Speculative Load Motion using Natural Language Hints

Alejandro Carbonara **Wenlu Hu** **Jiyuan Zhang**

1 Introduction

Compilers rely on instruction scheduling algorithms in order to decide how to allocate their processing cycles. Load instructions tend to be a challenge for such algorithms, as there can be a bottleneck when there are many load instructions for a limited number of load units. We aim to mitigate these stalls by moving the required loads earlier in our code. Any latency in stalls will be disguised by the additional time weve allocated to them. In the ideal scenario this will not take additional time because we are merely using time where our load unit would be otherwise unoccupied.

Merely moving these loads within the same basic block should not make a difference under a good instruction scheduling algorithm. Therefore, we aim to move our loads back to an earlier basic block. However, there is some inherent risk in this kind of code motion. First of all, there is additional register pressure. Therefore, we will not want our new position to be too far before our old positions. Second, moving a load instruction to another basic block may not necessarily win, as it may end up being executed unnecessarily if the load branch is not taken. However, if we can predict the probability that each branch is executed, we can determine where it is beneficial to speculatively move the load instructions earlier. Inspired by previous work [7] where variable/-function names are used to predict programmers coding decisions, we explore using variable names for branch prediction in this work.

1.1 Our Approach

In this project, we speculatively move load instructions with the help of branch prediction. Our system consists of two parts - a load-instruction motion program and a branch prediction classifier. The load-instruction motion codeflow is shown in the Figure 1a . We move load instructions in LLVM bit-code that satisfy certain requirements if the load branch is likely to be taken. To predict whether a branch is likely to be taken, it extracts the related variable/function names and passes them to the classifier for prediction. The classifier training part is shown in the Figure 1b. We instrument programs and run them to collect variable/function names and branch-taken ground truth. We then train a simple decision tree classifier on this data.

1.2 Our Contributions

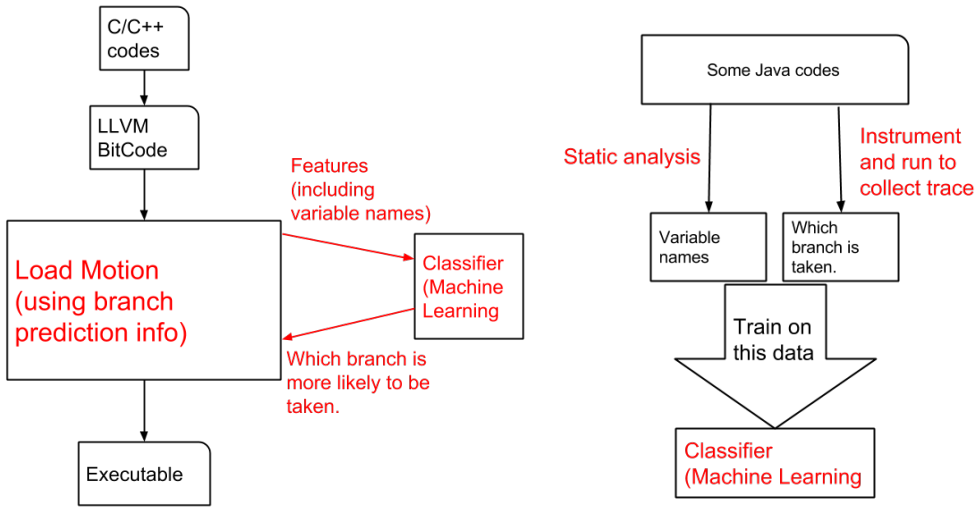
Our contributions include:

- We designed and implemented a load motion algorithm that improves code performance in experiments.
- We explored the possibility of basing branch prediction on natural language hints - function/variable names. This is novel to our best knowledge. We designed and implemented this idea into our load motion algorithm.
- We instrumented a body of Java code and run them to collect training data for branch prediction. We trained a decision tree classifier on this data and included it in all the experiments.
- We have evaluated the system we implemented and shown that it improves code performance by up to 9%.
- We have done a thorough study with experiments on what factors impact the performance of load movement.

1.3 Related Work

Previous work has shown the benefits of moving the computation of loads back. Srinivasans work [11] proved that instruction scheduling loads earlier can result in hiding latency for cache misses, avoiding a similar bottleneck. This has lead to other applications of scheduling loads as to safeguard against cache misses, such as in Winkles work [13] in applying the idea to software pipelining.

