# Static Load Classification for Improving the Value Predictability of Data-Cache Misses *

Martin Burtscher
Cornell University
burtscher@csl.cornell.edu

Amer Diwan
University of Colorado
diwan@cs.colorado.edu

Matthias Hauswirth
University of Colorado
hauswirt@cs.colorado.edu

## ABSTRACT

While caches are effective at avoiding most main-memory accesses, the few remaining memory references are still expensive. Even one cache miss per one hundred accesses can double a program's execution time. To better tolerate the data-cache miss latency, architects have proposed various speculation mechanisms, including load-value prediction. A load-value predictor guesses the result of a load so that the dependent instructions can immediately proceed without having to wait for the memory access to complete.

To use the prediction resources most effectively, speculation should be restricted to loads that are likely to miss in the cache *and* that are likely to be predicted correctly. Prior work has considered hardware- and profile-based methods to make these decisions. Our work focuses on making these decisions at compile time. We show that a simple compiler classification is effective at separating the loads that should be speculated from the loads that should not. We present results for a number of C and Java programs and demonstrate that our results are consistent across programming languages and across program inputs.

## Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors–Compilers, Optimization; C.4 [**Computer Systems Organization**]: Performance of Systems

## General Terms

Experimentation, Languages, Measurement, Performance

## Keywords

Load-value prediction, type-based analysis

*This work is supported by NSF ITR grant CCR-0085792, an NSF CAREER award, and an equipment gift from Intel. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.


*PLDI'02,* June 17-19, 2002, Berlin, Germany.


## 1. INTRODUCTION

Caches address the widening gap between processor and memory speeds by satisfying most memory requests quickly. However, because of the high cost of main-memory references, even a few cache misses can significantly degrade program performance. For this reason, prior work has proposed several speculation techniques, including load-value prediction [20], for hiding the latency of cache misses. Load-value predictors guess the result of a load as soon as the load starts executing. Instructions that need the loaded value can proceed using the guessed value without waiting for the memory access to complete (reduced latency) and are thus able to execute in parallel with the load (increased instruction-level parallelism). This paper focuses on the latency-reduction aspect of load-value prediction.

To make the most of speculation hardware for latency reduction, we should ideally restrict its use to the loads that miss in the cache *and* that are predictable. Applying speculation to loads that hit in the cache can destructively interfere with the speculation of loads that miss in the cache. Moreover, if speculation is done in software, speculating loads that hit in the cache unnecessarily increases the code size. Applying speculation to loads that are unlikely to be predicted correctly will incur a misspeculation penalty and slow down program execution.

The recent load-value prediction literature proposes complex predictors that combine multiple basic predictors (*hybrids*) [8, 25, 31] and incorporate confidence estimators to dynamically decide which loads are worth predicting and with which predictor [5, 7, 9, 20, 24]. The confidence estimators try to filter out loads that would be mispredicted since mispredictions lower program performance. While modern load-value predictors are quite effective in simulations, their complexity degrades various performance parameters such as critical path length (i.e., cycle time), energy consumption, heat dissipation, and chip area in a real hardware implementation. The goal of our research is to better understand how various static program characteristics relate to cache performance and value predictability, thus providing a foundation for making speculation decisions in compilers rather than in hardware.

Our approach is to statically partition all load instructions into 20 classes based on factors such as the type of the loaded value (e.g., pointer), the region of memory the load references (e.g., heap), and the kind of the load (e.g., array reference) and to measure the cache behavior and value predictability of these classes. We conduct our experiments using three two-way set-associative cache sizes (16K, 64K,

and 256K) and simulate five load-value predictors described in prior work: (i) LV, which predicts the last value for every load, (ii) L4V, which predicts one of the last four values for every load, (iii) ST2D, which uses strides to predict loads, (iv) FCM, which uses a representation of the context of preceding loads to predict a load, and (v) DFCM, which enhances FCM with strides. We simulate two sizes of load-value predictors: realistic (2048 entries) and infinite.

Our experiments use programs drawn from the SPECint95, SPECint00 and SPECjvm98, benchmark suites and demonstrate the following. First, the same classes make up the majority of cache misses across our benchmark programs. Six classes (representing about half of all references) account for most of the cache misses. Second, the load-value predictability of classes is consistent, i.e., some classes are highly predictable across benchmarks while others are highly unpredictable. We show how to exploit this class behavior to improve the load-value predictability of loads that miss in the cache. Third, we demonstrate that predictors that perform the best in prior work, FCM and DFCM, do not perform the best for cache misses. In other words, FCM and DFCM are most effective on the loads that are unimportant for performance. Finally, we demonstrate that our results are consistent across languages (C and Java) and across program inputs.

The remainder of this paper is organized as follows. Section 2 presents background information on load-value prediction. Section 3 describes our simulation framework and the benchmark programs. Section 4 presents our experimental results and discusses their implications for hardware and compilers. Section 5 reviews related work, and Section 6 concludes the paper.

## 2. BACKGROUND

At the outset, load-value prediction seems like a hopelessly hard problem: after all, a 32-bit word can hold over four billion distinct values and a 64-bit word over $10^{19}$ values. Fortunately, load values tend to cluster, repeat, occur in sequences, exhibit patterns, and correlate with one another. Prior work has proposed different predictors, each tailored to some kind of load-value locality. These predictors differ in the kind of information they retain and in the computations they perform on this information to produce a prediction.

The *last value predictor* LV [14, 20] simply predicts that a load instruction will load the same value that it did the previous time it executed. LV can only predict sequences of repeating values (e.g., 3, 3, 3, 3, ...). Such sequences are surprisingly frequent [14, 20]. All load instructions that load run-time constants such as starting addresses of data structures and floating-point constants fall into this category.

The *stride 2-delta predictor* ST2D [27] remembers the last value for each load (like LV) but also maintains a stride, which is the difference between the last two loaded values. To make a prediction, ST2D adds the stride to the last value of the load. When a load finishes, ST2D updates the last value. ST2D updates the stride only if it encounters the same stride twice in a row. Doing so eliminates the problem of making two consecutive mispredictions at every transition from one predictable sequence to another [5, 27].

Like LV, ST2D can also predict sequences of repeating values; the stride is simply zero in this case. In addition, ST2D can predict sequences that exhibit genuine stride behavior (e.g., -4, -2, 0, 2, 4, ...), i.e., sequences where the stride is a non-zero constant. Such sequences are not very frequent [14, 27] because register allocation assigns most induction variables to registers, but they do occur, for example, when a program uses global variables as counters.

The *last four value predictor* L4V [6, 19, 31] is similar to the last value predictor except that it retains the four most recently loaded values. At each load, L4V selects from its four possibilities the entry (not the value) that made the most recent correct prediction. Like the ST2D and the LV predictors, L4V can also predict repeating values. In addition, L4V can predict alternating values (e.g., -1, 0, -1, 0, -1, ...) or, more generally, any short repeating sequence that spans no more than four values (e.g., 1, 2, 3, 1, 2, 3, 1, ...). Such sequences are more frequent than true stride behavior [6, 19]. In particular, alternating sequences occur relatively often when variables toggle between two values.

The *finite context method predictor* FCM [26, 27] computes a hash value out of the last four values of a load using a select-fold-shift-xor function [24, 25, 26] to index the predictor's second level table. This table stores the values that follow every seen sequence of four values (modulo the table size). Since the table is shared, load instructions can communicate information to one another in this predictor. Hence, after observing a sequence of load values, FCM can predict any load that loads the same sequence.

