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

Chi-Keung Luk and Todd C. Mowry

Department of Computer Science
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada M5S 3G4
{luk,tcm}@eecg.toronto.edu

Technical Report CSRI-TR-359
February 1997

**Abstract**

To fully exploit the benefit of *software-based latency tolerance* techniques, one must be careful to apply them only to the dynamic references that are likely to miss - 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. 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 repeated patterns often exist in the cache outcomes for a given reference, and control-flow correlation succeeds because many cache outcomes are call-chain dependent.

## 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 processors that support *non-blocking loads*, the compiler can tolerate modest latencies by scheduling loads early relative to when their results are consumed [9, 16]. Compilers can also insert *prefetch* instructions to bring data lines into the cache before they are needed [12, 13]. Software-controlled *multithreading* has also been proposed through the use of *informing memory operations* [11], whereby software explicitly switches from one concurrent thread to another upon a cache miss.

While these software-based techniques provide latency-hiding benefits, they also typically incur runtime overheads. Aggressive instruction scheduling to exploit non-blocking loads increases register lifetimes which can lead to spilling. Software-controlled prefetching requires additional instructions to compute prefetch addresses and launch the prefetches themselves. Software-controlled multithreading incurs any overheads associated with using the informing memory operation mechanism.[1] While the benefit of a technique 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.

Previous studies on compiler-inserted prefetching for numeric applications [13] have demonstrated the importance of prefetching only those dynamic references that are likely to suffer misses, rather than indiscriminately

[1]For the condition-code implementation of informing memory operations, this overhead includes explicit branches after each memory reference. For the low-overhead trap mechanism, it potentially involves the overheads of setting the miss handler address register (MHAR), and of consuming branch resources more quickly.

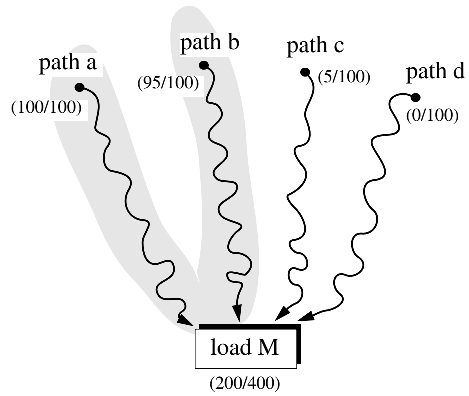



Figure 1: Illustration of how dynamic miss instances might be predicted by correlating misses with "path" information. ($X/Y$ means $X$ misses out of $Y$ references.)

prefetching all array references inside loops. For these numeric applications, the compiler can predict when cache misses are likely to occur through *locality analysis*, and can transform the code to statically isolate these dynamic miss instances through *loop splitting* techniques such as peeling and unrolling. In this paper, we focus instead on isolating dynamic miss instances in *non-numeric* applications, which is a considerably more difficult problem for the following two reasons. First, there is no analog to "locality analysis" for non-numeric codes, since complications such as pointers to heap-allocated objects and complex control flow make it difficult (if not impossible) for the compiler to statically analyze memory access patterns. Second, even if the compiler understands precisely when the dynamic misses will occur, isolating them is not as simple as applying "loop splitting" techniques. Despite these challenges, isolating dynamic miss instances in non-numeric codes is important if we are to achieve the full benefit of software-based latency tolerance techniques.

## 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 attractive method of collecting such information is to use *informing memory operations* [11], which allow software to directly observe and react to its cache miss behavior while retaining full access to its program state. One simple type of profiling information which can be collected using informing memory operations is the precise miss rate of all memory references. A previous study [10] has demonstrated that per-reference miss rates can be captured with low runtime overheads (less than a 25%) and tolerable data cache perturbations. Throughout the remainder of this paper, we will refer to this approach as *summary profiling*, since the miss rate is summarized as a single value.

If summary profiling indicates that all important memory reference instructions have miss rates 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 rates (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 rate. The current state-of-the-art approach for dealing with intermediate miss rates is to treat all static memory references with miss rates 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.

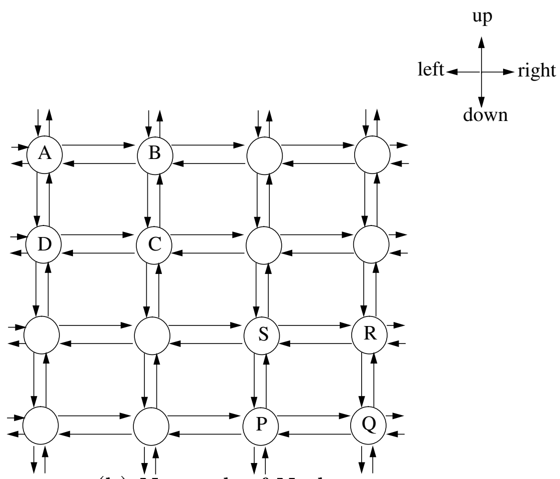
### 1.1.1 Correlation Profiling

Profiling tools based on informing memory operations can collect more sophisticated information than simply the per-reference miss rates. 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 a miss ratio of 50% (200 misses out of 400 references). However, depending on the abstract "path" which leads to the load, we may see more predictable behavior. In this example, paths "**a**" and "**b**" result in a

```
struct node {
  int data;
  struct node *left, *right, *up, *down;
};
void walk(node* p) {
  work(p→data); /*control-flow profiled*/
  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) Example Code




(b) Network of Nodes

Figure 2: Example of how control-flow correlation can detect data reuse.

high likelihood of the load missing, whereas paths "**c**" and "**d**" do not. Hence we would like to apply a latency tolerance technique along paths "**a**" and "**b**", but not along paths "**c**" and "**d**".

The abstract "paths" 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 construct these paths. 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* [5, 14, 21, 22, 23, 24]

### 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.

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 an analytical model for estimating the resulting performance improvements. Section 4 describes our experimental framework. In Section 5, we present our experimental results which quantify the performance advantages of correlation profiling in a collection of 22 non-numeric applications. In addition to presenting overall performance results, we also discuss individual applications in detail to develop a deeper understanding of when and how correlation profiling is effective. In Section 6, we discuss how compilers can exploit correlation profiling to enhance performance. Finally, we discuss related work and present conclusions in Sections 7 and 8.

## 2 Correlation Profiling Techniques

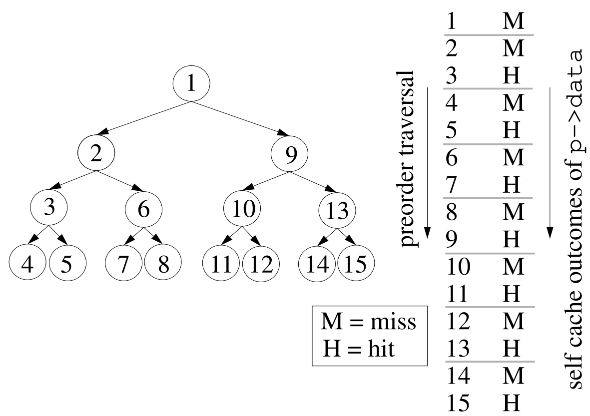
In this section, we propose and motivate three new correlation profiling techniques for predicting cache outcomes: *control-flow correlation*, *self correlation*, and *global correlation*. To illustrate the usefulness of these schemes, we show how they apply to a real-world application (`li`).

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

Figure 3: Example of how control-flow correlation can detect data replacement.

