Önder and R. Gupta. Automatic generation of microarchitecture simulators. In *IEEE International Conference on Computer Languages*, pages 80–89, Chicago, May 1998.

[17] S. Önder and R. Gupta. Dynamic memory disambiguation in the presence of out-of-order store issuing. *Journal of Instruction Level Parallelism*, Volume 4, June 2002. (www.microarch.org/vol4).

[18] V.-M. Pinait, A. Sasturkar, and W.-F. Wong. Static identification of delinquent loads. In *Proceedings of the International Symposium on Code Generation and Optimization*, San Jose, CA, Mar. 2004.

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

[20] 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.

[21] S. Steely, D. Sager, and D. Fite. Memory reference tagging. *US. Patent* 5,619,662, Filed Aug. 1994, Issued Apr. 1997.

[22] 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.

[23] 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.

[24] 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.

[25] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Washington, D.C., June 2004.