The informativeness of natural language hints is explored in Fus [7] work. They use the hints in Microsofts C# programs to predict whether a runtime exception should be logged. Their work uses these hints for software engineering problems instead of compiler optimization. By simply splitting variable names into natural words and training a simple decision-tree, they get a significant improvement in prediction accuracy. While their work inspired ours, our training part is much more difficult than that in their work. (Note that Wenlu is a coauthor of their paper.) First, our work requires modifying training program (our data source, the program that we instrument and collect training data from), compiling and running them, while they only need static analysis. Second, we need change our implementation for each open-source training program we found, as they are all different, e.g. directory organizations. They do not need to do this as they have access to a large amount of uniform commercial programs.



(a) Load-instruction Motion

(b) Classifier Training

Figure 1: System Workflow

2 Our Design

We instrument programs we get to help us collect necessary data. To be specific, we added logging instructions to each if statements in the programs as to document which then/else branch has been taken and to store the condition (e.g. `file != null`) leading to this branch. We use two different approaches to do this - one using Soot [12] Java optimization framework and the other using a Shell script. The first approach has a better coverage and is more extensible, while the second deals with programs with complicated compilation process.

With Soot we first find all if statements in a pass. In Soot IR, if statements take the form of `if (...)goto (...)` we add `System.out.println("!then!"+ condition)` right after the if statement, and `System.out.println("!else!"+ condition){` after the target of the goto. But if the target of the goto is also a goto statement, we will skip instrumenting this if as the added logging instruction will never be executed. This approach operates at Soot IR level and covers a variety of cases in source codes. But it involves changing compilation options to preserve the variable names in source code while converting to IR. Running codes instrumented by this approach also involves modifying the original java-to-jar compilation-linking process, as instrumenting only outputs .class files instead of .java files. So we use this approach for codes that compiles with javac. For codes that compiles with Apache Maven [2] or Apache Ant [1], the time cost to learn the compilation projects is too big for this course project, so we choose the second approach.

With Shell script, we use sed command to look for patterns `if (...)` . We insert a logging instruction `System.out.println("!checking!"+ condition)` before this line, and a logging instruction `System.out.println("!then!"+ condition)` after this line. In the log, if a !checking! is seen without a !then! following it, we know that the else branch has been taken. This approach covers fewer cases than the first one, such as multi-line condition of if. The advantage of this approach is that it changes source codes instead of IR, so the original compilation process works out of the box even after code changes. We used this approach to instrument benchmarks in DaCapo Benchmark Suite [5] which compiles with Apache Ant [1]. We also tried this approach on other projects as described in the Surprise Section.

```
!then! path != null
!else! file.canOpen() == null
```

Figure 2: Example Logs

2.1 Trace Collection

After instrumenting the programs, we recompile and run them, while collecting the logs they produce. Figure 2 shows some examples of the logs. The first log means the condition `path != null` is evaluated and the then branch is taken. The second log means the condition `file.canOpen()== null` is taken and the else branch is taken.

We then break the variable names in the logs into words. Non-alphabetical characters indicate word boundaries and capital letters indicate the beginning of new words. The variable/function names are mostly well formatted as file.canOpen(). Most words we get are indeed natural language words. We use the logs as our training data our training data. Our logs contain about 2687 + 159661 data points and a vocabulary of 22 words.

2.2 Training

We use the bag-of-words model to convert these words into features, that is, only the occurrences of the words are considered. The order in which they appear are ignored. On this data, we then train a decision tree classifier for which branch is more likely to be taken. We use the C4.5 [10] decision trees in Weka [8] machine learning framework. As our training data is limited 22-word vocabulary, we limit the decision tree to be of size 3, i.e. the classifier only looks at three words before making a decision. The small decision tree that learned on our limited training data chooses the word can as one of the three words it would look at. It decides if this word shows up in the variable names, the then clause is more likely to be taken. The classifier has a 99.9% accuracy on training data. Although more experiments on cross-validation and diverse testing data are needed to prove the accuracy of this approach, we believe this number still shows that this approach is promising given that we limit the size of the decision tree to only 3.

