Path-Based Reuse Distance Analysis*

Changpeng Fang[1], Steve Carr[2], Soner Önder[2], and Zhenlin Wang[2]

[1] PathScale, Inc., 2071 Stierlin Court Ste. 200, Mountain View, CA 94043, USA
cfang@pathscale.com
[2] Department of Computer Science, Michigan Technological University, Houghton, MI 49931, USA
{carr, soner, zlwang}@mtu.edu

Abstract. Profiling can effectively analyze program behavior and provide critical information for feedback-directed or dynamic optimizations. Based on memory profiling, reuse distance analysis has shown much promise in predicting data locality for a program using inputs other than the profiled ones. Both whole-program and instruction-based locality can be accurately predicted by reuse distance analysis.

Reuse distance analysis abstracts a cluster of memory references for a particular instruction having similar reuse distance values into a locality pattern. Prior work has shown that a significant number of memory instructions have multiple locality patterns, a property not desirable for many instruction-based memory optimizations. This paper investigates the relationship between locality patterns and execution paths by analyzing reuse distance distribution along each dynamic path to an instruction. Here a path is defined as the program execution trace from the previous access of a memory location to the current access. By differentiating locality patterns with the context of execution paths, the proposed analysis can expose optimization opportunities tailored only to a specific subset of paths leading to an instruction.

In this paper, we present an effective method for path-based reuse distance profiling and analysis. We have observed that a significant percentage of the multiple locality patterns for an instruction can be uniquely related to a particular execution path in the program. In addition, we have also investigated the influence of inputs on reuse distance distribution for each path/instruction pair. The experimental results show that the path-based reuse distance is highly predictable, as a function of the data size, for a set of SPEC CPU2000 programs.

1 Introduction

The ever-increasing disparity between memory and CPU speed has made effective operation of the memory hierarchy the single most critical element in the performance of many applications. To address this issue, compilers attempt to manipulate the spatial and temporal reuse in programs to make effective use of the cache. While static compiler analysis has achieved some success in improving memory performance, limited compile-time knowledge of run-time behavior decreases the effectiveness of static analysis. Profile analysis of a single run can yield more accurate information, however,

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

A. Mycroft and A. Zeller (Eds.): CC 2006, LNCS 3923, pp. 32–46, 2006.

the use of a single run does not catch memory issues that are sensitive to program input size.

To address the lack of sensitivity to data size in profile analysis, Ding et al. [1, 2] have developed techniques to predict *reuse distance* – the number of unique memory locations accessed between two references to the same memory location. Given the reuse distances of memory locations for two training runs, they apply curve fitting to determine the reuse distance for a third input using the data size of the third input. Ding et al. show that reuse distance is highly predictable given the data size of an input set.

Our previous work [3, 4] maps the reuse distance of memory locations to the instructions that reference those locations and shows that the reuse distance of memory instructions is also highly predictable. Our analysis abstracts a cluster of memory references for a particular instruction having similar reuse distance values into a locality pattern. The results of the analysis show that many memory operations exhibit more than one locality pattern, often with widely disparate reuse distances. Unfortunately, optimization based upon reuse distance often favors reuse distances to be consistently either large or small, but not both, in order to improve memory-hierarchy performance effectively.

In this paper, we extend our previous work to use execution-path history to disambiguate the reuse distances of memory instructions. Specifically, we relate branch history to particular locality patterns in order to determine exactly when a particular reuse distance will be exhibited by a memory operation. Our experiments show that given sufficient branch history, multiple locality patterns for a single instruction can be disambiguated via branch history for most instructions that exhibit such locality patterns.

Being able to determine when a particular locality pattern will occur for a memory instruction allows the compiler and architecture to cooperate in targeting when to apply memory optimizations. For example, the compiler could insert prefetches only for certain paths to a memory instruction where reuse distances are predicted to be large. In this case, the compiler would avoid issuing useless prefetches for short reuse distance paths.

We begin the rest of this paper with a review of work related to reuse-distance analysis and path profiling. Next, we describe our analysis techniques and algorithms for measuring instruction-based reuse distance with path information. Then, we present our experiments examining the effectiveness of path-based reuse-distance analysis and finish with our conclusions and a discussion of future work.

2 Related Work

Currently, compilers use either static analysis or simple profiling to detect data locality. Both approaches have limitations. Dependence analysis can help to detect data reuse [5, 6]. McKinley et al. design a model to group reuses and estimate cache miss costs for loop transformations based upon dependence analysis [7]. 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 [8]. These analytical models can capture high-level reuse patterns but may miss reuse opportunities due to a lack of run-time information and limited scope. Beyls and D'Hollander advance the static technique to encode conditions to accommodate

dynamic reuse distances for the class of programs that fit the *polyhedral model* [9]. Unfortunately, the model currently cannot handle spatial reuse and only works for a subset of numerical programs.