```
void createTree(treeNode* root, int height) {
   if (height > 1) {
      root→left = alloc(sizeof(treeNode));
      createTree(root→left, height-1);
      root→right = alloc(sizeof(treeNode));
      createTree(root→right, height-1);
   }
}
void preorder(treeNode* p) {
   if (p != NULL) {
      work(p→data);/* self-correlation profiled */
      preorder(p→left);
      preorder(p→right);
   }
}
```

(a) Example Code




(b) Tree Constructed and Traversed Both in Preorder

Figure 4: Example of using self-correlation profiling to detect spatial locality. Nodes numbered consecutively are adjacent in the memory.

## 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 in the following two situations.

**Detecting Data Reuse:** 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, where a network of nodes 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 when we access `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*), there is a high probability that the next reference of `p→data` will hit in the cache.

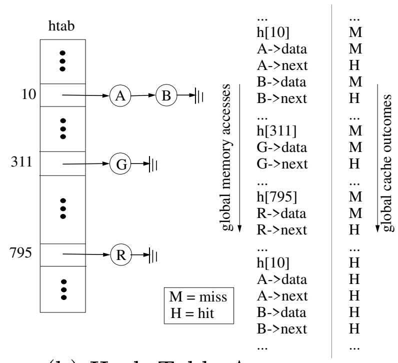
**Detecting Data Replacement:** Some control-flow paths may increase the likelihood of a cache miss by displacing a data line before it can be reused by the load. For example, if the "`x > 0`" condition is true in Figure 3, the subsequent `for` loop is likely to displace `*p` from the primary cache before 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 a pattern consisting of the $N$ previous cache outcomes of $L$ itself. This approach is particularly useful for detecting forms of spatial locality which are not readily obvious at compilation time. For example, consider the case in Figure 4 where a tree is constructed by `createTree()` in preorder. Assume that consecutive calls to `alloc()` 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

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

(a) Example Code




(b) Hash-Table Accesses

Figure 5: Example of using global-correlation profiling to detect bursty patterns of cache misses.

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 rate 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. Note that the cache outcomes of earlier instances of $L$ itself may potentially be contained in this global history pattern, and thus there may be some overlap between the behaviors detected by self correlation and by global 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 replaced 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 5 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 we access the same hash entry 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.

## 2.4 An Example: `li`

Having described the three forms of correlation profiling considered in this paper, we now demonstrate how they apply to a real non-numeric application: `li`, the "lisp interpreter" program taken from the SPEC95 integer benchmark suite. Over half of the total load misses experienced by `li` in a two-way set associative 8KB primary data cache 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 6.

The access patterns behave as follows. The procedure `mark()` traverses a binary tree through the three `while` loops shown in Figure 6(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 performed by the second inner `while` loop. The tree is allocated in preorder, similar to the one shown in Figure 4, 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 6(c).

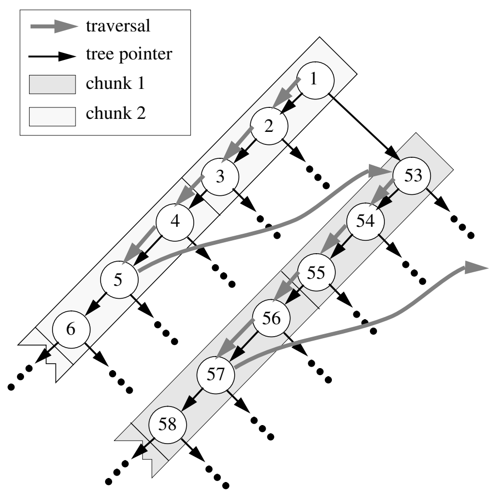
All three types of correlation profiling provide better cache outcome predictions than summary profiling for the `this→n_flags` reference in `mark()` for `li` (we will show the exact numbers later in our experimental results in Section 5). Self correlation detects this form of spatial locality effectively. 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.

```
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 6: 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.)

Control-flow correlation also performs well because it 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 6(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.)

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

# 3 Estimating Performance: An Analytical Model

In this section, we introduce the performance metrics which will be used later in Section 5 to evaluate our experimental results, and we also present an analytical model to provide insight into how correlation profiling can improve performance. Recall that execution time can be expressed as follows:

$$\text{Execution Time} = \text{Cycle Time} \times \text{Cycles Per Instruction } (\mathit{CPI}) \times \text{Instruction Count}.$$

We can break down the $CPI$ further into the following three components:

$$CPI = CPI_{execution} + CPI_{load\_stall} + CPI_{store\_stall} \tag{1}$$

where $CPI_{execution}$ is the cycles spent on the actual execution of instructions, and $CPI_{load\_stall}$ and $CPI_{store\_stall}$ are the stall cycles due to load and store cache misses, respectively. Since store latency can typically be hidden through buffering and pipelining, we would expect $CPI_{store\_stall}$ to be small (perhaps negligible).

We can further break down $CPI_{load\_stall}$ as follows:

$$CPI_{load\_stall} = \text{Loads Per Instruction } (\mathit{LPI}) \times \text{Stall Cycles Per Load } (\mathit{SCPL}). \tag{2}$$

When we apply a software-based latency tolerance technique to hide load latency, we expect $SCPL$ to decrease, but an overhead will also be incurred. To account for this overhead, equation (2) can be modified as follows:

$$\begin{aligned} CPI_{load\_stall} \text{ (with latency tolerance)} &= \mathit{LPI} \times (SCPL + \text{Overhead Cycles Per Load}) \\ &= \mathit{LPI} \times \text{Effective Stall Cycles Per Load } (\mathit{ESCPL}) \end{aligned} \tag{3}$$

The objective of any latency tolerance scheme is to minimize $ESCPL$ by paying as little overhead to hide as much latency as possible. (Note that for the ease of derivation, any instruction overhead is counted in $ESCPL$ and not as a direct increase in the instruction count.)

To gain insight into the potential impact of correlation profiling, we consider the performance of a given latency tolerance technique ($T$) when it is applied in the following five ways.

**None:** $T$ is never applied—i.e. we predict that all loads always hit.

**All:** $T$ is applied to every load of the program—i.e. we expect all loads to always miss.

**Single action per load:** For static load instructions with miss ratios greater than a threshold of $\frac{V}{L}$ (where $L$ is the load-miss latency and $V$ is the overhead of applying $T$ to each load reference), we apply $T$ to every dynamic instance of that load. We choose a threshold of $\frac{V}{L}$ because applying $T$ to every dynamic instance of a load with a smaller miss ratio will hurt performance. Notice that in this all-or-nothing scheme, $T$ is either always applied or never applied to a given load instruction. This scheme corresponds to the best that we can do with *summary profiling* information.

**Multiple actions per load:** We use *correlation profiling* to separate the dynamic instances of a given load into multiple "paths", where each path has its own miss ratio. We apply $T$ to paths with miss ratios greater than $\frac{V}{L}$. Hence, a given static load instruction can have *multiple* actions associated with it (i.e. either apply $T$ or not), each associated with a given abstract "path".

