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

Todd C. Mowry Chi-Keung Luk[†]

September 1997

CMU-CS-97-175

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[†]Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, M5S 3G4

An abbreviated version of this paper will appear in the proceedings of the 30th International Symposium on Microarchitecture, Research Triangle Park, North Carolina, December 1-3, 1997.

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.

Abstract

Software-based latency tolerance techniques offer the potential for bridging the ever-increasing speed gap between the memory subsystem and today's high-performance processors. However, to fully exploit the benefit of these techniques, one must be careful to apply them only to the dynamic references that are likely to suffer cache misses—otherwise the runtime overheads can potentially offset any gains. In this paper, we focus on isolating dynamic miss instances in *non-numeric* applications, which is a difficult but important problem. Although compilers cannot statically analyze data locality in non-numeric applications, one viable approach is to use profiling information to measure the actual miss behavior. Unfortunately, the state-of-the-art in cache miss profiling (which we call *summary profiling*) is inadequate for references with intermediate miss ratios—it either misses opportunities to hide latency, or else inserts overhead that is unnecessary. To overcome this problem, we propose and evaluate a new profiling technique that helps predict which dynamic instances of a static memory reference will hit or miss in the cache: *correlation profiling*.

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: *control-flow correlation*, *self correlation*, and *global correlation*. In addition, our detailed case studies illustrate that self correlation succeeds because a given reference's cache outcomes often contain repeated patterns, and control-flow correlation succeeds because cache outcomes are often call-chain dependent. We also demonstrate that software prefetching can achieve better performance on a modern superscalar processor when directed by correlation profiling rather than summary profiling information.

Keywords: B.3.2 Cache Memories, C.4 Performance of Systems (Measurement Techniques, Performance Attributes), E.1 Data Structures (Graphs, Lists, Trees), D.3.4 Compilers

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 complete solution. 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 [12], and can tolerate larger latencies by inserting *prefetch* instructions [7, 9].

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 [9], 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 Codes

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 intermediate 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.

1.1.1 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, whether recent memory references hit or missed in the cache, 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

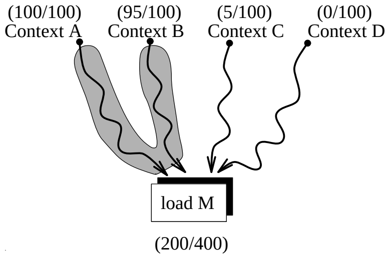


Figure 1: Example of how correlating cache misses with the dynamic context may improve predictability. (X/Y means X misses out of Y dynamic references.)

analogous forms of all three types of correlation profiling have been explored previously in the context of *branch prediction* [4, 10, 15, 16]

1.2 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, and in Section 3 we present a qualitative analysis of the expected performance benefits. In Section 4, we present our experimental results which quantify the performance advantages of correlation profiling in a collection of 22 non-numeric applications. In addition, in Section 5, we report the memory-access behaviors of individual applications which explain when and how correlation profiling is effective. In Section 6, we compare the performance of software prefetching guided by summary and correlation profiling on a modern superscalar processor. Finally, we discuss related work and present conclusions in Sections 7 and 8.

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

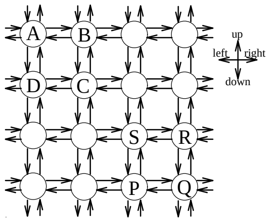
Our first profiling technique correlates cache outcomes with the recent control-flow paths. To collect this information, the profiling tool maintains the N most recent basic block numbers in a FIFO buffer, and matches this pattern against the hit/miss outcomes for a given memory reference. Intuitively, 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 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

```
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);
}
```

(a) Code with data reuse



(b) Example graph

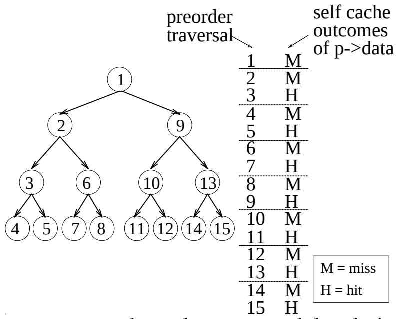
```
x = *p;
if (x > 0) {
  for (i = 0;
    i < 100000; i++)
    a[i] = foo(i);
}
y = *p;
```

(c) Code with cache displacement

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

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

(a) Example Code



(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.)

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

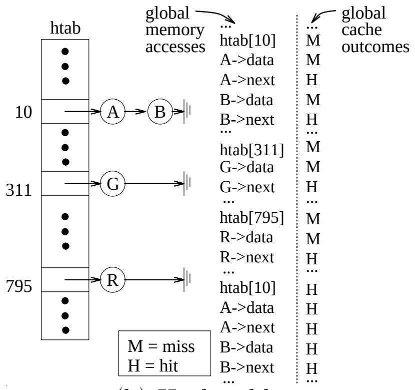
Under *self correlation*, we profile a load L by correlating its cache outcome with the N previous cache outcomes of L itself. This approach is particularly useful for detecting forms of spatial locality which are not apparent at compile time. For example, consider the case in Figure 3 where a tree is 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`.

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.

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

(a) Example code



(b) Hash table accesses

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

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.

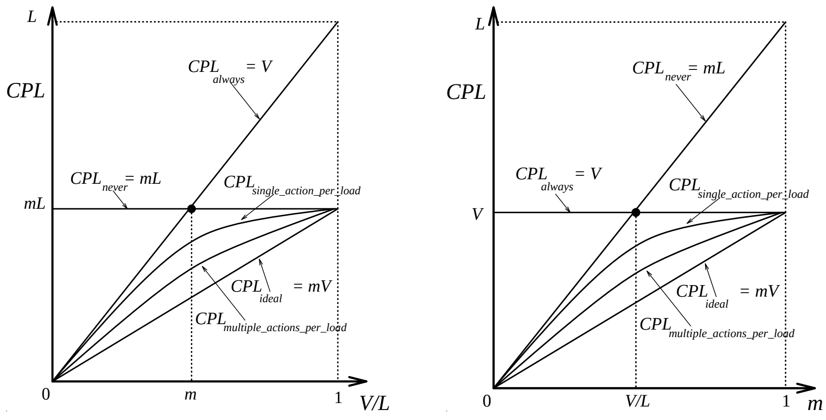
In summary, by correlating cache outcomes with the context in which the reference occurs—e.g., the surrounding control flow or the cache outcomes of prior references—we can potentially predict the dynamic caching behavior more accurately than what is possible with summarized miss ratios.

3 Qualitative Analysis of Expected Benefits

Before presenting our quantitative results in later sections, we begin in this section by providing some intuition on how correlation profiling can improve performance. A key factor which dictates the potential performance gain is the ratio of the latency tolerance overhead (V) to the cache miss latency (L). In the extreme cases where $\frac{V}{L} = 0$ or $\frac{V}{L} = 1$, there is no point in applying the latency tolerance technique (T) selectively, since it either has no cost or no benefit. When $0 < \frac{V}{L} < 1$, however, applying T selectively may be important.

Figure 5(a) illustrates how the average number of effective *stall cycles per load* (CPL) varies as a function of $\frac{V}{L}$ for various strategies for applying T. (Note that our CPL metric includes any overhead associated with applying T, but does not include the single cycle for executing the load instruction itself.) If T is never applied, then the CPL is simply mL, where m is the average miss ratio. At the other extreme, if we *always* apply T, then the latency will always be hidden, but *all* references (even those that normally hit) will suffer the overhead V: hence the $CPL = V$. Note that when $\frac{V}{L} > m$, it is better to never apply T rather than always applying it. Figure 5(b) shows an alternative view of CPL, where it is plotted as a function of m for a fixed $\frac{V}{L}$. Again, we observe that the choice of whether to always or never apply T depends on the value of m relative to $\frac{V}{L}$.