To address the limits of static analysis, much work has been done in developing feedback-directed schemes to analyze the memory behavior of programs using reuse distance. Mattson et al. [10] introduce reuse distance (or LRU stack distance) for stack processing algorithms for virtual memory management. Others have developed efficient reuse distance analysis tools to estimate cache misses [11, 12, 13, 14] and to evaluate the effect of program transformations [11, 15, 16]. Ding et al. [1] have developed a set of tools to predict reuse distance across all program inputs accurately, making reuse distance analysis a promising approach for locality based program analysis and optimizations. They apply reuse distance prediction to estimate whole program miss rates [2], to perform data transformations [17] and to predict the locality phases of a program [18]. Beyls and D'Hollander collect reuse distance distribution for memory instructions through one profiling run to generate cache replacement hints for an Itanium processor [19]. Beyls, D'Hollander and Vandeputte present a reuse distance visualization tool called RDVIS that suggests memory-hierarchy optimization [20]. Marin and Mellor-Crummey [21] incorporate reuse distance analysis in their performance models to calculate cache misses.

In our previous work, we propose a framework for instruction-based reuse distance analysis [3, 4]. We use *locality patterns* to represent the reuse distance distribution of an instruction, where a locality pattern is defined as a set of nearby related reuse distances of an instruction. Our work first builds the relationship between instruction-based locality patterns and the data size of program inputs, and extends the analysis to predict cache misses for each instruction, and identify *critical instructions*, those which produce most of the L2 cache misses. We find that a significant number of instructions, especially critical instructions, have multiple locality patterns. In this paper, we investigate the relationship between branch history and the occurrence of multiple locality patterns. To this end, we extend our reuse-distance analysis framework to path/instruction pairs to predict the path-based reuse distances across program inputs.

Previous research in path profiling usually aims at collecting accurate dynamic paths and execution frequencies for helping optimizing frequent paths [22, 23, 24]. Ammons and Larus use path profiles to identify *hot* paths and improve data flow analysis [22]. Ball and Larus present a fast path profiling algorithm that identifies each path with a unique number. Larus represents a stream of whole program paths as a context-free grammar which describes a program's entire control flow including loop iteration and interprocedural paths [24]. All of the aforementioned work takes basic block paths. We instead track only branch history since many modern superscalar processors already record branch history for branch prediction, allowing us to use the reuse-distance analysis in a dynamic optimization framework in the future.

With the intention of applying latency tolerating techniques to the specific set of dynamic load instructions that suffer cache misses, Mowry and Luk [25] propose a profiling approach to correlate cache misses to paths. While correlation profiling motivates our work, we focus on path-based reuse distance analysis. Reuse distance analysis exposes not only the results of hits or misses of cache accessing, but also the relevant

reasons. Further, our analysis can predict locality change on paths across program inputs, which is not mentioned in Mowry and Luk's paper [25].

3 Analysis

In this section we first describe previous work on whole-program and instruction-based reuse distance analysis. We then relate branch history to locality patterns at the instruction level. We further discuss locality pattern prediction with respect to the branch history of each instruction.

3.1 Reuse Distance Analysis

Reuse distance is defined as the number of distinct memory references between two accesses to the same memory location. In terms of memory locations, reuse distance has different levels of granularity, e.g. per memory address or per cache line. With the intention of analyzing data locality, this work focuses on cache-line based reuse distance. According to the access order of a reuse pair, reuse distance has two forms: *backward* reuse distance and *forward* reuse distance. Backward reuse distance is the reuse distance from the current access to the previous one addressing the same memory location. Similarly, forward reuse distance measures the distance from the current to the next access of the same memory location. In this paper, we report only backward reuse distances. The results for forward reuse distance are similar.

Ding et al. [1] show that the reuse distance distribution of each memory location accessed in a whole program is predictable with respect to the program input size. They define the *data size* of an input as the largest reuse distance and use a histogram describing reuse distance distribution. Each bar in the histogram consists of the portion of memory references whose reuse distance falls into the same range. Given two histograms with different data sizes, they predict the histogram of a third data size and find that those histograms are predictable in a selected set of benchmarks. Typically, one can use this method to predict reuse distance for a large data input of a program based on training runs of a pair of small inputs.

Previously, we have extended Ding et al.'s work to predict reuse distance for each instruction rather than memory location. We map the reuse distances for a memory location to the instructions that cause the accesses and show that the reuse distances of each memory instruction are also predictable across program inputs for both floating-point and integer programs [3, 4]. In our approach, the reuse distances for each instruction are collected and distributed in logarithmic scaled bins for distances less than 1K and in 1K-sized bins for distances greater than 1K. The minimum, maximum, and mean reuse distances together with the access frequency are recorded for each bin. We scan the original bins from the smallest distance to the largest distance and iteratively merge any pair of adjacent bins i and $i+1$ if

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

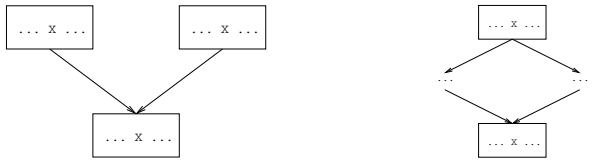
This inequality is true if the difference between the minimum distance in bin $i+1$ and the maximum distance in bin i is no greater than the length of bin i. The merging process stops when it reaches a minimum frequency and starts a new pattern for the

next bin. We call the merged bins *locality patterns*, which accurately and efficiently represent the reuse distance distribution on a per instruction basis. Furthermore, we have shown that locality patterns can be predicted accurately for a third input using two small training inputs and curve fitting, and can be used to predict cache misses for fully and set associative caches. These results show that locality patterns are an effective abstraction for data locality analysis.