**Ideal:** $T$ is applied only to the precise set of dynamic load references that suffer cache misses. Achieving this in practice is unlikely, but it represents an upper bound on how well we might do with perfect cache outcome prediction.

The derivation of the $ESCPL$ for these five schemes (denoted by $ESCPL_{none}$, $ESCPL_{all}$, $ESCPL_{single\_action\_per\_load}$, $ESCPL_{multiple\_actions\_per\_load}$, and $ESCPL_{ideal}$) is given in the Appendix. Figure 7 illustrates how they change as a function of $\frac{V}{L}$. Both $ESCPL_{single\_action\_per\_load}$ and $ESCPL_{multiple\_actions\_per\_load}$ are non-linear with respect to $\frac{V}{L}$ because the fraction of load references to which $T$ is applied depends on the value of $\frac{V}{L}$ itself (see equation (7) in the Appendix). Note that they are bounded by $ESCPL_{none}$, $ESCPL_{all}$, and $ESCPL_{ideal}$.

# 4 Experimental Methodology

We now describe the applications used in our study and our methodology for collecting experimental results. We measured the impact of correlation profiling on the following 22 non-numeric applications: the entire SPEC95 integer benchmark suite [7], the additional integer benchmarks contained in the SPEC92 suite [6], uniprocessor versions of two graphics applications from SPLASH-2 [20], eight applications from Olden [15] (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 [17] to instrument these binaries, and piped the resulting trace into our simulator which produces detailed performance statistics.

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

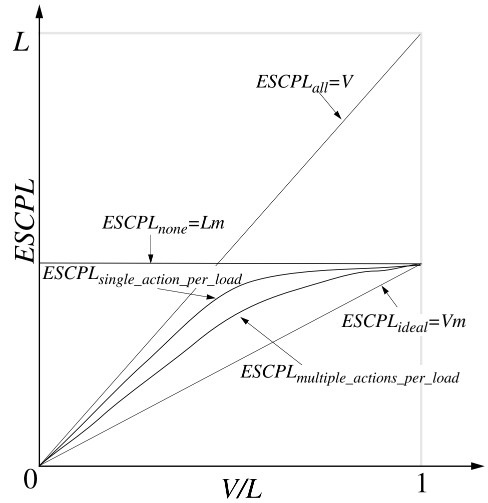



Figure 7: Illustration of the $ESCPL$ for the five different approaches of applying a latency tolerance scheme. In this figure, $m$ is the overall miss ratios of all loads in the program, $L$ is the load miss latency, and $V$ is the overhead of the latency tolerance technique.

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 |

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 rates. We will refer to these loads as the *correlation-profiled loads*.

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). In addition, we made the following methodological decisions to make our experiments feasible.

**Data Cache Size:** 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 rate 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 data cache with 32 byte lines.

**Path Pruning:** In some cases, we are forced to discard paths for the sake of saving memory in our simulator. We organize the correlation paths as trees within our simulator, where two paths belong in the same subtree if they share a common prefix path ("prefix" meaning the most recent portion of the path). We can prune all paths under a given subtree if their miss ratios fall onto the same side of a given threshold (we use 50% in our experiments), thus treating them as a single path with the combined miss ratio. (This methodology is quite similar to the path pruning algorithm described by Young and Smith [24] within the context of maintaining paths for branch prediction.)

**Rarely Executed Paths:** In some cases, we find paths with very small reference counts. Although these rarely executed paths may appear to produce strong correlations with cache outcomes (in the extreme case, a path executed only once will have perfect correlation), they are in fact useless for optimization purposes since the cost of isolating them is prohibitively high. To account for this, our simulator considers a path to be indistinguishable and ignores it for correlation profiling purposes if it is executed fewer than ten times (the statistics for all such paths are lumped together as a single value).

# 5 Experimental Results

In this section, we present experimental results to quantify the performance benefits offered by correlation profiling. We begin with an overview of how correlation profiling reduces the effective stall cycles per load ($ESCPL$) relative to summary profiling, we then present detailed case studies to understand the issues in greater depth, and finally we summarize what we have learned from these experiments.

## 5.1 Overall Improvement in Effective Stall Cycles Per Load ($ESCPL$)

Figure 8 shows how summary profiling (**P**) and the three correlation profiling schemes—control-flow (**C**), self (**S**), and global (**G**)—reduce the $ESCPL$ relative to the original code (without latency tolerance). Each bar is normalized with respect to $ESCPL_{none}$ (described earlier in Section 3), and is broken down into the following three 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 sum of these two sections is the *misprediction penalty* of a particular scheme. The bottom section ("*Predict MISS / Actual MISS*") is the *useful* overhead paid for tolerating references that really do miss. Hence the overall reduction in the height of the bar is the net gain from latency tolerance.

As discussed earlier in Section 3, our threshold for predicting whether a reference misses 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 8 shows results with two values of $\frac{V}{L}$: 0.25 and 0.5.

Overall, we see in Figure 8 that correlation profiling can offer a significantly lower $ESCPL$ than summary profiling by reducing both aspects of the misprediction penalty. For $\frac{V}{L} = 0.25$, summary profiling tends to apply latency tolerance aggressively, thus resulting in a noticeable amount of wasted overhead (the "*Predict MISS / Actual HIT*" section). In contrast, for $\frac{V}{L} = 0.50$, summary profiling tends to be more conservative, thus resulting in many untolerated misses (the "*Predict HIT / Actual MISS*" cases).

Among the three correlation profiling schemes, self correlation offers the best performance in the largest number of cases. Global correlation is often competitive with self correlation (surpassing it only in `gcc` and `radiosity`), which is not surprising since the global history sequences often capture parts of the self history. Control-flow correlation provides significant improvements over the other schemes in `compress`, `voronoi` and `m88ksim`. An interesting observation from Figure 8 is that in more than half of the applications, *all three* of the correlation profiling schemes provide substantial improvements over summary profiling. This suggests that some cache outcome patterns may be predictable by more than one form of correlation—we will examine this possibility further in the next section as we look at individual case studies.