To achieve better performance than this all-or-nothing approach, we apply the same decision-making process (i.e. comparing the miss ratio with $\frac{V}{L}$) to more refined sets of loads. In the ideal case, we would consider and optimize each dynamic reference individually (the resulting CPL of mV is shown in Figure 5). However, since this is impractical for software-based techniques, we must consider aggregate collections of references. Since summary profiling provides only a single miss ratio per static reference, the finest granularity



(a) CPL vs. $\frac{V}{L}$ (b) CPL vs. m

Figure 5: Illustration of the CPL for different approaches of applying a latency tolerance scheme (m = overall average load miss ratio, V = latency tolerance overhead, and L = load miss latency).

at which we can decide whether or not to apply T is once for all dynamic instances of a given static reference. Figure 5 illustrates the potential shape of this "*single action per load*" curve, which is bounded by the cases where T is never, always, and ideally applied. Since *correlation* profiling distinguishes different sets of dynamic instances of a static load based on path information, it allows us to make decisions at a finer granularity than with summary profiling. Therefore we can potentially achieve even better performance, as illustrated by the "*multiple actions per load*" curve in Figure 5. (Further details on the actual CPL equations for the summary and correlation profiling cases can be found in the Appendix)

4 Quantitative Evaluation of Performance Gains

In this section, we present experimental results to quantify the performance benefits offered by correlation profiling. We begin by measuring and understanding the potential performance advantages for a generic latency tolerance scheme. Later, in Section 6, we will focus on software-controlled prefetching as a specific case study.

4.1 Experimental Methodology

We measured the impact of correlation profiling 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 [14], eight applications from Olden [11] (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, and Table 2 shows some relevant dynamic statistics of these applications.

We compiled each application with `-O2` optimization using the standard MIPS C compilers under IRIX 5.3. We used the MIPS `pixie` utility [13] 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) in this section. (Later, in Section 6, we model a modern superscalar processor: the MIPS R10000).

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%. Using this criteria, we focus only on the most significant loads which have intermediate miss ratios. We will refer to these loads as the *correlation-profiled loads*. The fraction of dynamic load references in each application that is correlation profiled is shown in Table 2.

Table 1: Benchmark characteristics.

Suite	Name	Description	Input Data Set	Cache Size
SPEC95 Integer	m88ksim	Motorola 88000 CPU simulator	train	8 KB
	perl	Unix script language Perl	train (scrabbl)	128 KB
	go	Computer game "Go"	train	8 KB
	ijpeg	Graphic compression and decompression	train	8 KB
	vortex	Database program	train	8 KB
	compress	Compresses and decompresses file in memory	train	16 KB
	gcc	GNU C compiler	train (amptjp.i)	64 KB
	li	LISP interpreter	train	8 KB
SPEC92 Integer	sc	Spreadsheet program	loada1	128 KB
	espresso	Minimization of boolean functions	cps	16 KB
	eqntott	Translation of boolean equations into truth tables	int_pri_3.eqn	8KB
SPLASH-2	raytrace	Ray-tracing program	car	4KB
	radiosity	Light distribution using radiosity method	batch	8KB
Olden	bh	Barnes-Hut's N-body force-calculation	4K bodies	16KB
	mst	Finds the minimum spanning tree of a graph	512 nodes	8KB
	perimeter	Computes perimeters of regions in images	4K x 4K image	16KB
	health	Simulation of the Columbian health care system	max. level = 5 max. time = 50	16KB
	tsp	Traveling salesman problem	100,000 cities	8KB
	bisort	Sorts and merges bitonic sequences	250,000 integers	8KB
	em3d	Simulates the propagation of E.M. waves in a 3D object	2000 H-nodes, 100 E-nodes	32KB
	voronoi	Computes the voronoi diagram of a set of points	20,000 points	8KB
UNIX Utilities	awk	Unix script language AWK	Extensive test of AWK's capabilities	32KB

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 a path length of 32 previous cache outcomes (either self or global).

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.

4.2 Improvements in Prediction Accuracy and Performance

Figure 6 shows how the three correlation profiling schemes—control-flow (**C**), self (**S**), and global (**G**)—improve the prediction accuracy of correlation-profiled loads. Each bar 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.

As discussed earlier in Section 3, our threshold for deciding whether to apply latency tolerance to a reference is that its miss ratio must exceed $\frac{V}{L}$, where V is the latency tolerance overhead and L is the miss latency. 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 references along individual paths. Figure 6 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

Table 2: Dynamic benchmark statistics (the column "Insts" is the number of dynamic instructions, the column "Loads" is the number of dynamic loads (its percentage out of "Insts" is also given), the column "Load Miss Rate" is the data-cache miss rate of loads, the column "CP Loads" is the fraction of dynamic loads that are correlation profiled, and the column "CP Load Misses" is the fraction of load misses that are correlation profiled).

Suite	Name	Dynamic Statistics				
		Insts	Loads	Load Miss Rate	CP Load Refs	CP Load Misses
SPEC95 Integer	m88ksim	132M	22M (17%)	0.8%	1%	48%
	perl	79M	15M (18%)	12.3%	21%	95%
	go	568M	121M (21%)	7.1%	10%	23%
	ijpeg	1438M	266M (18%)	2.7%	2%	17%
	vortex	2838M	830M (29%)	3.3%	7%	48%
	compress	39M	8M (20%)	3.9%	6%	87%
	gcc	282M	61M (22%)	1.4%	2%	40%
	li	228M	54M (24%)	4.0%	8%	73%
SPEC92 Integer	sc	833M	143M (17%)	9.2%	26%	92%
	espresso	560M	112M (20%)	2.2%	6%	70%
	eqntott	986M	210M (21%)	5.5%	14%	77%
SPLASH-2	raytrace	2105M	588M (28%)	4.8%	10%	53%
	radiosity	996M	236M (24%)	0.4%	1%	32%
Olden	bh	2326M	667M (29%)	1.0%	3%	82%
	mst	90M	14M (16%)	6.9%	17%	91%
	perimeter	123M	17M (14%)	2.3%	5%	88%
	health	8M	2M (25%)	9.0%	20%	84%
	tsp	825M	239M (29%)	1.0%	1%	37%
	bisort	732M	132M (18%)	2.5%	6%	74%
	em3d	420M	73M (17%)	1.4%	4%	98%
	voronoi	263M	87M (16%)	1.3%	4%	57%
UNIX Utilities	awk	70M	9M (7%)	7.6%	16%	90%

profiling can significantly reduce both types of misprediction.

To quantify the performance impact of this increased prediction accuracy, Figure 7 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[1] 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`), 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.

5 Case Studies

To develop a deeper understanding of when and why correlation profiling succeeds, we now examine a number of the applications in greater detail. In addition to discussing the memory access patterns for these applications, we also show the impact of the correlation-profiled loads on three performance metrics: the *miss ratio distribution*, the stall cycles per load (CPL) due to correlation-profiled loads only, and the overall CPI. While CPL and CPI measure the impacts on execution time, the miss ratio distribution gives us insight into how effectively correlation profiling has isolated the dynamic hit and miss instances of static load instructions.

[1] 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 $\frac{V}{L} = 0.25$)

