More than Meets the Eye: Invisible Instructions

Guilherme V. Leobas
Researcher
DCC
UFMG
6627 Antônio Carlos Avenue
Belo Horizonte, Minas Gerais
31.270-213, Brazil
guihermel@dcc.ufmg.br

Breno C. F. Guimarães
Researcher
DCC
UFMG
6627 Antônio Carlos Avenue
Belo Horizonte, Minas Gerais
31.270-213, Brazil
brenosfg@dcc.ufmg.br

Fernando M. Q. Pereira
Professor
DCC
UFMG
6627 Antônio Carlos Avenue
Belo Horizonte, Minas Gerais
31.270-213, Brazil
fernando@dcc.ufmg.br

ABSTRACT

In modern software development, high-level languages are becoming progressively more feature-rich. The expressiveness and increased abstraction provided by these features allow programmers to be more productive and less concerned with low-level details. It is then the compiler's job to strip these layers of abstraction to actually implement its language's features. In contrast to average developers, compiler engineers operate within the compiler's infrastructure, looking for opportunities to optimize code or analyze programs. However, given their vantage point, these developers often assume that the program representation they use contains near-complete information of what will end up in the program's binary. We show that this is not quite the case. To this end, we introduce the notion of invisible instructions, which are present in the binary but are not visible in the program's compiler-generated intermediate representation. We use static analysis and profiling techniques to measure the prevalence of these instructions for a wide variety of programs in several benchmark suites, and show that for some instruction types, up to 36% of their occurrences on average are invisible.

CCS CONCEPTS

•Software and its engineering → Compilers;

KEYWORDS

Invisible Instructions, Profiling, Compilers

ACM Reference format:
Guilherme V. Leobas, Breno C. F. Guimarães, and Fernando M. Q. Pereira. 2018. More than Meets the Eye: Invisible Instructions. In *Proceedings of XXII Brazilian Symposium on Programming Languages, SAO CARLOS, Brazil, September 20–21, 2018 (SBLP 2018),* 8 pages.
DOI: 10.1145/3264637.3264641

1 INTRODUCTION

One of the major developments stemming from the advances in compiler technology has been the advent of increasingly feature-rich programming languages. Modern languages are more expressive and provide a much greater degree of abstraction than their predecessors. For instance, compilers are now capable of generating programs based only on a high-level description of their input/output [5–7]. Even for older languages, the addition of new features in later standards paired with the application of more elaborate compiler optimizations solidify this trend towards more abstraction as the norm. This frees programmers from having to worry about the minutiae of implementation, while the compiler does the heavy lifting.

However, while this increased expressiveness enhances productivity, it also widens the gap between the programs developers write and the code that is actually executed. Even for compiler engineers, who supposedly work in lower abstraction layers, there is no guarantee that what one sees in an intermediate representation (IR) is exactly what ends up in the binary. The compiler might insert or remove instructions due to several culprits, such as register allocation, stack management, program loading, etc., all of which take place after the program analysis/optimization stages. Therefore, even for the most mindful of developers, there is still more to a program than meets the eye at first glance.

In this paper, we investigate the problem of distinguishing which parts of a binary come from its source code and which have been inserted by the compiler. Our goal is to not only identify but also quantify this discrepancy. In a sense, we are trying to answer the question: "How far from its original code can a program end up being?". To this end, we define the notion of visible and invisible instructions. Roughly speaking, an instruction is said to be visible if it has an observable correspondent in the intermediate representation of the program. By exclusion, invisible instructions are all instructions which are not visible. Our approach towards counting these is twofold. First, we instrument the program's intermediate representation, marking instructions of interest which are visible, and run it to count their occurrences. Then we profile a non-instrumented version of the same program, counting all executions of instructions of interest. By comparing these measurements, we can estimate the size of the compiler-specific overhead.

Knowing the ratio between visible and invisible instructions has several benefits. The most important is the knowledge it brings about the impact of compiler transformations on the dynamic behavior of programs. Given that a compiler optimization can only touch visible instructions, there is a limit to what one can expect from the effects of said code transformation. This observation bears consequences similar to that of Amdahl's Law [1]: if an optimization is expected to bring X% improvement onto the code, and the

SBLP 2018, SAO CARLOS, Brazil

DOI: 10.1145/3264637.3264641

optimizable part of the binary is Y% of its executable trace, then one should expect an improvement of no more than X*Y/100.

We have implemented our ideas on top of the LLVM Compiler Infrastructure [10] and Pin [12], a dynamic binary instrumentation tool from Intel. Section 3 gives the necessary details of our implementation. We have collected 98 C and C++ programs to assess visible and invisible instructions. These benchmarks come mostly from the LLVM test-suite and span different areas of computation. For instance, Dhrystone [18] and Whetstone were developed to perform intense integer and floating-point computation, MediaBench [11] implements multimedia algorithms while MiBench [8] features graph network and security algorithms.