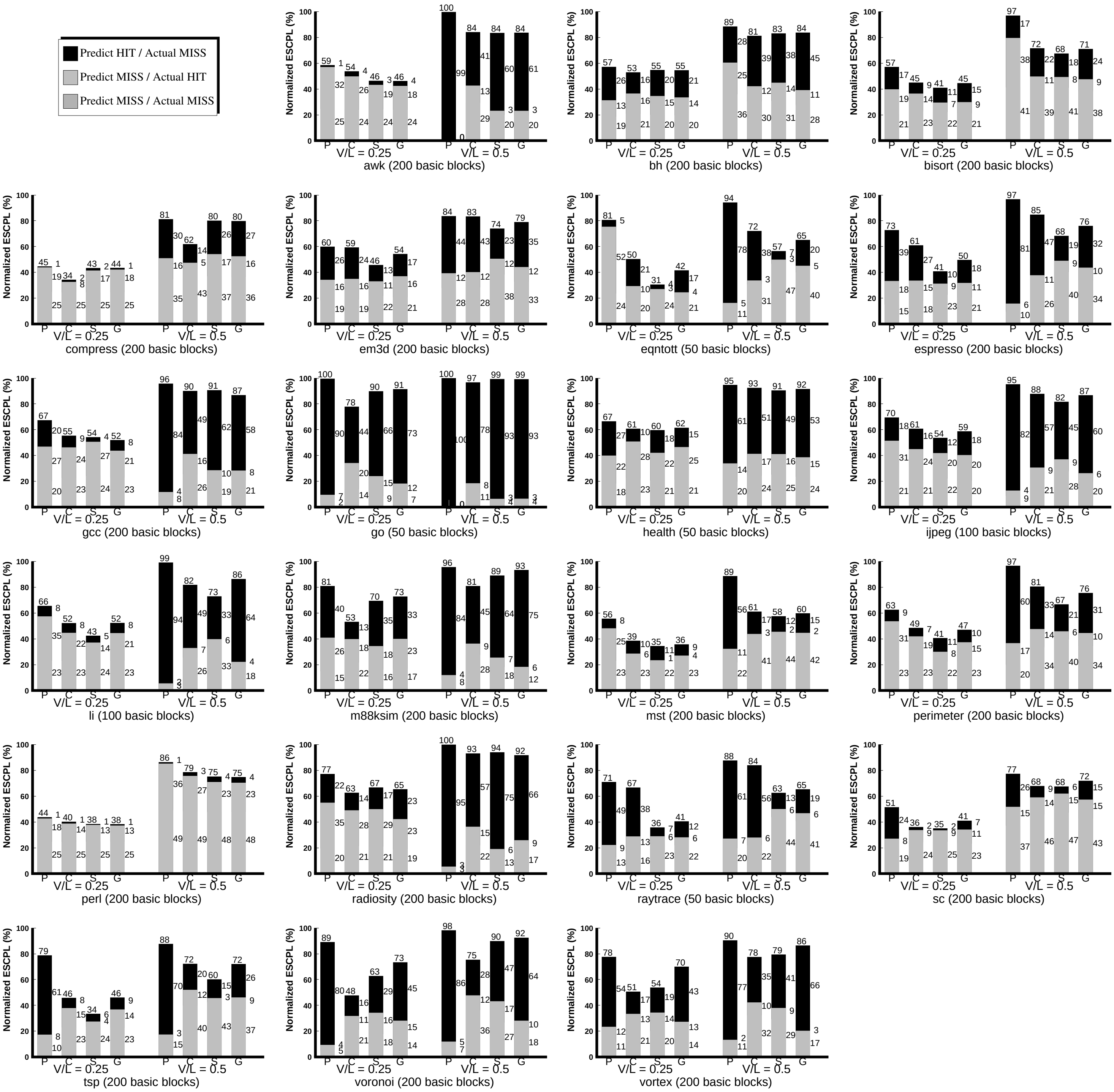
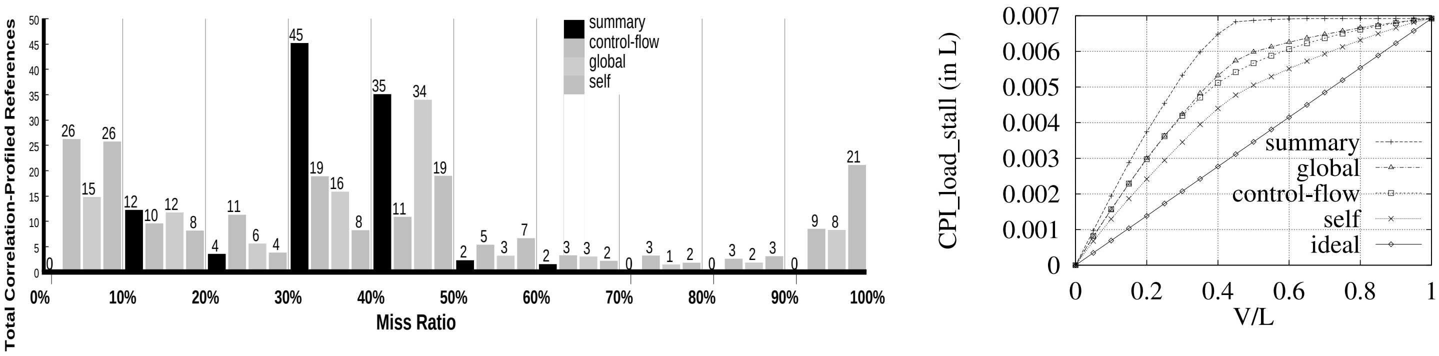



Figure 8: Effective stall cycles per load ($ESCPL$), normalized to the case without latency tolerance, under four profiling schemes (**P** = summary profiling, **C** = control-flow correlation, **S** = self correlation, **G** = global correlation). Two ratios of latency tolerance overhead ($V$) to miss latency ($L$) are shown ($\frac{V}{L} = 0.25, 0.5$). The number next to the benchmark name is the maximum path length used in control-flow correlation profiling.

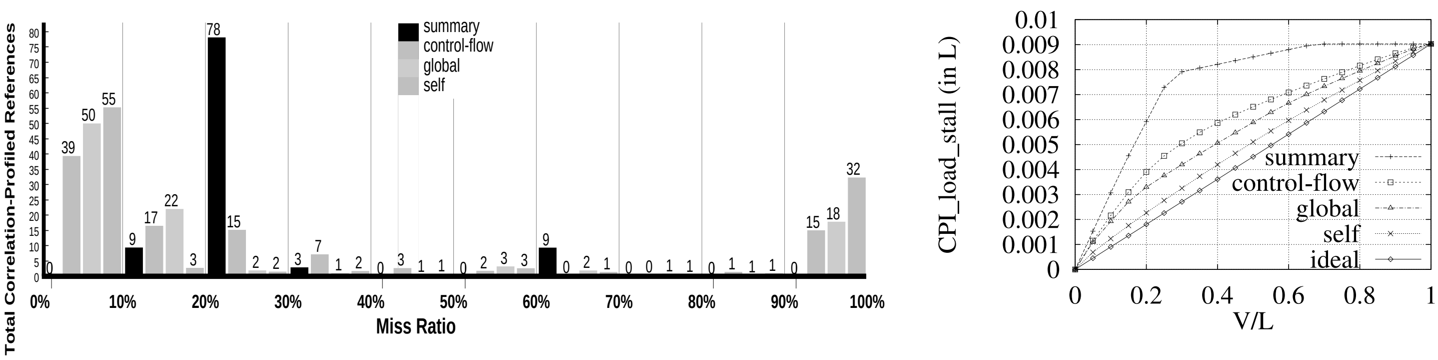
## 5.2 Case Studies

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 two performance metrics: $CPI_{load\_stall}$, and the *miss ratio distribution*. While $CPI_{load\_stall}$ measures the impact 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.




(a) Miss ratio distribution of correlation-profiled load references (b) $CPI_{load\_stall}$ due to correlation-profiled loads

Figure 9: Detailed performance results for `li`.




(a) Miss ratio distribution of correlation-profiled load references (b) $CPI_{load\_stall}$ due to correlation-profiled loads

Figure 10: Detailed performance results for `eqntott`.

