Predicting Data Cache Misses in Non-Numeric Applications Through Correlation Profiling

Todd C. Mowry

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
tcm@cs.cmu.edu

Chi-Keung Luk

Department of Computer Science
University of Toronto
Toronto, Canada M5S 3G4
luk@eecg.toronto.edu

Abstract

*To maximize the benefit and minimize the overhead of software-based latency tolerance techniques, we would like to apply them precisely to the set of dynamic references that suffer cache misses. Unfortunately, the information provided by the state-of-the-art cache miss profiling technique (*summary profiling*) is inadequate for references with intermediate miss ratios—it results in either failing to hide latency, or else inserting unnecessary overhead. To overcome this problem, we propose and evaluate a new technique—*correlation profiling*—which improves predictability by correlating the caching behavior with the associated dynamic context. Our experimental results demonstrate that roughly half of the 22 non-numeric applications we study can potentially enjoy significant reductions in memory stall time by exploiting at least one of the three forms of correlation profiling we consider.*

1 Introduction

As the disparity between processor and memory speeds continues to grow, memory latency is becoming an increasingly important performance bottleneck. Cache hierarchies are an essential step toward coping with this problem, but they are not a panacea. To further tolerate latency, a number of promising software-based techniques have been proposed. For example, the compiler can tolerate modest latencies by scheduling *non-blocking loads* early relative to when their results are consumed [11], and can tolerate larger latencies by inserting *prefetch* instructions [7].

While these software-based techniques provide latency-hiding benefits, they also typically incur runtime overheads. For example, aggressive scheduling of non-blocking loads increases register lifetimes which can lead to spilling, and software-controlled prefetching requires additional instructions to compute prefetch addresses and launch the prefetches themselves. While the benefit of a technique typically outweighs its overhead whenever a miss is tolerated, the overhead hurts performance in cases where the reference would have enjoyed a cache hit anyway. Therefore to maximize overall performance, we would like to apply a latency-tolerance technique *only* to the precise set of dynamic references that would suffer misses. While previous work has addressed this problem for numeric codes [7], this paper focuses on the more difficult but important case of isolating dynamic miss instances in *non-numeric* applications.

1.1 Predicting Data Cache Misses in Non-Numeric Applications

To overcome the compiler's inability to analyze data locality in non-numeric codes, we can instead make use of profiling information. One simple type of profiling information is the precise miss ratios of all static memory references. Throughout the remainder of this paper, we will refer to this approach as *summary profiling*, since the miss ratio of each memory reference is summarized as a single value.

If summary profiling indicates that all significant memory reference instructions (i.e. those which are executed frequently enough to make a non-trivial contribution to execution time) have miss ratios close to 0% or 100%, then isolating dynamic misses is trivial—we simply apply the latency-tolerance technique only to the static references which always suffer misses. In contrast, if the important references have *intermediate* miss ratios (e.g., 50%), then we do not have sufficient information to distinguish which dynamic instances hit or miss, since this information is lost in the course of summarizing the miss ratio. The current state-of-the-art approach for dealing with inter-

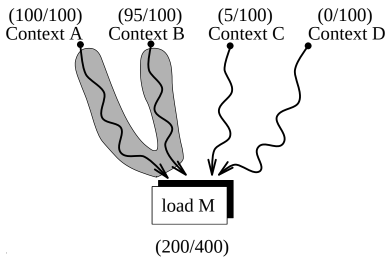


Figure 1: Example of how correlating cache misses with the dynamic context may improve predictability. (X/Y means X misses out of Y dynamic references.)

mediate miss ratios is to treat all static memory references with miss ratios above or below a certain threshold as though they always miss or always hit, respectively [2]. However, this all-or-nothing strategy will fail to hide latency when references are predicted to hit but actually miss, and will induce unnecessary overhead when references are predicted to miss but actually hit. Rather than settling for this sub-optimal performance, we would prefer to predict dynamic hits and misses more accurately.

1.2 Correlation Profiling

By exposing caching behavior directly to the user, *informing memory operations* [6] enable new classes of lightweight profiling tools which can collect more sophisticated information than simply the per-reference miss ratios. For example, cache misses can be correlated with information such as recent control-flow paths, the cache outcomes of previous references, etc., to help predict dynamic cache miss behavior. We will refer to this approach as *correlation profiling*.