2 BACKGROUND

The notion of instruction visibility is vital to our work. In this section, we formally define which instructions are considered visible/invisible, and provide a working example to demonstrate their occurrences in real programs.

2.1 Definitions

Informally, we classify as visible those instructions in the program's intermediate representation that lead to the direct creation of code in the program's binary representation. Below we provide a loose definition of this notion:

Given a program P, its intermediate representation I, binary code B and a function $M : I \rightarrow B$ which maps intermediate representation instructions to their binary counterpart, an instruction $i \in B$ is said to be visible in P if $\exists\ x \in I \mid M(x) = i$. Intuitively, an instruction is said to be invisible in P if $\nexists\ x \in I \mid M(x) = i$.

Notice that this definition is rather abstract: the distinction between visible and invisible instructions is highly dependent on the compiler implementation, and on the kind of instruction. For instance, if we restrict ourselves to LLVM and x86, we find that this compiler's intermediate representation (IR) provides an opcode for addition, which has a direct counterpart in virtually any machine language, and, particularly, in x86. Therefore, the `addl` instruction in the x86 version of a program produced due to the `add` opcode in that program's IR is considered visible. However, instructions like memory fences, or transcendental functions, might not have direct correspondence between the compiler's IR and the architecture's binary language. Thus, henceforth, we shall ask the reader to believe that, for each instruction that we deal with in Section 4, we have a suitable implementation of the function M. This implementation provides a concrete separation between visible and invisible instructions.

Invisible instructions can come from one of the following: (i) from the loader, (ii) inserted by the compiler or (iii) from libraries linked dynamically to build the executable. In the first case, the loader is the program responsible to load the code and data from the executable object file, preallocate the necessary structures (i.e. stack), adjust the file descriptors and jump to the first instruction of the binary. As we show in Section 4, the number of instructions executed by the loader is almost constant between programs of the same language. In the second case, the compiler might insert or remove instructions due to various reasons, including stack management, spilling when doing register allocation or any other compiler optimization which transforms the program. In the last case, as an example, the C/C++ runtime library is dynamically linked by default in most compilers to reduce the binary size.

2.2 Why IR?

As mentioned in the previous section, our definition of visibility is based on a program's compiler-generated intermediate representation, not its source code. The reason we chose this representation is a matter of scope: our goal is not to identify instructions invisible to developers who write high-level code. Programmers who make use of language features or library abstractions trade low-level knowledge for programmability. Therefore, they are conscious that the compiler will implement these in whichever way it sees fit.

Rather, our focus is on developers who write compiler analyses and optimizations. Since these take place much closer to the code generation stage, they are often written under the assumption that they have near-complete information of the program's code. As we show, this is not always the case.

It is also important to note that the programs we analyze are all in Static Single Assignment (SSA) form [2]. A vast amount of optimizations in mainstream compilers such as GCC, LLVM and IntelCC rely on programs being in SSA form [13, 19]. Therefore, if one were to perform program analysis or implement an optimization in any of these compilers, the program would most likely be in this format. Thus, we are not analyzing straw man programs, but rather their most common representation.

2.3 Working example

We shall use the snippet of code from Figure 1 to illustrate the problem of not accounting for invisible instructions. This example contains a simple function which recursively calculates Fibonacci numbers, using memoization. At first glance, one might think that the number of memory-related instructions generated for this particular function is simply three loads and one store[1] for each call of fib. However, that is not quite the case, as we shall see later.

```
int fib(int *cache, int n){
    if(cache[n] != -1)
        return cache[n];
    cache[n] = fib(cache, n-1) +
               fib(cache, n-2);
    return cache[n];
}
```

Figure 1: Recursive C function that calculates Fibonacci numbers.

Figure 2 shows the intermediate representation (IR) for the *fib* function generated by LLVM. By default, variables in the IR are treated as stack variables and each operation on them requires the usage of a load and/or a store. Therefore, we will not use this raw version to compute invisible instructions, but rather an optimized version of it.

Figure 3 shows the bytecode generated in Static Single Assignment (SSA) form. By transforming the program into SSA, variables are all stored in virtual registers, whose quantity is unlimited. With

[1] Load/store are LLVM IR abstractions for any instruction which reads from/writes to memory.

this format, we get a program that better represents the binary generated at the end. We will not go any further in explaining the process for computing SSA form in this paper. LLVM has an optimization pass (*mem2reg*) that performs this transformation.