### 5.2.1 li

Figure 9 shows the detailed performance results for `li` (its memory access behavior has already been discussed in Section 2.4). 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

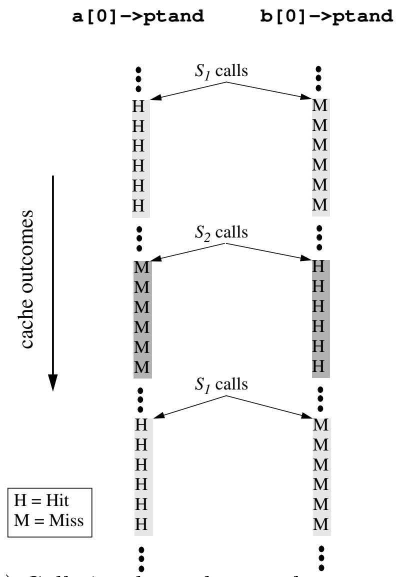
[2]Recall that we only profile loads with miss ratios between 10% and 90% among the top 15 ranked loads in terms of their contributions to total misses. Therefore the summary profiling case will never have loads outside of this miss ratio range.

the this→n_flags and p→n_flags references shown earlier in Figure 6). In contrast, with self correlation 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 translates into a reduction in $CPI_{load\_stall}$, as shown in Figure 9(b). As we see in this graph, where $CPI_{load\_stall}$ is plotted normalized to $L$ (the miss latency) over a range of overhead-to-latency ratios ($\frac{V}{L}$), correlation profiling partially closes the gap between summary profiling and ideal prediction. (To compute the actual impact on performance for a given latency tolerance technique and system, one can choose the appropriate $\frac{V}{L}$ ratio, and multiply the resulting $CPI_{load\_stall}$ by $L$ to compute the actual contribution to $CPI$.)

```
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]→ptand[i] and b[0]→ptand[i] in the original cmppt() into the four loads (i.e. a[0]→ptand, a_ptand[i], b[0]→ptand, and b_ptand[i]) shown in (a).

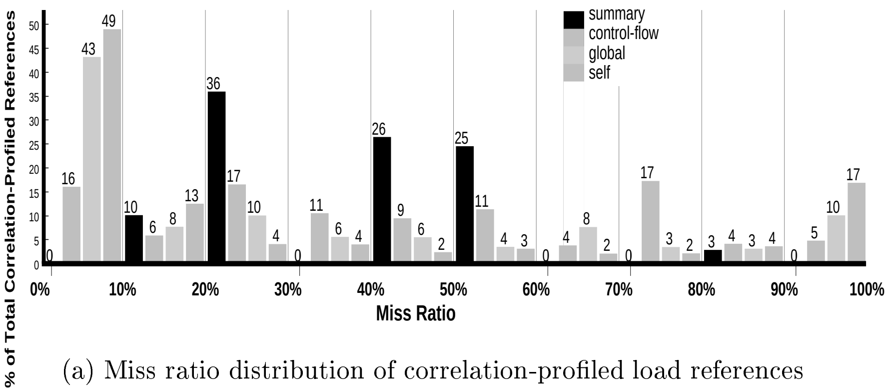
### 5.2.2 eqntott

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 $CPI_{load\_stall}$. 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]→ptand and b[0]→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().

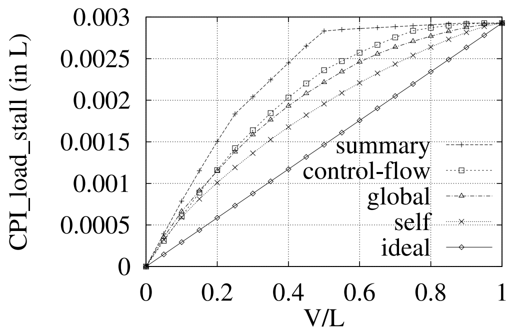
The cache outcomes of a[0]→ptand also help predict those of a_ptand[i]—if a[0]→ptand is a hit, it implies that the array a_ptand[] has been loaded recently, and therefore the a_ptand[i] references are likely to also hit. (Similar correlation also exists between b[0]→ptand and b_ptand[i]). Hence global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

### 5.2.3 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 4) for these applications. 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.

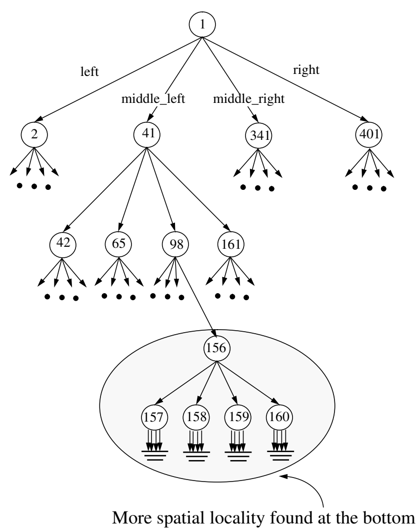



(a) Miss ratio distribution of correlation-profiled load references




(b) $CPI_{load\_stall}$ due to correlation-profiled loads

Figure 12: Detailed performance results for `perimeter`.

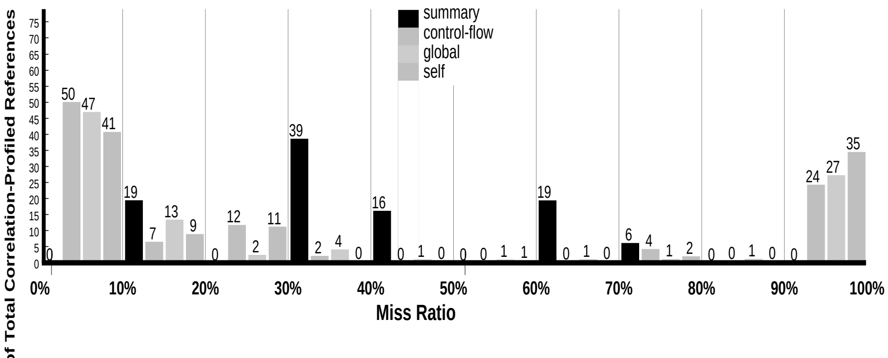



(a) A quadtree allocated in preorder

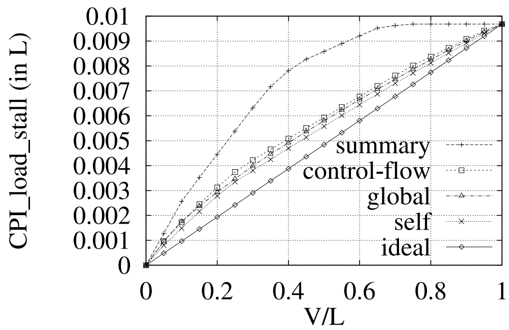
```
void middle_first(quadTree* p) {
  if (p == NULL)
    return;
  work(p->data);
  middle_first(p->middle_left);
  middle_first(p->middle_right);
  middle_first(p->left);
  middle_first(p->right);
}
```

(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.




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



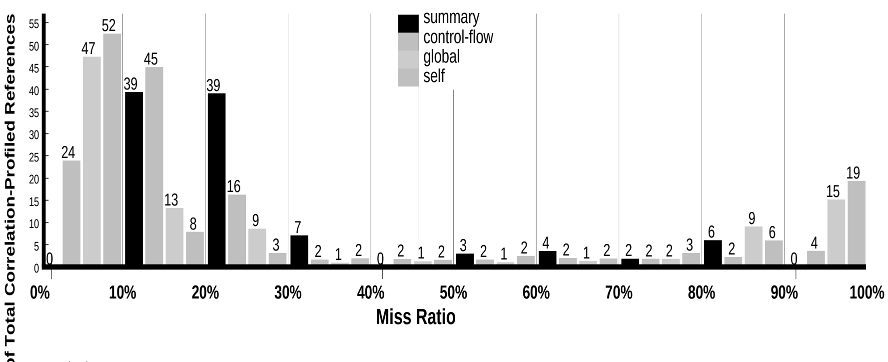

(b) $CPI_{load\_stall}$ due to correlation-profiled loads

Figure 14: Detailed performance results for `mst`.

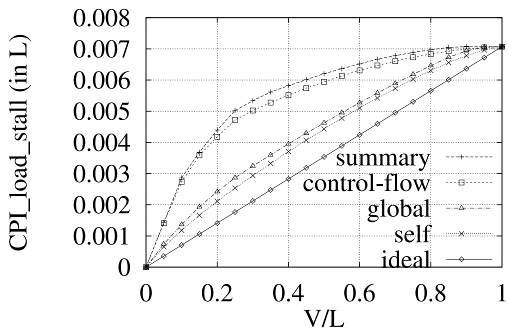
```
void *HashLookup(unsigned int key, Hash hash) {
  int j;
  HashEntry ent;
  j = (hash→mapfunc)(key);
  for (ent = hash→array[j];
    ent && ent→key !=key;
    ent = ent→next);
  if (ent) return ent→entry;
  return NULL;
}
```