Figure 1 illustrates how correlation profiling information might be exploited. The load instruction shown in Figure 1 has an overall miss ratio of 50%. However, depending on the dynamic context of the load, we may see more predictable behavior. In this example, contexts **A** and **B** result in a high likelihood of the load missing, whereas contexts **C** and **D** do not. Hence we would like to apply a latency tolerance technique within contexts **A** and **B** but not **C** or **D**.

The dynamic contexts shown in Figure 1 should be viewed simply as non-overlapping sets of dynamic instances of the load which can be grouped together because they share a common distinguishable pattern. In this paper, we consider three different types of information which can be used to distinguish these contexts. The first is *control-flow* information—i.e. the sequence of N basic block numbers preceding the load. The other two are based on sequences of cache access outcomes (i.e. hit or miss) for previous memory references: *self* correlation considers the cache outcomes of the previous N dynamic instances of the given static reference, and *global* correlation refers to the previous N dynamic references across the entire program. Note that analogous forms of all three types of correlation profiling have been explored previously in the context of *branch prediction* [4, 9, 14, 15]

1.3 Objectives and Overview

The goal of this paper is to determine whether *correlation profiling* can predict data cache misses more accurately in non-numeric codes than *summary profiling*, and if so, can we translate this into significant performance improvements by applying software-based latency tolerance techniques with greater precision. We focus specifically on predicting *load* misses in this paper because load latency is fundamentally more difficult to tolerate (store latency can be hidden through buffering and pipelining). Although we rely on simulation to capture our profiling information in this study, correlation profiling is a practical technique since it could be performed with relatively little overhead using informing memory operations [6].

The remainder of this paper is organized as follows. We begin in Section 2 by discussing the three different types of history information that we use for correlation profiling. Sections 3 and 4 present our experimental methodology and our experimental results which quantify the performance advantages of correlation profiling. Finally, we discuss related work and present conclusions in Sections 5 and 6.

2 Correlation Profiling Techniques

In this section, we propose and motivate three new correlation profiling techniques for predicting cache outcomes: *control-flow correlation*, *self correlation*, and *global correlation*.

2.1 Control-Flow Correlation

Our first profiling technique correlates cache outcomes with the recent control-flow paths. To collect this information, the profiling tool maintains the N most recent basic block numbers in a FIFO buffer, and matches this pattern against the hit/miss outcomes for a given memory reference. Intuitively, control-flow correlation is useful for detecting cases where either *data reuse* or *cache displacement* are likely.

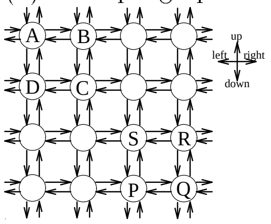
If we are on a path which leads to *data reuse*—either temporal or spatial—then the next reference is likely to be a cache hit. Consider the example shown in Figure 2(a)-(b), where a graph is traversed by the recursive procedure `walk()`. Any cyclic paths (e.g., A→B→C→D→A or P→Q→R→S→P) will result in

(a) Code with data reuse

```
struct node {
   int data;
   struct node
      *left, *right,
      *up, *down;
};

void walk(node* p) {
   work(p→data);
   if (go_left(p→data))
      p = p→left;
   elsif (go_right(p→data))
      p = p→right;
   elsif (go_up(p→data))
      p = p→up;
   elsif (go_down(p→data))
      p = p→down;
   else p = NULL;
   if (p != NULL) walk(p);
}
```

(b) Example graph



(c) Code with cache displacement

```
x = *p;
if (x > 0) {
   for (i = 0;
      i < 100000; i++)
      a[i] = foo(i);
}
y = *p;
```

Figure 2: Examples of how control-flow correlation can detect data reuse and cache displacement. (Control-flow profiled loads are underlined.)

temporal reuse of p→data. In this example, control-flow correlation can potentially detect that if the last four traversal decisions lead to a cycle (e.g., *right*, *down*, *left*, and *up*), then there is a high probability that the next p→data reference will enjoy a cache hit.

Some control-flow paths may increase the likelihood of a cache miss by displacing a data line before it is reused. For example, if the "`x > 0`" condition is true in Figure 2(c), then the subsequent `for` loop is likely to displace `*p` from the primary cache before it can be loaded again. Note that while paths which access large amounts of data are obvious problems, the displacement might also be due to a mapping conflict.

