Static Identification of Delinquent Loads[*]

Vlad-Mihai Panait
Dept. of Computer Science
Politehnica University of
Bucharest, Romania
vpanait@cs.pub.ro

Amit Sasturkar[†]
Dept. of Computer Science
Stony Brook University
Stony Brook, NY
amits@cs.sunysb.edu

Weng-Fai Wong
Dept. of Computer Science
National University of Singapore
Singapore
wongwf@comp.nus.edu.sg

Abstract

The effective use of processor caches is crucial to the performance of applications. It has been shown that cache misses are not evenly distributed throughout a program. In applications running on RISC-style processors, a small number of delinquent load *instructions are responsible for most of the cache misses. Identification of delinquent loads is the key to the success of many cache optimization and prefetching techniques. In this paper, we propose a method for identifying delinquent loads that can be implemented at compile time. Our experiments over eighteen benchmarks from the SPEC suite shows that our proposed scheme is stable across benchmarks, inputs, and cache structures, identifying an average of 10% of the total number of loads in the benchmarks we tested that account for over 90% of all data cache misses. As far as we know, this is the first time a technique for static delinquent load identification with such a level of precision and coverage has been reported. While comparable techniques can also identify load instructions that cover 90% of all data cache misses, they do so by selecting over 50% of all load instructions in the code, resulting in a high number of false positives. If basic block profiling is used in conjunction with our heuristic, then our results show that it is possible to pin down just 1.3% of the load instructions that account for 82% of all data cache misses.*

1. Introduction

The introduction of caches in processors has been an important step in alleviating the problem of ensuring sufficient supply of data into the processor. However, with ever increasing processor speeds and the use of massive instruction level parallelism within processors, the bottleneck in performance once again turns to the memory hierarchy. This well known problem has attracted much attention from the computer systems research community. Many hardware, software and hybrid schemes to alleviate the problem have been proposed. One of the keys to the success of these schemes is the ability to control the overheads involved. This in turn relies on the precise triggering of the necessary computation. An example is prefetching. Performing a prefetch for every load instruction in the program will be too costly and almost certainly will not yield good results. The key to containing the overhead is the correct identification of the load instructions that are most likely to benefit from the prefetch operation.

Previous studies [1, 4] found that in most applications, data cache misses were not uniformly distributed. In particular, a small number of load instructions are responsible for the majority of data cache misses. These load instructions have come to be known as *delinquent loads*. If these delinquent loads can be successfully identified, many cache optimization schemes can precisely target the bottlenecks, thereby minimizing overheads.

In this paper, we propose a static scheme for identifying delinquent loads. The scheme is based largely on the analysis and classification of the address computation subtree of load instructions that is done after code generation but before execution. Thus no activity of the compiler, in particular code optimization, is affected, and unlike memory profiling, there is no overhead incurred during runtime. Our experiment shows that the method is robust with respect to changes in input and cache configuration as well as compiler optimizations. Tested over eighteen large SPEC benchmarks, the heuristic was able to single out 10% of the load instructions in the benchmarks that were responsible for over 90% of the level one data cache misses. In addition, we show that basic block profiling is an excellent means of identification of delinquent loads. When used in combination, basic block profiling together with our heuristic can identify the 1.3% of load instructions that are respon-

* This work is supported in part by A*STAR Grant 012/106/0046.
† This author contributed to the work while he was still with the Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, India.

sible for over 80% of cache misses.

This paper is organized as follows. Following the introduction, we will survey the related work in this area. In Section 3 we will dwell deeper into the motivation of our approach and some of the terminologies we use in this paper. In Section 5 we will describe the overall framework for our approach as well as the ideas that motivated our heuristic. They are by no means exhaustive and the overall framework allows for the inclusion of other heuristics that we may have missed. In Section 6 we will show how we actually implemented the scheme as well as describe the experimental setup we used to verify our approach. Section 8 discusses the results we obtained in our experiments. This is followed by the conclusion.

2. Related Work

There is a rich literature on data prefetching. All hardware and software prefetching techniques necessitate some form of delinquent load identification. We refer the interested reader to Van der Wiel and Lilja's survey [13] for a comprehensive survey of this field.

There are a number of specific works that motivated us. Mehrotra and Harrison [9] proposed a hardware scheme that attempts to discover address computation patterns and subsequently to prefetch based on these patterns. This motivated an important aspect of our method. However, during runtime, since hardware can only perform limited computations, the patterns they used were relatively simple.

Burtscher, Diwan, and Hauswirth [3] (which we shall call the 'BDH-method' in the rest of the paper) proposed a static classification system for improving value prediction. In the BDH-method, load instructions are classified according to the following criteria:

- The region of memory the reference accesses: whether the area is the stack (S), the heap (H) or the global data space (G).
- The kind of reference: whether the reference loads a scalar (S), an element of an array (A) or a field in a structure (F).
- The type of the reference: whether the reference loads a pointer value (P) or a non-pointer value (N).

Thus in the BDH-method, each load instruction belongs to a class that is denoted by a three letter string, representing the region of memory it accesses, the kind and the type of references. The heuristic that was suggested [3] is that loads belonging to the union of the classes GAN, HSN, HFN, HAN, HFP and HAP account for most of the misses in a program. It should be noted that their study was performed on traces. In our implementation of the scheme in the compiler, at times it is difficult to get accurate information about the area of memory being accessed during runtime by a load instruction. Our results showed that this scheme by itself, however, lack the precision of our proposed scheme.

There are also attempts to model the cache's behavior, such as the cache miss equations [6], that may be used. However, most of these models involve statistical averaging making it difficult to extend them to work on specific load instructions. Abstract interpretation has also been used to predict cache behavior [5]. However, the technique can only identify loads that 'always hit', or 'always miss', and uses some 'simple heuristics' to estimate the upper bounds of the miss rates of instructions that do not fall into these two categories.

The work that comes closest in comparison to ours is that by Ozawa, Kimura and Nishizaki [10] (which we shall call the 'OKN-method' in the rest of the paper). They observed that for the SPEC 92 benchmarks, using three simple heuristics, namely whether the load involves a pointer derefencing, a strided reference or none of the above, allowed them to capture a significant portion of the data cache misses. Our scheme goes much further than their proposal and in general subsumes it. As a result, the experiments show that our method is significantly more precise.

3. Delinquent Loads

A load is said to be *delinquent* if it accounts for a significant amount of the cache misses experienced by a program. We are concern only with loads because as far as we know there is no scheme analogous to prefetching for store instructions. Furthermore, processor stalls due to store instructions are generally well controlled by means of write buffers. In essence, a delinquent load is an important memory and hence performance bottleneck. Identification of delinquent loads therefore allows for mechanisms for alleviating this bottleneck to be precisely targeted thereby reducing overhead. Identification of delinquent loads may be done on-line or off-line. Hardware techniques for prefetching, for example, would use hardware mechanisms such as state machines to predict delinquency. However, specialized hardware is required. Furthermore, in order not to have an impact on the critical path of instruction processing, either a enormous amount of hardware that exploits parallelism is needed, or only very simple predictions can be made.