```
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) $CPI_{load\_stall}$ due to correlation-profiled loads

Figure 16: Detailed performance results for `raytrace`.

### 5.2.4 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.

```
ELEMENT **prims_in_box2(pepa, ...){
ELEMENT **pepa;
  ...
  k = 0;
  npepa = alloc(...); /* memory allocation */
  for (j = 0; j < n_in; j++){
    tmp = pepa[j];
    bb = tmp->bv;
    ...
    /* computes the value of ovlap */
    /* no change in pepa[j]'s value */
    if (ovlap == 1) {
      npepa[k++] = pepa[j]; /* copying */
      ...
    }
  }
  return (npepa);
}
```

```
VOID subdiv_bintree(BTNODE* btn, ...){
  ...
  /* btn1 and btn2 are the two children of btn */
  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);   /* slings t into a list */
                     /* consisting of all nodes of t */
  for (; l; l=donext) {
    work(l->data);
    donext = l->next;
  }
  ...
}
```

Figure 18: Pseudo codes drawn from tsp.

#### 5.2.5 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.

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.

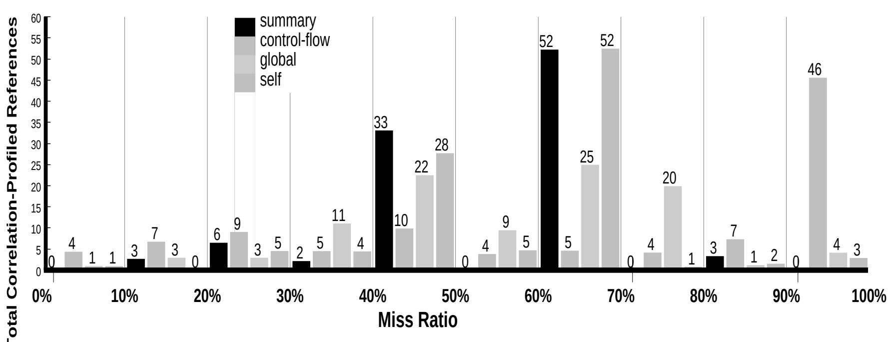
```
EDGE_PAIR do_merge(...)  {
  ...
  v = ldi→next;
  b = ldi;
  splice(a, b) /* call site 1*/
  ...
  /* no dereferences of ldj before */
  b = ldj;
  splice(a, b) /* call site 2*/
  ...
  /* no dereferences of ldk before */
  b = ldk;
  splice(a, b) /* call site 3*/
  ...
}
splice(QUAD_EDGE a, QUAD_EDGE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

(a) Code fragment in `voronoi`

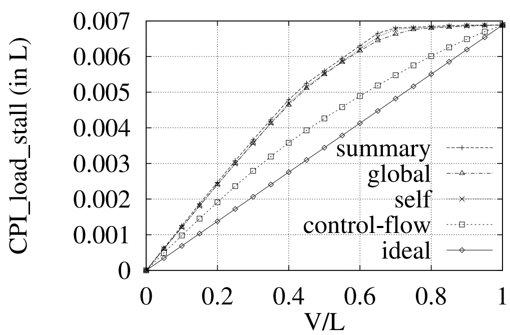
```
compress() {
  ...
  while ((c = getbyte()) != EOF) {
    ...
    fcode = (long) (((long) c << maxbits) + ent);
    i = (xor((c << hshift), ent)); /* xor hashing */
    if (htab[i] == fcode) {
      ent = codetab[i];
      continue;
    } else
      ...  /* store fcode into an entry of htab */ ...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

(b) Code fragment in `compress`

Figure 19: Pseudo codes drawn from (a) `voronoi` and (b) `compress`.




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



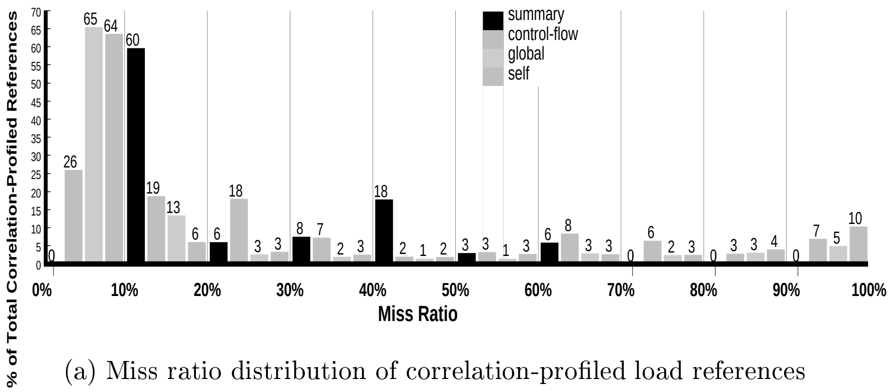

(b) $CPI_{load\_stall}$ due to correlation-profiled loads

Figure 20: Detailed performance results for `compress`.

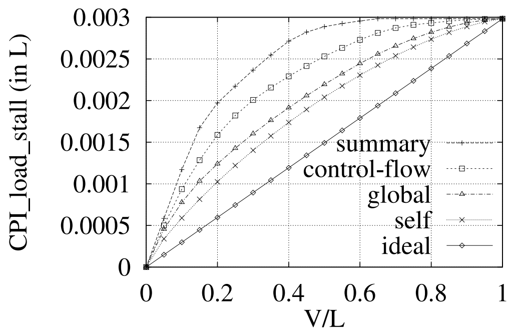
#### 5.2.6 voronoi and compress

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 19(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 20 for its performance results), roughly half of the misses are caused by the hash table access `htabof[i]` in the procedure `compress()` (see Figure 19(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.




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




(b) $CPI_{load\_stall}$ due to correlation-profiled loads

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.