3.2 Using Paths to Disambiguate Reuse Patterns

Although previous results show that over half of the instructions in SPEC CPU2000 contain only one pattern, a significant number of instructions exhibit two or more locality patterns. For the purposes of memory-hierarchy optimization, the compiler may need to know when each locality pattern occurs in order to tailor optimization to the pattern exhibiting little cache reuse.

Typically two backward reuse distance locality patterns of a load come either from different sources which meet at the current load through different paths (otherwise, one access will override the other) as shown in Figure 1(a), or a single source that reaches the current load through distinct paths as shown in Figure 1(b). This suggests that a dynamic basic block history plus the source block can uniquely identify each reuse. However, it is expensive to track a basic block trace at run time and apply it to memory optimizations. Branch history is a close approximation to basic block history and available on most modern superscalar architectures.



(a) (b)

Fig. 1. Path-related reuses, (a) from two separate sources, (b) from a single source

In this work, we use a stack to keep track of the branch history during profiling and collect reuse distances for each distinct branch history of an instruction. During a load, our reuse distance collection tool calculates reuse distance and records the distance with respect to the current branch history. If an instruction has multiple locality patterns and the reuse distances for each branch history form at most one pattern, the branch history can differentiate the multiple patterns and make it possible to determine when each pattern will occur.

Due to the existence of loops in a program, the history stack tends to be quickly saturated, making it difficult to track the reuses from outside the loops. To solve this problem, we detect loop back-edges during profiling and keep the branch history for up to l iterations of a loop, where l is the number of data elements that fit in one cache-line. Note that we choose l based on the cache-line size to differentiate between spatial (references within the same cache line) and temporal reuse patterns (references to the same

memory location). After l iterations or at the exit of a loop, the branch history in the loop is squashed with all corresponding history bits cleared from the stack. In this way, we efficiently use a small number of history bits to represent long paths. Furthermore, by squashing loops, the branch histories to an instruction tend to be consistent across program inputs, making it feasible to predict reuse distances along execution paths.

```
if (...)
  for (i = 0; i < n; i++)  // loop 1
    ...A[i]...
else
  for(i = 0; i < n; i++)   // loop 2
    ...A[i]...

for(i = 0; i < n; i++)     // loop 3
  ...A[i]...
```

Fig. 2. Multiple reuses

As an example, consider the program shown in Figure 2. `A[i]` in the third loop has spatial reuse from within the loop $l-1$ out of every l iterations. Additionally, `A[i]` has temporal reuse once every l iterations from either loop 1 or loop 2, depending on the condition of the `if`-statement. For this case, a history of $l+1$ bits are enough to differentiate all reuse patterns – l bits for reuse from within the loop and one bit for reuse from outside the loop.

After the reuse distances for all paths of each instruction are collected, the patterns are formulated following the merging process discussed in Section 3.1. We then examine whether the path history can help to uniquely identify a pattern and whether the path-based patterns can be predicted for a third input.

3.3 Path-Based Reuse Distance Prediction

Previous work has shown that reuse distances are likely to change across program inputs. To make the above analysis useful for optimizations, it is essential to predict reuse distances along execution paths. Our path-based reuse distance prediction is very similar to that for whole programs [1] and instructions [3, 4], except that we form patterns for each path in the two training runs and predict the patterns for the path in the validation run.

In the two training runs, the reuse patterns of each instruction are created for each profiled path. If a path does not occur in both of the two training runs, the reuse patterns for that path are not predictable. Our prediction also assumes a path has an equal number of patterns in the two training runs. We define the *coverage* of the prediction as the percentage of dynamic paths whose reuse distances are predictable based upon the above assumptions.

Given the reuse patterns of the same path in two runs, the predicted patterns for the path in the validation run can be predicted using curve fitting as proposed by Ding et al. [1]. The prediction *accuracy* is computed by comparing the predicted patterns with the observed ones in the validation run. Here accuracy is defined as the percentage of

the covered paths whose reuse distances are correctly predicted. A path's reuse distance distribution is said to be correctly predicted if and only if all of its patterns are correctly predicted. The prediction of a reuse pattern 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

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

4 Experiment

In this section, we report the results of our experimental evaluation of the relationship between locality patterns and execution paths. We begin with a discussion of our experimental methodology and then, we discuss the effectiveness of using path information in differentiating multiple locality patterns of an instruction. Finally, we report the predictability of the reuse distance distribution along execution paths.