FCM can predict long sequences of arbitrary reoccurring values (e.g., 3, 7, 4, 9, 2, ..., 3, 7, 4, 9, 2, ...). These sequences occur, for instance, during the repeated traversal of dynamic data structures. Note that this predictor can also predict alternating sequences and sequences exhibiting stride behavior as long as the sequence repeats and its length does not exceed the table size.

The *differential finite context method predictor* DFCM [16] improves on FCM by retaining strides instead of absolute values. This approach reduces the chance of detrimental aliasing in the second-level table, often increases the predictor's capacity, and enables it to predict values it has never before seen. Thus, DFCM combines the strengths of FCM and ST2D at the cost of some additional complexity.

## 3. METHODOLOGY

Our approach is to use compiler and binary instrumentation to generate a detailed trace. For each load, the trace give the *class* of the load. The cache and load-value predictor simulators consume these traces and for each load they determine if the predictor would have predicted the load correctly, update the state, and attribute the prediction or misprediction to the class of the load. The cache simulators determine if the load hits or misses in the cache and also attributes the hit or miss to the class of the load. At the end of the run, we output statistical information for each class, including their cache and load-value predictor behavior.

Sections 3.1 and 3.2 present the classes and our classification technique, respectively. Section 3.3 describes how we use the class information in our simulations. Section 3.4 describes our benchmark programs.

### 3.1 Classes

We distinguish between two kinds of references: *high-level references* that are visible at the source level and *low-level references* that are only visible in the assembly or some other low-level representation of programs. We consider two kinds of low-level references for C programs: loads of return ad-

dresses (RA) and restores of callee-saved registers (CS), and one kind of low-level load for Java programs: memory copies by the run-time system (MC). We consider three dimensions when classifying the high-level references:

- The **region** of memory the reference accesses: is the reference loading from a location in the stack, the heap, or the global space?
- The **kind** of reference: is the reference loading an object field, an array element, or a scalar variable?
- The **type** of the reference: is the reference loading a value of type pointer or non-pointer?

We picked these dimensions based on our intuition and the description of load-value predictors in prior work. More specifically, one of the strengths of FCM and DFCM (given in prior work) is that they can successfully predict repeated traversals of linked data structures. Thus FCM and DFCM will probably be the most successful at predicting pointer-typed loads from object fields. In subsequent work we are studying other classifications also, such as ones based on simple program analyses.

We use three-letter abbreviations for high-level references. The first letter stands for the region of memory (**S**tack, **H**eap, or **G**lobal). The second letter denotes the kind of reference (**A**rray, **F**ield, or **S**calar). The third letter indicates the type of the reference (**P**ointer or **N**on-pointer). For example, an HFP reference loads the value of a pointer-typed field in a heap-allocated object.

## 3.2 Load Classification

Figure 1 shows our data-collection setup for C programs. We first translate the benchmark programs to the SUIF v.1 representation [17]. Next, we add instrumentation to loads that are visible at the SUIF level (*high-level loads*). The instrumentation communicates the type, kind, address, and virtual program counter[1] of each load to the *VP library* (Section 3.3). Then, we compile the instrumented programs on an Alpha/OSF workstation and instrument the low-level loads in the resulting binary using ATOM [29]. Finally, we link the instrumented binary with the *VP library* and run it to collect our data.

There are two sources of imprecision in our methodology for C programs. First, we assume that all references (except references to local scalar variables whose address is not taken) result in loads. This is potentially imprecise since a compiler may be able to eliminate some references to non-local or non-scalar variables [11] or may be unable to assign some local scalar variables to registers. Since this work is primarily concerned with understanding the behavior of variables and not so much with evaluating the performance impact of a load-value predictor or cache, we feel that this simplification is acceptable. Second, our instrumentation of high-level loads may perturb later compiler optimizations. We have made careful choices in our instrumentation to mitigate this problem. For example, since passing parameters consumes registers and thus affects register allocation, we communicate all information between the instrumentation and the *VP library* using a set of scalar global variables.

[1]The program counter values are not available in SUIF so we sequentially number all the loads of the program and use that as the program counter for our simulations.

The organization for Java programs is similar to that for C programs except that we instrument using the Jikes RVM [4] from IBM Research instead of SUIF and ATOM. We use the two-generational copying garbage collector for Java. For our experiments, Java programs differ from C programs in four ways. First, Java programs have only scalar local variables, which are usually allocated in registers. Thus, the classes S__ are empty. Second, unlike in C programs, only objects and arrays are allocated in the heap. Hence, the HSN and HSP classes are empty for Java programs. Third, there are no global arrays and global scalars in Java programs, and thus classes GS_ and GA_ are empty. Fourth, we do not yet have a convenient mechanism for measuring low-level loads except for memory copies and therefore do not report data for classes RA and CS.

## 3.3 The VP Library

For C programs, the *VP library* simulates the caches and load-value predictors and determines the **region** of memory a load touches (i.e., stack, heap, or global space) by examining the address of the load. While we can easily determine an approximation to the **region** of loads in the compiler [10], we opted to use a precise run-time classification in order to avoid polluting our data with artifacts of an imperfect points-to analysis. Our experience indicates that the **region** of most loads stays constant across executions of the load and thus a compile-time analysis should be effective at determining the **region** of loads. For Java programs, the bytecode (such as *aload*) directly tells us the **region** of memory a load accesses.

We simulate caches with a write-no-allocate policy, two-way associativity, 64-bit word size and 32-byte block size. We consider cache sizes of 16K, 64K, and 256K. We picked these sizes because they are representative of L1 data caches for modern processors (e.g., 64K in the Alpha 21264, the Athlon XP, and the UltraSPARC III, and 16K in the Pentium III). We simulate value predictors of two sizes. (i) The 2048-entry predictors have 2048 entries in their tables. FCM and DFCM have 2048 entries in both the first and second-level tables. (ii) The infinite predictors have a sufficiently large size to eliminate any conflicts.

## 3.4 Benchmarks

For C programs, we use programs from the SPECint95 [3] and SPECint00 [1] integer benchmark suites compiled on an Alpha/OSF workstation for our measurements (Table 1). For Java programs, we use programs from the SPECjvm98 [2] suite compiled on a PowerPC running Linux.[2] These programs are well understood, non-synthetic, and compute-intensive.

Table 2 shows the percentages of the measured loads that fall into each of the 20 classes using the "reference" inputs for the SPECint95 programs and the "train" inputs for the SPECint00 programs. If a particular class makes up 2% or more of the total references in a program, we highlight it in **bold**. Note that all but four of the twenty classes make up at least 2% of the total loads in at least one benchmark program. Also, loads in some classes, for example GSN and CS, occur frequently in the majority of the programs. Table 3 presents similar data for the Java benchmarks except that

[2]Unfortunately, since parts of our infrastructure are platform specific, we could not run the C and Java programs on the same architecture.