2.2 Self Correlation

Under *self correlation*, we profile a load L by correlating its cache outcome with the N previous cache outcomes of L itself. This approach is particularly useful for detecting forms of spatial locality which are not apparent at compile time. For example, consider the case in Figure 3 where a tree is constructed in preorder, assuming that consecutive calls to the memory allocator return contiguous memory locations, and that a cache line is large enough to hold exactly two `treeNodes`. Depending on the traversal order (and the extent to which the tree is modified after it is created), we may experience spatial locality when the tree is subsequently traversed. For example, if the tree is also traversed in preorder, we will expect p→data to suffer misses on every-other reference as cache line boundaries are crossed. Therefore despite the fact that the overall miss ratio of p→data is 50% and the compiler would have difficulty recognizing this as a form of spatial locality, self correlation profiling would accurately predict the dynamic cache outcomes for p→data.

(a) Example code

```
preorder(treeNode* p) {
   if (p != NULL) {
      work(p→data);
      preorder(p→left);
      preorder(p→right);
   }
}
```

(b) Tree constructed and traversed both in preorder

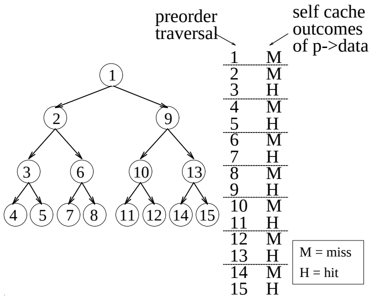


Figure 3: Example of using self-correlation profiling to detect spatial locality for p→data. (Consecutively numbered nodes are adjacent in memory.)

(a) Example code

```
while (1) {
   int i = hash(get());
   listNode* curr = htab[i];
   while (curr != NULL) {
      work(curr→data);
      curr = curr→next;
   }
}
```

(b) Hash table accesses

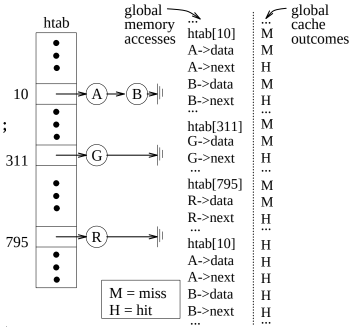


Figure 4: Example of using global-correlation profiling to detect bursty cache misses for curr→data.

2.3 Global Correlation

In contrast with self correlation, the idea behind *global correlation* is to correlate the cache outcome of a load L with the previous N cache outcomes regardless of their positions within the program. The profiling tool maintains this pattern using a single N-deep FIFO which is updated whenever dynamic cache accesses occur. Note that since earlier instances of L itself may appear in this global history pattern, global correlation may capture some of the same behavior as self correlation (particularly in extremely tight loops).

Intuitively, global correlation is particularly helpful for detecting *bursty* patterns of misses across multiple references. One example of this situation is when we move to a new portion of a data structure that has not been accessed in a long time (and hence has been displaced from the cache), in which case the fact that the first access to an object suffers a miss is a good indication that associated references to neighboring objects will also miss. Figure 4 illustrates such a case where a large hash table (too large to fit in the cache) is organized as an array of linked lists. In this case, we might expect a strong correlation between whether `htab[i]` (the list head pointer) misses and whether subsequent accesses to `curr`→`data` (the list elements) also miss. Similarly, if the same entry is accessed twice within a short interval (e.g., `htab[10]`), the fact that the head pointer hits is a strong indicator that the list elements (e.g., `A`→`data` and `B`→`data`) will also hit.

In summary, by correlating cache outcomes with the context in which the reference occurs—e.g., the surrounding control flow or the cache outcomes of prior references—we can potentially predict the dynamic caching behavior more accurately than what is possible with summarized miss ratios.

3 Experimental Methodology

To evaluate the potential performance benefit of correlation profiling, we measured its impact on the following 22 non-numeric applications: the entire SPEC95 integer benchmark suite, the additional integer benchmarks contained in the SPEC92 suite, uniprocessor versions of two graphics applications from SPLASH-2 [13], eight applications from Olden [10] (a suite of pointer-intensive benchmarks), and the standard UNIX utility `awk`. Table 1 briefly summarizes these applications, including the input data sets that were run to completion in each case.