4.1 Methodology

In this work, we execute our benchmark suite on the SimpleScalar Alpha simulator [26]. We modify *sim-cache* to generate the branch history and collect the data addresses and reuse distances of all memory instructions. Ding and Zhong's reuse-distance collection tool [1, 2] is used to calculate the reuse distance for each memory access. During profiling, our analysis records a 32-byte cache-line-based backward reuse distance for each individual memory instruction with the current branch history of varying lengths. Given the 32-byte cache-line size, we squash the branch history for loops every 4 iterations to help differentiate spatial and temporal reuse.

Our benchmark suite consists of 10 of the 14 floating-point programs and 11 of the 12 integer programs from SPEC CPU2000, as shown in Figure 3. The remaining five benchmarks (178.galgel, 187.facerec, 191.fma3d, 200.sixtrack and 252.eon) in SPEC CPU2000 are not included because we could not get them to compile correctly with version 5.5 of the Alpha compiler using optimization level -O3. For all benchmarks we use the test and train input sets for the training runs. For floating-point programs, we use the reference input sets for verification. However, for integer programs, we use the MinneSpec workload [27] in order to save profiling time due to the large memory requirements of the reference input set. We collect the reuse distance distribution by running all programs to completion.

4.2 Differentiating Multiple Locality Patterns

In this section, we experimentally analyze the ability of using branch history to differentiate between multiple locality patterns for a single instruction on our benchmark suite. We examine branch histories of length 1, 2, 4, 8, 16 and 32 bits using the history collection described in Section 3.2.

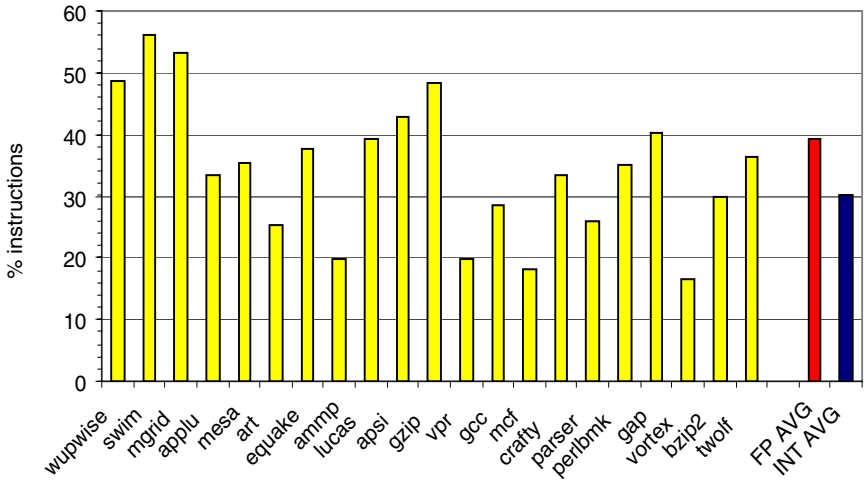


Fig. 3. % instructions having multiple reuse patterns

Figure 3 presents the percentage of instructions that have multiple locality patterns in a program. On average, 39.1% of the instructions in floating-point programs and 30.2% of the instructions in integer programs have more than one locality pattern. Floating-point programs, especially 168.wupwise, 171.swim, 172.mgrid and 301.apsi, tend to have diverse locality patterns. Many instructions in these programs have both temporal reuse from outside the loop that corresponds to large reuse distances, and spatial reuse from within the loop that normally has short reuse distances. In integer programs, a high number of conditional branches tends to cause multiple locality patterns. This phenomenon occurs often in 164.gzip, 186.crafty, 254.gap and 300.twolf.