#### 5.2.7 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 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.

### 5.3 Summary

Despite the wide diversity among the applications we studied, we have noticed some common themes in the memory access behaviors across these applications. We conclude this section by summarizing the lessons we have learned from our experiments and case studies.

- Roughly half of the applications enjoyed significant improvements from *both* control-flow and self correlation. This raises the interesting question of whether it is the *same* load references that are benefiting from both types of correlation. As we have discovered during our case studies, the answer appears to yes in many cases—a number of load references can be predicted through more than one form of correlation. At the same time, we have also seen some cases where only one form of correlation can accurately predict cache outcomes. We have also found that although global correlation makes superior predictions in some cases by observing correlations across different load instructions (e.g., `eqntott`), in many cases it essentially assimilates self correlation, but does not perform quite as well because self correlation records a longer sequence of cache outcomes for a given load.

- Self correlation is very effective for applications that experience repeated patterns in their cache outcome results. A common example of such a pattern is a sequence of hits followed by a miss due to the spatial locality between array elements (e.g., in `eqntott`, `espresso`, `vortex`, and `li`), or between accesses of elements of recursive data structures that are somewhat "linearized" in the memory (e.g., in `li`, `perimeter`, `bisort`, and `mst`). Another common predictable pattern is a long run of either all hits or all misses (e.g., in `eqntott`, `mst`, `tsp`, and `raytrace`). The good news is that we typically do not need long self-correlation histories to capture these patterns—in fact, we find that remembering the last four cache outcomes of a given reference is sufficient to achieve good predictability in many of these applications.

- We observe that call site information is very useful in predicting the cache outcomes of loads inside procedures in many applications (`eqntott`, `espresso`, `vortex`, `m88ksim`, `go`, `mst` and `voronoi`). We also find that in tree-based applications (e.g., `perimeter`, `bisort`, and `tsp`), the cache outcomes often depend highly on the level inside the tree at which the references occur. Control-flow correlation accurately predicts both call-site dependent and tree-level dependent cache outcomes by capturing call chain information.

- Array-like references with regular strides still play an important role in a considerable number of these non-numeric applications (e.g., `eqntott`, `li`, `espresso`, `vortex`, `m88ksim` and `go`). However, these array references are often written in more complex forms than in numeric applications, including using pointer arithmetic to compute addresses and passing array indices as procedure parameters. This suggests that some latency-hiding techniques more commonly applied to numeric applications (e.g., prefetching array references [13] and optimizing data locality [4, 19]) may potentially be applicable to non-numeric applications, provided that the compiler can recognize the complex access patterns (which may be possible with the help of interprocedural pointer analysis [8, 18]).

## 6 Exploiting Correlation Profiling in Practice

Now that we have demonstrated correlation profiling's potential for predicting cache outcomes more accurately than summary profiling, the next question is how do we exploit these potential performance gains in practice? The first step is to collect the correlation profiles themselves, which can be done either through simulation (as we have done in this study) or through a runtime profiling tool based on informing memory operations [11]. To minimize the overhead of correlation profiling, we found it useful to first use summary profiling information to focus only on the most significant loads with intermediate miss rates. With so few loads to profile, it was reasonable to maintain relatively long sequences of basic block numbers or previous cache outcomes in the history patterns.

Assuming that we discover that interesting correlations do exist, the next step is to transform the code to statically isolate the "paths" (either control-flow paths or different patterns in previous cache outcomes) which lead to the different cache outcome predictions. One viable approach for doing this is to exploit *code duplication*, which has been used to implement static correlated branch prediction [24]. We could also potentially duplicate sections of code to isolate paths with similar cache outcome behavior, and apply individual latency tolerance actions to each path. Applying code duplication to separate control-flow paths is very similar to what has been done for static correlated branch prediction. Using code duplication to distinguish paths which differ based on previous cache outcomes requires an architectural mechanism that enables software to observe cache outcomes directly, such as informing memory operations.

## 6.1 Exploiting Control-Flow Correlations

While we can reuse the code duplication algorithms developed for correlated static branch prediction [24], we may also potentially suffer from the same kinds of performance overhead due to *code expansion*. Fortunately, our results suggest that only a relatively small amount of code expansion would be required in many applications, since the key distinctions are *call sites* involved in the calling chain. By *cloning* [3] a small number of procedures, we can effectively isolate these paths.

```
foo(tree* root){
   work(root→data);
   foo(root→left);
   foo(root→right);
}
```

(a) Original procedure foo()

clones ⟹

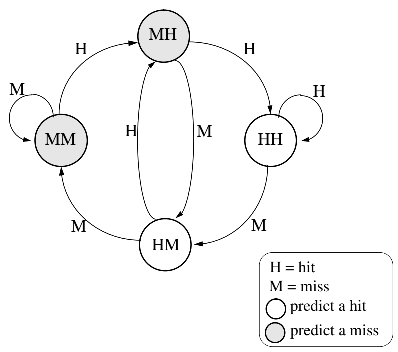
```
foo_0(tree* root){
   work(root→data);
   foo_1(root→left);
   foo_1(root→right);
}

foo_1(tree* root){
   work(root→data);
   foo_2(root→left);
   foo_2(root→right);
}

foo_2(tree* root){
   work(root→data);
   foo_3(root→left);
   foo_3(root→right);
}
   ⋮
foo_n(tree* root){
   work(root→data);
   foo_n(root→left);
   foo_n(root→right);
}
```

(b) n clones of foo()

Figure 23: Example of using procedure cloning to determine the current tree level

```
loop_begin: if (p == 0) goto loop_exit
                    ⋮
            R <- load *p
            R <- R + 10
                    ⋮
            goto loop_begin
loop_exit:
```

(a) Original code




(b) Predicting the next cache outcome based on the last two

```
HH_loop_begin: if (p == 0) goto loop_exit
                    ⋮
               R <- info_load *p
               R <- R + 10      /*just seen HH*/
jump upon a miss    ⋮
               goto HH_loop_begin
               R <- R + 10      /*just seen HM*/
                    ⋮
HM_loop_begin: if (p == 0) goto loop_exit
                    ⋮
               R <- info_load *p
               R <- R + 10      /*just seen MH*/
jump upon a miss    ⋮
               goto MH_loop_begin
               R <- R + 10      /*just seen MM*/
                    ⋮
MM_loop_begin: if (p == 0) goto loop_exit
                    ⋮
               R <- info_load *p
               R <- R + 10      /*just seen MH*/
jump upon a miss    ⋮
               goto MH_loop_begin
               R <- R + 10      /*just seen MM*/
                    ⋮
               goto MM_loop_begin
MH_loop_begin: if (p == 0) goto loop_exit
                    ⋮
               R <- info_load *p
               R <- R + 10      /*just seen HH*/
jump upon a miss    ⋮
               goto HH_loop_begin
               R <- R + 10      /*just seen HM*/
                    ⋮
               goto HM_loop_begin
loop_exit:
```

(c) Duplicated code with informing memory operations

Figure 24: Example of using informing memory operations to implement self correlated prediction. The single loop in (a) is duplicated into the four loops shown in (c), each of them corresponds to a different state in (b).