We compiled each application with `-O2` optimization using the standard MIPS C compilers under IRIX 5.3. We used the MIPS `pixie` utility [12] to instrument these binaries, and piped the resulting trace into our detailed performance simulator. To increase simulation speed and to simplify our analysis, we model a perfectly-pipelined single-issue processor (similar to the MIPS R2000).[1] To reduce the simulation time, our simulator performs correlation profiling only on a selected subset of load instructions. Our criteria for profiling a load is that it must rank among the top 15 loads in terms of total cache miss count, and its miss ratio must be between 10% and 90%.

We attempt to maintain as much history information as possible for the sake of correlation. For control-flow correlation, we typically maintained a path length of 200 basic blocks—in some cases this resulted in such a large number of distinct paths that we were forced to measure only 50 basic blocks. For the self and global correlation experiments, we maintained patterns of 32 previous cache outcomes (either self or global).

[1] In our technical report [8], we also model a modern superscalar processor (the MIPS R10000), but space constraints prevent us from showing that data here.

Table 1: Benchmark characteristics.

Suite	Name	Input Data Set	Cache Size
SPEC95 Integer	m88ksim	train	8 KB
	perl	train (scrabbl)	128 KB
	go	train	8 KB
	ijpeg	train	8 KB
	vortex	train	8 KB
	compress	train	16 KB
	gcc	train (amptjp.i)	64 KB
	li	train	8 KB
SPEC92 Integer	sc	loada1	128 KB
	espresso	cps	16 KB
	eqntott	int_pri_3.eqn	8KB
SPLASH-2	raytrace	car	4KB
	radiosity	batch	8KB
Olden	bh	4K bodies	16KB
	mst	512 nodes	8KB
	perimeter	4K x 4K image	16KB
	health	max. level = 5 max. time = 50	16KB
	tsp	100,000 cities	8KB
	bisort	250,000 integers	8KB
	em3d	2000 H-nodes, 100 E-nodes	32KB
	voronoi	20,000 points	8KB
UNIX Utilities	awk	Extensive test of AWK's capabilities	32KB

We focus on the predictability of a single level of data cache (two levels makes the analysis too complicated). The choice of data cache size is important because if it is either too large or too small relative to the problem size, predicting dynamic misses becomes too easy (they either always hit or always miss). Therefore we would like to operate near the "knee" of the miss ratio curve, where predicting dynamic hits and misses presents the greatest challenge. Although we could potentially reach this knee by altering the problem size, we had greater flexibility in adjusting the cache size within a reasonable range. We chose the data cache size as follows. We first used summary profiling to collect the miss ratios of all loads within the application on different cache sizes ranging from 4KB to 128KB. We then chose the cache size which resulted in the largest number of significant loads having intermediate miss ratios—these sizes are shown in Table 1. In all cases, we model a two-way set-associative cache with 32 byte lines.

4 Experimental Results

Figure 5 shows how the three correlation profiling schemes—control-flow (**C**), self (**S**), and global (**G**)—improve the prediction accuracy of correlation-profiled loads. Due to space constraints, we show only the eleven cases most significant cases (the other cases can be found in our technical report [8]). Each bar