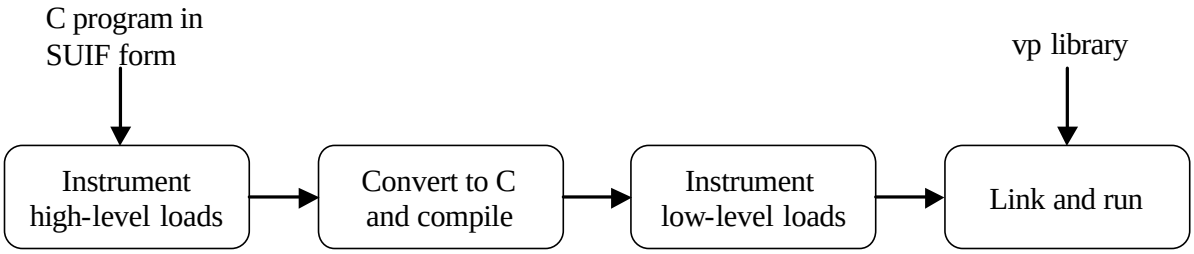


**Figure 1: Experimental setup**

| Program name | Source | Description |
|---|---|---|
| compress | SPECint95 | Compresses and decompresses a file in memory |
| gcc | SPECint95 | C compiler that builds SPARC code |
| go | SPECint95 | Plays the game of "GO" |
| ijpeg | SPECint95 | Compression and decompression of graphics |
| li | SPECint95 | Lisp interpreter |
| m88ksim | SPECint95 | Motorola 88000 chip simulator, runs a test program |
| perl | SPECint95 | Manipulates strings (anagrams) and prime numbers in Perl |
| vortex | SPECint95 | An object oriented database program |
| bzip2 | SPECint00 | Compression of an image |
| gzip | SPECint00 | Compression utility using LZ77 |
| mcf | SPECint00 | Combinatorial optimizations |
| compress | SPECjvm98 | Utility to compress/uncompress large files based on Lempel-Ziv method |
| jess | SPECjvm98 | Java expert system shell based on NASA's CLIPS expert system |
| raytrace | SPECjvm98 | Single-threaded raytracer |
| db | SPECjvm98 | Small data-management program on memory-resident databases |
| javac | SPECjvm98 | The JDK 1.0.2 Java compiler |
| mpegaudio | SPECjvm98 | MPEG-3 audio stream decoder |
| mtrt | SPECjvm98 | Multi-threaded raytracer (calls raytrace) |
| jack | SPECjvm98 | Parser generator with lexical analysis, early version of JavaCC |

**Table 1: Benchmark programs**

some of the classes do not exist (Section 3.2). We use input "size10" for Java benchmarks.

# 4. RESULTS

Section 4.1 presents our results for C programs and Section 4.2 for Java programs. Section 4.3 validates our observations by comparing to results from a different set of program inputs. When presenting results we omit data for benchmark/class combinations if the class comprises less than 2% of the references in the benchmark program.

## 4.1 Results for C Programs

Section *4.1.1* examines the behavior of our classes with respect to data-cache performance. Section *4.1.2* presents results for load-value predictor performance. Section *4.1.3* combines the cache and value-prediction results to explore how value predictors perform on cache misses.

### *4.1.1 Cache Performance*

Since cache misses benefit the most from latency tolerance techniques, we start by examining the cache performance of our classes. Figure 2 shows the average percentage of total cache misses incurred by each class. The "error" bars present the highest and lowest percentage of cache misses for each class. We picked this metric instead of the more traditional cache miss or hit rates because it emphasizes classes that contribute most to cache misses rather than classes that contribute few misses but have high miss rates. Later on we also present cache hit rates.

For each class we only consider those benchmarks in which the class makes up at least 2% of the references. Thus, the sum of the bars of a given cache size often adds up to more than 100%. To see how this happens, imagine that there are two programs and two classes of loads such that each class occurs in only one program and for that program makes up 100% of the cache misses. Then, both classes will have their bars at 100%, and the sum of the two bars will add up to more than 100% (namely 200%). Figure 2 has three bars for each class for cache sizes of 16K, 64K, and 256K. For example, looking at the 16K GAN bar we see that GAN loads account (on average) for 43% of all the cache misses in programs that have a non-trivial number of GAN loads. We also see that the range for GAN reaches from contributing nearly 0% to almost 100% of the cache misses, depending on the program.

The numbers next to the class names along the horizontal axis give the number of programs for which that class makes up at least 2% of the total references. To put these numbers in perspective, Table 4 gives the data-cache miss rates for our benchmark programs.

From Figure 2 we see that the classes have fairly consistent cache behavior across benchmark programs. In particular, the vast majority of cache misses are in six classes: GAN, HSN, HFN, HAN, HFP, and HAP. The other classes (e.g., the low-level classes) contribute little to the number of cache misses. Table 5 gives the percentage of cache misses that come from the six classes mentioned above.

Figure 3 presents the cache hit rates for each class using cache sizes of 16K, 64K, and 256K. The "error" bars present the range of hit rates for a class. The y-axis starts at 40% for improved readability. We present cache hit rates rather than miss rates to make the graph easily comparable to Figure 4. From Figure 3 we see that the classes that account for the

| Benchmark | 16K | 64K | 256K |
|---|---|---|---|
| compress | 8.5 | 6.2 | 3.3 |
| gcc | 3.0 | 1.1 | 0.3 |
| go | 5.0 | 1.1 | 0.0 |
| ijpeg | 1.5 | 0.6 | 0.4 |
| li | 3.1 | 2.5 | 1.4 |
| m88ksim | 0.2 | 0.0 | 0.0 |
| perl | 0.9 | 0.0 | 0.0 |
| vortex | 1.6 | 0.7 | 0.3 |
| bzip | 2.0 | 1.9 | 1.6 |
| gzip | 5.8 | 2.6 | 0.1 |
| mcf | 27.2 | 25.1 | 21.5 |

**Table 4: Load miss rates for data caches**

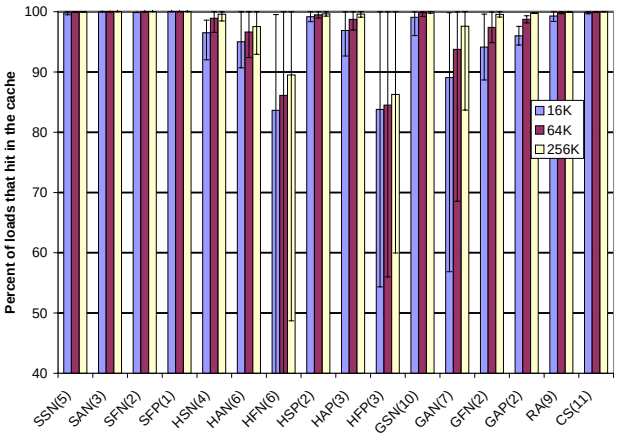


**Figure 3: Cache hit rates for all loads (average over all benchmarks, minimum, and maximum)**

| Benchmark | 16K | 64K | 256K |
|---|---|---|---|
| compress | 98 | 98 | 97 |
| gcc | 78 | 83 | 85 |
| go | 86 | 88 | 94 |
| ijpeg | 95 | 98 | 98 |
| li | 69 | 74 | 77 |
| m88ksim | 41 | 77 | 100 |
| perl | 50 | 96 | 96 |
| vortex | 86 | 96 | 99 |
| bzip | 100 | 100 | 100 |
| gzip | 96 | 96 | 89 |
| mcf | 68 | 68 | 67 |