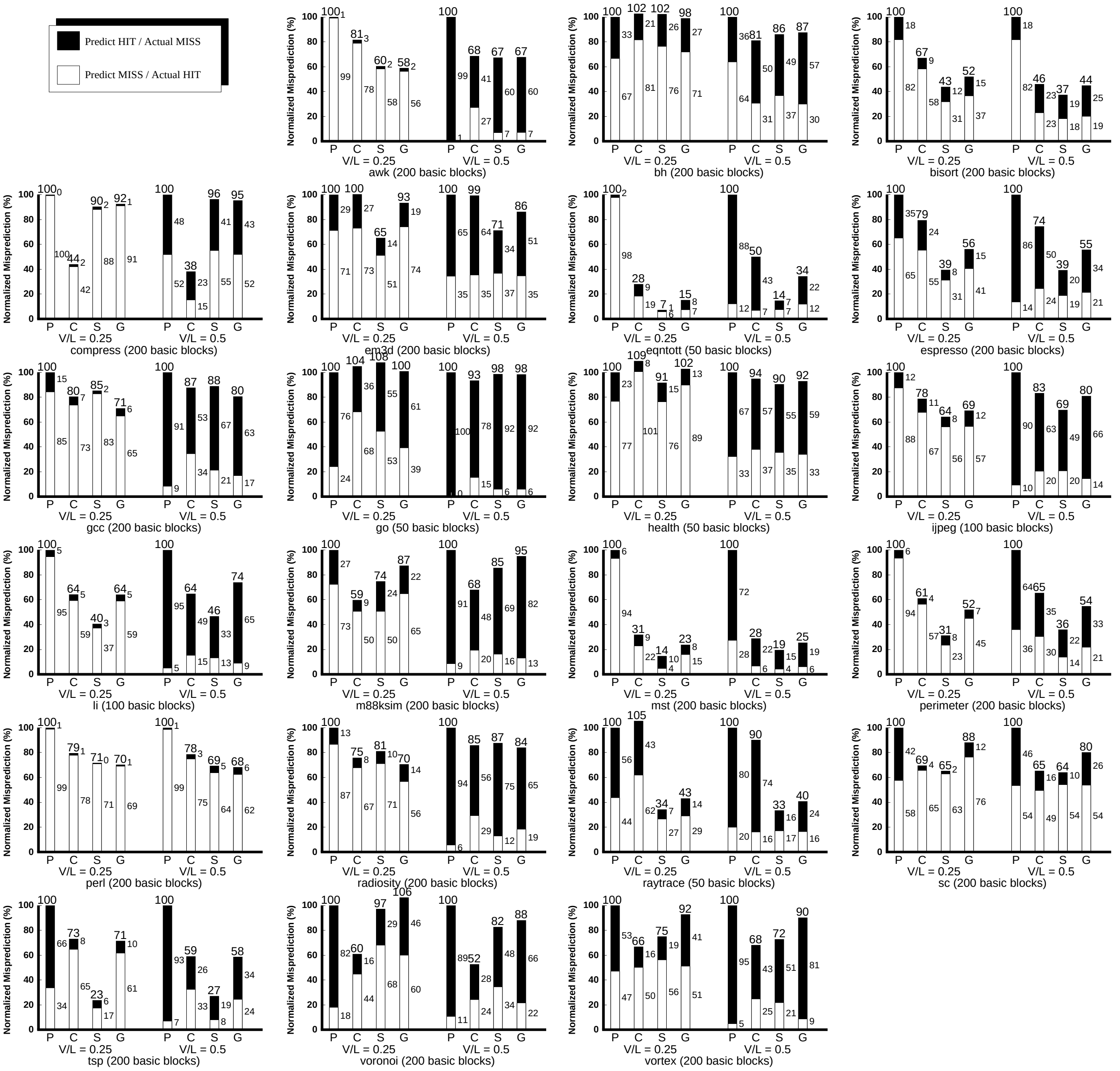


Figure 6: Number of mispredicted correlation-profiled loads, normalized to summary profiling (**P** = summary profiling, **C** = control-flow correlation, **S** = self correlation, **G** = global correlation). Maximum path lengths used in control-flow correlation are indicated next to the benchmark names.

5.1 li

Over half of the total load misses are caused by two pointer dereferences: `this`→`n_flags` in `mark()`, and `p`→`n_flags` in `sweep()`, as illustrated by the pseudo-code in Figure 8.

The access patterns behave as follows. The procedure `mark()` traverses a binary tree through the three `while` loops shown in Figure 8(a). Starting at a particular node, the first inner `while` loop continues descending the tree—choosing either the left or right child as it goes—until it reaches either a marked node or a leaf node. At this point, we then backup to a node where we can continue descending through a search

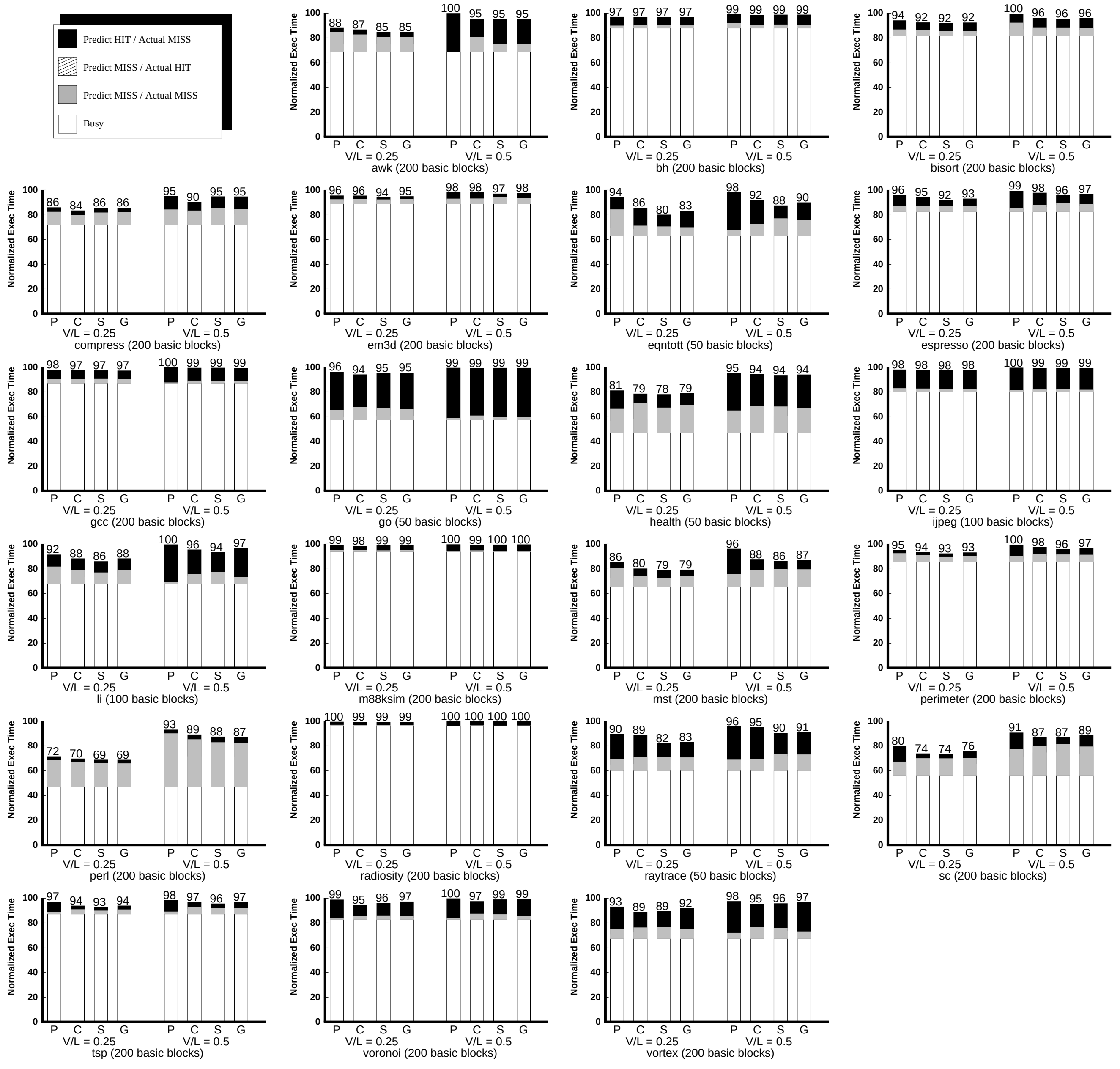


Figure 7: Impact of the profiling schemes on execution time, assuming a 50 cycle miss latency (L). (**P** = summary profiling, **C** = control-flow correlation, **S** = self correlation, and **G** = global correlation.)

performed by the second inner `while` loop. The tree is allocated in preorder, similar to the one shown in Figure 3, except much larger. Therefore we enjoy spatial locality as long as we continue following left branches in the tree, but spatial locality is disrupted whenever we backup in the second inner `while` loop, as illustrated by Figure 8(c).