The most common off-line method for identifying delinquent loads involves *memory profiling*. There are two main ways of doing memory profiling. One can use a full instruction level simulation to execute the program, thereby obtaining the complete details of the execution behavior of the application. Another approach is to instrument the code such that a memory trace is produced even as the application executes natively on the processor [8]. Still it is necessary to run the output memory trace through a cache simulator in order to obtain the cache miss data necessary to identify the delinquent loads. Unfortunately, this process is time and space consuming and therefore is not generally applicable. It is worth noting that the latest generation of the Intel Xeon and Pentium 4 processors support hardware event registers that allow for the recording of data cache misses [7]. These can be used through the appropriate library support [11] to obtain actual cache events.

The approach taken in this paper is to apply post-compilation static analysis together with a heuristic function in order to guess which load instructions are likely to be delinquent. It should be noted that although modifications are needed to the compiler, none of the major activities of the compiler are interfered with. For example, we do not require the compiler to not perform certain optimizations in order for our scheme to work. Although this will give less accurate results than detailed memory profiling, the approach has several key advantages. The first advantage of our scheme over hardware schemes is that it is done at the software level, thus reducing the complexity of the hardware. Secondly, this approach allows for greater flexibility in adapting the delinquent load identification method to different classes of programs. Depending on static information, programs may be divided into classes and different classification, weights and heuristic functions for delinquent load identification may be applied to these different application classes.

Besides the disadvantage of being less accurate with false positives being reported, the introduction of the analysis also affects compilation time. However, the increase is not significant as the analysis is largely local in nature.

Benchmark	$\lvert\Lambda\rvert$	**Ideal** $\lvert\Delta\rvert$ **(π)**	**Profiling** $\lvert\Delta\rvert$ **(π)**	ρ
008.espresso	16354	79 (0.48%)	215 / 16354 (1.31%)	98%
022.li	6326	36 (0.57%)	234 (3.70%)	94%
072.sc	7189	30 (0.42%)	379 (5.27%)	97%
099.go	26985	172 (0.64%)	2598 (9.63%)	79%
101.tomcatv	3972	53 (1.33%)	255 (6.42%)	99%
124.m88ksim	11749	27 (0.23%)	199 (1.69%)	18%
126.gcc	121112	1256 (1.04%)	7555 (6.24%)	85%
129.compress	2542	14 (0.55%)	80 (3.15%)	86%
132.ijpeg	22812	72 (0.32%)	711 (3.12%)	79%
147.vortex	46281	126 (0.27%)	913 (1.97%)	86%
164.gzip	6041	60 (0.99%)	143 (2.37%)	96%
175.vpr	16529	64 (0.39%)	500 (3.02%)	92%
179.art	3517	25 (0.71%)	150 (4.26%)	93%
181.mcf	3642	27 (0.74%)	136 (3.73%)	97%
183.equake	4688	82 (1.75%)	505 (10.77%)	99%
188.ammp	20568	66 (0.32%)	1142 (6.93%)	91%
197.parser	12179	239 (1.96%)	1051 (8.63%)	88%
300.twolf	31075	131 (0.42%)	911 (2.93%)	98%
AVERAGE		**0.73%**	**4.73%**	**87.5%**

Table 1. Use of profiling in identifying delinquent loads

4. What Profiling Brings

Basic block execution profiling is fast becoming a widely accepted method for optimization techniques that require feed back. `prof` and `gprof` are standard tools that perform basic profiling in Unix and Linux at the subroutine level while tools like `pixie` on SGI Irix that perform basic block profiling have been around for quite some time. Not surprisingly, delinquent loads tend to occur in frequently executed basic blocks. Table 1 shows the usefulness of profiling in identifying delinquent loads. The details of the experimental framework will be given later. Here, by means of memory profiling and execution tracing, we consider what would happen if we consider *all* of the load instructions in the basic blocks that culmulatively accounts for 90% of the total compute cycles of a benchmark to be possibly delinquent (the set Δ). We will explain the terminologies used later but essentially by profiling, we can identify (on average) 4.73% of all the (static) load instructions in a benchmark (Λ) that accounts for 87.5% of all (level 1) data cache misses. In the third column of Table 1 we compute the 'ideal' number of load instructions it would take cover the same amount of misses. This is done by sorting the load instructions in descending order of their number of misses and then greedily picking out the required number so as to reach the same ρ. As can be seen, there is certainly room for improvement.

It should be pointed out that throughout this paper, basic block profiling was done within the simulator and the training input for profiling is the same as that of experimental runs. Therefore, there is a high level of fidelity between the profile and the actual run. Furthermore, while basic block profiling can identify the blocks that are entered most frequently, this is not necessary the same as the blocks that accounts for most of the execution (compute, memory and I/O) cycles. We believe this is the reason for the poor coverage for 124.m88ksim. The main contribution of this paper is the proposal of a complimentary heuristic that can either be effective by itself or be combined with basic block profiling or other feedback mechanism such as sampling-based profiling [12]. For the case of basic block profiling, we will show that our proposed heuristic adds value by sharpening the focus and overcome the weakness of a pure profiling approach.

5. Our Proposed Framework

Our scheme is implemented as a post-compilation pass. This loose coupling with the compiler allows for the use of disassemblers in place of the compiler. However, it should be pointed out that the compiler possesses all the necessary structures and information to implement the heuristic. From the assembly code of the program, *address patterns* for each load instruction in the program are created.

Using the structure of the address patterns as well as other features of the instruction (which will be described later), the load instruction is identified as belonging to a number of *classes*. By means of a system of weights assigned to the classes, a heuristic function is computed. If the value of the heuristic function exceeds a constant called the delinquency threshold, the load instruction is classified as 'possibly delinquent'.

The heuristic we developed can be used by itself or in combination with basic block profiling. We will first describe our heuristic, establish its effectiveness and stability, and then show how it can significantly sharpen profiling making static identification of delinquent loads without memory profiling a real possibility.

5.1. Address Patterns

For each load instruction, control flow and data flow analysis is used to compute an expression called the *address pattern*. This computation is performed on the assembly output of the compiler and is therefore applicable to both optimized as well as unoptimized code. The address pattern essentially summarizes the data-flow subgraph corresponding to the computation of the address source operand of the load instruction. Written as a context free grammar an address pattern (*AP*) is:

$$\begin{aligned} AP &\rightarrow AP(AP) \mid AP * AP \mid AP + AP \mid AP - AP \mid \\ & AP << AP \mid AP >> AP \mid \text{const} \mid BR \\ BR &\rightarrow \text{gp} \mid \text{sp} \mid \text{reg}_{\text{param}} \mid \text{reg}_{\text{ret}} \end{aligned}$$

Most of the operators have the standard meanings, i.e. '+' is addition, '-' is subtraction, '*' is multiplication, '<<' and '>>' are the left and right shift operation, respectively. However, in the address patterns, parenthesis represents dereferencing. For example, an address pattern like "45(sp)+30" means that the effective address is the content of the memory location sp+45, added with the constant 30. In terms of precedence, dereferencing has the highest precedence, followed by multiplication, addition, subtraction and shifting, in that order. The intermediate registers used for the computation are eliminated and the address pattern is expressed only in terms of *basic registers* (BR). Note also that multiple address patterns for a single load instruction is possible if there are multiple control paths reaching the load instruction and between each of these paths, the address computation differs. In this case, there will be an address pattern for each of the control paths reaching the load instruction.