**Table 5: Percentage of cache misses that come from classes GAN, HSN, HFN, HAN, HFP, and HAP**

| Class | compress | gcc | go | ijpeg | li | m88ksim | perl | vortex | bzip | gzip | mcf | mean |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SSN | 0 | 1.28 | **3.50** | 0.42 | **4.40** | **12.10** | **6.23** | **7.26** | 0.12 | 0.15 | 0.15 | 2.97 |
| SAN | 0 | 0.63 | 1.01 | **16.61** | 0 | 0.45 | **2.58** | 0.00 | **12.73** | 0.01 | 0 | 2.84 |
| SFN | 0 | 0.67 | 0 | **3.62** | 0.00 | 0.30 | 0 | **2.60** | 0 | 0 | 0 | 0.60 |
| SSP | 0 | 0.37 | 0 | 0.17 | 1.40 | 0.00 | 0.00 | 0.33 | 0 | 0.02 | 0 | 0.19 |
| SAP | 0 | 0.25 | 0 | 0.17 | 0 | 0 | 0 | 0 | 0 | 0.00 | 0 | 0.04 |
| SFP | 0 | 0.29 | 0 | 0.25 | 0.01 | 0.24 | **2.15** | 0.05 | 0 | 0 | 0 | 0.25 |
| HSN | 0 | 0.88 | 0 | **14.75** | **3.51** | 0.00 | **8.07** | **7.32** | 0.27 | 0.01 | 0.20 | 2.92 |
| HAN | 0 | **7.39** | 0 | **48.55** | 0.00 | 0.00 | **4.30** | **5.39** | **31.83** | 0.00 | **2.75** | 8.35 |
| HFN | 0 | **16.37** | 0 | 0.76 | **8.80** | **6.11** | **8.42** | 0.85 | 0 | **3.54** | **27.35** | 6.02 |
| HSP | 0 | 0.33 | 0 | 0.00 | 1.82 | 0.00 | **20.01** | **7.64** | 0 | 0 | 0 | 2.48 |
| HAP | 0 | **9.42** | 0 | 1.33 | 0.56 | 0 | **3.02** | **4.97** | 0 | 0 | 0.88 | 1.68 |
| HFP | 0 | 1.82 | 0 | 0.11 | **24.44** | 0.57 | **6.29** | 0.16 | 0 | 0.01 | **17.47** | 4.24 |
| GSN | **43.46** | **11.10** | **14.23** | 0.45 | **12.76** | **17.49** | **16.81** | **27.79** | **43.71** | **43.75** | **3.12** | 19.56 |
| GAN | **19.27** | **6.51** | **52.03** | **3.00** | 0.00 | **21.86** | 0.00 | 0.03 | **3.63** | **26.24** | 0 | 11.05 |
| GFN | 0 | 0.81 | 0 | 0.41 | 0.00 | **10.96** | 0.00 | 0.16 | 0 | 0.00 | **2.79** | 1.26 |
| GSP | 0 | 0.68 | 0 | 0.04 | 0.00 | 0.00 | 0.00 | 0.00 | 0 | 0 | 0.48 | 0.10 |
| GAP | 0 | **2.17** | 0.00 | 0.00 | 0.00 | 0.86 | 0.00 | 0.60 | 0.41 | 0.00 | **4.72** | 0.73 |
| GFP | 0 | 0.77 | 0 | 0.20 | 0.00 | 0.07 | 0.00 | 0.00 | 0 | 0.00 | 0.26 | 0.11 |
| RA | **7.65** | **5.16** | **3.68** | 0.91 | **8.84** | **4.58** | **4.11** | **4.60** | 0.76 | **2.52** | **7.29** | 4.17 |
| CS | **29.62** | **33.10** | **25.55** | **8.27** | **33.46** | **24.40** | **18.01** | **30.24** | **6.54** | **23.75** | **32.55** | 22.12 |

**Table 2: Dynamic distribution of total references in C benchmarks runs (ref inputs for SPECint95, train inputs for SPECint00)**

| Class | compress | jess | raytrace | db | javac | mpegaudio | mtrt | jack | mean |
|---|---|---|---|---|---|---|---|---|---|
| GFN | 0.14 | **3.20** | 0.87 | 1.73 | **14.43** | 0.39 | 0.36 | **3.65** | 3.10 |
| GFP | 1.53 | 0.76 | 0.40 | 0.42 | 1.57 | **2.00** | 0.42 | 0.82 | 0.99 |
| HAN | **14.68** | **2.36** | **3.38** | **15.66** | **11.28** | **32.42** | **4.49** | **2.43** | 10.84 |
| HAP | 0.07 | **18.01** | **13.38** | **9.69** | 1.88 | **11.36** | **11.68** | **11.37** | 9.68 |
| HFN | **49.01** | **57.90** | **54.51** | **48.65** | **48.30** | **47.07** | **54.05** | **65.08** | 53.07 |
| HFP | **34.25** | **17.63** | **27.27** | **23.37** | **15.56** | **6.74** | **28.69** | **15.23** | 21.09 |
| MC | 0.31 | 0.13 | 0.19 | 0.46 | **6.97** | 0.02 | 0.29 | 1.42 | 1.23 |

**Table 3: Dynamic distribution of total references in Java benchmarks runs (size 10 inputs for SPECjvm98)**

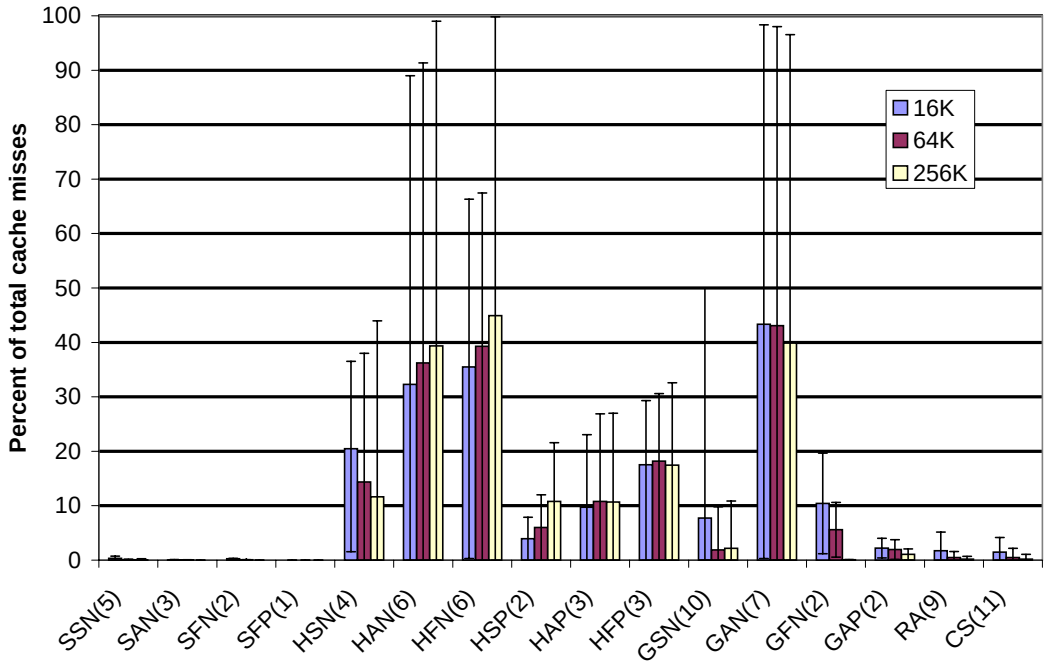


**Figure 2: Contribution to cache misses by class (eleven programs)**

vast majority of the loads (e.g., HFN) have low cache hit rates compared to the other classes.