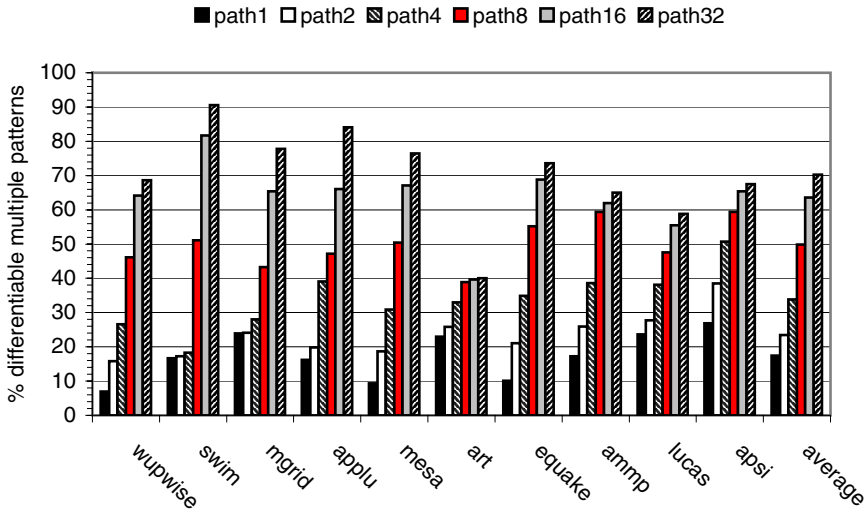


Fig. 4. % multiple patterns differentiable by paths for CFP2000

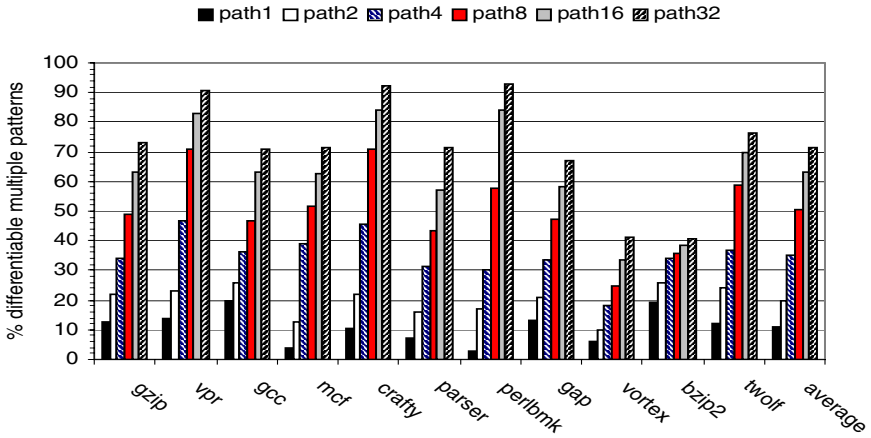


Fig. 5. % multiple patterns differentiable by paths for CINT2000

Figures 4 and 5 show the percentage of multiple locality patterns that can be differentiated using branch histories with various lengths. The bars labeled "path*n*" show the differentiation results for a path with *n* bits of history. We see from these two figures that, for both floating-point and integer programs, using execution path context can differentiate a significant percentage of multiple patterns for an instruction. This percentage increases with the increase in the number of history bits used. On average, paths with an 8-bit history can disambiguate over 50% of the multiple patterns. Whereas paths with 32 bits of history can disambiguate over 70% of the multiple patterns.

There are still some multiple patterns that cannot be differentiated by our approach even though a 32-bit history is used. Several factors have been observed to be responsible for this non-differentiability. The first is branch history *aliasing*, where different execution paths have the same branch history. Branch history aliasing occurs when executions from different block traces share the last several bits of the branch history, as shown in Figure 6. In this case, when using a 2-bit history both paths have the history of 01. However, a 3-bit history will solve the problem.

To examine the effect of branch history aliasing on our scheme, we report the percentage of multiple patterns that cannot be differentiated because of history aliasing, as

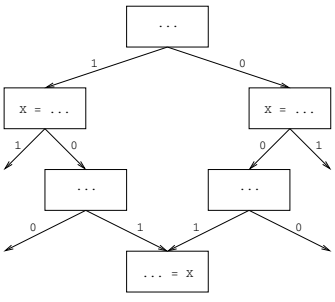


Fig. 6. Branch history aliasing

Table 1. % multiple patterns that are non-differentiable because of history aliasing for CFP2000

Benchmark	path1	path2	path4	path8	path16
168.wupwise	62.6	53.7	42.9	23.4	5.4
171.swim	70.3	69.8	68.7	36.0	5.4
172.mgrid	51.1	50.9	46.9	31.7	9.6
173.applu	52.3	48.7	29.5	21.4	2.4
177.mesa	76.4	67.1	54.9	35.3	18.6
179.art	40.8	38.0	30.9	24.9	24.1
183.equake	74.9	64.0	50.1	29.8	16.2
188.ammp	51.4	42.7	30.1	9.2	6.7
189.lucas	51.9	47.8	37.4	28.0	20.1
301.apsi	60.0	48.3	36.2	27.5	21.4
average	59.2	53.1	42.7	26.7	13.0

Table 2. % multiple patterns that are non-differentiable because of history aliasing for CINT2000

Benchmark	path1	path2	path4	path8	path16
164.gzip	74.0	64.6	52.6	37.7	23.4
175.vpr	81.3	71.8	48.2	24.0	12.0
176.gcc	65.0	58.7	48.5	37.9	21.7
181.mcf	91.8	83.0	56.5	44.3	33.3
186.crafty	87.6	75.9	52.7	27.3	13.9
197.parser	82.8	74.0	59.0	46.7	32.8
253.perlbmk	89.1	74.8	61.6	34.4	8.0
254.gap	73.6	66.2	53.5	40.1	28.6
255.vortex	70.1	66.5	58.1	51.4	42.4
256.bzip2	59.6	52.9	44.5	43.2	40.2
300.twolf	74.9	63.0	50.3	28.2	17.4
average	77.3	68.3	53.2	37.7	24.9

listed in Tables 1 and 2. We identify whether or not a particular history is an aliased one by tracking the block trace associated with this history. We experimentally collect the data for $path16$. For $pathn$ where $n < 16$, the non-differentiable multiple patterns due to aliasing are those for $path16$ plus all patterns that can be differentiated by a 16-bit history but not the n-bit history. We see from Tables 1 and 2 that the branch history aliasing problem is more severe in integer programs than in floating-point programs, and increasing the number of history bits can greatly reduce the number of non-differentiable patterns.