5.2. The Decision Criteria

The classes used by the heuristic function are drawn from five criteria. Each of these five criteria captures some particular intuition about the structure or complexity of the load instruction. They are:

- (H1) **Register usage in an address pattern.** The intuition behind this decision criterion is that the way registers are used indicates the region of memory the load is likely to access as well as the complexity of the dereferencing needed. To this end, we counted the number of occurences of the basic registers (defined above) in an address pattern.
- (H2) **The type of operations used in the address computation.** Another decision criteria meant to capture the complexity of the dereferencing operation involved is the type of arithmetic operations used in the address computation. We keep track of the multiplications, shifts and other arithmetic operations that appear in the computation of an address pattern. This criterion is especially useful when dealing with loads operating on contiguous data structures in memory such as arrays.
- (H3) **Maximum level of dereferencing.** In order to deal with pointer intensive benchmarks, we keep track of the number of dereferencing involved in an address pattern.
- (H4) **Recurrence.** A recurrence in the address pattern is indicative of an iterative walk through the memory space.
- (H5) **Execution frequency.** Basic block profiling allows us to classify load instructions as 'rarely executed', 'seldom executed', 'fair amount of execution', and 'in a program hotspot'. The last category is used in the profiling filter described in Section 9. In our heuristic, the third category, which actually accounts for most of the loads, is not used at all, and the first two categories are used only in the negative sense - it is used to eliminate infrequently executed load instructions. This part of our heuristic is therefore not very dependent on the fidelity of the profiling information. As such it is entirely possible to replace profiling with static heuristic approximations [15, 14] in identifying infrequently executed load instructions if it is desired to run the heuristic without basic block profiling.

Each decision criterion gives rise to a set of distinct *classes* that are used by the heuristic. A class is a set of address patterns that possesses a certain property. Each address pattern will belong to at most one class of each decision criterion. As an example, consider criterion H3. Each address pattern of a load instruction will be classified as one of the following classes: 'no dereferencing', 'one level of dereference', ... , 'n-level dereference'. The following summarizes the classes of each of the decision criterion:

- (H1) The classes for this criterion are the enumeration of the possible number of occurences of each of the four basic registers in an address pattern. Eventually, however, some of these classes were combined.
- (H2) Rather than an exhaustive enumeration of all possible combinations of arithmetic operators in the address patterns, using observations made in our experimentations, we came up with just one positive class. In this class, the address patterns involve multiplication or shift operations.
- (H3) There are as many classes as the possible number of levels of dereferencing in the address patterns.
- (H4) There are two classes for this criterion depending on whether the address pattern involves a recurrence or not.
- (H5) There are three classes for this criterion as mentioned above: 'rarely executed', 'seldom executed', 'fairly frequently executed', and 'in a program hotspot'. In our experiments, this meant that each of the load instructions were executed less than 100 times during the program's execution.

Each class is given a *weight*. We will discuss how we obtained the weights in Section 7. A heuristic function based on these weights decides if a particular load instruction is possibly delinquent.

6. Implementation and Experimental Setup

We used the SimpleScalar cache simulator [2] for gathering statistics including the number of cache hits, and misses. SimpleScalar implements a close derivative of the MIPS architecture. The utilities that we have used are the C to MIPS GNU C compiler and the MIPS `objdump` utility that disassembles MIPS executables, both included in the SimpleScalar toolkit.

We shall now describe the implementation of our classification framework. The C benchmark is first compiled to a MIPS executable. The MIPS executable is disassembled using `objdump`. This gives us the assembly code for the benchmark as well as any library functions. In order to compute the address patterns needed for our heuristic, the control and data flow graphs have to be re-constructed. If a load's address computation is dependent on values computed outside the basic block it is in, we perform a data flow analysis to obtain all reaching definitions for the temporaries involved. For each of these definitions we create a distinct address pattern. Thus, a load may have more than one address pattern.

Our experimentation is divided into two phases. In the training phase, we simulated a set of benchmark programs using the SimpleScalar cache simulator. For each class we obtained the full memory profiling data, including the number of times the load executed, number of misses, number of hits, etc. Using these statistics, we calculate the weights for each of the classes. The details of this process will be explained later. In the learning phase, we used a split level one cache structure with a four-way associative data cache having 256 cache sets of 32 bytes cache blocks, implementing a LRU replacement policy. The benchmarks were compiled unoptimized. The runtime characteristics of the benchmarks we used is summarized in Table 2. Fortran benchmarks were first converted to C code by `f2c`.

The second phase is the testing phase. Here, we experimented with the heuristic function on inputs different from the ones used in the training phase, different cache sizes, and a completely new set of benchmarks.

Benchmark	Instr executed	No. of L1 data cache accesses	Total no. of L1 cache misses
008.espresso	7.29×10^8	1.51×10^8	6.08×10^6
022.li	6.44×10^7	3.35×10^7	7.64×10^4
072.sc	3.28×10^8	9.23×10^7	2.91×10^6
099.go	1.22×10^9	3.42×10^8	4.64×10^5
101.tomcatv	2.01×10^{11}	8.06×10^{10}	1.67×10^9
124.m88ksim	4.36×10^8	1.88×10^8	1.23×10^6
126.gcc	1.93×10^9	7.84×10^8	4.81×10^6
129.compress	5.86×10^7	2.14×10^7	4.94×10^5
132.ijpeg	2.76×10^9	9.93×10^8	2.14×10^6
147.vortex	3.63×10^9	1.98×10^9	1.16×10^7
164.gzip	1.31×10^{11}	3.90×10^{10}	1.63×10^9
175.vpr	2.44×10^{11}	1.06×10^{11}	2.08×10^9
179.art	1.82×10^{11}	9.73×10^{10}	6.13×10^9
181.mcf	1.33×10^{11}	6.79×10^{10}	6.94×10^9
183.equake	5.98×10^{11}	2.92×10^{11}	7.27×10^9
188.ammp	1.92×10^{12}	7.84×10^{11}	9.89×10^9
197.parser	6.23×10^{11}	3.34×10^{11}	4.73×10^9
300.twolf	2.76×10^9	1.39×10^9	4.08×10^7

Table 2. Typical runtime characteristics of the SPEC benchmarks we used

7. Heuristic for the Static Identification of Delinquent Loads

7.1. Types of classes

From the decision criteria, a number of classes are formed. These classes may be one of three natures: positive, negative, and neutral. Membership in a positive class is evidence that a particular load is possibly delinquent while membership in a negative class is evidence that it is not. The latter allows us to reduce the set of load instructions that needs to be considered. The nature of the classes is reflected in the weights attached to it - positive classes carry positive weight values, negative classes carry negative weight values, and neutral classes have weights equal to zero. In this subsection, we will decide on the nature of each class while in the next subsection, we will show through an example, how weights are derived.