2.3 Speculative Load Motion

We implement our code motion in LLVM [9]. To simplify the problem, we limit the code motion to only one basic block away. In order to move a load from Basic Block BB1 to an earlier Basic Block BB2, we require all of the following conditions to hold:

Basic Block BB2 is a predecessor of Basic Block BB1 , and dominates BB1. We get the dominance information in advance from a previous pass. The code is recycled from Homework 3.

From Basic Block BB2, the branch BB1 is more likely to be taken than other branches. We use machine learning to predict which branch is more likely to be taken. We extract the variable names in the condition of if statement, and query the classifier we trained from the last step. The condition of if statements in C code is usually broken into several instructions in LLVM. So we start from the branch instruction, and trace back recursively. Some original variable names are changed when converted to LLVM, but function names are not. So we use the name of the closest function call instead.

While calculating the address operand of the load, no side effects are involved. For example, if the address is calculated by a function call, we should not move this load as this function may introduce side effects. We start from the load instruction and trace backwards to put all instructions in the same basic block that the load instruction depends on into a stack. In the process, we check if any of these instructions has a side effect.

If all these requirements are satisfied, we move a load instruction to the earliest position in Basic Block BB2 that data dependency permits. Along with the load instruction, we also move the instructions that it depends on. These instructions are put into the stack when we look for side effects. For data dependency issues, we move these instructions in the order of their appearance in the source code, i.e. popping them out of the stack.

2.4 Advanced Branch prediction

Additionally, we created an advanced version of our load motion algorithm that could take into consideration more precise branch prediction information. If we are able to know the exact probability that each branch is taken, we can then use that to better determine the expected benefit from moving a load back. In cases where we might move our load back several blocks, we would need to consider the probability that our load may have be needed coming many different paths.

For this, we developed a dataflow computation that would predict the probability that a load was expected at any given point of our code. Our dataflow differed from the ones in class in several ways.

Lattice: It was more complex, associated a float representing the probability of being expected to each instruction. Due to the nature of floats, our lattice had no top element and the height was potentially infinite. Getting an exact solution would be potentially impossible, so we simply ran our pass a number of times in order to get an approximate solution.

Meet function: When we met up along two paths, the new value for an instruction would be the weighted average of the value of the instruction on each path. Our meet function depended highly upon the structure of the lattice.

Transfer function: If we have a load within a basic block, we set the probability that the load was expected as 1. Our transfer function was monotonic, so we could guarantee that there would be no oscillatory effects.

We found that this dataflow algorithm was impractical due to the difficulty of getting exact values for our branch prediction. In the end, our dataflow algorithm was implemented but left unused.

3 Experimental Setup

We instrumented two benchmarks from the DaCapo Benchmark Suite [5] using source-code-level script, and two example Java programs, Airline Example and Simple Word Count, from [3] using Soot [12] Java optimization framework. Speculative Load Motion

We use -O0 option to disable all automatic optimization for all experiments as to show the improvement we make. We run the m2r memory-to-register pass to all test programs, both original test program and our optimized version. Simulator

The performance gain of load scheduled program is hard to measure directly on real system. Therefore, we resort to a simulator to get more accurate results. The simulator we use is Sniper [6], which can simulate x86 architecture using trace-driven methods. We also use the Pin tool to help us collect the execution trace for analysis.

3.1 Micro-benchmark

The performance of load scheduling is hard to measure on normal JAVA program. Therefore, we wrote our own microbenchmark by adopting some JAVA program features. The microbenchmark is written in C with Java naming style, where a load instruction is in a branch that is almost always executed. The pseudo-code for this program is shown in Figure 3. We use this as our test data for all further experiments except otherwise noted.

```
int loop(someList) {
   for i = 0 to 90000
      fileIndex = some computation of i
      if canOpen(fileIndex)
         someElement = someList[some map of
             i] //LOAD
         printf(someElement)
}
int main() {
   int weightList[10000];
   loop(weightList);
}
```

Figure 3: Pseudocode of the Test Program

	Manually	With LLVM
Original	237725846 (5228815)	201952681 (14958511)
Optimized	219967129 (8424704)	184153968 (6105917)
Improvement	7.5%	8.8%

Figure 4: Whole System Evaluation

4 Experimental Evaluation

We compare the cycles taken to run the original test program and the optimized version where load instructions are moved. We experimented with many different parameters of the experiment and tuned our system accordingly. Figure 4 shows the performance improvement where our system works the best. Results are average of 3 runs for all experiments, and the numbers in the brackets are standard deviation. We compare our system with a baseline where load instructions are moved manually in the C code. We show that moving a load instruction with our LLVM implementation works as well as moving the load instruction manually. Our LLVM implementation gives 8.8% performance boost, where the manual approach give 7.5%. This difference is within error bar, given that the variance of some measurement is high.