We can also potentially use procedure cloning to determine the current tree level for predicting tree-level dependent cache outcomes. Figure 23 illustrates how a recursive procedure `foo()` could be cloned $n$ times, where each copy corresponds to a particular depth in the tree. If most misses occur near the root of the tree (as we have observed in several of our cases studies), we could apply a latency tolerance scheme at higher levels in the tree (`foo_0()`, `foo_1()`, etc.), but disable it near the bottom (`foo_n()`). The number of clones that we should make can potentially be determined from the control-flow correlation profiling information.

### 6.2 Exploiting Self Correlations or Global Correlations

A viable technique for exploiting self correlations or global correlations is to use *informing memory operations* [11], which are essentially a memory operation combined with a conditional branch-and-link operation that is taken only if the reference suffers a cache miss. With informing memory operations, we can apply code duplication to separate paths based on previous cache outcomes. Figure 24 shows an example of how this might be accomplished. Assume that self correlation has detected three common cache outcome patterns for the load of `*p` in Figure 24(a): (i) a long sequence of hits, (ii) a long sequence of misses, and (iii) an alternating sequence of hits and misses. Given these patterns, we could use the previous two cache outcomes to predict the next cache outcome, as illustrated by the state diagram in Figure 24(b). Figure 24(c) shows how the original loop could be replicated into four copies, where each copy encodes one of the four states. The original load of `*p` is replaced by an *informing* version of the same load. As a result of each informing load, one of two state transitions occur: (i) if the load hits, we will continue as normal and jump directly to next state associated with a hit; (ii) if the load *misses*, the informing mechanism will directly trigger a branch which we will set to take us to the next state associated with a miss.

Again, code expansion is a concern here. In the worst case, we might expand the code exponentially with respect to the length of the history pattern that is needed to provide the correlation. Fortunately, our experimental results indicate that relatively short history lengths are sufficient in many applications. In addition, we do not expect it to be necessary to enumerate all possible paths of a particular history length, as we do in our example. In many situations, we only need to differentiate paths that occur frequently but have substantially different miss ratios. Enumerating only these paths can reduce the code expansion factor significantly.

## 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 the cache miss rates 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*, which can be translated into the effective stall cycles per load ($ESCPL$) that we use in our study. 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.

The three forms of correlation which we explore in this study (*control-flow*, *self*, and *global*) were inspired by earlier work on using correlation to enhance *branch prediction* accuracies [5, 14, 21, 22, 23, 24]. 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 would like to apply them to the precise set of dynamic references that suffer misses. Unfortunately, compilers are not able to statically predict data cache locality in non-numeric applications, and the state-of-the-art profiling technique (summary profiling) is inadequate for references with intermediate miss ratios. To address this problem, 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 translates 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. Our case studies also indicate that array-like references are still important sources of cache misses in some non-numeric programs, although they are usually written in more complicated forms which may be difficult for the compiler to recognize. Finally, we illustrate that through the use of code duplication—and with the support of informing memory operations to exploit self or global correlation—we can implement software transformations to exploit correlation profiling information. We believe that these promising results may lead to further innovations in optimizing the memory performance of non-numeric applications.

## Appendix: Derivation of the Effective Stalled Cycle Per Load Access ($ESCPL$) under Five Latency-Tolerance Schemes

Denote the $ESCPL$ under a particular tolerance scheme $S$ by $ESCPL_S$. Let $ESCPL_S^i$ be the $ESCPL_S$ 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:

$$ESCPL_S = \sum_i ESCPL_S^i \times f_i \tag{4}$$

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.

$\underline{ESCPL_{none}}$: A load reference is stalled only when it is a cache miss, so:

$$ESCPL_{none} = Lm \tag{5}$$

$\underline{ESCPL_{all}}$: $T$ fully tolerates the latencies of all load references but always incurs the overhead, so:

$$ESCPL_{all} = V \tag{6}$$

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

$$ESCPL^i_{single\_action\_per\_load} = \begin{cases} Lm_i & \text{if } m_i \le \frac{V}{L} \text{ (i.e. not apply T)} \\ V & \text{otherwise (i.e. apply T)} \end{cases} \tag{7}$$

$$\begin{aligned} ESCPL_{single\_action\_per\_load} &= \sum_{i \in A} ESCPL^i_{single\_action\_per\_load} \times f_i + \sum_{i \in NA} ESCPL^i_{single\_action\_per\_load} \times f_i \\ &= V \sum_{i \in A} f_i + L \sum_{i \in NA} m_i f_i \qquad \text{by (7)} \end{aligned} \tag{8}$$

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

$\underline{ESCPL_{multiple\_actions\_per\_load}}$: $T$ is only applied to references of load $i$ that belong to paths with miss ratios $> \frac{V}{L}$. The formula for $ESCPL^i_{multiple\_actions\_per\_load}$ can be simply obtained adding an extra level to Equation (8) to capture the notion of paths within load $i$. That is:

$$ESCPL^i_{multiple\_actions\_per\_load} = V \sum_{j \in A_i} f_{i,j} + L \sum_{j \in NA_i} m_{i,j} f_{i,j} \tag{9}$$

where $A_i$ is the set of paths of load $i$ with miss ratios $> \frac{V}{L}$, $NA_i$ is the set of paths of load $i$ of miss ratios $\le \frac{V}{L}$, $m_{i,j}$ is the miss ratio of path $j$ of load $i$, and $f_{i,j}$ is the fraction of references of load $i$ that are on path $j$. $ESCPL_{multiple\_actions\_per\_load}$ can be obtained by substituting $ESCPL^i_{multiple\_actions\_per\_load}$ into Equation (4).

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

$$ESCPL_{ideal} = Vm \tag{10}$$

# 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 *Proceedings of the 26th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 139–152, December 1993.

[3] D. F. Bacon, S. L. Graham, and O. L. Sharp. Compiler transformations for high-performance computing. *Computing Surveys*, 26(4), 1994.

[4] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, October 1994.

[5] P. Chang, E. Hao, T. Yeh, and Y. Patt. Branch classification: a new mechanism for improving branch predictor performance. In *Proceedings of the 27th Annual IEEE/ACM International Symposium on Microarchitecture*, November 1994.

[6] Standard Performance Evaluation Corporation. *The SPEC92 benchmark suite*. http://www.specbench.org.

[7] Standard Performance Evaluation Corporation. *The SPEC95 benchmark suite*. http://www.specbench.org.

[8] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[9] K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 211–222, April 1994.

[10] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing loads: Enabling software to observe and react to memory behavior. Technical Report CSL-TR-95-673, Stanford University, July 1995.

[11] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing memory operations: Providing memory performance feedback in modern processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 260–270, May 1996.

[12] 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.

[13] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[14] S. Pan, K. So, and J. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, October 1992.

[15] 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.

[16] A. Rogers and K. Li. Software support for speculative loads. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, October 1992.

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

[18] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

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

[20] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–38, June 1995.

[21] T.-Y. Yeh and Y. Patt. Two-level adaptive branch prediction. In *Proceedings of the 24th Annual IEEE/ACM International Symposium on Microarchitecture*, November 1991.

[22] T.-Y. Yeh and Y. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266, May 1993.

[23] C. Young, N. Gloy, and M. Smith. A comparative analysis of schemes for correlated branch prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 276–286, May 1995.

[24] C. Young and M. Smith. Improving the accuracy of static branch prediction using branch correlation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 232–241, October 1994.