We begin with some terminologies. Let P be a program and $P(I)$ be the execution of P with input I. Let $E(i)$ be the number of times that instruction i is executed, and $M(i, C)$ be the total number of misses experienced by instruction i when running under cache configuration C. Note that if i is not a memory accessing instruction, then $M(i, C) = 0$. We extend the definition of M to a set such that if S is a set of instruction, then $M(S, C) = \sum_{i \in S} M(i, C)$.

Let F be a class. We define the *miss probability of class F in benchmark j* (running under configuration C), $m_j(F, C)$, to be

$$m_j(F, C) = \frac{M(F, C)}{\sum_{i \in F} E(i)}$$

We also define the *amount of misses accounted for by members of class F in benchmark j*, $n_j(F, C)$, to be

$$n_j(F, C) = \frac{M(F, C)}{M(P(I), C)}$$

$m_j(F, C)$ represents the likelihood of an instruction of class i in benchmark j experiencing a cache miss. A large number for a particular class would mean that members of this class experience a high miss probability in the particular benchmark. High probability alone is not enough as it is entirely possible that a particular load instruction has a high miss probability but is insignificant because it is not executed often enough. The later property is represented by $n_j(F, C)$ which shows the proportion of the total number of misses that members of F account for.

Using $m_j(F, C)$ and $n_j(F, C)$, we define a *strength index*, r, to be the ratio of $m_j(F, C)$ to $n_j(F, C)$.

The following shows how classes are formed:

- A benchmark is *irrelevant with respect to class F* if both $m_j(F, C)$ and $n_j(F, C)$ are below certain thresholds. Otherwise, it is considered *relevant* to class F. A class is a positive class if for all relevant benchmarks, $r \geq \frac{1}{20}$.
- A class is a negative class if for all the eleven benchmarks, $n_j(F, C) < 0.50\%$.
- A class is a neutral class if for at least one benchmark, $r < \frac{1}{20}$.

7.2. Computing the Weights - An Example

In this subsection, using the H1 decision criteria as an example, we will show how weights for the classes are derived.

According to decision criterion H1, fifteen classes were formed. They are listed in Table 3. Recall that in H1, we are interested in the occurences of ‘basic registers’ in the address patterns. The second column shows the characteristics of each class. For example, class 5 corresponds to those loads in which at least one of their address patterns involves using the stack pointer (sp) and the global pointer (gp) exactly once each. The third column of Table 3 shows, for each class, the number of benchmarks (out of the eleven) in which load instructions with address patterns possessing the class’ feature were found. The fourth column is the number of benchmarks in which the class is relevant. Although, as described in Section 4.3, we mentioned that we counted the occurences of basic registers, it turns out that address patterns featuring basic registers other than the stack and global pointer showed a low level of relevance. We therefore merged all these other classes into class 15.

Class	Feature	Found in	Relevant in
1	gp=1	11 benchmarks	1 benchmark
2	gp=2	1 benchmark	0 benchmark
3	gp=3	1 benchmark	0 benchmark
4	sp=1	11 benchmarks	5 benchmarks
5	sp=1, gp=1	7 benchmarks	5 benchmarks
6	sp=1, gp=2	1 benchmark	1 benchmark
7	sp=2	11 benchmarks	10 benchmarks
8	sp=2, gp=1	4 benchmarks	4 benchmarks
9	sp=3	4 benchmarks	1 benchmark
10	sp=3, gp=1	2 benchmarks	2 benchmarks
11	sp=4	2 benchmarks	1 benchmark
12	sp=4, gp=3	1 benchmark	1 benchmark
13	sp=5	1 benchmark	0 benchmark
14	sp=6, gp=3	1 benchmark	1 benchmark
15	any others	11 benchmarks	1 benchmark

Table 3. Criteria H1 applied to the eleven training benchmarks

For each of the classes, we compute $m_j(F, C)$ and $n_j(F, C)$. Table 4 shows the $m_j(F, C)$ and $n_j(F, C)$ values for class 5. This class appears in seven of the eleven benchmarks and it is relevant in five, namely 147.vortex, 175.vpr, 179.art, 183.equake and 197.parser. Let us denote this set by R_{F_5}. On benchmarks 099.go and 164.gzip, the class is irrelevant. An inspection of the values of $m_j(F, C)$ and $n_j(F, C)$ for the relevant benchmarks shows that class 5 is a positive class.

The weight of class F_k, W_{F_k}, is computed as follows:

$$W(F_k) = \frac{1}{|R_{F_k}|} \sum_{j \in R_{F_k}} \frac{m_j(F_k, C)}{n_j(F_k, C)}$$

where $| \cdot |$ is the cardinality of the set. As an example, the weight of F_5 is calculated as

$$W(F_5) = \frac{4/48 + 6/25 + 30/67 + 6/6 + 8/13}{5} = \frac{2.37}{5} = 0.47$$

Benchmark	$m_j(F_5, C)$ (%)	$n_j(F_5, C)$ (%)
099.go	0.16	0.13
147.vortex	4.34	48.19
164.gzip	0.28	0.03
175.vpr	6.27	25.14
179.art	30.44	67.17
183.equake	6.83	6.72
197.parser	8.07	13.17

Table 4. $m_j(F_5, C)$ and $n_j(F_5, C)$ values of class 5 'sp=1, gp=1' of the criteria H1 on seven benchmarks

For negative classes, the above formula is not applicable because the benchmarks are almost always irrelevant. The method for assigning the weights to the negative classes used will be described in the next section.

7.3. The Heuristic

Table 5 shows the weights of all the positive and negative classes selected to constitute the heuristic. We simplified the classes further by merging some of the compatible classes that have very similar weights. The resulting *aggregate classes* are the final ones used in the classification scheme we implemented:

- (AG_1) The set of address patterns in which both the stack pointer and the global pointer are used at least once each. This is a class from the H1 criteria.
- (AG_2) The set of address patterns in which only the stack pointer is used and is used two times or more. This is a class from the H1 criteria.
- (AG_3) The set of address patterns in which either multiplication or shift operations are present. Note that we have dropped the consideration for addition of small constants. This is a class from the H2 criteria.
- (AG_4) One level dereferencing is found in the address pattern. This is a class from the H3 criteria.
- (AG_5) Two level dereferencing is found in the address pattern.
- (AG_6) Three level dereferencing is found in the address pattern.
- (AG_7) A recurrence is found in the address pattern. This is a class from the H4 criteria.
- (AG_8) The loads in this class have a low frequency of execution. For our experiments, this is defined as a load instruction that is executed 100 to 1000 times. This is a class from the H5 criteria.
- (AG_9) The loads in this class are rarely executed. These are loads that are executed less than 100 times.