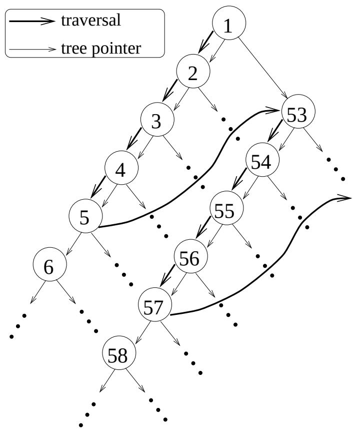
All three types of correlation profiling provide better cache outcome predictions than summary profiling for the `this→n_flags` reference in `mark()` for `li`. Self correlation detects this form of spatial locality effectively. Global correlation is more accurate than summary profiling but less accurate than self correlation in this case because the cache outcomes of other references (which do not help to predict this reference) consume wasted space in the global history pattern. Control-flow correlation also performs well because it

```
void mark(NODE *ptr) {
  ...
  while (TRUE) { /* outer while loop */
    while (TRUE) {/* 1st inner while loop */
      if (this→n_flags & MARK)
        break; /* a marked node */
      else {
        ...;
        if (livecar(this)) {
          ...;
          prev = this;
          this = car(prev); /* go left */
          ...;
        } else if (livecdr(this)) {
          ...;
          prev = this;
          this = cdr(prev); /* go right */
          ...;
        }else break; /* a leaf node* /
      } /* ends if-else */
    } /* ends 1st inner-while */
    while (TRUE) {/* 2nd inner while loop */
      /* backup to a point where we
      can continue descending */
    } /* ends 2nd inner while */
  } /* end 1st outer while */
```

(a) Procedure `mark()`

```
LOCAL sweep(){
  ...
  for (seg = segs; seg != NULL;
    seg = seg→sg_next) {
    p = &seg→sg_nodes[0];
    for (n = seg→sg_size; n--; p++)
      if (!(p→n_flags & MARK)) {
        ...
      }
  }
}
```

(b) Procedure `sweep()`



(c) Tree traversal order in `mark()`

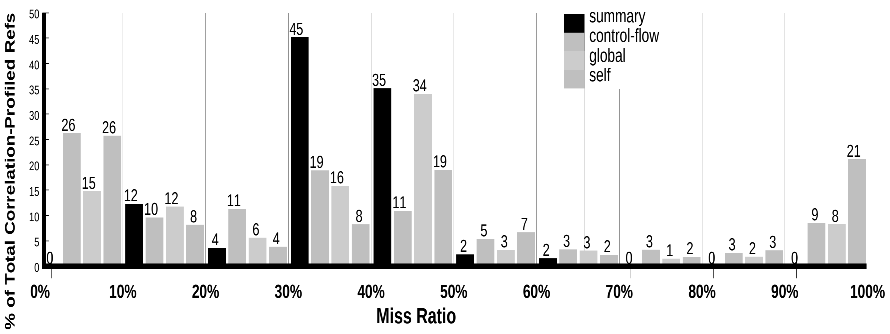
Figure 8: Procedures `mark()` and `sweep()` in `li`, and the memory access patterns of `mark()`. (Note: consecutively numbered nodes in part (c) correspond to adjacent addresses in memory.)

observes that `this→n_flags` is more likely to suffer a miss if we begin iterating in the first inner `while` loop immediately following a backup performed in the second inner `while` loop (in the preceding outer `while` loop iteration).

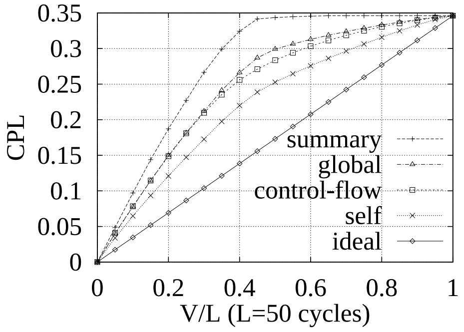
Finally, the reference `p→n_flags` in `sweep()` (shown in Figure 8(b)) is in fact an array reference written in pointer form. Both self correlation and global correlation detect the spatial locality caused by accessing consecutive elements within the array. (Although the compiler could potentially recognize this spatial locality through static analysis if it can recognize that `p→n_flags` is effectively an array reference, this is not always possible for all such cases.)

Figure 9 shows the detailed performance results for `li`. The miss ratio distribution in Figure 9(a) has ten ranges of miss ratios, each of which contains four bars corresponding to the fraction of total dynamic correlation-profiled load references that fall within this range. The bars for summary profiling represent the inherent miss ratios of these load instructions, and the other three cases represent the degree to which correlation profiling can effectively group together dynamic instances of the loads into separate paths with similar cache outcome behavior. For a correlation scheme to be effective, we would like to see a "U-shaped" distribution where references have been isolated such that they always have very high or very low miss ratios—we refer to such a case as being *strongly biased.* In contrast, if most of the references are clustered around the middle of the distribution, we say that this is *weakly biased.* Correlation profiling can outperform summary profiling by increasing the degree of bias, which we do observe in Figure 9(a). With summary profiling, 80% of the loads that we profile[2] have miss ratios in the range of 30-50% (these include the `this→n_flags` and `p→n_flags` references shown earlier in Figure 8). In contrast, with self correlation

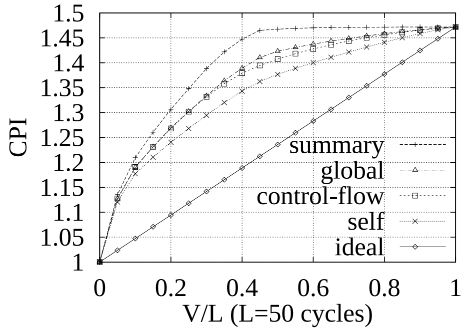
[2]Recall that we only profile loads with miss ratios between 10% and 90% among the top 15 ranked loads in terms of their contributions to total misses. Therefore the summary profiling case will never have loads outside of this miss ratio range.



(a) Miss ratio distribution of correlation-profiled load references



(b) CPL due to correlation-profiled loads



(c) Overall CPI

Figure 9: Detailed performance results for `li`.

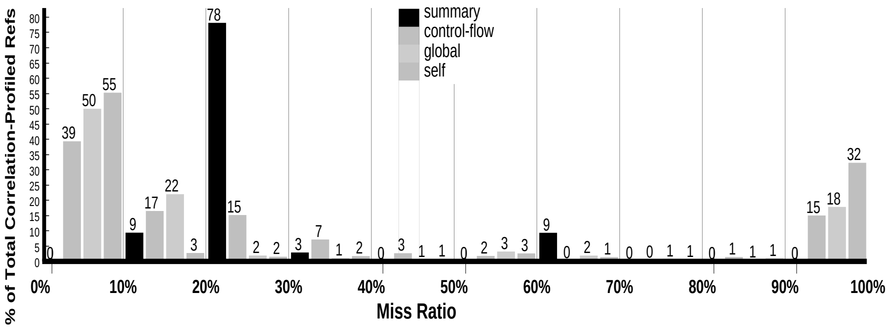
profiling only 27% of the isolated loads have miss ratios in the 30-50% range, and over 45% are either below 10% or above 90%. All three correlation schemes increase the degree of bias in this case.

This increased degree of bias of correlation-profiled loads translates into a reduction in CPL, as shown in Figure 9(b) where the CPL due to correlation-profiled loads is plotted over a range of overhead-to-latency ratios ($\frac{V}{L}$), assuming a miss latency of 50 cycles. As we have discussed in Section 3, correlation profiling partially closes the gap between summary profiling and ideal prediction. The overall CPI is also shown in Figure 9(c).

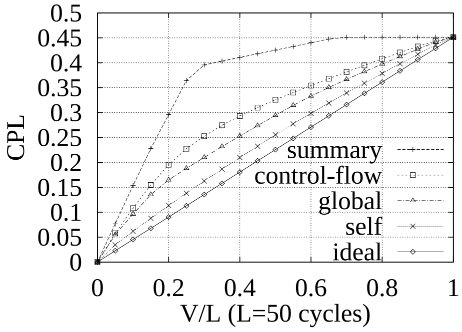
5.1.1 eqntott

Figure 10 shows detailed performance results for `eqntott`, where we see that all three forms of correlation profiling successfully increase the degree of bias and reduce CPL (and hence CPI). We now focus on the memory access behavior. Most of the load misses are caused by the four loads in `cmppt()` shown in Figure 11(a), two of which are array references (`a_ptand[i]` and `b_ptand[i]`). Clearly the spatial locality enjoyed by these two array references can be detected through self correlation (and hence global correlation). However, the access patterns of the other two loads (`a[0]`$\rightarrow$`ptand` and `b[0]`$\rightarrow$`ptand`) are more complicated. The procedure `cmppt()` has multiple call sites, and two of them, say S_1 and S_2, invoke it very frequently. Whenever `cmppt()` is called at S_1, `a[0]` will very likely be unchanged but `b[0]` will have a new value. In contrast, whenever `cmppt()` is called at S_2, `b[0]` will very likely be unchanged but `a[0]` will have a new value. Moreover, both S_1 an S_2 repeatedly call `cmppt()`. This call-site dependent behavior results in the streams of cache outcomes illustrated in Figure 11(b). Self correlation captures these streaming behavior, and control-flow correlation also predicts the cache outcomes accurately by distinguishing the two call sites of `cmppt()`.