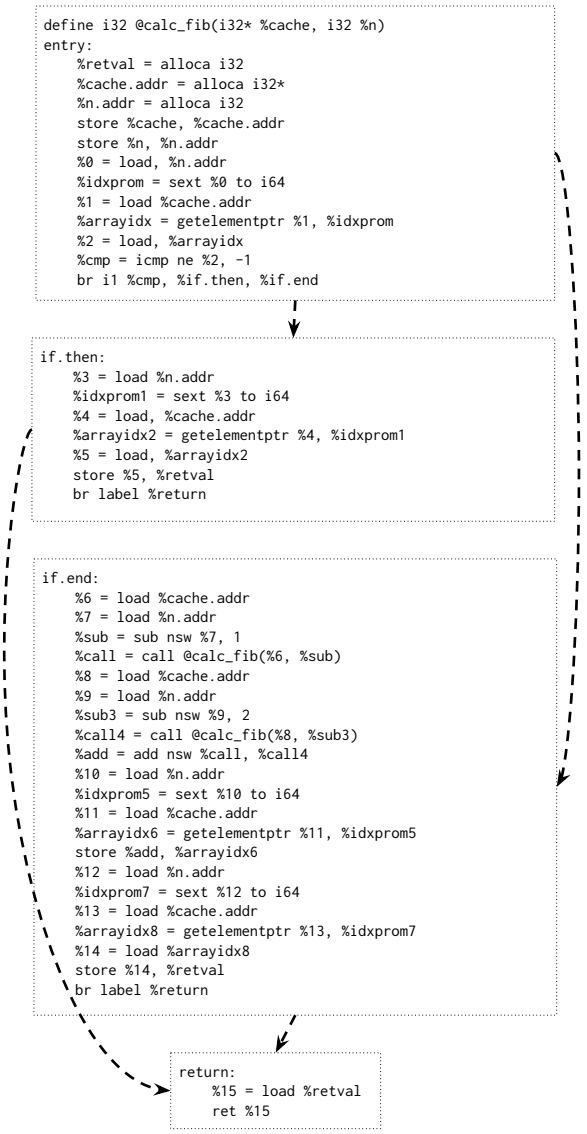


Figure 2: IR generated by LLVM.

Figure 4 shows the trace of an execution for the aforementioned function in an x86 architecture processor. Instructions were associated with a prefix indicating whether it is a store ([S]) or a load ([L]). The arrows show which instructions were originated from each part of the source code, and which instructions are due to stack management. As one can see, the number of memory-related instructions that are actually executed in the binary increases significantly.

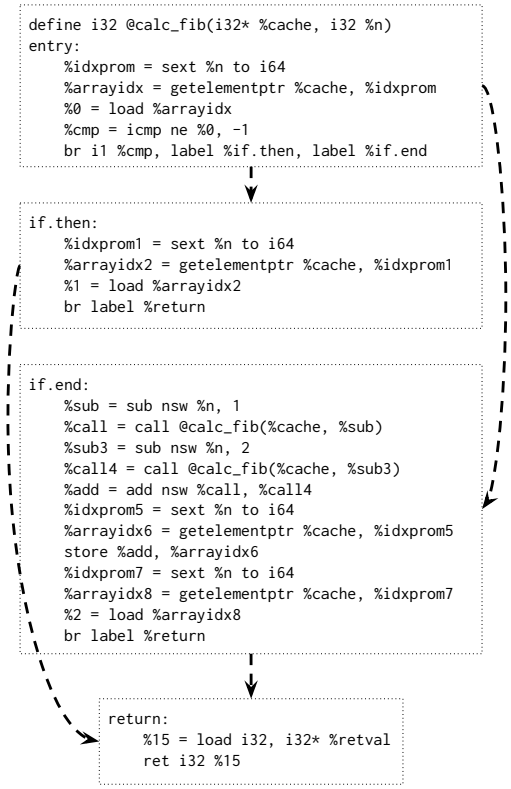


Figure 3: LLVM IR in Static Single Assignment form.

3 SOLUTION

To count the occurrences of invisible instructions, we first determine which type of instruction will be observed. These generally fall into three categories: memory manipulation, arithmetics/logic and control-flow. In our current implementation, we count all occurrences of each of these categories separately.

Once the instructions of interest have been determined, their measurement takes place in two major stages. The first one computes the set of visible instructions in a program. We achieve this by instrumenting the program's intermediate representation through an LLVM transformation pass, which marks all instances of the instruction type. We then generate an executable for the instrumented program and run it, counting how many markers are reached during execution. The second stage computes the set of all instructions of the given type which are actually executed in the program. To this end, we run a non-instrumented version of the same program, profiling its execution with Pin. This allows us to dynamically determine an instruction's type as it runs, so we count the executions of the type of interest.