The above results indicate that we can use readily-available compiler information to focus the mechanism for tolerating load latency on only a few of the classes. The six classes that contribute the most to the cache misses make up between 38% and 73% of the loads executed in the benchmarks (arithmetic mean 55%). Moreover, in a 64K cache, these classes account for 68% to 100% of the cache misses (arithmetic mean 89%). In other words, it suffices to use mechanisms such as load-value prediction on only about half the loads, thus reducing conflicts in the predictor's tables.

From the bars for the three cache sizes in Figure 2, we see that as the cache size increases the *contribution* of a particular class may decrease (e.g., GAN) or increase (e.g., HAN). This happens because increasing the cache size will not necessarily remove the same percentage of misses from all classes.

It is not surprising that the heap classes have poor cache behavior, though to our knowledge, no one has demonstrated this empirically. Class GAN performs poorly in the cache because global arrays often hold hash tables that are accessed throughout the lifetime of the program.

### 4.1.2 Class Predictability

Tables 6 (a) and (b) show which predictors performs best for each class. The *Class* column gives the name of the class. The number in parentheses is the number of programs (out of a total of eleven programs) for which this class makes up at least 2% of the references. The tables omit classes that make up less than 2% of the references in all the benchmarks. The LV, L4V, ST2D, FCM, and DFCM columns give the number of benchmarks for which the predictor is predictability-wise within 5% of the best predictor for the class. An empty entry means that the corresponding predictor does not perform within 5% of the best predictor for any benchmark program. A **bold** entry indicates that the corresponding predictor is one of the most consistent predictors for the class. For example, from Table 6 (a) we see that SSN makes up at least 2% of the references in five benchmark programs and DFCM performs the best (or within 5% of the best) in these programs. Table 6 (a) presents data for 2048-entry predictors and Table 6 (b) for infinite-size predictors.

Table 6 (b) shows that DFCM is the best predictor if the predictor size is unlimited. For realistic predictor sizes, DFCM (and to a lesser extent FCM) significantly outperforms the other predictors particularly for pointer loads as well as non-pointer loads from the stack. Since register allocators eliminate the easily predictable loads from the stack (such as references to induction variables), it is not surprising that the simpler predictors (LV, ST2D, and L4V) perform poorly for non-pointer loads from the stack. Since register allocators are less effective at eliminating loads from the heap and global space, enough simple load-value locality remains for the simpler predictors to perform well.

For classes HAN, GSN, GFN, RA, and CS, the simpler predictors (particularly L4V and ST2D) are comparable or sometimes even better than the more complex predictors (DFCM and FCM). Even when L4V and ST2D perform as well but no better than FCM they may be preferable because they require much simpler and smaller hardware. For class RA, L4V is the most consistent realistic predictor (Table 6(a)). Since RA represents the loads of return PC values, it is not

| Class | | LV | L4V | ST2D | FCM | DFCM |
|---|---|---|---|---|---|---|
| SSN | (5) | 1 | 2 | 2 | 4 | **5** |
| SAN | (3) | 1 | | 1 | 1 | **2** |
| SFN | (2) | | | 1 | **2** | **2** |
| SFP | (1) | | | | | **1** |
| HSN | (4) | 1 | 2 | 1 | 3 | **4** |
| HAN | (6) | 2 | 2 | 4 | 4 | **5** |
| HFN | (6) | 2 | 3 | 2 | 4 | **6** |
| HSP | (2) | 1 | 1 | 1 | **2** | **2** |
| HAP | (3) | | 1 | | **2** | **2** |
| HFP | (3) | | | 1 | 2 | **3** |
| GSN | (10) | 2 | 2 | **8** | 2 | 7 |
| GAN | (7) | 3 | 3 | 4 | **5** | **5** |
| GFN | (2) | **1** | **1** | **1** | **1** | **1** |
| GAP | (2) | | 1 | | **2** | **2** |
| RA | (9) | 5 | **8** | 5 | 4 | 4 |
| CS | (11) | 2 | 3 | 7 | 1 | **9** |

(a) 2048

| Class | | LV | L4V | ST2D | FCM | DFCM |
|---|---|---|---|---|---|---|
| SSN | (5) | 1 | 1 | 1 | **5** | **5** |
| SAN | (3) | | | | 1 | **3** |
| SFN | (2) | | | 1 | 1 | **2** |
| SFP | (1) | | | | **1** | |
| HSN | (4) | | | | 2 | **4** |
| HAN | (6) | 1 | | | 5 | **6** |
| HFN | (6) | | | | 5 | **6** |
| HSP | (2) | 1 | 1 | 1 | **2** | **2** |
| HAP | (3) | | 1 | | 2 | **3** |
| HFP | (3) | | | | **3** | **3** |
| GSN | (10) | 1 | 1 | 4 | 6 | **10** |
| GAN | (7) | 1 | 1 | 1 | **6** | **6** |
| GFN | (2) | 1 | 1 | 1 | **2** | **2** |
| GAP | (2) | | | | **2** | **2** |
| RA | (9) | 2 | 4 | 2 | 8 | **9** |
| CS | (11) | | | 2 | 7 | **11** |

(b) infinite

**Table 6: Best predictor for predictor sizes 2048 and infinite. Inputs: ref for SPECint95 and train for SPECint00 (eleven programs)**

| Class | | Number of benchmarks |
|---|---|---|
| SSN | (5) | 4 |
| SAN | (3) | 1 |
| SFN | (2) | 1 |
| SFP | (1) | 1 |
| HSN | (4) | 2 |
| HAN | (6) | 3 |
| HFN | (6) | 4 |
| HSP | (2) | 2 |
| HAP | (3) | 2 |
| HFP | (3) | 2 |
| GSN | (10) | 9 |
| GAN | (7) | 2 |
| GFN | (2) | 1 |
| GAP | (2) | 0 |
| RA | (9) | 6 |
| CS | (11) | 7 |

**Table 7: Number of benchmarks for which the best 2048-entry predictor for the class predicts more than 60% of the loads (eleven programs)**

surprising that RA loads take on one of the last few values of the load. If a procedure is always called from the same site, then even LV or ST2D will be effective at predicting RA loads (and indeed for some of the programs, ST2D and LV work quite well).

DFCM and ST2D are the best predictor for CS (Table 6 (a)). Note, however, that FCM performs relatively poorly on CS (it is the best predictor for CS for only one benchmark program). CS is an unusual class because members of this class have mixed types. For example, imagine a procedure that uses register `r1` and is called from two call sites. When it is called from the first call site `r1` may contain a pointer and when it is called from the second call site `r1` may contain an integer. The integer may be an induction variable and thus favor ST2D since compilers eagerly allocate induction variables to registers. The pointer, on the other hand, will most likely favor FCM or DFCM. Thus, DFCM performs well on CS because it can predict both strides and repeated traversals through linked data structures.

Figure 4 presents the average percentage of correct predictions for each class using the five load-value predictors. Taller bars in Figure 4 are better. For some classes, such as SSP, there are no bars since these classes make up less than 2% of the total references in all benchmark programs. Table 7 presents for each class the number of benchmarks where the best value predictor for that class can correctly predict at least 60% of the references in that class. Figure 4 and Table 7 together suggest that some classes are more predictable than others. For example, GAN appears in seven benchmark programs, but in only two of them is a predictor able to predict more than 60% of the GAN references. On the other hand, GSN appears in ten benchmark programs and in nine of them a predictor is able to correctly predict more than 60% of the GSN references.