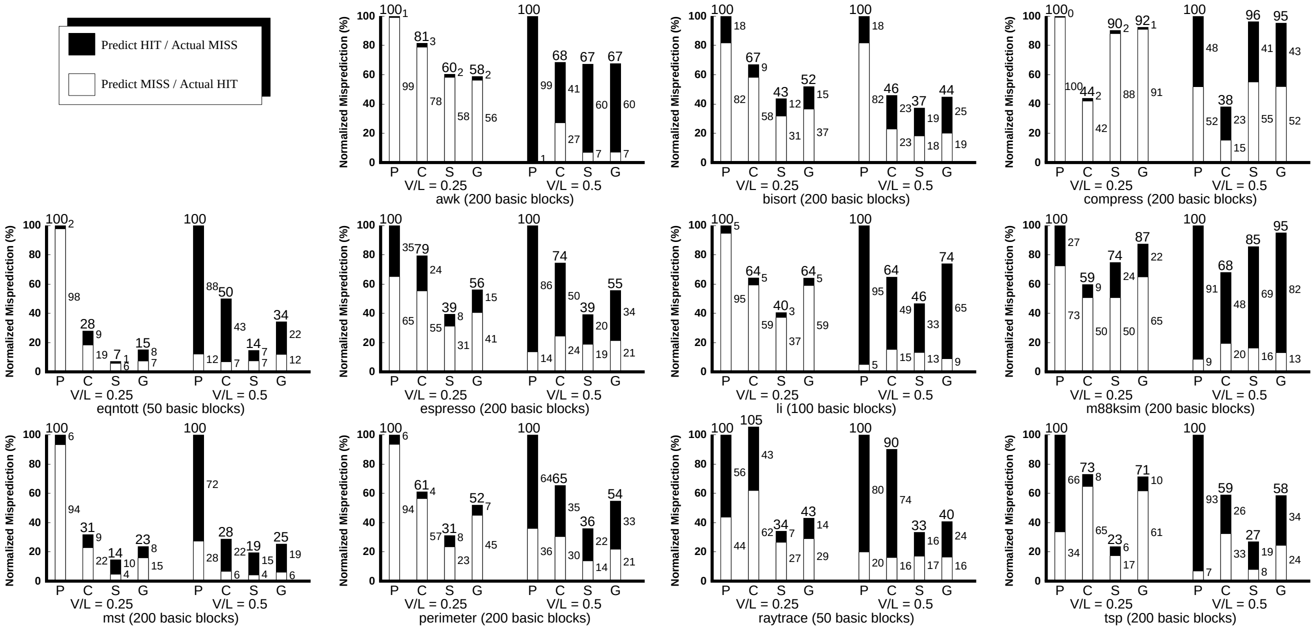


Figure 5: Number of mispredicted correlation-profiled loads, normalized to summary profiling (**P** = summary profiling, **C** = control-flow correlation, **S** = self correlation, **G** = global correlation). Maximum path lengths used in control-flow correlation are indicated next to the benchmark names.

is normalized with respect to the number of mispredicted references in summary profiling (**P**), and is broken down into two categories. The top section ("*Predict HIT / Actual MISS*") represents a *lost opportunity* where we predict that a reference hits (and thus do not attempt to tolerate its latency), but it actually misses. The "*Predict MISS / Actual HIT*" section accounts for *wasted overhead* where we apply latency tolerance to a reference that actually hits.

The decision of whether to apply a latency tolerance technique (T) to a load works as follows. At a certain granularity (dictated by our ability to statically isolate different contexts), we must choose a single strategy for a set of dynamic instances of the load: either to apply T or not. If we apply T, then the net gain per dynamic reference is $mL - V$, where m is the miss ratio, L is the average cache miss latency, and V is the average latency tolerance overhead. Hence the break-even point occurs when $m = \frac{V}{L}$: if $m > \frac{V}{L}$, we should apply T, and if $m < \frac{V}{L}$, we should not. For summary profiling, this threshold is applied to the overall miss ratio of an instruction; for correlation profiling, it is applied to groups of dynamic instances of a reference within distinguishable contexts.

Figure 5 shows results with two values of $\frac{V}{L}$: 0.25 and 0.5. For $\frac{V}{L} = 0.25$, summary profiling tends to apply latency tolerance aggressively, thus resulting in a noticeable amount of wasted overhead. In contrast, for $\frac{V}{L} = 0.50$, summary profiling tends to be more conservative, thus resulting in many untolerated misses. Overall, correlation profiling can significantly reduce both types of misprediction.

To quantify the performance impact of this improved prediction accuracy, Figure 6 shows the resulting execution time of the four profiling schemes, assuming a cache miss latency of 50 cycles. Each bar is normalized to the execution time without latency tolerance, and is broken down into four categories. The bottom section is the *busy* time. The section above it ("*Predict MISS / Actual MISS*") is the *useful* overhead paid for tolerating references that normally miss. The top two sections represent the *misprediction penalty*, including wasted overhead ("*Predict MISS / Actual HIT*") and untolerated miss latency ("*Predict HIT / Actual MISS*").