Figure 6 shows a breakdown of the flow of our analyses. In the following sections, we go over the workings of each of these stages in greater detail.

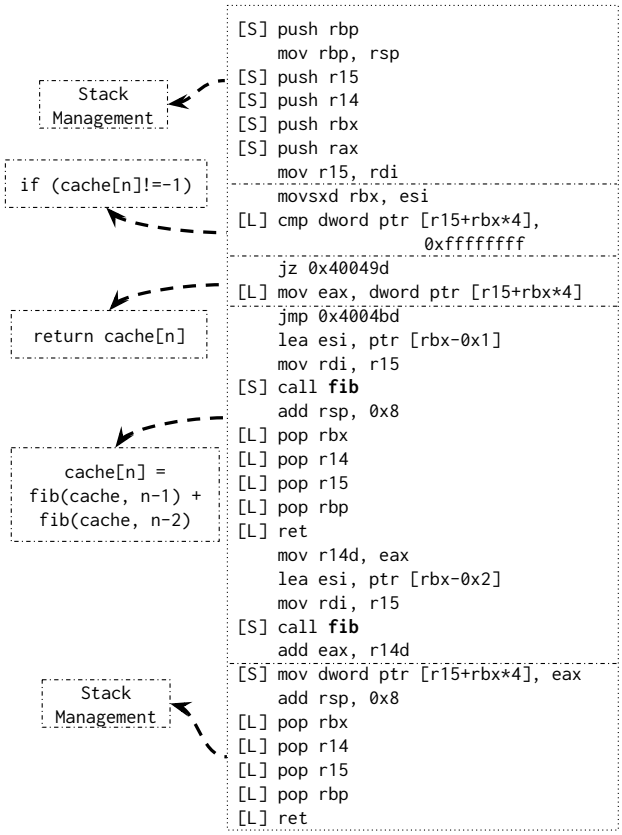


Figure 4: Assembly Instructions executed for the fib function in an x86 architecture. The prefix [S] indicates that the instruction writes to memory while the [L] prefix indicates a memory read.

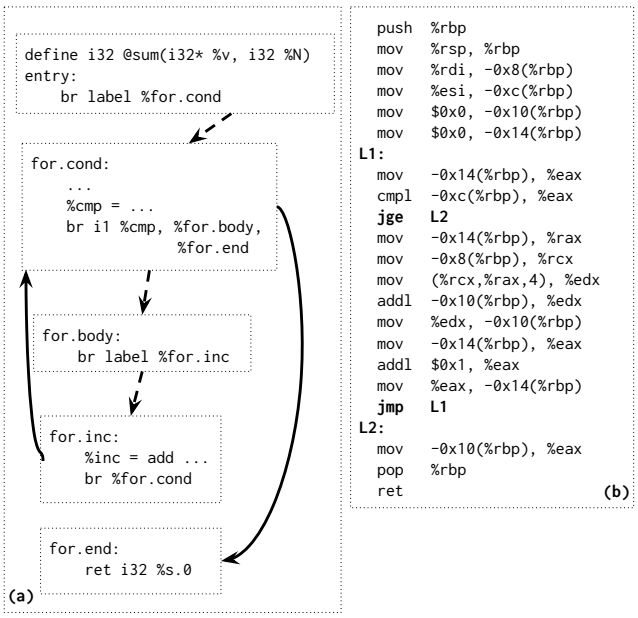


Figure 5: (a) Control Flow Graph for a function that sums the values of an array and (b) its x86 assembly code with jump instructions highlighted.

Table 1: Mapping between LLVM IR instructions and its x86 equivalent

LLVM IR	x86
ADD	ADD, INC
UDIV, SDIV, UREM, SREM	DIV, IDIV
FADD	ADDPD, ADDPS, ADDSD, ADDSS, FADD, FADDP, FIADD
FMUL	MULPD, MULPS, MULSD, MULSS, FMUL, FMULP, FIMUL
BR	JMP, JE, JNE, JG, JGE, JA, JAE, JL, JLE, JB, JBE, JO, JNO, JZ, JNZ, JS, JNS
LOAD	*
STORE	*

3.1 LLVM Instrumentation

Our LLVM instrumentation works as follows: first, we transform the IR into SSA form. Then, for every target instruction, we add an operation immediately preceding it to increment a variable defined in an external C library. This library is a hash map where each key is associated with an opcode and its value stores how many times an instruction with that opcode was executed. We link this library with the instrumented program's object code when generating the binary. We also insert a call to a function to dump this hash map to a file before any program exit points.