We have observed that, branch history aliasing most commonly occurs when all locality patterns represent short reuse distances. This is not a severe problem for determining cache misses. There are two ways to reduce the influence of the branch history aliasing problem. We can use more history bits and focus only on those critical instructions which produce most of the cache misses. Or, for those applications considering only short reuse distances, a block trace can be used instead of branch history for path representation. If only short reuse distances are involved, the memory requirement needed for basic block traces will not be excessive.

In addition to branch history aliasing, cache-line alignment causes some multiple patterns to be not differentiable using our scheme. The following code occurs in the function `fullGtu(Int32 i1, Int32 i2)` in program 256.bzip2.

```
1. c1=block[i1];
2. c2=block[i2];
3. if(c1!=c2) return(c1>c2);
4. i1++;i2++;

5. c1=block[i1];
6. c2=block[i2];
7. if(c1!=c2) return(c1>c2);
8. i1++;i2++;
```

Depending on the value of `i1`, `block[i1]` at line 5 may reuse the data touched at line 1 having a short reuse distance, or data from beyond the function having a long reuse distance. The change in the value of `i1` may cause the load at line 5 to be in a different cache line from the load at line 1. In this case, our scheme cannot differentiate between the two patterns. Notice that this piece of code is similar to an unrolled loop, which suggests that compile-time loop unrolling may also influence our scheme. Indeed, we have found cases of loop unrolling that cause some multiple patterns to be not differentiable by paths in floating-point programs such as 168.wupwise and 173.applu.

For cache-related optimization, it is important to differentiate multiple patterns of an instruction having both short and long reuse distances. This would allow the compiler to optimize the long reuse distance that is likely a cache miss and ignore the path where the reuse distance is short and a cache hit is likely. Table 3 lists the percentage of multiple patterns that cannot be differentiated and have both short and long patterns. Here we use a threshold of 1K, which corresponds to the size of a 32K-byte L1 cache, to classify reuse distances as short or long. We can see that non-differentiable multiple patterns

Table 3. % multiple patterns that cannot be differentiated and have both short and long distances

CFP2000	**path4**	**path8**	**path16**	**CINT2000**	**path4**	**path8**	**path16**
168.wupwise	28.5	24.3	17.5	164.gzip	36.5	28.7	21.9
171.swim	37.1	21.5	7.2	175.vpr	0.3	0.2	0.2
172.mgrid	10.4	8.2	6.1	176.gcc	0.3	0.3	0.2
173.applu	11.5	10.8	6.6	181.mcf	16.8	16.7	13.9
177.mesa	4.8	4.6	4.6	186.crafty	0.0	0.0	0.0
179.art	18.3	12.1	11.5	197.parser	0.1	0.1	0.0
183.equake	4.6	4.2	4.1	253.perlbmk	1.4	0.0	0.8
188.ammp	0.4	0.3	0.2	254.gap	1.5	1.2	1.1
189.lucas	42.6	37.1	30.2	255.vortex	0.4	0.3	0.3
301.apsi	5.2	4.6	3.8	256.bzip2	14.0	14.0	11.2
				300.twolf	6.8	5.0	4.2
average	16.3	12.8	9.2	average	7.0	6.1	4.9

with both short and long reuse distances only account for a small portion of the total number of the multiple patterns in a program, and on average, integer programs have a lower percentage than floating-point programs.

4.3 Path Based Reuse Distance Prediction

Tables 4 and 5 list the path-based reuse distance prediction coverage and accuracy for floating-point and integer programs, respectively. For comparison, we also list the instruction-based reuse distance prediction results in the columns labeled "inst". Due to the excessive memory requirements of simulation via SimpleScalar and profile collection, we cannot generate the prediction results for the path32 for integer programs.

Table 4. CFP2000 path-based reuse-distance prediction

Benchmark	coverage (%)							accuracy(%)						
	inst	path						inst	path					
		1	2	4	8	16	32		1	2	4	8	16	32
168.wupwise	92.9	94.2	93.7	94.9	96.6	97.5	98.5	98.1	99.0	99.1	99.4	99.4	99.5	99.7
171.swim	95.5	98.6	98.7	98.9	99.2	99.7	99.8	89.0	93.6	93.5	93.6	95.9	95.7	95.7
172.mgrid	96.6	97.9	97.3	97.7	96.6	97.1	94.1	91.9	94.8	95.0	95.8	96.4	96.5	96.2
173.applu	96.4	94.0	92.2	92.3	91.9	87.8	77.0	96.0	97.0	96.2	96.2	96.1	96.3	97.1
177.mesa	96.9	97.0	97.1	99.2	99.2	99.9	99.8	98.6	98.6	99.3	99.3	99.3	98.9	98.9
179.art	94.6	96.2	96.2	97.3	99.5	99.5	99.5	96.5	95.6	95.6	95.7	94.6	94.6	94.7
183.equake	99.2	99.6	99.6	99.6	99.2	98.9	98.0	98.3	98.6	98.6	98.8	98.6	99.0	98.9
188.ammp	99.9	99.9	99.9	99.9	99.9	99.8	99.4	89.6	92.8	92.8	93.9	94.0	94.1	94.3
189.lucas	71.7	66.5	65.3	63.3	62.4	60.1	59.3	94.1	97.5	98.6	98.6	98.3	98.8	98.8
301.apsi	96.6	96.6	96.8	97.1	96.1	91.4	85.9	93.0	96.5	97.0	97.2	96.9	97.2	97.7
average	94.0	94.1	93.7	94.0	94.1	93.17	91.1	94.5	96.4	96.6	96.9	97.0	97.1	97.2