As mentioned before, the way of computing weights for positive classes is not applicable to negative classes because the benchmarks are almost always irrelevant. Instead, we chose a value that is close to the approximate mean of all the positive weights, except the highest and the lowest weights, and negated it to arrive at the weight for AG_9. This value is halved for AG_8.

Aggregate Classes	Feature	Weight
AG_1	sp, gp	+0.28
AG_2	sp more than 2 times	+0.33
AG_3	multiplication shifts	+0.47
AG_4	dereferenced once	+0.16
AG_5	dereferenced twice	+0.67
AG_6	dereferenced thrice	+1.72
AG_7	recurrent	+0.10
AG_8	seldom executed	-0.20
AG_9	rarely executed	-0.40

Table 5. Aggregate classes and their weights used in the heuristic function

Let i be a load instruction and A_i be its set of address patterns. We define the heuristic function $\phi(i)$ as follows:

$$\phi(i) = \max_{j \in A_i} \sum_{k=\mathrm{AG}_1}^{\mathrm{AG}_9} W(k) \times d(j,k)$$

where

$$d(j,k) = \begin{cases} 1 & \text{if } j \in k \\ 0 & \text{otherwise} \end{cases}$$

We define a *delinquency threshold*, δ, such that if a load instruction i has $\phi(i) > \delta$, then we consider i to be 'possibly delinquent'. Unless otherwise specified, a δ value of 0.10 is used in this paper.

8. Results

In this section, we shall describe the results of our experiments. In order to facilitate the description, we shall use the following terminologies. Let P be a program executable. Let Λ be the set of all load instructions in P. Let $\Delta \subseteq \Lambda$ be the subset of Λ which a heuristic scheme, H say, outputs as the set of possibly delinquent loads. We define

$$\pi(H) = \frac{|\Delta|}{|\Lambda|}$$

to be the *precision measure* of heuristic scheme H. Intuitively, the lower the $\pi(H)$ value, the more precise or focus is H. This measure is important since we are aiming for static identification of delinquent loads. Let $M(P(I), C)$ be the total number of data cache misses incurred when program P is executed with input I on a particular processor-memory configuration, C. Let $M_\Delta(P(I), C)$ be the total number of data cache misses caused by members of Δ when program P is executed with input I on a particular processor-memory configuration, C. We define

$$\rho(H) = \frac{M_\Delta(P(I), C)}{M(P(I), C)}$$

to be the *coverage* of heuristic scheme H. We will present both π and ρ as percentages.

8.1. Observations about the Criteria

In deriving the aggregate classes and the weights described in Table 5, we were able to make a number of observations which we shall elaborate here.

The aggregate classes AG_1 and AG_2 of criterion H1 give us a fairly good indication of the data spaces involved in the load operation. The experiments show that operations that use the both global data region and the stack make positive classes. This is consistent with the assumptions of the BDH-method. In unoptimized code, temporary variables are held on the stack. More occurences of the stack pointer in the address pattern is indicative of complex addressing. For example, consider `A[i][j]` where `A`, `i` and `j` are all on the stack. Then several access to the stack is needed to compute the address of the corresponding load instruction. This is evident of a complex addressing attempted and thus increases the likelihood of a cache miss. While compiler optimization will allocate registers for many of these temporary stack variables, still if stack variables are used in the address computation, it is likely that the addressing is complex.

The type of operations needed to compute the effective address is another indication of the complexity of the addressing. Aggregate class AG_3 of decision criterion H2 captures this intuition. As an example, consider the array access `A[45]`. In MIPS assembly code, we will see a fetching of the base address of the array followed by the addition of the offset (45). Initially, we experimented with checking for the addition of constants as well as multiplications and shift operations. It turned out that because modern caches fetch entire blocks at a time, we could not come up with a constant that was stable across different cache configurations of different block sizes, even though it worked fairly well for our training configuration. Hence, we reverted to just checking for the presence of multiplications and shift operations in the address patterns. These operations are likely to yield an address that is fairly different from previous accesses and hence the increased likelihood of cache misses.

A simple intuition lies behind aggregate classes AG_4 to AG_7 of criterion H3. Namely, the levels of dereferencing involved in accessing a piece of data, the more likely one is not to find it in the cache although lower levels of dereferencing occur more frequently. For example, a simple read from an array, `A[k]` say, which involves a single dereference is more likely to hit the cache than a complex dereferencing like `A[k]->field`, which involves two levels of dereferencing. The higher the level of dereferencing, the higher is the likelihood that the data structure involved is large, complex, and runtime allocated, making contiguity and thus locality unlikely.

The aggregate class AG_7, derived from criterion H4, is linked to loops. For example, if in a loop we access `A[k]`, where `k` is a value that is updated at each iteration, the address to be fetched from will change with each iteration of the loop. We classify the pattern as recurrent. The experiments show a direct relation between recurrent patterns and delinquency. This is consistent with the findings of Mehrotra and Harrison [9].

Aggregate classes AG_8 and AG_9 from criterion H5 are straightforward - if a load instruction is rarely or seldom executed, then it can never delinquent.

Benchmark	Input 1	Input 2
008.espresso	`bca.in`	`cps.in`
099.go	`50 9 2stone9.in`	`60 20 9stone21.in`
129.compress	`test.in`	`bigtest.in`
147.vortex	`input1_lendian`	`input3_lendian`
164.gzip	`input.source 60`	`input.log 60`
175.vpr	`input_ref`	`input_train`
179.art	`input_ref1`	`input_ref2`
181.mcf	`input_ref`	`input_test`
183.equake	`input_ref`	`input_test`
188.ammp	`input_ref`	`input_test`
197.parser	`input_ref`	`input_test`

Table 6. The inputs used in the experiments

Benchmark	Input 1 π / ρ	Input 2 π / ρ
008.espresso	11% / 85%	11% / 86%
099.go	19% / 86%	25% / 99%
129.compress	4% / 99%	4% / 99%
147.vortex	6% / 89%	7% / 91%
164.gzip	6% / 96%	6% / 91%
175.vpr	11% / 99%	11% / 99%
179.art	7% / 99%	8% / 99%
181.mcf	9% / 99%	9% / 99%
183.equake	18% / 99%	18% / 99%
188.ammp	7% / 99%	7% / 99%
197.parser	15% / 99%	13% / 99%
AVERAGE	**10% / 95%**	**11% / 96%**

Table 7. Performance on different inputs

8.2. Results on different sets of inputs

Table 6 shows the inputs used for testing our heuristic. We used 'Input 1' to train the heuristic. The benchmark were compiled unoptimized. Then using the same cache configuration, we tested the heuristic on the same eleven benchmarks, also compiled unoptimized, but with a different set of inputs. We refer the reader to the SPEC benchmark documentation for a detailed comparison of the two input sets.