Branch instructions require some extra care. We cannot simply add markers immediately before each to count them, because jump instructions in the LLVM IR are not guaranteed to end up in the binary. Take the Control Flow Graph of a simple array sum function in Figure 5.a for example. Since each edge in a CFG corresponds to a branch, one would expect the same amount of jump instructions in the assembly code. However, it is clear to see that there are more edges in the graph than there are branch instructions in the assembly in Figure 5.b. This is because Basic Blocks can be rearranged in such way that jumps are no longer necessary. For instance, the Basic Blocks *entry*, *for.body* and *for.inc* can be arranged sequentially in the binary, causing the branches in dashed edges to become unnecessary. To avoid mistakenly counting these, our approach works by inserting the increment at the beginning of each Basic Block that has at least two entry points. This way we guarantee that at least one of the entry points comes from a jump.

3.2 Pin Analysis

We use Pin to profile the execution of a non-instrumented version of the Program using what is called a Pintool. A Pintool can be seen as a plug-in made up of two major components: instrumentation code and analysis code. The former is a mechanism that decides which part of the binary is analyzed and the latter is the code to be executed. Pin works by intercepting the first instruction of the program right after it loads into memory, then small sections of the program are analyzed and just in time recompiled together with analysis code.

We have implemented three Pintools to profile program execution: *ProfileArithmetic*, *ProfileBranches* and *ProfileMemoryAccess*. *ProfileBranches* and *ProfileMemoryAccess* work by leveraging the Pin

API to determine if a given instruction is a branch or manipulates memory, respectively.

ProfileArithmetic's implementation is a bit more elaborate. Since there is no method available in the Pin API to programmatically determine whether an instruction is a binary arithmetic operation, we need to define a mapping of our own. To this end, we have computed a set of relations between LLVM IR instructions and their x86 counterparts. While not exhaustive (due to the extensiveness of the x86 instruction set), these relations encompass all of the most commonly used arithmetic instructions. *ProfileArithmetic* then works by simply counting the occurrences of instructions that are contained in this set. Table 1 shows the $LLVMIR \rightarrow x86$ mappings we have used to determine instruction visibility.

Note that for load/store instructions there is no reasonably-sized set of x86 counterparts. This is due to fundamental differences between the two levels of abstraction. While LLVM defines a strictly typed low level language, x86 assembly is much more permissive. Therefore, most x86 instructions can receive memory addresses as one of its operands, essentially making most of them capable of reading from/writing to memory. Fortunately, Pin allows us to check when this is the case, so we can compute memory manipulation instructions properly.

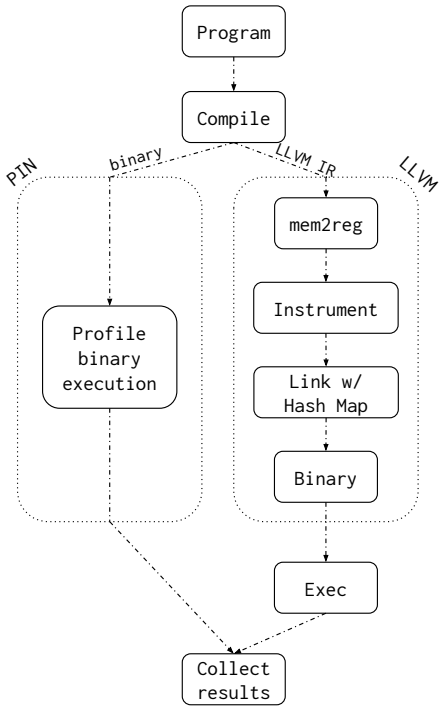


Figure 6: Flow chart of execution for LLVM instrumentation and Pin profiling.

4 RESULTS

In this section, we evaluate the prevalence of invisible instructions in programs from a wide variety of application areas. We analyze instruction types separately, and present results for each type across the entire collection of benchmarks we have used.

4.1 Experimental Setup

We have implemented our analysis on top of LLVM 3.8 and Intel Pin 3.6. Our hardware setup consists of a 12-core Intel(R) Xeon(R) E5-2620, at 2.00GHz, with 16GB of RAM running Ubuntu 16.04. To take advantage of the 12 cores our machine disposes of, we developed a test-framework which can compile and execute benchmarks in parallel.

We run both the instrumented and non-instrumented versions of each program with the same inputs until completion, collecting the number of visible instructions and the execution traces to count the total number of instructions executed.

Benchmarks: To test our ideas, we have collected 98 programs from 24 different test suites. Most of these benchmarks come from the LLVM test-suite and are widely used by compiler engineers to evaluate the effectiveness of optimizations and analyses.