Table 5. CINT2000 path-based reuse-distance prediction

Benchmark	coverage (%)						accuracy(%)					
	inst	path					inst	path				
		1	2	4	8	16		1	2	4	8	16
164.gzip	99.2	99.2	99.2	99.0	99.2	98.8	95.1	95.5	95.8	97.2	97.0	97.5
175.vpr	97.7	99.2	98.9	98.3	95.8	90.0	93.9	93.7	93.9	93.4	94.2	95.6
176.gcc	95.6	96.7	96.8	96.4	93.6	90.2	93.3	95.2	95.2	94.9	95.3	94.8
181.mcf	94.5	95.0	95.0	95.0	94.6	92.7	88.9	89.9	90.6	89.3	90.0	90.6
186.crafty	97.7	98.5	99.0	99.2	99.1	97.9	93.2	93.3	94.4	93.8	94.4	94.6
197.parser	83.3	85.5	87.1	84.8	79.1	66.4	84.9	84.4	85.2	88.5	91.6	97.0
253.perlbmk	99.8	99.8	99.8	99.8	99.8	99.2	97.2	97.2	97.2	97.2	98.0	97.9
254.gap	86.8	86.6	86.9	85.2	82.1	77.5	91.5	92.6	92.7	94.7	97.0	99.6
255.vortex	99.7	99.7	99.8	99.8	99.8	99.7	97.3	97.3	97.3	97.4	97.2	96.4
256.bzip2	99.9	99.9	99.9	99.9	99.9	99.9	98.0	97.8	97.8	97.8	98.1	97.9
300.twolf	95.6	96.2	96.1	95.4	94.1	90.1	93.3	93.3	93.4	94.0	94.0	95.0
average	95.4	96.0	96.2	95.7	94.3	91.1	93.3	93.7	94.0	94.4	95.2	96.1

For floating-point programs, on average, our mechanism can predict reuse distances for over 91% of the paths with accuracies all above 96%, with the number of history bits ranging from 1 up to 32. With less than or equal to 8 bits of branch history, the path-based prediction coverage compares well with using no branch history. When more than 8 bits are used, the prediction coverage decreases slightly. The rightmost part of Table 4 shows that using branch history improves the accuracy of reuse-distance prediction.

Integer programs exhibit similar coverage and accuracy results, as listed in Table 5. On average, we can predict reuse distances for over 91% of the paths with accuracies above 93.5%. While the coverage decreases with the increase in the number of bits used, the path-based reuse distance prediction coverage is higher than the instruction-based one when less than 8 bits are used. With a single-bit history, the average prediction accuracy is 93.7%, while the accuracy for a 16-bit history improves to 96.1% of the covered paths.

We have observed two major factors that influence the prediction coverage. First, our prediction assumes all paths appear in both training runs. However, some paths may only occur when using the reference input set (we call these paths *missing paths*). For example, a conditional branch may be taken when the reference input is used but not when the test input is used. Long execution paths will experience this phenomenon more often than short paths. Another factor determining the predictability is *pattern matching*. For a path or instruction, if the number of locality patterns is not equal in the two training runs, we cannot accurately match the corresponding patterns and thus cannot predict the reuse distances. For this case, relating reuse distances to paths has an advantage because most paths tend to have a single locality pattern.

For 168.wupwise, 171.swim and 179.art, the pattern matching problem dominates the cases where we do not predict reuse correctly. Thus, the coverage monotonically increases with the increase in the number of history bits used. When missing paths are the major factor, the prediction coverage decreases with the path length, as is the case for 173.applu and 189.lucas. For 173.applu, 22.8% of the paths are missing in the training runs for the 32-bit history, leading to a low coverage. For 189.lucas, 197.parser and 254.gap, there is a significant number of instructions that do not appear in the two training runs. Thus, the corresponding paths do not occur in both training runs, resulting in a low coverage for all evaluation cases.

5 Conclusions and Future Work

In this paper, we have proposed a novel approach for path-based reuse-distance analysis. We use execution-path history to disambiguate the reuse distances of memory instructions. Specifically, we relate branch history to particular locality patterns in order to determine exactly when a particular reuse distance will be exhibited by a memory operation.

Our experiments show that given sufficient branch history, multiple locality patterns for a single instruction can be disambiguated via branch history for most instructions that exhibit such locality patterns. On average, over 70% of the multiple patterns for static instructions can be differentiated by execution paths with a 32-bit branch history, for both floating-point and integer programs. In addition, we also show that the path

based reuse distances can be more accurately predicted than the instruction based reuse distances across program inputs, without a significant decrease in prediction coverage.