The result as shown in Table 7 indicates that for the eleven benchmarks, the heuristic is insensitive to inputs. This is not too surprising as both standard profiling and memory profiling have been found to be fairly stable with respect to inputs [1].

Benchmark	π	Assoc 2	Assoc 4	Assoc 8
		ρ	ρ	ρ
008.espresso	14%	86%	90%	84%
099.go	43%	94%	95%	92%
129.compress	5%	99%	99%	99%
147.vortex	5%	82%	80%	72%
164.gzip	6%	92%	94%	91%
175.vpr	19%	98%	99%	99%
179.art	8%	99%	99%	99%
181.mcf	10%	88%	88%	88%
183.equake	16%	99%	99%	99%
188.ammp	13%	94%	93%	92%
197.parser	20%	80%	80%	82%
AVERAGE	**14%**	**91%**	**92%**	**90%**

Table 8. Performance of heuristic on different associativites of the cache

8.3. Varying associativity, size, and turning on compiler optimization

Next we investigated the stability of the heuristic function with respect to different cache configurations. Using the same set of inputs, we varied the associativity of the cache. For this set of experiments, we also turned on compiler code optimization using the '`-O`' option of the GNU C compiler. As the input is the same, the value of π is the same in for all the runs reported in Table 8. Table 9 shows the results of the same optimized code running tested on 8Kbyte, 16KByte, 32KByte and 64KByte caches.

The result shows that our method is stable with respect to associativities and sizes that are typical in modern processor caches. However, for 099.go, optimization turned out to have a negative impact. After code optimization, many of the loads exhibited the features that we consider positive evidence of delinquency. Nonetheless, this seems to be an isolated case. In general, our heuristic is insensitive to compiler optimizations. Compilers generally first generate unoptimized code and then optimize on it. This insensitivity makes it possible to use our heuristic to guide later code optimization phases.

Benchmark	π	8k	16k	32k	64k
		ρ	ρ	ρ	ρ
008.espresso	14%	92%	90%	87%	87%
099.go	43%	94%	95%	94%	94%
129.compress	5%	99%	99%	99%	99%
147.vortex	5%	82%	80%	73%	73%
164.gzip	6%	92%	94%	91%	91%
175.vpr	19%	99%	99%	99%	99%
179.art	8%	99%	99%	99%	99%
181.mcf	10%	88%	88%	88%	88%
183.equake	16%	99%	99%	99%	99%
188.ammp	13%	94%	94%	92%	92%
197.parser	20%	80%	80%	82%	82%
AVERAGE	**14%**	**92%**	**92%**	**91%**	**91%**

Table 9. Performance on different cache sizes

8.4. Performance on new benchmarks

An important litmus test for the heuristic function is its performance on benchmarks that were not used in the training and weight computation. We tested the heuristic against a set of seven new benchmarks, namely 022.li, 072.sc, 101.tomcatv, 124.m88ksim, 126.gcc, 132.ijpeg, and 300.twolf, all from the SPEC suite of benchmarks. Table 10 shows that the heuristic function achieves an average π value of 9.06% but a slightly lower average ρ value of 88%. This shows that the heuristic works in general.

Benchmark	$\lvert\Delta\rvert$ / $\lvert\Lambda\rvert$ (π)	ρ
022.li	309 / 6326 (4.88%)	96%
072.sc	389 / 7189 (5.41%)	83%
101.tomcatv	240 / 3972 (6.04%)	99%
124.m88ksim	594 / 11749 (5.06%)	84%
126.gcc	17340 / 121112 (14.32%)	85%
132.ijpeg	2243 / 22812 (9.83%)	72%
300.twolf	5553 / 31075 (17.87%)	99%
AVERAGE	**9.06%**	**88.29%**

Table 10. Performance of the heuristic function on a new set of benchmarks

8.5. Comparision with related works

To better gauge the performance of our method, we compared it with two schemes previously reported in the literature. Table 11 summarizes the π and ρ values for our heuristic running under the baseline 8KByte data cache configuration. The benchmarks are unoptimized. Table 11 also shows

Benchmark	With AG_8 and AG_9			Without AG_8 and AG_9	
	π	ρ	ξ	π	ρ
008.espresso	11.19%	85%	37%	30.17%	87%
022.li	4.88%	96%	16%	15.44%	97%
072.sc	5.41%	83%	8%	13.74%	83%
099.go	19.36%	86%	26%	30.17%	83%
101.tomcatv	6.04%	99%	4%	13.27%	99%
124.m88ksim	5.06%	84%	12%	21.84%	85%
126.gcc	14.32%	85%	15%	33.03%	85%
129.compress	4.88%	99%	8%	12.82%	99%
132.ijpeg	9.83%	72%	14%	28.38%	72%
147.vortex	6.80%	89%	25%	22.86%	89%
164.gzip	6.22%	96%	1%	13.79%	96%
175.vpr	11.62%	99%	10%	14.97%	99%
179.art	7.88%	99%	7%	13.51%	99%
181.mcf	9.58%	99%	8%	18.40%	99%
183.equake	18.73%	99%	32%	23.42%	99%
188.ammp	7.18%	99%	13%	21.11%	99%
197.parser	15.88%	99%	8%	21.50%	99%
300.twolf	17.87%	99%	9%	26.30%	99%
AVERAGE	**10.15%**	**92.61%**	**14.04%**	**20.82%**	**92.89%**

Table 11. Performance summary of our heuristic method

Benchmark	OKN Method		BDH Method	
	π	ρ	π	ρ
008.espresso	53.90%	85%	42.98%	99%
022.li	39.77%	98%	42.33%	96%
072.sc	53.12%	72%	71.11%	88%
099.go	50.15%	89%	39.10%	84%
101.tomcatv	61.68%	99%	75.86%	99%
124.m88ksim	53.86%	86%	50.72%	93%
126.gcc	60.00%	86%	45.68%	86%
129.compress	57.99%	91%	70.89%	99%
132.ijpeg	47.06%	72%	33.91%	73%
147.vortex	65.44%	90%	35.52%	99%
164.gzip	54.16%	96%	58.78%	97%
175.vpr	53.89%	99%	38.97%	99%
179.art	58.97%	99%	65.37%	99%
181.mcf	54.61%	99%	55.79%	69%
183.equake	62.63%	99%	60.60%	98%
188.ammp	62.91%	99%	32.80%	99%
197.parser	45.12%	99%	38.04%	99%
300.twolf	70.51%	99%	54.61%	98%
AVERAGE	**55.88%**	**92.06%**	**50.73%**	**93.00%**

Table 12. Performance of the OKN and BDH methods

the performance of our heuristic when AG_8 and AG_9 is removed leaving behind AG_1 to AG_7 which can be implemented without any information of runtime control profile. The fourth column of Table 11 gives a measure of dynamic impact of false positives. The measure ξ is the percentage of dynamic load instructions that was mis-labeled as delinquent by our heuristic. We used a strict definition of 'false positive': a load instruction is a false positive if it is in Δ of the heuristic but not in the ideal Δ set (second column of Table 1). This gives an indication of the damage done if say prefetching is applied to loads wrongly identified as delinquent.