These benchmarks are all written in C/C++. We chose these languages for two main reasons. First, they provide a relatively small degree of abstraction compared to other modern languages. Therefore, one would expect the binaries for programs written in these to be fairly close to their intermediate representation. One of our goals is to evaluate whether this is truly the case. Second, because of the wide variety of compiler optimizations written for these languages. C/C++ compilers are notorious for performing aggressive code transformations. For instance, LLVM currently features over 50 optimization passes[2], all of which are performed on a program's intermediate representation. Given that these transformations can only affect visible instructions, the impact of all of these is limited to what portion of the program is available to them. Therefore, measuring this portion is also important.

4.2 Experiments

Given a program P and an instruction type T. We shall use T_{LLVM} to refer to the number of visible instructions for a type T and T_{Pin} as the total number of instructions for the same type T. The number of invisible instructions is simply the difference between the total and the visible portion.

$$T_{\texttt{inv}} = T_{\texttt{Pin}} - T_{\texttt{LLVM}}$$

The percentage difference is the ratio between the amount of invisible instructions and the total.

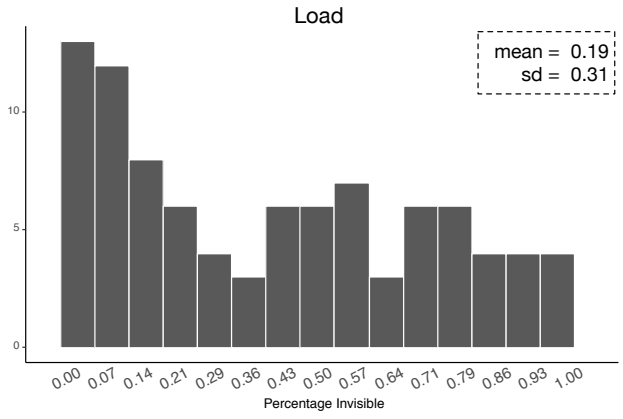
$$\%T_{\texttt{inv}} = \frac{T_{\texttt{inv}}}{T_{\texttt{Pin}}} = \frac{T_{\texttt{Pin}} - T_{\texttt{LLVM}}}{T_{\texttt{Pin}}}$$

Figures 7 to 10 show histograms alongside their geometric mean and standard deviation for the set of instructions studied in this work. Each histogram was built using 15 as the number of bins. The x-axis represents the percentage of invisible instructions while the y-axis the amount of programs whose percentage of invisible instructions fall within that range. Intuitively, programs in the 0.0 column have all instructions of that type visible in their intermediate representation, whereas for those in the 1.0 column all of them are invisible. We omit the cases when T_{LLVM} was larger than T_{Pin} to plot the histogram. These cases represents no more than 6

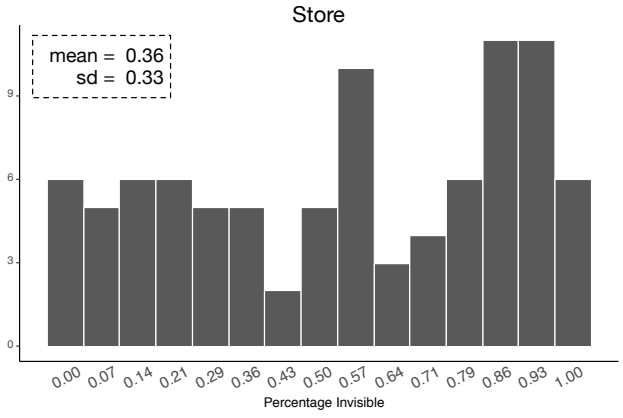
[2] https://llvm.org/docs/Passes.html

benchmarks out of 98 and the computed $\%T_{inv}$ value was close to 0.0.

Figures 7a and 7b show the distribution for memory read (*Load*) and memory write (*Store*) instructions. Interestingly, instructions that write to memory are much more likely to be invisible than memory reads, having a mean ratio of 19%, as opposed to 37% for the latter. Their variation in prevalence across different programs, however, seems to be close to equal, with both having similar standard deviations. In fact, for any given instruction type, we observed similar deviations across all of instructions of that type.