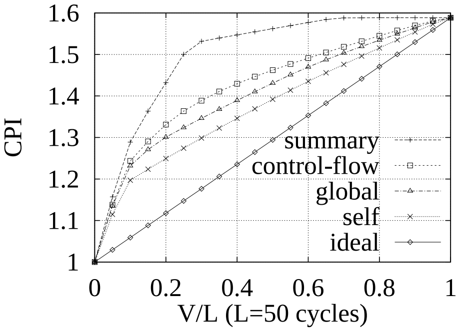
The cache outcomes of `a[0]`$\rightarrow$`ptand` also help predict those of `a_ptand[i]`—if `a[0]`$\rightarrow$`ptand` is a hit, it implies that the array `a_ptand[]` has been loaded recently, and therefore the `a_ptand[i]` references are likely to also hit. (Similar correlation also exists between `b[0]`$\rightarrow$`ptand` and `b_ptand[i]`). Hence global correlation is quite effective in this case. Control-flow correlation also predicts the cache outcomes of `a_ptand[i]` and



(a) Miss ratio distribution of correlation-profiled load references



(b) CPL due to correlation-profiled loads

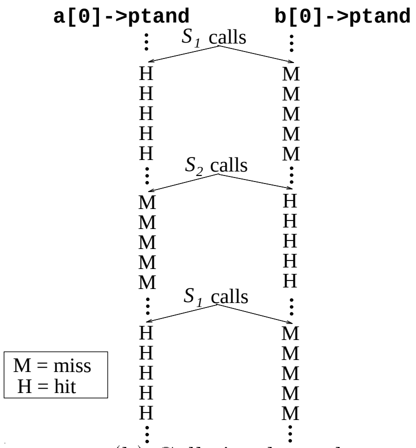


(c) Overall CPI

Figure 10: Detailed performance results for `eqntott`.

```
extern int ninputs, noutputs;
int cmppt (a, b)
PTERM *a[], *b[]; {
  register int i, aa, bb;
  register int* a_ptand, *b_ptand;
  a_ptand = a[0]->ptand;
  b_ptand = b[0]->ptand;
  for (i = 0; i < ninputs; i++) {
    aa = a_ptand[i];
    bb = b_ptand[i];
    /* the famous correlated branches */
  }
  return (0);
}
```

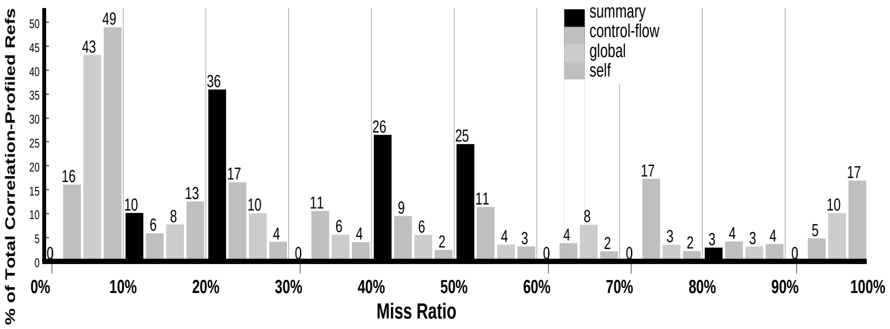
(a) Procedure `cmppt()` which causes most load misses



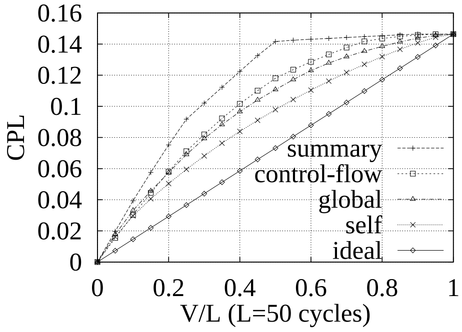
(b) Call-site dependent cache outcome patterns

Figure 11: The memory access behavior in `eqntott`. To make all loads explicit, we rewrite the two expressions `a[0]`$\rightarrow$`ptand[i]` and `b[0]`$\rightarrow$`ptand[i]` in the original `cmppt()` into the four loads (i.e. `a[0]`$\rightarrow$`ptand`, `a_ptand[i]`, `b[0]`$\rightarrow$`ptand`, and `b_ptand[i]`) shown in (a).

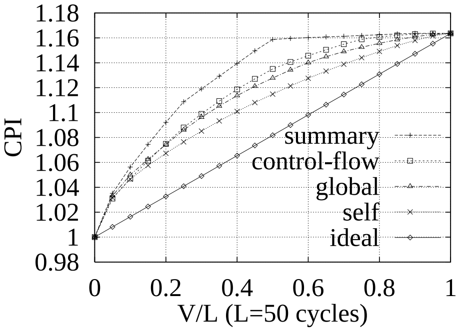
`b_ptand[i]` in an indirect fashion, by virtue of predicting those of `a[0]`$\rightarrow$`ptand` and `b[0]`$\rightarrow$`ptand`.



(a) Miss ratio distribution of correlation-profiled load references

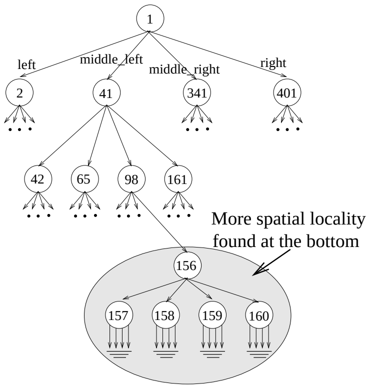


(b) CPL due to correlation-profiled loads



(c) Overall CPI

Figure 12: Detailed performance results for `perimeter`.



(a) A quadtree allocated in preorder

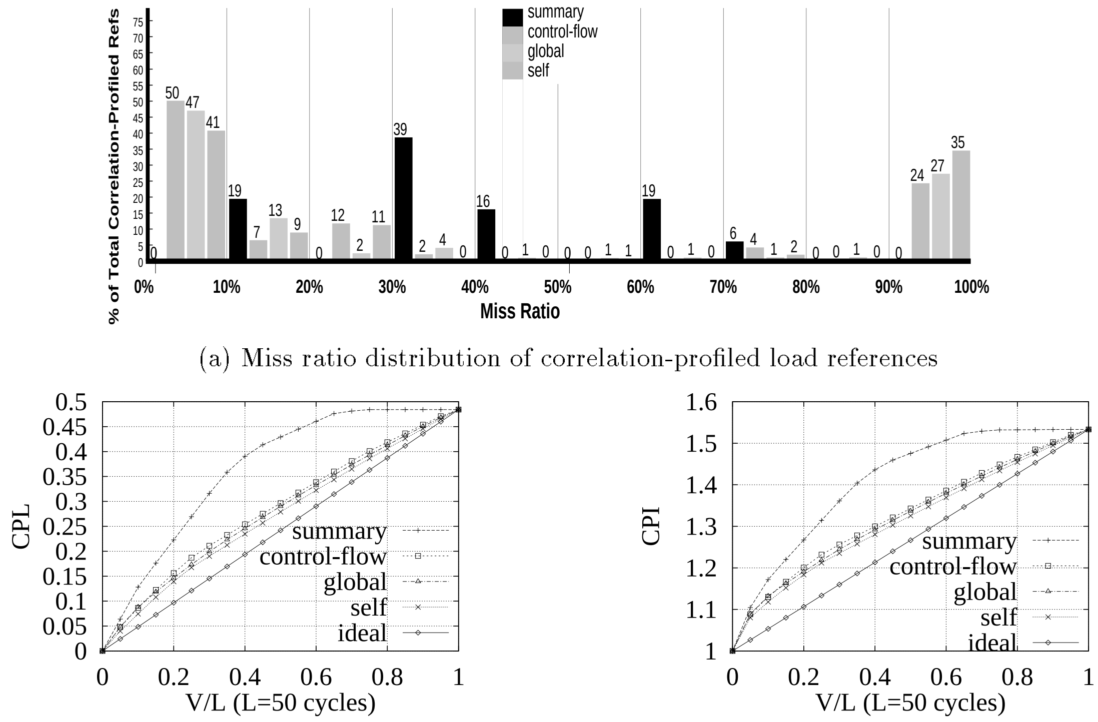
```
void middle_first(quadTree* p) {
  if (p == NULL)
    return;
  work(p→data);
  middle_first(p→middle_left);
  middle_first(p→middle_right);
  middle_first(p→left);
  middle_first(p→right);
}
```

(b) Code for traversing the quadtree in (a)

Figure 13: Example of a case where more spatial locality is found at the bottom of a tree. This example assumes that one cache line can hold three tree nodes and the tree is allocated in preorder. Nodes having consecutive numbers are adjacent in the memory.