Table 12 reports the values of π and ρ using the heuristics proposed by Ozawa, Kimura and Nishizaki [10] (the 'OKN-method') and the BDH-method [3]. The same binary and cache configurations were used in all the tests.

In their paper, Ozawa, Kimura and Nishizaki reported a π value between 30% and 60% [10]. For our simulations, the average value of π is 56%. Even though we used a different simulator and tool chain, this is in general agreement with their observations. This however is significantly higher than the π value of 11% achieved by our method. The ρ value of the OKN-method is almost the same as ours. However, for certain benchmarks, our method performs significantly better. For example, for 129.compress, by just pointing out 124 load instructions, we were able to account for 99% of all data cache misses. On the other hand, the OKN-method singled out more than 1474, or ten times more, load instructions but could only account for 91% of the misses.

We have also implemented a static version of the BDH-method. In order to decide if a load will access global data or stack data, the registers used in the load instruction are examined. If the base register is \$s8[30], which is the MIPS stack pointer, the data is loaded from stack and if the base register is \$gp[29], the MIPS global pointer, the data is loaded from the global data area. In order to identify heap accessing loads, we performed value propagation to determine if a particular load instruction uses pointers initialized by dynamic memory allocation routines like `malloc()` or `calloc()`.

Next we have to identify the structure of the load, i.e. whether the load accesses a scalar value, an array element or the field of a structure. We also have to identify the type of the load, i.e. whether it is a pointer dereference or not. In order to do these two identifications, type analysis of the MIPS assembly code is done with the help of the symbol table. Each entry for a function in the symbol table contains a list of variables, their types and their offset into the program stack when this function is called. Based on these offset values, we are able to establish the type of the data being accessed by a load instruction. Structures have two offsets. The first is the offset to the base of the structure in the stack and the other is the offset to the field from the base address of the structure. Using these offsets, the types of individual fields in structures can also be found. The type information for a variable also informs us whether the variable is a pointer or a non-pointer. In addition, if a value loaded from memory is used as part of the address in a subsequent load, the first load is assumed to be a pointer reference. Thus by doing a type analysis with the help of the symbol table we can perform the BDH classification statically.

Note that in original paper [3] this classification was done over the execution trace. The simulator was instrumented to produce type information for loads. Thus no type analysis or symbol table usage was needed. We used the same classes as reported by the paper. The main difference is that we do the classification statically. However, the area of memory accessed by a load instruction, a key component of the BDH-method, is not always discernable by a compiler. Examples of when complications will arise include pointer parameters and multiple level dereferencing.

As suggested by the authors, we used the GAN, HSN, HFN, HAN, HFP and HAP classes. Our results, as shown in Table 12, from simulating the BDH-method are very close to that reported in the original paper. The reason why the six classes cover so much of the total number of misses is that generally they correspond to the nature of the benchmarks. For example, if for most of the time, a benchmark works with structures allocated in the heap, generally most of the misses will be generated by the HFP. Although the coverage (ρ) is very good, the problem comes from the large proportion of load instructions identified as possibly delinquent, i.e. the π value, which is 50% of the total number of loads.

In summary, compared to both methods, ours is more precise and has similar if not better coverage.

8.6. Varying the delinquency threshold

The heuristic can also be fine-tuned by varying the weights of the classes. However, it is not an easy process. An easier approach is to vary the delinquency threshold. With a delinquency threshold, δ, set at 0.10, we were able to achieve a π of 10% and a ρ value of 92%. Table 13 shows the effect of varying the value of δ on some of the benchmarks. The simulations used a 16KByte cache configuration using optimized code. With a higher δ, both π and ρ are reduced. However, if we examine the effect for individual benchmarks, then we see that the impact of increasing δ varies significantly. For example, for 179.art, 183.equake and 188.ammp, reducing δ results in lower π with no significant impact on ρ. However, for 164.gzip and 197.parser, the reduction in ρ is very significant. For 164.gzip, ρ is reduced from 94% when $\delta = 0.10$ to 34% when δ is set to 0.40. This points to the possibility of using a different δ value for different benchmarks. Further investigation is warranted.

Benchmark	δ = **0.10**	δ = **0.20**	δ = **0.30**	δ = **0.40**
	π / ρ	π / ρ	π / ρ	π / ρ
008.espresso	14 / 90	12 / 90	10 / 58	6 / 55
099.go	43 / 95	38 / 95	32 / 88	16 / 51
129.compress	5 / 99	5 / 97	2 / 97	1 / 58
147.vortex	5 / 80	4 / 79	3 / 79	2 / 79
164.gzip	6 / 94	6 / 93	4 / 72	3 / 34
175.vpr	19 / 99	18 / 89	15 / 84	12 / 84
179.art	8 / 99	4 / 99	4 / 99	2 / 99
181.mcf	10 / 88	8 / 74	6 / 58	3 / 58
183.equake	16 / 99	15 / 99	13 / 99	10 / 99
188.ammp	13 / 94	9 / 90	7 / 90	6 / 90
197.parser	20 / 80	18 / 79	8 / 41	8 / 41
AVERAGE	**14 / 92**	**12 / 89**	**9 / 78**	**6 / 68**