Comparing Figures 4 and 3 we see that classes that suffer from low hit rates in caches (e.g., classes HFN, HFP, and GAN) also often suffer from low predictability in value predictors.

To summarize, we observe that some classes are much more predictable than others. Moreover, for the predictable classes, there is usually a realistic predictor that performs best for most of the benchmarks. Given this consistency of predictor performance, it should be possible to build an effective hybrid predictor that uses static instead of dynamic predictor selection. After all, the classes for which the simpler predictors perform almost as well or even better than FCM and DFCM represent over a quarter of all loads. We further note that classes that perform poorly for caches also perform poorly for load-value prediction.

### 4.1.3 *Improved Latency Tolerance Through Compiler Analysis*

We now consider how to use our results from Section *4.1.1* and *4.1.2* to improve the performance of programs. The full benefit will be greater once we consider more uses of the results, such as for prefetching.

Figure 5 gives the performance of load-value prediction on loads that miss in a 64K cache. A predictor with a taller bar performs better than one with a shorter bar. We use the 2048-entry configurations for all load-value predictors. To speed up simulations, we ignored the low-level loads in these experiments since they rarely miss in the cache. Figure 5 presents results only for the classes that cause the

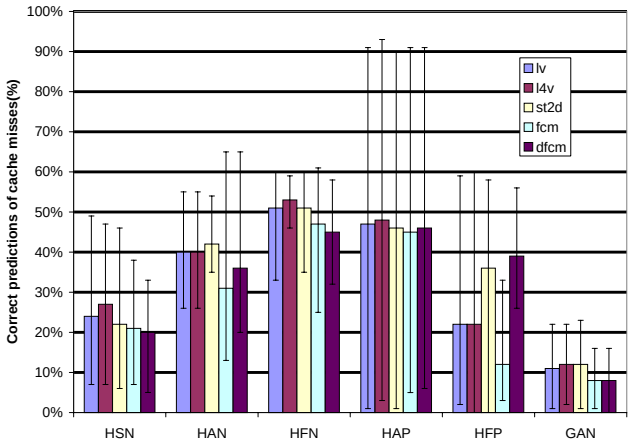


**Figure 5: Prediction rates for loads missing in the cache (average over all benchmarks, minimum, and maximum)**

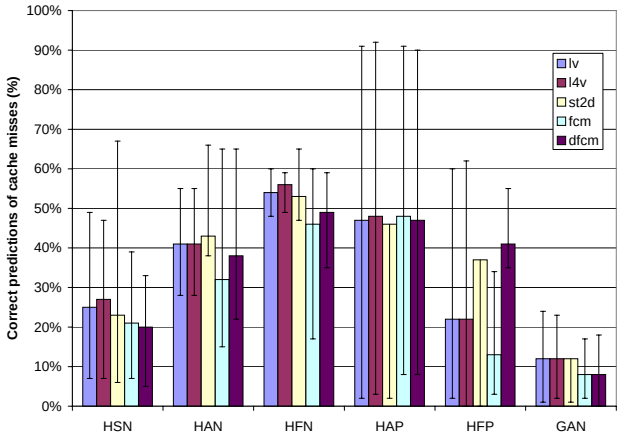


**Figure 6: Prediction rates for loads missing in cache and designated by compiler to be predicted (average over all benchmarks, minimum and maximum)**

majority of the cache misses. The "error" bars give the minimum and maximum correct predictions across our benchmark programs. On inspecting the data we noted that the low and high points of the predictors matched up. For example, when one predictor performed its worst for a class in a benchmark, it was usually the case that the other predictors also performed their worst for that same class and benchmark.

Figure 5 shows that FCM and DFCM perform about the same or slightly worse than the simpler predictors on the loads that miss in the cache. This is surprising because in Section *4.1.2* we saw that DFCM is one of the strongest predictors when we consider all loads. For example, FCM and DFCM perform much better than the other predictors on class HAP (Figure 4), but when we consider only cache misses, the simpler predictors perform slightly better than FCM and DFCM. In other words, FCM and DFCM, despite their relative complexity, are outperformed by the simpler predictors on the loads that matter the most.

One explanation for the relatively poor performance of

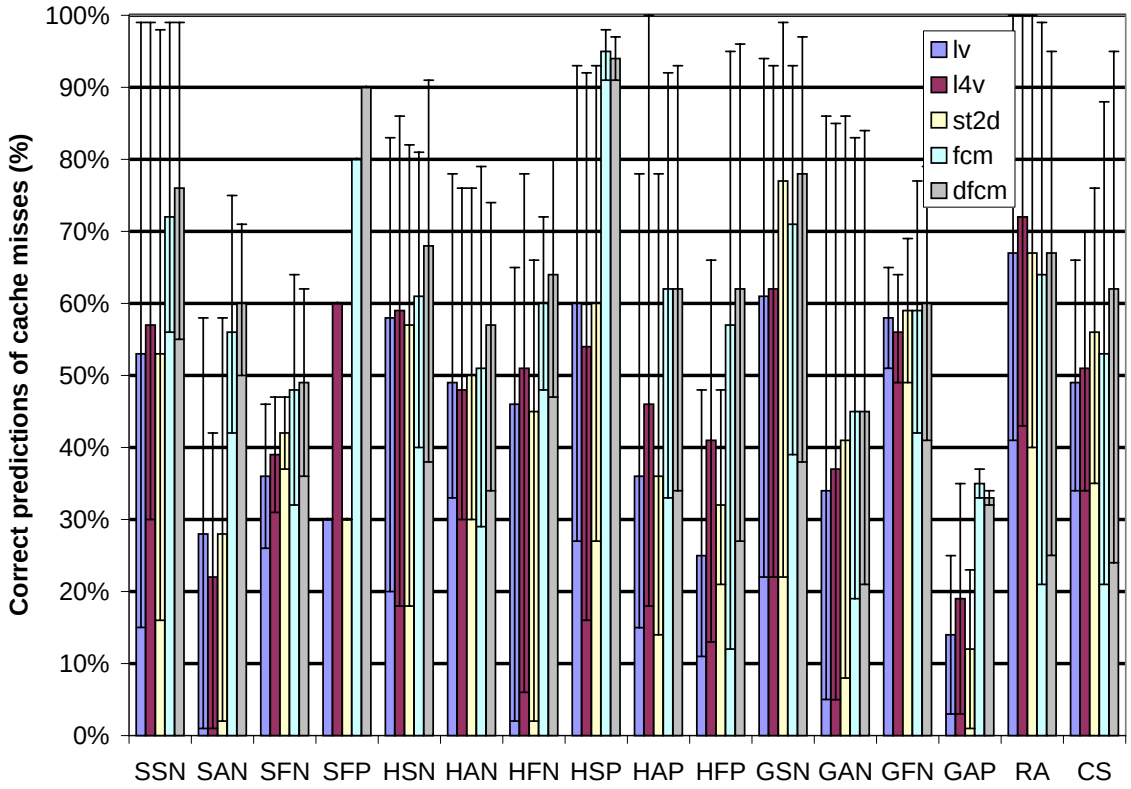


**Figure 4: Prediction rates for all loads (average over all eleven benchmarks, minimum, and maximum)**

FCM and DFCM is that their tables are not large enough. To determine if this is the case, we increased the size of the FCM and DFCM from 2048 entries to practically infinite tables. With infinite tables, DFCM and FCM perform better than the simpler predictors. In other words, if it is possible to build a really large predictor, FCM and DFCM may be preferable to the simpler predictors. Otherwise, the simpler predictors perform just as well (and sometimes even better) than FCM and DFCM. It is also worth keeping in mind that a 2048-entry FCM or DFCM will be larger and more complex than a 2048-entry ST2D.