The degree to which improved prediction accuracy translates into reduced execution time depends not only on the relative importance of load stalls but also the fraction of loads that are correlation profiled. When both factors are favorable (e.g., `eqntott`),

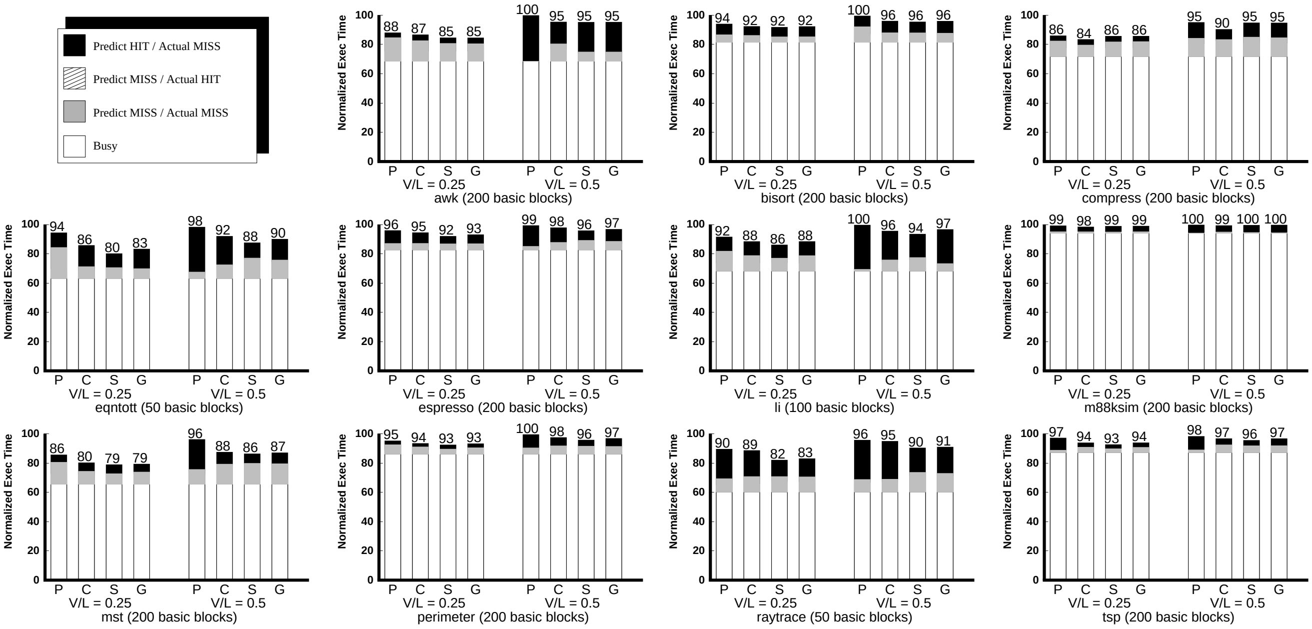


Figure 6: Impact of the profiling schemes on execution time, assuming a 50 cycle miss latency (L). **P** = summary profiling, **C** = control-flow correlation, **S** = self correlation, and **G** = global correlation.

we see large performance improvements—when either factor is small (e.g., `perimeter` and `tsp`), the performance gains are modest despite large improvements in prediction accuracies. Since failing to hide a miss is more expensive than wasting overhead, it is possible to improve performance by replacing more expensive with less expensive mispredictions, even if the total misprediction count increases (e.g., `raytrace` with control-flow correlation when $\frac{V}{L} = 0.25$).

4.1 Lessons Learned

We carefully studied the memory access patterns of each application by hand to develop a deeper understanding of when and why correlation profiling succeeds. Although space constraints prevent us from presenting the full details here, we briefly summarize the major results (for more detail, we refer the reader to an earlier publication [8]). Although global correlation makes excellent predictions in some cases by correlating behavior across different load instructions (e.g., `eqntott`), in most cases it essentially assimilates self correlation, but does not perform quite as well since it records less history for a given load. Self correlation is often successful since it recognizes forms of spatial locality which are not recognizable at compile time (e.g, `li`, `perimeter`, `bisort`, and `mst`), and also long runs of either all hits or all misses (e.g., `eqntott`, `mst`, `tsp`, and `raytrace`). We often find that as few as four previous cache outcomes per reference are sufficient to achieve good predictability with self correlation. By capturing call chain information, control-flow correlation can distinguish behavior based on call sites (e.g., `eqntott`, `espresso`, `vortex`, `m88ksim`, `go`, `mst` and `voronoi`) and the depth of the recursion while traversing a tree (e.g., `perimeter`, `bisort`, and `tsp`).