Different factors that impacts our performance The load scheduling performance is affected by multiple factors. In this part, we evaluate the performance of our load scheduling optimization under different configurations.

4.1 Cache size

The cache size has an impact on the miss rate of data access, inducing longer load latency. This can influence how much benefits the achieve by moving load instructions ahead. Figure 5 shows the performance of our system under different cache size settings.

As the L1 cache size increases, we see less performance improvements by moving load instructions ahead. The reason is with smaller cache size, the program encounters more L1 cache miss. The load instruction has larger average latency. Therefore, it benefits more by moving the load instructions ahead. When the L1 cache size exceeds 16KB, it can hold the entire array in L1. The benefits drop as the cache size increases.

L1 Cache	2 kB	16 kB	32 kB
Original	201952681	57494225	55288154
Optimized	184153968	56021472	55097846
Improvement	8.8%	2.6%	0.3%
Miss rate	10.8%	0.22%	0.08%

Figure 5: Varying Cache Size

L1 Cache access latency	4	16
Original	201952681	280078006
Optimized	184153968	261150173
Improvement	8.8%	6.7%

Figure 6: Varying Cache Latency

4.2 Cache/Memory access latency

The Cache/Memory latency factor will have similar effects on the load scheduling optimization as the cache size factor. It influences the load scheduling performance by affecting the average load latency. Figure 6 shows the performance of our system when we change the L1 cache access latency.

Increase the L1 cache access latency can highlight the benefits of moving load forward. But on the other hand, the latency should not be too large, otherwise the memory access would dominate the program execution time.

4.3 Window size

We show the performance of our system with different window sizes in Figure 7. In out-of-order processor, multiple instructions can be dispatched at the same time. As long as there are no dependences between them and there are available resource, the instructions can be issued simultaneously. This is hardwares mechanism to dynamically exploit instruction level parallelization. The dispatch window size puts a limitation the parallelization level. The larger the window size, more instructions can be potentially executed simultaneously and thus more likely the load latency can be overlapped.

As can be seen from the experiment results, the larger the window size, the less gain can be achieved by moving load instruction forward, because the hardware can still issue the instructions that are not depend on the load when a load instruction is stalled.

4.4 Computation overlap

We explore how our system performs when the number of computations overlapping with the moved load instruction changes. The results are shown in Figure 8. How early can we move the instruction forward is the tradeoff between two

window size	8	20
Original	201952681	166992948
Optimized	184153968	169497842
Improvement	8.8%	1.5%

Figure 7: Varying Window Size

	Original	Double
Original	8331314	9359179
Optimized	8203833	9694601
Improvement	1.5%	-3.4%

Figure 8: Varying overlapped computation

# of iterations	900	90000
Original	4067345	201952681
Optimized	4023783	184153968
Improvement	1.0%	8.8%

Figure 9: Varying Number of Loop Iterations

Size of array	100	10000
Original	18813495	201952681
Optimized	19483878	184153968
Improvement	-3.2%	8.8%

Figure 10: Varying Memory FootPrint

factors: 1). The earlier we move the load instruction, more computation instructions can overlap with load latency; 2). The earlier we move the load instruction, the larger register pressure we will suffer. Actually, if we move the load instruction too early such that the loaded data will not be used in the recent future, the compiler might split the register to memory. When the loaded element finally be used, it has to be re-loaded from the stack, which is again an memory access.