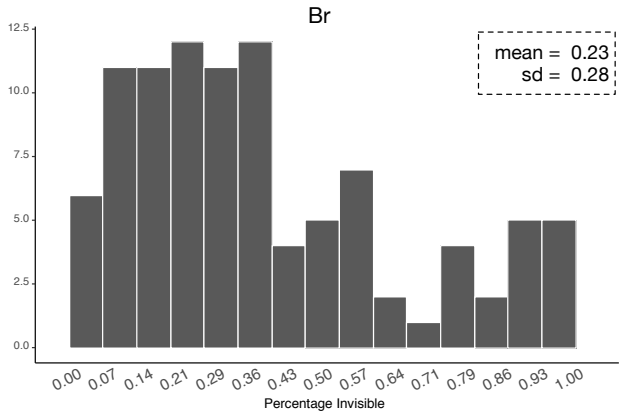
(a) *Load Instruction*



(b) *Store Instruction*

Figure 7: (a) Histogram for the proportion of invisible memory read (*Load*) instructions. (b) Histogram for the proportion of invisible memory write (*Store*) instructions.

Figure 8a shows the histogram for branch instructions. As pointed out in section 3, our LLVM pass for this particular set of instructions is likely to not be completely accurate, due to how the amount of jump instructions in the binary may vary depending on how basic blocks are organized in the sequential code. Still, our measurements show a considerable amount of jumps as being invisible, with a mean prevalence of 23%.



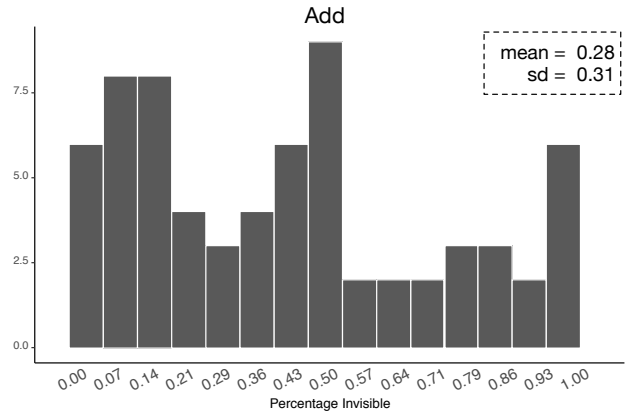
(a) *Branch Instruction*

Figure 8: Ratio of invisible *Branch* instructions across all benchmarks.

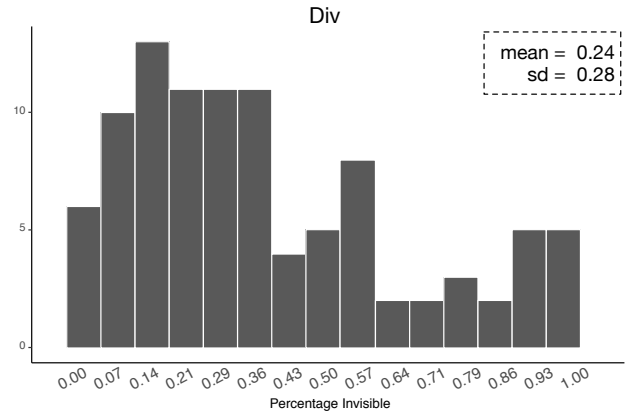
Figures 9a and 9b show the distributions for integer addition and division. We chose to display results for these two due to their contrasting traits. While addition is the simplest of arithmetic operations and usually the cheapest to run, division is usually considered expensive. Therefore, one would expect compilers to be more reluctant to insert division operations, and thus their likelihood to be invisible to be much lower. Our results show that while div instructions are indeed less likely to be invisible, they are not so by such wide a margin. The mean prevalence of invisible additions is 28%, while the mean ratio for divisions is 24%, which are surprisingly close.

We also analyzed floating-point arithmetic operations. Figures 10a and 10b show the histograms for floating-point addition (*FAdd*) and floating-point multiplication (*FMul*) instructions. As one can see, the column 0.0 aggregates nearly all of the data, therefore, the likelihood of floating-point instructions being invisible is close to zero. Intuitively this result makes sense, given that floating-point operations are inherently more complex and less likely to be the target of backend optimizations. Moreover, integer instructions are general-purpose, being used for tasks such as indexing memory or determining control-flow. Their floating-point counterparts, on the other hand, are more expensive and tend to only be used in their own niche, and are thus less likely to be inserted by the compiler when implementing other features. Hence, the results presented in these figures are expected, with most floating-point instructions being visible. It is also worth noting that no floating-point instructions were executed by any program before its entry point (*main* function) or after it finished running (*return* or *exit* calls).

Finally, we measured the number of instructions executed before a program's *main* function begins running and after its execution halts. The former are usually related to the loader, which sets up the environment where the program will run. The latter are usually associated with resource cleanup. Due to the differences in supporting infrastructure between the two languages, for this experiment we executed only the benchmarks written in C. Table 2 shows



(a) ***Integer Addition Instruction***



(b) ***Integer Division Instruction***

Figure 9: (a) Distribution of invisible integer addition (*Add*) instructions. (b) Distribution of invisible integer division (*Div*) instructions.