Table 13. Varying the delinquency threshold. All values are in percentages

Benchmark	ϵ = **0**	ϵ = **0.10**	ϵ = **0.20**	ϵ = **0.30**
	π / ρ / ρ^*	π / ρ	π / ρ	π / ρ
008.espresso	0.96 / 84 / 66	1.99 / 84	3.01 / 84	4.03 / 85
022.li	0.93 / 93 / 15	1.33 / 93	1.72 / 94	2.12 / 95
072.sc	0.77 / 81 / 7	1.24 / 81	1.70 / 81	2.17 / 81
099.go	3.17 / 72 / 46	4.79 / 74	6.41 / 75	8.03 / 75
101.tomcatv	1.51 / 99 / 23	1.96 / 99	2.42 / 99	2.87 / 99
124.m88ksim	0.48 / 7 / 5	0.94 / 7	1.40 / 72	1.86 / 72
126.gcc	2.02 / 74 / 26	3.25 / 75	4.48 / 77	5.71 / 79
129.compress	0.59 / 86 / 5	1.02 / 86	1.46 / 86	1.89 / 89
132.ijpeg	0.74 / 54 / 17	1.65 / 65	2.56 / 66	3.47 / 69
147.vortex	0.58 / 78 / 23	1.22 / 80	1.82 / 80	2.44 / 83
164.gzip	0.40 / 93 / 18	0.99 / 94	1.57 / 94	2.15 / 94
175.vpr	0.59 / 92 / 16	1.70 / 96	2.80 / 96	3.90 / 98
179.art	0.85 / 93 / 24	1.56 / 94	2.27 / 98	2.99 / 96
181.mcf	1.04 / 97 / 27	1.92 / 98	2.77 / 98	3.62 / 98
183.equake	4.67 / 99 / 42	6.08 / 99	7.49 / 99	8.90 / 99
188.ammp	1.22 / 91 / 13	1.81 / 92	2.41 / 89	3.00 / 92
197.parser	2.07 / 88 / 22	3.46 / 88	4.84 / 89	6.22 / 89
300.twolf	0.72 / 98 / 25	2.44 / 98	4.15 / 98	5.87 / 98
AVERAGE	**1.30 / 82 / 23**	**2.19 / 84**	**3.07 / 88**	**3.95 / 88**

Table 14. Varying the ϵ factor. All values are in percentages

9. Combining with Profiling

In Section 4, assuming a high degree of fidelity, we have shown that basic block profiling alone yields very good results. In the sections following it, we introduced our heuristic, and showed it stability and merit over previously proposed heuristics. Certainly, the question is whether one can do even better by combining it with profiling. Even at 4.7%, the actual number of loads that may be delinquent can be in the thousands. For our benchmarks, this represented an average of 955 load instructions.

Let Δ_P and Δ_H be the set of possibly delinquent loads identified by profiling and our heuristic, respectively. Let $\Delta_d = \Delta_H - (\Delta_P \cap \Delta_H)$ be the remaining part of Δ_H that is not found the intersection of Δ_P and Δ_H. We sort the loads in Δ_d in descending heuristic score. Now, we define an ϵ-factor and a Δ_ϵ which is a subset of Δ_d consists of the highest scoring loads in Δ_d such that $|\Delta_\epsilon| = \epsilon \times |\Delta_d|$. The combined heuristic will then report $(\Delta_P \cap \Delta_H) \cup \Delta_\epsilon$ as the set of possibly delinquent loads. When $\epsilon = 0$, this is just $(\Delta_P \cap \Delta_H)$.

The basic idea here is to use our heuristic to both sharpen the set returned by profiling as well as add a small fraction of loads that are not in the hotspots. Table 14 shows the efficacy of the combined heuristics under different ϵ-factors.

The results show that it is possible to pinpoint 1.3% (an average of 262 load instructions for our eighteen benchmarks) of loads that account for 82% of all cache misses. In the column for $\epsilon = 0$, besides π and ρ, there is a third set of values, ρ^*. This is the coverage obtained by *randomly* labelling the same number of load instructions in the hotspots as possibly delinquent. The value reported is the average of three random sampling runs. The disappointing coverage is strong evidence of the value our heuristic brings.

As pointed out in Section 4, the high fidelity of the basic block profiling we used is generally not reproducible in practice. Furthermore, it suffers from the drawback that counting basic block entries alone does not necessarily give the exact hotspots where the bulk of the cache stall cycles occur. It is beyond the scope of this paper to consider the fidelity of profiling, but given these weakness, our heuristic which takes a completely different perspective, compliments profiling, resulting in a doubling of precision without sacrificing coverage to bring it closer to the optimum.

10. Conclusion

In this paper we proposed a scheme for identifying delinquent loads during compile time. It is motivated by the observation made in previous studies that data cache misses in an application is not uniformly distributed. As far as we know, this is the first time a static method has been proposed whereby it is possible to identify 10% of the load instructions in a program responsible for 90% of level one data cache misses. In contrast, while two comparable schemes, one based on three simple heuristics, and another that is not easy to implement at compile time, also achieved coverages exceeding 90%, they were far less precise, classifying 56% and 51%, respectively, of all load instructions as delinquent. We implemented our heuristic as a post-compilation pass of a production C compiler, namely the GNU C compiler, and evaluated its effectiveness on the SimpleScalar instruction-level simulator. Through experimentation using eighteen SPEC benchmarks, we have found that the method to be stable across different inputs, benchmarks, compiler optimization, and cache structures. In essence, we have shown that delinquent loads are few and far between and yet they exhibit structure and patterns that can be detected at compile time. If high fidelity basic block profiling information is available, it is possible to use our heuristic to sharpen the identification process such that it is possible to pinpoint the 1.3% of the tens of thousands of load instructions in a benchmark that accounts for 82% of data cache misses. We believe the ability to precisely identify memory bottlenecks at compile time will allow many hardware and software techniques to target them with greater precision and hence lower overhead. Also, our heuristic may be used in Worse Case Execution Time (WCET) analysis [5].

References

[1] Santosh Abraham and Bob R. Rau. Predicting load latencies using cache profiling. Technical Report HPL-94-110, Hewlett Packard Laboratory, 1994.

[2] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. http://www.cs.wisc.edu/ mscalar/simplescalar.html.

[3] Martin Burtscher, Amer Diwan, and Matthias Hauswirth. Static load classification for improving the value predictability of data cache misses. In *Proceedings of ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 222–233, June 2002.

[4] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher J. Hughes, Yong fong Lee, Dan Lavery, and John P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of 28th Annual International Symposium on Computer Architecture*, pages 14–25, July 2001.

[5] Christian Ferdinard, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2-3):163–189, 1999.

[6] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: an analytical representation of cache misses. In *Proceedings of the 11th international conference on Supercomputing*, pages 317–324, July 1997.

[7] Intel Inc. *IA-32 Intel Architecture software developer's manual (order nu mber 245472).* http://www.intel.com/design/pentium4/manuals/245472.htm.

[8] Teresa Johnson. Automatic annotation of instructions with profiling information, 1995.

[9] Sharad Mehrotra and Luddy Harrison. Examination of a memory access classification scheme for pointer-intensive and numeric programs. In *Proceedings of the 10th International Conference on Supercomputing*, pages 133–140, May 1996.

[10] Toshihiro Ozawa, Yasunori Kimura, and Shin'ichiro Nishizaki. Cache miss heuristics and preloading techniques for general-purpose programs. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 243 – 248, 1995.

[11] PAPI: Performance application programmer interface. http://icl.cs.utk.edu/projects/papi.

[12] S. Subramanya Sastry, Rastislav Bodik, and James E. Smith. Rapid profiling via stratified sampling. In *Proceedings of 28th Annual International Symposium on Computer Architecture*, pages 278–289, July 2001.

[13] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Computing Survey*, 32(2):174–199, June 2000.

[14] Weng-Fai Wong. Source level static branch prediction. *The Computer Journal*, 42(2):142–149, 1999.

[15] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 1–11, 1994.