Figure 6 is similar to Figure 5 except that we use compiler information to predict only loads in the classes that account for most of the misses in the cache. Comparing Figure 6 to Figure 5, we see that there is a modest benefit to filtering loads, i.e., preventing them from accessing the predictor. For example, LV correctly predicts up to 3% more cache misses if only classes HAN, HFN, HAP, HFP, and GAN access the predictor. Reducing predictor accesses eliminates conflicts and thus allows predictors to be more effective on the remaining accesses.

To understand how the cache size affects our results, we repeated the above experiments with a 256K cache instead of a 64K cache. A 256K cache should have fewer misses than a 64K cache and thus use the load-value predictor for fewer loads. Interestingly, we found that the relative performance of the predictors did not change. However, the percentage of correct predictions for the predictors improved by several percent over Figure 6.

The above results use filtering based on which loads are important with respect to the cache. However, another kind of filtering is also possible: filtering out loads that are poorly predicted with value predictors because they provide little potential (and possibly significant harm from the misprediction penalties) for speculation. When we stopped predicting class GAN because it is by far the least predictable of the classes in Figure 6, our results improved: most of the predictors performed better (by up to 7%) than in Figure 6 because there were fewer conflicts in the predictors' tables.

## 4.2 Results for Java programs

In Section 4.1 we demonstrated for C programs that our load classes are useful in separating loads that frequently hit in the cache from loads that frequently miss in the cache. We cannot, however, meaningfully use the same classification for Java programs for two reasons. First, the vast majority of loads in Java programs are from the heap[3] (Table 3) and thus the distinction between heap and non-heap loads is not interesting. Second, our current framework for Java programs misses loads in some low-level classes, which causes the heap loads to dominate the behavior even more. For these reasons, we only report a partial set of results for Java programs.

When we consider the value predictability of all loads, the relative performance of the predictors is similar to that of C programs. DFCM usually has the best predictability and FCM the second-best predictability. However the difference

[3]Our traces do not contain all the low-level loads in Java programs. The balance between heap and non-heap loads may be different in a full memory trace.

between the context-based predictors (DFCM and FCM) and the other predictors (e.g., LV) are not as dramatic as with C programs. The only class for which FCM and DFCM are much better is HAP, where DFCM predicts 80% of the loads correctly, FCM predicts 60% of the loads correctly, and the other predictors are at least 10% worse than FCM.

When we consider the value predictability of loads that miss in the cache, we see a similar behavior to C programs: DFCM and FCM offer little benefit over the simpler predictors and for all classes except HAP and HFP, one of the simpler predictors outperforms both DFCM and FCM.

We also conducted an experiment using a different infrastructure that provides a trace of all loads for Java programs (including loads that are missing in the above results). By instrumenting at the very end of optimizing compilation, even after register allocation, we are able to trace all loads including loads belonging to the classes CS and RA. However, at this late phase of compilation, we do not have enough information to reliably partition loads into classes and thus, we report only overall performance. Our results using these traces are consistent with our other results. In particular, we found that when we consider only cache misses, the simpler predictors are close in performance to DFCM and FCM. More specifically, the simpler predictors perform much better (by at least 10%) than DFCM and FCM for one benchmark (mpegaudio) and slightly better (by less than 5%) for one benchmark (compress). DFCM or FCM perform much better than the simpler predictors for two benchmarks (db and mtrt) and only slightly better for the remaining four benchmarks.

## 4.3 Validation

To ensure wide applicability of our results, we conducted our experiments on a variety of programs written in C and Java. As noted above, our results are consistent across the two programming languages; for example DFCM is the best predictor for pointer loads in both Java and C programs.

We also repeated many of our experiments with C programs using another set of inputs and computed tables similar to Table 6. We found that while the absolute numbers differed, our main conclusions were the same: a predictor that performs well (poorly) with one set of inputs also performs well (poorly) with a different set of inputs.

# 5. RELATED WORK

We first describe related work in load-value prediction and then related work in cache performance.

## 5.1 Load-Value Prediction

Two independent research efforts [14, 20] first recognized that load instructions exhibit *value locality* and concluded that there is potential for prediction.

Lipasti et al. [20] investigated why load values are often predictable and studied the predictability of different kinds of load instructions. They found that while all loads exhibit significant value predictability, address loads have slightly better value locality than data loads, instruction address loads hold an edge over data address loads, and integer data values are more predictable than floating-point data values. Our approach separates loads into many more classes that exhibit greatly varying predictability behavior.

Gabbay and Mendelson [15] explore the possibility of using program profiles to enhance the efficiency of value prediction. They use profiling to insert opcode directives to filter out highly unpredictable values from being allocated in the load-value predictor, which considerably reduces the amount of aliasing. Our approach of filtering loads based on how important and predictable they are achieves the same goal without the need for profiling. Furthermore, Gabbay and Mendelson found that training runs generally correlate with test runs, indicating that a program's input values do not significantly affect the value locality. A more detailed study about predictability by Sazeides and Smith [28] illustrates that most of the locality originates in the program control structure and immediate values, which explains the observed independence of program input. Our methodology can also be used with profiles. Profiling may, however, result in insufficient data to classify loads that are never or hardly ever executed during the profile run. Our static approach does not suffer from this problem.

Rychlik et al. [25] address the problem of useless predictions. They introduce a simple hardware mechanism that inhibits predictions that were never used (because the true load value became available before the predicted value was consumed) from updating the predictor, which results in improved performance due to reduced predictor pollution. This filtering is complementary to the filtering with our compiler-based approach.

Fu et al. [13] propose a mixed hardware- and software-based approach to value speculation that leverages advantages of both hardware schemes for value prediction and compiler schemes for exposing instruction-level parallelism. They propose adding new instructions to explicitly load values from the predictor and to update the predictor. Our approach does not require any new instructions and is geared towards hybrid predictors. While the approach of Fu et al. supports hybrids, they defer the problem of which component to select to the hardware. Static classification, as presented in this paper, can be used to accomplish this in software.

Calder et al. [9] examine selection techniques to minimize predictor capacity conflicts by prohibiting unimportant instructions from using the predictor. At the same time, they classify instructions depending on their latency so that the confidence threshold can be adapted to the potential gain of predicting a given instruction. Hence, operations with small gains are only predicted if the predictor's confidence is very high, whereas operations with potentially large gains are predicted even if the confidence is rather low. Interestingly, they found that loads are responsible for most of the latency in the critical path and hence predicting only loads represents a good filtering criterion. We implicitly use this criterion because we only predict load values. Note that Calder et al. classify loads by latency while we classify by type of memory access.

Morancho et al. [21] propose separating the confidence estimator from the predictor so that only the confidence estimator has to be large enough to handle "all" load instructions, whereas the predictor itself can be designed smaller because it only has to hold the predictable loads. By performing confidence pre-estimation at compile time, our approach allows both the value predictor and the confidence estimator to be designed smaller because they only have to be large enough to hold the predictable loads.