Doubling the computation instructions, we intended to see more computation being overlapped with memory access and thus result in more performance gain. But to our surprise, doubling the computation even get our worse performance. More experiments are needed to gain a thorough understanding of this result. One explanation of this result might be that although memory access latency is 75 cycles, the latency to access L1 and L2 are only 4 and 20 cycles separately. If a load instruction only accesses L2, the original computation () is enough to hide the latency. So doubling the computation would not hide the latency better. Meanwhile, after doubling the computation and moving the load instruction before all the computations, the register live range is further extended. This side effect may have caused the optimized version to slow down.

4.5 Number of loop iterations

When we first designed the testing program, we used a small iteration number and small array size. We expected to see the performance gain but failed. The results are shown in Figure 9. When the iteration number or array size is small (900 iterations and 100 array size), almost no difference can be observed.

By comparing the runtime trace and assembly code, we finally figured out the reason is that besides those array load instructions in the program, these are also many other unexpected load instructions that are either created by the compilers or induced in the execution runtime (i.e., load/store instructions to split intermediate register variables from/to stack because of lack of registers; initialization related instruction at the start of a function/program; instructions related to dynamically linking to library). When the program is small, those loads took a large part in the program. As a result, they overshadowed the optimization of load movement.

We made modifications to the program intending to highlight the benefits of load movements: We tried to use register types, in order to force the compiler to keep the intermediate variables in register, not to repetitively split into memory. We increase the iteration count of the program, in order to offset the effects of initialization overheads. We increase the array size, such that it is much larger than the L1 cache size. This is to magnify the array load latency.

4.6 Memory footprint

To understand how the LOAD motion optimization reacts to the memory footprints of test programs, we change the size of the arrays in the test programs and measure the performance improvement again. The effects are similar to changing cache sizes. When the memory footprint is small, almost everything fit into the L1 cache. Most load instructions are very fast as the data is already in L1. So moving LOADs around does not help. Instead, it hurts performance (likely due to the expansion of variable live ranges).

4.7 Distances the instructions are moved

Our basic version of load motion only moves load instructions to a spot one block away. This limitation prevent it from taking advantage of all optimizing opportunities. To study whether allowing a bigger moving distance will improve the performance even more, we try different load motion distance in terms of number of basic blocks, in this experiment. The test program we used in previous experiments does not provide opportunities to move more than one basic block. So we create a new test program with nested ifs just for this experiment. The pseudo code is shown in Figure 11 and the results are shown in Figure 12

We use window size of 20 in this experiment. Although we do not see performance improvement in any case, we do see that moving load instructions further away only makes things worse. This is consistent with our observation in experiment d). These two experiments show that our decision of moving load instructions only one block away is a close-to-optimal solution.

5 Surprises and Lessons Learned

5.1 Address-calculation Instructions

The very first time we ran our algorithm over our test data, we found that no loads were set to move. Looking into the bitcode of our program we found that that was because for even the simplest load actions, there would often be computations needed in order to get the memory location to load. For example, a single instruction accessing the nth position of an array would result in two instructions in the bit-code: one to get the memory location and a second to load. The loads dependency on the first instruction made it no longer a candidate for motion.

```
int loop(someList) {
   for i = 0 to 90000
      fileIndex = some computation of i
      if canOpen(fileIndex)
         fileIndex = some computation of i
         if canOpen(fileIndex)
            fileIndex = some computation of i
            if canOpen(fileIndex)
               fileIndex = some computation
                   of i
               if canOpen(fileIndex)
                  someElement =
                      someList[some map of
                      i] //LOAD
                  printf(someElement)
}
int main() {
   int weightList[10000];
   loop(weightList);
}
```

Figure 11: Another Test Program

Distance	1	2	4
Original	163316968	163316968	163316968
Optimized	174213794	183802317	179825892
Improvement	-6.6%	-12.5%	-10.1%

Figure 12: Varying Motion Distance

This problem lead to the development of a more active movement strategy. Once we found a load that was a candidate for motion, we would seek out all instructions within the block that the load depended upon. We would consider all these instructions as candidates for motion together.

5.2 Performance of load motion

To evaluate the performance gain of load movement program is a non-trivial task. It requires careful tuning and specially targeted benchmark to observe desired performance.

When we conducted the evaluation, we found that a lot of hardware optimization tricks should be taken into account when judging if moving the load forward can actually achieve the goal of overlapping load latency with computing instructions. We had to concern ourselves with such optimizations such as hardware prefetch, out-of-order execution, and prediction. The program itself needs to be carefully designed to bypass those optimizations.