5.1.2 perimeter and bisort

Figure 12 shows the detailed performance results for `perimeter`. The main data structures used in both `perimeter` and `bisort` are trees: quadtrees in `perimeter`, and binary trees in `bisort`. These trees are allocated in preorder, but the orders in which they are traversed are rather arbitrary. As a result, we do not see very regular cache outcome patterns (such as the one illustrated in Figure 3) for these applications.



(a) Miss ratio distribution of correlation-profiled load references

(b) CPL due to correlation-profiled loads

(c) Overall CPI

Figure 14: Detailed performance results for `mst`.

Nevertheless, there is still a considerable amount of spatial locality among consecutively accessed nodes while we are traversing around the *bottom* of a tree that has been allocated in preorder. For example, if we traverse a quadtree using the procedure `middle_first()` shown in Figure 13, we will only miss twice upon accessing nodes 156 through 160 at the tree's bottom, assuming that nodes 156 through 158 are in one cache line and nodes 159 through 161 are in another. In contrast, there is relatively little spatial locality while we are traversing the middle of the tree. Self correlation (and hence global correlation) can discover whether we are currently in a region of the tree that enjoys spatial locality. Control-flow correlation can also potentially detect whether we are close to the bottom of the tree by noticing the number of levels of recursive descent.

5.1.3 mst

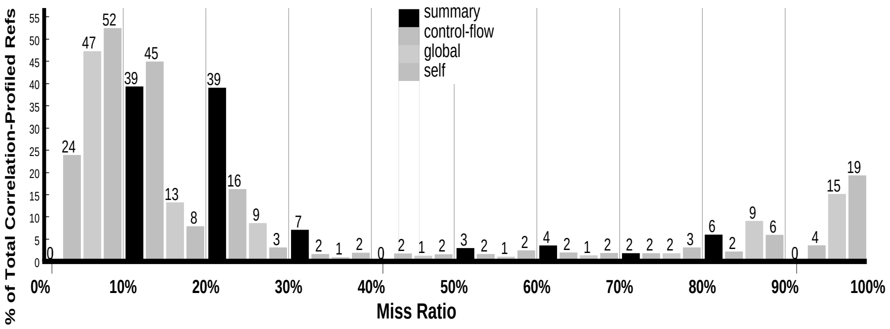
Most of the misses in `mst` (see the detailed performance results in Figure 14) are caused by loads in `HashLookup()` and the `tmp`→`edgehash` load in `BlueRule()`, as illustrated in Figure 15. The `mst` application consists of two phases: a creation phase and a computation phase. Both phases invoke `HashLookup()`, but the creation phase causes most of the misses when it calls `HashLookup()` to check whether a key already exists in the hash table before allocating a new entry for it. During the computation phase, much of the data has already been brought into the cache, and hence there are relatively few misses. Both self correlation and global correlation accurately predict the cache outcomes of these two distinct phases, since they appear as repeated streams of either hits or misses. Control-flow correlation is also effective since it can distinguish the call chains which invoke `HashLookup()`.

The load of `tmp`→`edgehash` in `BlueRule()` accesses a linked lists whose nodes are in fact allocated at contiguous memory locations. Consequently, self correlation detects this spatial locality accurately, but control-flow correlation is not helpful.

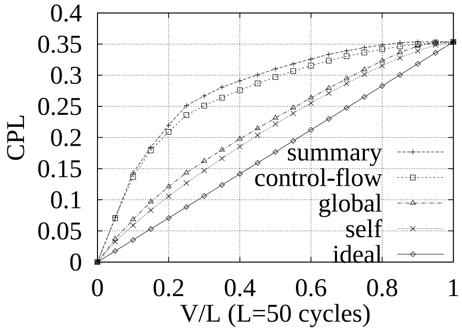
```
void *HashLookup(int key, Hash hash) {
  int j;
  HashEntry ent;
  j = (hash→mapfunc)(key);
  for (ent = hash→array[j];
    ent && ent→key !=key;
    ent = ent→next);
  if (ent) return ent→entry;
  return NULL;
}
```

```
static BlueReturn BlueRule(...)  {
  ...
  for (tmp=vlist→next; tmp;
    prev=tmp,tmp=tmp→next) {
    ...
    hash = tmp→edgehash;
    ...
  }
}
```

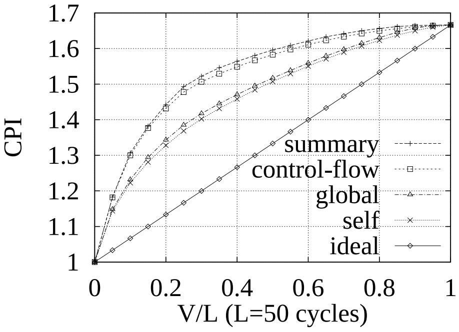
Figure 15: Pseudo codes drawn from `mst`.



(a) Miss ratio distribution of correlation-profiled load references



(b) CPL due to correlation-profiled loads



(c) Overall CPI

Figure 16: Detailed performance results for `raytrace`.

5.1.4 raytrace and tsp

In `raytrace` (refer to Figure 16 for its performance results), over 30% of load misses are caused by the pointer dereference of `tmp→bv` in `prims_in_box2()` (see Figure 17). In `subdiv_bintree()`, the two calls to `prims_in_box2()` copy part of the array `pe` of the current node `btn` to the arrays `btn1→pe` and `btn2→pe`, where `btn1` and `btn2` are the children of `btn`. This process of copying `pe` is performed recursively on the whole tree by `create_bintree()`. As a result, when `prims_in_box2()` is called upon a node n, we may have used all values in the array `pe` (referred to as `pepa` in `prims_in_box2()`) of n before at some antecedent of n and hence hopefully most data loaded by `tmp→bv` is already in the cache. In this case, most references of `tmp→bv` will hit in the cache. In contrast, if the values in `pepa` are new, all `tmp→bv` references will miss. Hence self correlation captures these streams of hits and streams of misses. In theory, control-flow correlation could also achieve good predictions by observing whether any copying occurred in the parent node—unfortunately, the profiling tool cannot record enough state across the many control-flow changes in `subdiv_bintree()` and `prims_in_box2()` to know what decisions were made in the parent node.

```
ELEMENT **prims_in_box2(pepa, ...){
ELEMENT **pepa;
  ...
  k = 0;
  npepa = alloc(...);
  for (j = 0; j < n_in; j++){
    tmp = pepa[j];
    bb = tmp→bv;
    ...
    /* computes ovlap */
    /* no change in pepa[j] */
    if (ovlap == 1) {
      npepa[k++] = pepa[j];
      ...
    }
  }
  return (npepa);
}
```

```
VOID subdiv_bintree(BTNODE* btn, ...){
  ...
  /* btn1 and btn2 are btn's children */
  btn1→pe = prims_in_box2(btn→pe, ...);
  ...
  btn2→pe = prims_in_box2(btn→pe, ...);
  ...
}
VOID create_bintree(BTNODE* root, ...){
  ...
  if (...)  {
    subdiv_bintree(root, ...);
    create_bintree(root→btn[0], ...);
    create_bintree(root→btn[1], ...);
    ...
  }
  ...
}
```

Figure 17: Pseudo codes drawn from `raytrace`.

```
Tree tsp(Tree t,int sz, ...)  {
  ...
  if (t→size <= sz) return conquer(t);
  ...
  leftval = tsp(t→left, sz, ...);
  rightval = tsp(t→right,sz, ...);
  return merge(leftval, rightval, t, ...);
}
```

```
static Tree conquer(Tree t) {
  ...
  l = makelist(t);
  for (; l; l=donext) {
    work(l→data);
    donext = l→next;
  }
  ...
}
```