The literature describes several hybrid value predictors. For example, Wang and Franklin propose a hybrid between

a last-distinct-four-value and a stride predictor [31], Rychlik et al. use a finite context method and stride 2-delta hybrid [25], and Burtscher and Zorn propose a three-component hybrid that includes a register file, a stride, and a last three value component [8]. The data in this paper suggests that the best predictor for a load can often be picked at compile time rather than at run time in hardware.

## 5.2 Cache Performance

While there has been significant prior work in understanding the cache behavior of programs, we are not aware of any study that correlates cache behavior to high-level properties such as types. Some prior work tries to understand and improve the cache behavior of heap loads by measuring the cache impact of garbage collection [12, 18, 23, 30, 32, 33].

Mowry and Luk [22] also attempt to improve the effectiveness of latency-tolerance techniques by applying them only to cache misses. They identify instructions that are likely to miss in the cache using correlation profiling, which, for instance, predicts whether a load will hit or miss in the cache based on whether previous loads hit or miss in the cache. Rather than using profiling (particularly on-line profiling), our approach uses static properties to predict whether or not a load will miss. The advantage of Mowry and Luk's approach when compared to ours is that it can adapt dynamically if a load's behavior changes during a run. They present numbers in their paper only for the loads that they use correlation profiling on (the top 15 loads in terms of misses) and thus we cannot directly compare our accuracy to theirs. However, the disadvantage of their approach is that it incurs run-time overhead and requires additional hardware to collect and act on the correlation profiles. Our approach has no run-time overhead besides the cost of incorrect predictions. We believe that the two techniques are complementary and it would be interesting to compare and combine the two.

# 6. CONCLUSIONS

We show that a compiler can divide a program's load instructions into classes that exhibit consistent cache and value-predictability behavior. For example, most (arithmetic mean 89%) of the cache misses stem from six classes that represent an average of 55% of the loads in our benchmark programs. Moreover, our classes are largely consistent with respect to load-value predictability: the best predictor for each class seems to be independent of the program.

Interestingly, we found that load-value predictors behave quite differently on loads that miss in the cache than on loads that hit. In particular, predictors such as the FCM and DFCM, which are believed to be the best predictors in the literature, actually perform well only on loads that hit in the cache. For loads that miss, these more complex predictors are no better than the much simpler ones. In other words, for the loads that need speculation the most, the simpler, smaller, and faster predictors perform as well as the more complex predictors.

We use our results to determine at compile time which loads to speculate. More specifically, we only speculate loads from classes that miss frequently in the cache and that are predictable. This approach improves the predictor accuracy by up to 8% on the loads that cause cache misses, i.e., on the loads that matter the most.

# 7. REFERENCES


[1] SPECcpu2000 benchmarks. http://www.spec.org/osg/cpu2000/CINT2000.

[2] SPECjvm98 benchmarks. http://www.spec.org/osg/jvm98/.

[3] In *SPECcpu95*, 1995.

[4] M. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. C. Sreedhar, and H. Srinivasan. The jalapeno dynamic optimizing compiler for Java. In *ACM Java Grande Conference*, San Francisco, CA, June 1999.

[5] M. Burtscher. *Improving Context-Based Load Value Prediction*. PhD thesis, University of Colorado, Boulder, 2000.

[6] M. Burtscher and B. G. Zorn. Exploring Last $n$ Value Prediction. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 66–76, 1999.

[7] M. Burtscher and B. G. Zorn. Prediction Outcome History-based Confidence Estimation for Load Value Prediction. *Journal of Instruction-Level Parallelism*, 1999.

[8] M. Burtscher and B. G. Zorn. Hybridizing and Coalescing Load Value Predictors. In *International Conference on Computer Design*, pages 81–92, 2000.

[9] B. Calder, G. Reinman, and D. M. Tullsen. Selective value prediction. In *Proceedings of the 26th annual international symposium on computer architecture*, pages 64–74. ACM, 1999.

[10] S. Cho, P. Yew, and G. Lee. A high-bandwidth memory pipeline for wide issue processors. *IEEE Transactions on Computers*, 50(7):709–723, July 2001.

[11] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. In *ACM conference on programming language design and implementation*, pages 106–117, Montréal, Canada, May 1998.

[12] A. Diwan, D. Tarditi, and E. Moss. Memory subsystem performance of programs with intensive heap allocation. *ACM Transactions on Computer Systems*, 1995.

[13] C. Y. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte. Value speculation scheduling for high performance processors. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[14] F. Gabbay. Speculative execution based on value prediction. Technical Report 1080, Department of Electrical Engineering, Technion–Israel Institute of Technology, 1996.

[15] F. Gabbay and A. Mendelson. Can program profiling support value prediction? In *30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[16] B. Goeman, H. V. Dierendonck, and K. DeBosschere. Differential fcm: Increasing value prediction accuracy by improving table usage efficiency. In *HPCA-7*, Jan. 2001.

[17] S. U. S. R. Group. Suif compiler system version 1.x. suif.stanford.edu/suif/suif1/index.html.

[18] P. J. Koopman, Jr., P. Lee, and D. P. Siewiorek. Cache behavior of combinator graph reduction. *Transactions on Programming Languages and Systems*, 14(2):265–277, Apr. 1992.

[19] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 19th IEEE/ACM international symposium on microarchitecture*, pages 226–237, 1996.

[20] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value Locality and Load Value Prediction. In *Proceedings of the second international conference on architectural support for programming languages and operating systems*, pages 138–147, 1996.

[21] E. Morancho, J. M. Llaberia, and A. Olive. Split last-address predictor. In *1998 International Conference on Parallel Architectures and Compilation Techniques*, 1998.

[22] T. C. Mowry and C.-K. Luk. Predicting data cache misses in non-numeric applications through correlation profiling. In *International Symposium on Microarchitecture*.

[23] M. B. Reinhold. *Cache Performance of Garbage-Collected Programming Languages*. PhD thesis, Laboratory for Computer Science, MIT, Sept. 1993.

[24] G. Reinman and B. Calder. Predictive techniques for aggressive load speculation. In *Proceedings of the 31st IEEE/ACM international symposium on microarchitecture*, pages 127–137, 1998.

[25] B. Rychlik, J. Faistl, B. Krug, and J. P. Shen. Efficacy and performance impact of value prediction. In *Proceedings of the international conference on parallel architectures and compilation techniques*, 1998.

[26] Y. Sazeides and J. E. Smith. Implementations of context based value predictors. Technical Report ECE-97-8, University of Wisconsin, Madison, Wisconsin, 1997.

[27] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the thirteenth IEEE/ACM international symposium on microarchitecture*, pages 248–258, 1997.

[28] Y. Sazeides and J. E. Smith. Modeling program predictability. In *25th International Symposium on Computer Architecture*, 1998.

[29] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *ACM conference on programming language design and implementation*, pages 196–205, Orlando, FL, June 1994.

[30] D. Tarditi and A. Diwan. Measuring the cost of memory management. *Lisp and Symbolic Computation*, 1996.

[31] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[32] P. R. Wilson, M. S. Lam, and T. G. Moher. Caching considerations for generational garbage collection. In *1992 ACM Conference on Lisp and Functional Programming*, pages 32–42, San Francisco, California, June 1992.

[33] B. Zorn. The effect of garbage collection on cache performance. Technical Report CU-CS-528-91, University of Colorado at Boulder, May 1991.