Also on the software side, when moving the load instructions too early, the compiler will require to split the loaded register into memory, and we would have to manually force it not to split.

We originally intended to use some real JAVA projects to evaluate the performance. But normal JAVA programs or most widely open-source JAVA projects rarely has the patterns that would exhibit the benefits of moving load ahead (or can not obviously exhibit), so we finally decided to write a micro-benchmark on our own to better study the load movement.

To evaluate the performance, we tried a couple tools: Sniper simulator and Pin tools. We explored how different hardware features affects the load scheduling performance. We tuned different parameters and used the tools to look into assembly codes and runtime traces. These experiments help us gain a deeper understanding of how the compiled program actually runs on the hardware and those hardware features should be taken into account when doing load scheduling by compiler.

5.3 Profiling real-world open-source programs is hard

As Wenlu has experience analyzing large real-world commercial C# projects in [7], and experience working on Hadoop, she did not expect profiling large real-world open-source Java projects to be a lot harder.

We tried Apache Hadoop and HttpClient at first. After spending several hours configuring it and running Hadoop, we found out that it is not easy to re-compile the project. It uses Apache Maven for compilation, which is known for steep learning curve [4]. If we use Soot to instrument it, .java are automatically converted to .class. This means we need to change the Maven compilation process of Hadoop, and have it taking .class files as input instead of .JAVA. This may take a long time and is not even the focus of our work.

After learning this lesson, we came up with the Shell script instrumenting approach. This approach covers fewer cases, but is easy to implement and does not require understanding of the compilation process. Although we still fail to profile Hadoop as even unmodified code needs lots of tricks to successfully compile, we are able to instrument and profile six benchmarks from DaCapo Benchmark Suite - fop, luindex, lusearch, pmd, sunflow, and xalan. Among the six, only two, luindex and lusearch, actually produce logs. Others did not produce logs due to the coverage shortcoming of this approach.

6 Conclusions and Future Work

We explored load motion with natural language hints, and showed that it can improve performance by up to 9% when load stalls are considerable in programming running time, i.e. when memory footprint is large and/or cache size is small. We have also shown that under other conditions, the optimization may hurt performance but only by a little bit. Neither moving a load too close nor too far are desirable. The sweet spot may depend on hardware parameters like memory access cost.

There are two aspects in where we expect that we can improve our system. First of all, we believe we can improve the machine learning. As of now the partially-implemented advanced version of load motion can not run because the classifier only gives discrete result on which branch is more likely to be taken. If more data can be collected, an advanced machine learning model can be used to predict the continuous probability of taking each branch. We may be able to run and test the advanced load motion algorithm. The system will be then more knowledgeable of the cost of moving load instructions and thus make better decisions.

Second of all, we believe work can be done in improving our system's awareness of the hardware. The sweet spot of how far a load instruction should be moved tends to be hardware dependent. The benefit depends on how long each type of instruction takes. If we can add the hardware parameters to the input of the algorithm, the algorithm may be able to make smarter decisions.

References

[1] Apache ant. http://ant.apache.org/. [Online; accessed 28-April-2015].

[2] Apache maven. https://maven.apache.org/. [Online; accessed 28-April-2015].

[3] Java coding samples. https://www.cs.utexas.edu/ scottm/cs307/codingSamples.htm. [Online; accessed 28-April-2015].

[4] Stackoverflow: Why do so few people use maven? are there alternative tools? http://stackoverflow.com/questions/1077477/why-do-so-few-people-use-maven-are-there-alternative-tools. [Online; accessed 28-April-2015].

[5] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[6] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 52:1–52:12, New York, NY, USA, 2011. ACM.

[7] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 24–33, New York, NY, USA, 2014. ACM.

[8] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[9] LLVM. Llvm — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/LLVM, 2010. [Online; accessed 28-April-2015].

[10] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[11] Srikanth T. Srinivasan and Alvin R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, pages 148–159, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[12] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[13] Sebastian Winkel, Rakesh Krishnaiyer, and Robyn Sampson. Latency-tolerant software pipelining in a production compiler. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 104–113, New York, NY, USA, 2008. ACM.