Roughly half of the applications enjoy significant improvements from *both* control-flow and self correlation, and in many of these cases we observe that the same load references can be successfully predicted by both forms of correlation. This is good news, since control-flow correlation profiling is the easiest case to exploit in practice by using *procedure cloning* [5] to distinguish call-chain dependent behavior. We applied this technique to software-controlled prefetching, and observed that correlation profiling offers superior performance over summary profiling when prefetching on a modern superscalar processor [8].

5 Related Work

Abraham *et al.* [2] investigated using summary profiling to associate a single latency tolerance strategy (i.e. either attempt to tolerate the latency or not) with each profiled load. They used this approach to reduce the cache miss ratios of nine SPEC89 benchmarks, including both integer and floating-point programs. In a follow-up study [1], they also report the improvement in *effective cache miss ratio*. In contrast with this earlier work, our study has focused on *correlation profiling*, which is a novel technique that provides superior prediction accuracy relative to summary profiling.

Ammons *et al.*[3] used path profiling techniques to observe that a large fraction of primary data cache misses in the SPEC95 benchmarks occur along a relatively small number of frequently executed paths.

The three forms of correlation explored in this study (*control-flow*, *self*, and *global*) were inspired by earlier work on using correlation to enhance *branch prediction* accuracies [4, 9, 14, 15]. While branch outcomes and cache access outcomes are quite different, it is interesting to observe that correlation-based prediction works well in both cases.

6 Conclusions

To help achieve the full potential of software-based latency tolerance techniques, we have proposed *correlation profiling*, which is a technique for isolating the dynamic instances of a static memory reference that are likely to suffer cache misses. Our experiments demonstrate that by exploiting at least one of the three forms of correlation information that we consider (i.e. *control-flow*, *self*, or *global*), correlation profiling outperforms summary profiling by improving the prediction accuracy, thereby reducing memory stall times. We observe that *self correlation* works well because the cache outcome patterns of individual references often repeat in predictable ways, and that *control-flow correlation* works mainly because many cache outcomes are call-chain dependent. Although *global correlation* offers superior performance in some cases, it typically does a less effective job of capturing the same behaviors as self correlation. Through the use of informing memory operations and techniques such as procedure cloning, we can exploit correlation profiling information in practice. We hope that these promising results will lead to further innovations in optimizing the memory performance of non-numeric applications.

7 Acknowledgments

This work is supported by grants from the Natural Sciences and Engineering Research Council of Canada, and by a grant from IBM Canada's Centre for Advanced Studies. Todd C. Mowry is partially supported by a Faculty Development Award from IBM. Chi-Keung Luk is partially supported by a Canadian Commonwealth Fellowship.

References

[1] S. G. Abraham and B. R. Rau. Predicting load latencies using cache profiling. Technical Report HPL-94-110, Hewlett-Packard Company, November 1994.

[2] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of load/store instruction latencies. In *MICRO-26*, pages 139–152, December 1993.

[3] G. Ammons, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI'97*, June 1997.

[4] P. Chang, E. Hao, T. Yeh, and Y. Patt. Branch classification: a new mechanism for improving branch predictor performance. In *MICRO-27*, November 1994.

[5] K.D. Cooper, M.W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2), April 1993.

[6] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing memory operations: Providing memory performance feedback in modern processors. In *ISCA'96*, pages 260–270, May 1996.

[7] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS-V*, pages 62–73, October 1992.

[8] T. C. Mowry and C.-K. Luk. Predicting data cache misses in non-numeric applications through correlation profiling. Technical Report CMU-CS-97-175, Carnegie Mellon University, September 1997.

[9] S. Pan, K. So, and J. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *ASPLOS-V*, pages 76–84, October 1992.

[10] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Trans. on Programming Languages and Systems*, 17(2), March 1995.

[11] A. Rogers and K. Li. Software support for speculative loads. In *ASPLOS-V*, pages 38–50, October 1992.

[12] M. D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford University, November 1991.

[13] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA'95*, pages 24–38, June 1995.

[14] T.-Y. Yeh and Y. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *ISCA'93*, pages 257–266, May 1993.

[15] C. Young and M. Smith. Improving the accuracy of static branch prediction using branch correlation. In *ASPLOS-VI*, pages 232–241, October 1994.