Being able to determine when a particular locality pattern will occur for a memory instruction allows the compiler and architecture to cooperate in targeting when to apply memory optimizations. Our next step is to apply the analysis for optimizations like prefetching. Specifically, we are developing software/hardware cooperative approaches to invoke prefetches only when certain paths with large reuse distances are executed. These approaches aim to avoid useless prefetches while achieving high performance.

References

1. Ding, C., Zhong, Y.: Predicting whole-program locality through reuse distance analysis. In: Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation, San Diego, California (2003) 245–257
2. Zhong, Y., Dropsho, S., Ding, C.: Miss rate prediction across all program inputs. In: Proceedings of the 12^{th} International Conference on Parallel Architectures and Compilation Techniques, New Orleans, LA (2003) 91–101
3. Fang, C., Carr, S., Önder, S., Wang, Z.: Reuse-distance-based miss-rate prediction on a per instruction basis. In: Proceedings of the Second ACM Workshop on Memory System Performance, Washington, D.C. (2004) 60–68
4. Fang, C., Carr, S., Önder, S., Wang, Z.: Instruction based memory distance analysis and its application to optimization. In: Proceedings of the 14^{th} International Conference on Parallel Architectures and Compilation Techniques, St. Louis, MO (2005)
5. Goff, G., Kennedy, K., Tseng, C.: Practical dependence testing. In: Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Canada (1991) 15–29
6. Pugh, W.: A practical algorithm for exact array dependence analysis. Communications of the ACM **35**(8) (1992) 102–114
7. McKinley, K.S., Carr, S., Tseng, C.: Improving data locality with loop transformations. ACM Transactions on Programming Languages and Systems **18**(4) (1996) 424–453
8. Wolf, M.E., Lam, M.: A data locality optimizing algorithm. In: Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Canada (1991) 30–44
9. Beyls, K., D'Hollander, E.: Generating cache hints for improved program efficiency. Journal of Systems Architecture **51**(4) (2005)
10. Mattson, R.L., Gecsei, J., Slutz, D., Traiger, I.L.: Evaluation techniques for storage hierarchies. IBM Systems Journal **9**(2) (1970) 78–117
11. Almasi, G., Cascaval, C., Padua, D.: Calculating stack distance efficiently. In: Proceedings of the first ACM Workshop on Memory System Performance, Berlin, Germany (2002)
12. Cascaval, C., Padua, D.: Estimating cache misses and locality using stack distance. In: Proceedings of the 17th International Conference on Supercomputing, San Francisco, CA (2003) 150–159
13. Sugumar, R.A., Abraham, S.G.: Efficient simulation of caches under optimal replacement with applications to miss characterization. In: Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems, Santa Clara, CA (1993) 24–35
14. Zhong, Y., Ding, C., Kennedy, K.: Reuse distance analysis for scientific programs. In: Proceedings of Workshop on Language, Compilers, and Runtime Systems for Scalable Compilers, Washington, DC (2002)

15. Beyls, K., D'Hollander, E.: Reuse distance as a metric for cache behavior. In: Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems. (2001)
16. Ding, C.: Improving effective bandwidth through compiler enhancement of global and dynamic reuse. PhD thesis, Rice University (2000)
17. Zhong, Y., Orlovich, M., Shen, X., Ding, C.: 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. (2004)
18. Shen, X., Zhong, Y., Ding, C.: Locality phase prediction. In: Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI), Boston, MA (2004)
19. Beyls, K., D'Hollander, E.: Reuse distance-based cache hint selection. In: Proccedings of the 8th International Euro-Par Conference. (2002)
20. Beyls, K., D'Hollander, E., Vandeputte, F.: RDVIS: A tool that visualizes the causes of low locality and hints program optimizations. In Sunderam, V.e.a., ed.: Computational Science – ICCS 2005, 5th International Conference. Volume 3515., Atlanta, Springer (2005) 166–173
21. Marin, G., Mellor-Crummey, J.: 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 (2004)
22. Ammons, G., Larus, J.R.: Improving data-flow analysis with path profiles. In: Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation, Montreal, Canada (1998) 72–84
23. Ball, T., Larus, J.R.: Efficient path profiling. In: Proceedings of the 29th International Symposium on Microarchitecture, Paris, France (1996) 46–57
24. Larus, J.R.: Whole program paths. In: Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation, Atlanta, GA (1999) 259–269
25. Mowry, T., Luk, C.K.: Predicting data cache misses in non-numberic applications through correlation profiling. In: Proceedings of the 30th International Symposium on Microarchitecture, North Carolina, United States (1997) 314–320
26. Burger, D.C., Austin, T.M.: The SimpleScalar tool set, version 2.0. Computer Architecture News **25**(3) (1997) 13–25
27. KleinOsowski, A., Lilja, D.: Minnespec: A new spec benchmark workload for simulation-based computer architecture research. Computer Architecture Letters **1** (2002)