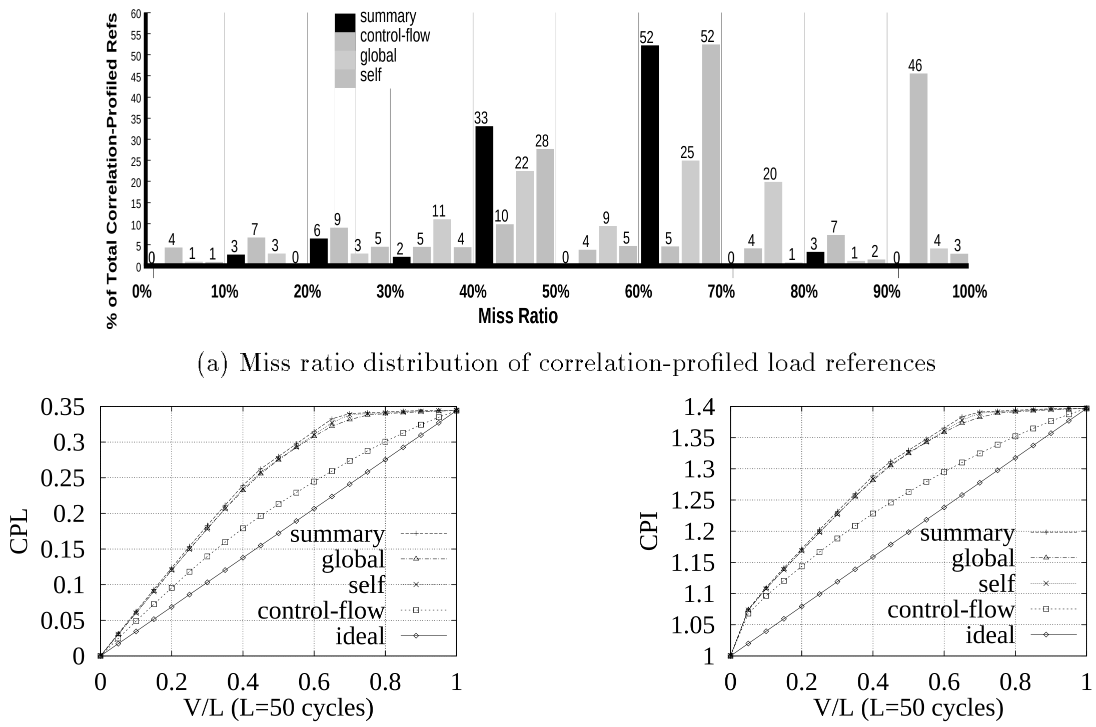
Figure 18: Pseudo codes drawn from `tsp`. Procedure `makelist(Tree t)` slings `t` into a list consisting of all nodes of t.

Similar to `raytrace`, `tsp` also traverses a binary tree recursively, and some data which is read by the current node will be read again by its descendents. As illustrated in Figure 18, the procedure `tsp()` recursively traverses the tree `t` and calls `conquer(t)` if the size of `t` is not greater than `sz`. The procedure `conquer(t)` uses `makelist(t)` to sling *every* node of `t` into a list which is then traversed by the `for` loop. Therefore since all descendents of `t` are brought into the cache whenever `conquer(t)` is called, subsequent recursion down `t→left` and `t→right` within `tsp()` results in many cache hits. Hence the `l→data` references either mainly hit or mainly miss for a given node `t`. Self correlation captures this pattern effectively. Control-flow correlation is also quite effective because it can observe the number of times `conquer()` has been called in a given recursive descent—most misses occur the first time it is invoked.

5.1.5 voronoi and compress

Control-flow correlation offers the best prediction accuracy in both of these applications. Most of the misses in `voronoi` are caused by loading `b→next` in `splice()`, which is called from three different places in `do_merge()`, as illustrated in Figure 20(a). When `splice()` is called from *call site 1*, `b→next` will hit since `ldi→next` loaded this same data into the cache just prior to the call. When `splice()` is called from the other two call sites, `b→next` is more likely to miss. Hence control-flow correlation distinguishes the behavior of these different call sites accurately. Self correlation is less effective since `b→next` does not have regular cache outcome patterns.

In `compress` (see Figure 19 for its performance results), roughly half of the misses are caused by the hash



(a) Miss ratio distribution of correlation-profiled load references

(b) CPL due to correlation-profiled loads

(c) Overall CPI

Figure 19: Detailed performance results for `compress`.

table access `htabof[i]` in the procedure `compress()` (see Figure 20(b)). The index `i` to the hash table `htab` is a function of the combination of the prefix code `ent` and the new character `c`. If this combination has been seen before, the hash probe test (`(htab[i] == fcode)`) will be true—if it has been seen *recently*, the load of `htab[i]` is likely to hit in the cache. Since the input file we use (provided by SPEC) is generated from a frequency distribution of common English texts, some strings will appear more often than others. Because of this, we expect that the condition `(htab[i] == fcode)` should be true quite frequently once many common strings have been entered into `htab`. If the last few tests of `(htab[i] == fcode)` are false, the probability that the next one is true will be high, which also implies that the next reference of `htab[i]` is more likely a hit. Therefore, control-flow correlation can make accurate predictions by examining the last several outcomes of this branch.

5.1.6 espresso, vortex, m88ksim, and go

For these four applications, correlation profiling mainly improves the cache outcome predictions for *array* references. In `espresso` (see Figure 21 for its detailed performance results), many load misses are due to array references, written in pointer form, with variable strides. Figure 22(a) shows one such example. Inside the `for` loop, `p` is incremented by `BB→wsize`, whose value depends on the call chain of `setup_BB_CC()` and ranges from 4 to 24 bytes. Different values result in different degrees of spatial locality, but all can be captured by self correlation (and hence global correlation). Control-flow correlation can also make enhanced predictions by exploiting the call-chain information.

In `vortex`, `m88ksim`, and `go`, many load misses are caused by array references located inside procedures, where array indices are passed as procedure parameters. See Figure 22(b) for an example drawn from `vortex`. Each of these procedures have multiple call sites, and the cache outcomes of those array references are mainly call-site dependent. This explains why control-flow correlation offers the highest cache outcome prediction accuracy for these three benchmarks. In `vortex`, the array index parameter values at a given call are very close or even identical most of the time, but values passed at different call sites are quite different. Consequently, references made through the same call sites will enjoy temporal and/or spatial

```
EDGE_PAIR do_merge(...) {
  ...
  v = ldi→next;
  b = ldi;
  splice(a, b) /* call site 1*/
  ...
  /* no dereferences of ldj before */
  b = ldj;
  splice(a, b) /* call site 2*/
  ...
  /* no dereferences of ldk before */
  b = ldk;
  splice(a, b) /* call site 3*/
  ...
}
splice(QUAD_EDGE a, QUAD_EDGE b) {
  ...
  beta = rot(b→next);
  ...
}
```

(a) Code fragment in `voronoi`

```
compress() {
  ...
  while ((c = getbyte()) != EOF) {
    ...
    fcode = (((long) c << maxbits) + ent);
    i = (xor((c << hshift), ent));
    if (htab[i] == fcode) {
      ent = codetab[i];
      continue;
    } else {
      ... /* store fcode into htab */ ...
    }
  }
  ...
}
```

(b) Code fragment in `compress`

Figure 20: Pseudo codes drawn from (a) `voronoi` and (b) `compress`.

locality, but those made through different call sites will not. Since a procedure is usually invoked multiple times by the same call site before being invoked by another call site, this results in a streaming pattern of a miss followed by several hits—hence self correlation also performs well in `vortex` by capturing these cache outcome patterns.

5.2 Lessons Learned from All Case Studies

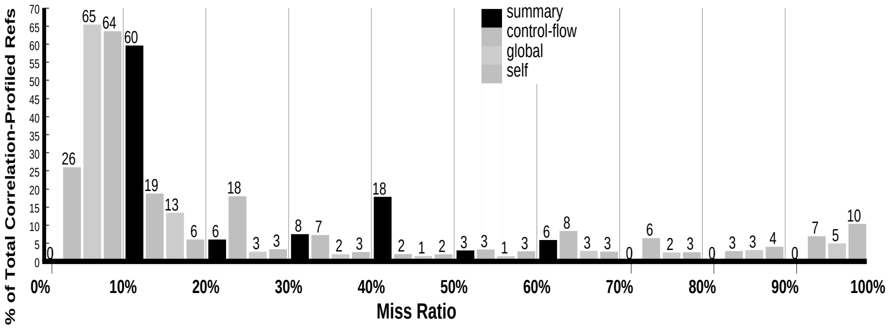
Although global correlation makes excellent predictions in some cases by correlating behavior across different load instructions (e.g., `eqntott`), in most cases it essentially assimilates self correlation, but does not perform quite as well since it records less history for a given load. Self correlation 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.

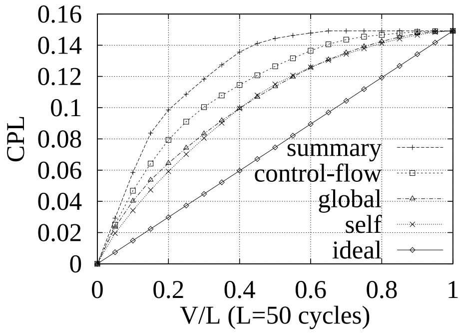
6 Applying Correlation Profiling to Prefetching

To demonstrate the practicality of correlation profiling, we used both summary and correlation profiling to guide the manual insertion of prefetch instructions into three applications: (`eqntott`, `tsp`, and `raytrace`). In the case of correlation profiling, we used *procedure cloning* [5] to isolate different dynamic instances of a static reference, and adapted the prefetching strategy accordingly with respect to the call sites. We assumed that $\frac{V}{L} = 0.1$ when deciding whether to insert prefetches,[3] and we performed fully-detailed simulations of a processor similar to the MIPS R10000 [8] (details of the memory hierarchy are shown in Figure 23(a)).

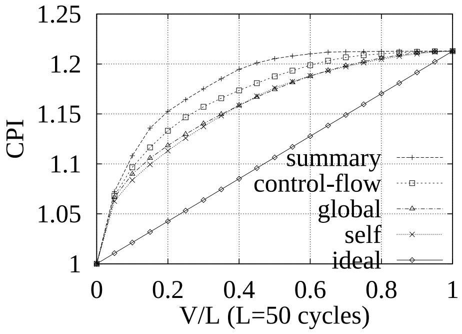
[3] We assume an average prefetch overhead (V) of two cycles, and an average miss latency (L) of 20 cycles.



(a) Miss ratio distribution of correlation-profiled load references



(b) CPL due to correlation-profiled loads



(c) Overall CPI

Figure 21: Detailed performance results for `espresso`.

```
void setup_BB_CC(pcover BB,
                 pcover CC){
    ...
    for(p=BB→data,
        last=p+BB→count*BB→wsize;
        p<last;p+=BB→wsize)
          p[0] = p[0] | ACTIVE;
    ...
}
```

(a) Code fragment in `espresso`

```
boolean ChkGetChunk(numtype ChunkNum, ...)  {
    ...
    if (((Theory→Flags[ChunkNum] & ...))
        && ...
    ...
}
```

(b) Code fragment in `vortex`

Figure 22: Pseudo codes drawn from (a) `espresso` and (b) `vortex`.

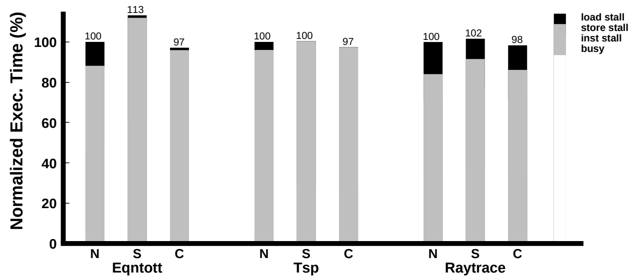
Figure 23(b) shows the resulting execution times, normalized to the case without prefetching. For these applications, summary-profiling directed prefetching actually hurts performance due to the overheads of unnecessary prefetches. In contrast, correlation profiling provides measurable performance improvements by isolating dynamic hits and misses more effectively, thereby achieving similar benefits with significantly less overhead. We would also like to point that these numbers do not represent the limit of what correlation can achieve. For example, with an 8KB primary data cache, correlation profiling offers a 10% speedup over summary profiling in the case of `eqntott`.

7 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

Memory Parameters for the MIPS R10000 Simulator	
Primary Instr and Data Caches	32KB, 2-way set-assoc.
Unified Secondary Cache	2MB, 2-way set-assoc.
Line Size	32B
Primary-to-Secondary Miss Latency	12 cycles
Primary-to-Memory Miss Latency	75 cycles
Data Cache Miss Handlers (MSHRs)	8
Data Cache Banks	2
Data Cache Fill Time (Requires Exclusive Access)	4 cycles
Main Memory Bandwidth	1 access per 20 cycles

(a) Memory Parameters



(b) Execution Time

Figure 23: Impact of correlation profiling on prefetching performance (**N** = no prefetching, **S** = prefetching directed by summary profiling, **C** = prefetching directed by correlation profiling).

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, 10, 15, 16]. While branch outcomes and cache access outcomes are quite different, it is interesting to observe that correlation-based prediction works well in both cases.

8 Conclusions

To achieve the full potential of software-based latency tolerance techniques, we have proposed *correlation profiling*, which is a technique for isolating which dynamic instances of a static memory reference are likely to suffer cache misses. We have evaluated the potential performance benefits of three different forms of correlation profiling on a wide variety of non-numeric applications. Our experiments demonstrate that correlation profiling techniques always outperform summary profiling by increasing the degree of bias in the miss ratio distribution, and this improved prediction accuracy can translate into significant reductions in the memory stall time for roughly half of the applications we study. Detailed case studies of individual applications show 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, for the most part it mainly assimilates self correlation. Finally, we observe that correlation profiling offers superior performance over summary profiling when prefetching on a superscalar processor. We believe that these promising results may lead to further innovations in optimizing the memory performance of non-numeric applications.

Appendix: Derivation of the Stall Cycles Per Load (CPL) under Five Latency-Tolerance Schemes

Denote the CPL under a particular tolerance scheme S by CPL_S. Let CPL_S^i be the CPL_S of load i in the program and f_i be the fraction of references made by load i out of the total references of all loads. Then:

$$CPL_S = \sum_i CPL_S^i \times f_i \qquad (1)$$

Let L be the cycles stalled upon a load miss, V be the overhead of applying the latency-tolerance technique T to a load reference, m_i is miss ratio of load i and m is the overall miss ratio of all loads.

CPL_{never}: A load reference is stalled only when it is a cache miss, so:

$$CPL_{never} = mL \tag{2}$$

CPL_{always}: T fully tolerates the latencies of all load references but always incurs the overhead, so:

$$CPL_{always} = V \tag{3}$$

$CPL_{single\_action\_per\_load}$: The miss ratio m_i decides whether T should be applied to load i:

$$CPL^{i}_{single\_action\_per\_load} = \begin{cases} m_i L & \text{if } m_i \leq \frac{V}{L} \text{ (i.e. not apply T)} \\ V & \text{otherwise (i.e. apply T)} \end{cases} \tag{4}$$

$$\begin{aligned} CPL_{single\_action\_per\_load} &= \sum_{i \in A} CPL^{i}_{single\_action\_per\_load} \times f_i + \sum_{i \in NA} CPL^{i}_{single\_action\_per\_load} \times f_i \\ &= V \sum_{i \in A} f_i + L \sum_{i \in NA} m_i f_i \qquad \text{by (4)} \end{aligned} \tag{5}$$

where A is the set of loads with miss ratios $> \frac{V}{L}$ and NA is the set of loads with miss ratios $\leq \frac{V}{L}$.

$CPL_{multiple\_actions\_per\_load}$: T is only applied to references of load i that belong to contexts with miss ratios $> \frac{V}{L}$. The formula for $CPL^{i}_{multiple\_actions\_per\_load}$ can be simply obtained adding an extra level to Equation (5) to capture the notion of contexts within load i. That is:

$$CPL^{i}_{multiple\_actions\_per\_load} = V \sum_{j \in A_i} f_{i,j} + L \sum_{j \in NA_i} m_{i,j} f_{i,j} \tag{6}$$

where A_i is the set of contexts of load i with miss ratios $> \frac{V}{L}$, NA_i is the set of contexts of load i of miss ratios $\leq \frac{V}{L}$, $m_{i,j}$ is the miss ratio of context j of load i, and $f_{i,j}$ is the fraction of references of load i that are on context j. $CPL_{multiple\_actions\_per\_load}$ can be obtained by substituting $CPL^{i}_{multiple\_actions\_per\_load}$ into Equation (1).

CPL_{ideal}: Under this ideal scheme, load-miss latencies are fully tolerated and the overhead is only incurred to miss references:

$$CPL_{ideal} = mV \tag{7}$$

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] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, October 1996.

[8] MIPS Technologies Inc. *R10000 Microprocessor User's Manual, Version 1.1*, 1996.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

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