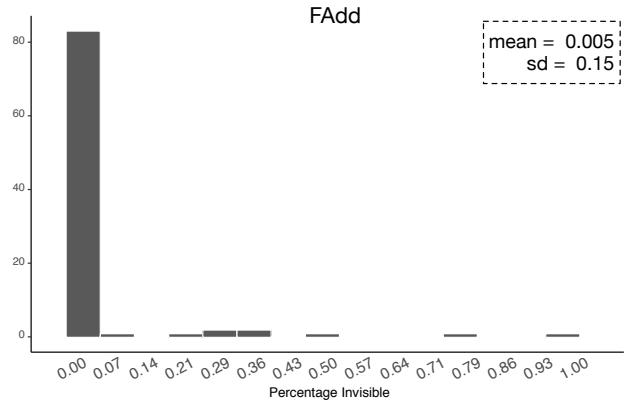
the geometric mean for the number of each type of instruction which are executed in these two scenarios across all benchmarks. All instruction types seem equally likely to occur before/after the program's execution, with only memory reads (*Load*) being exceptionally more likely to occur in the "before" stage. The number of these instructions executed also varies very little across benchmarks, with very low standard deviation values for all instruction types. These instructions also represent an extremely low percentage of the total number of operations of that type executed by each program. All of these indicate that the loader/cleanup overhead is nearly negligible and nearly constant, when it comes to C binaries.

5 RELATED WORK

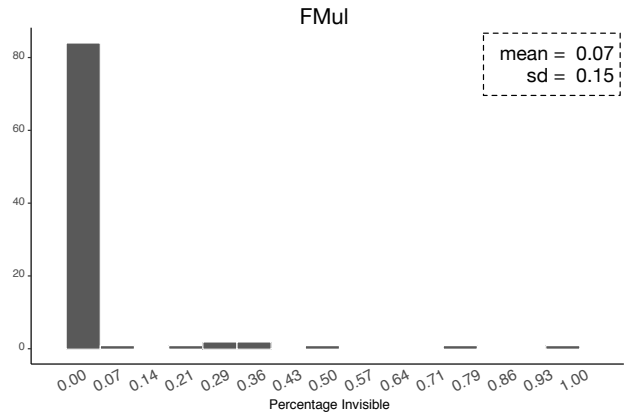
Categorizing and tallying instruction executions at the binary level is not a novel approach. In fact, Patterson and Hennessy's seminal book on Computer Organization has an entire subsection dedicated to this [15]. In it, they present the execution times and number of instructions executed for the 12 integer benchmarks from SPEC2006 [9]. Guthaus et al. [8] go further and present an insightful instruction distribution for the MiBench and SPEC2000 benchmark suites, split between four major instruction types. The categories we use were based on their approach.

Table 2: Mean and standard deviation of the number of instructions of type T executed before *main* and after *return* or calls to *exit*. Third column represents the percentage of those instructions in the program

	Before main			After main		
Instruction	**mean**	**sd**	**%**	**mean**	**sd**	**%**
ADD	14 629	192	0.0004%	283	129	0.000008%
DIV	18 635	198	0.0002%	236	111	0.000002%
FADD	0	0	0%	0	0	0%
FMUL	0	0	0%	0	0	0%
BR	19 741	245	0.0002%	552	259	0.000006%
LOAD	28 635	326	0.0001%	352	162	0.000001%
STORE	14 120	122	0.0002%	433	197	0.000006%



(a) ***Float Add Instruction***



(b) ***Float Multiplication Instruction***

Figure 10: (a) Distribution of invisible floating-point addition (*FAdd*) instructions. (b) Distribution of invisible floating-point multiplication (*FMul*) instructions.

Nevertheless, to the best of our knowledge, the literature does not contain the kind of study that this paper presents. Our personal communication with several different researchers reveal that there is much guessing, and very little experimentation when it comes to separating visible and invisible instructions. This fact is unfortunate, not only because it complicates the analysis of the benefits of compiler optimizations, but also the reach of techniques to make software more secure. For instance, techniques like AddressSanitizer [16] and Low-Fat pointers [4] protect source code: they require the recompilation of programs, and guarantee safety only for the compiled part. Thus, what is the actual impact they have on the entire software stack? We have not found any study on the literature concerning this aspect of such tools.

Regardless, even though we do not know any study that is similar to ours, the systems community already has lots of tools in place to carry it out. For instance, `perf`[3] lets us count the number of binary instructions executed by a program. Furthermore, where we perform our experiments using PIN [12], valgrind would also do [14]. We have used LLVM [10] to count the visible part of the instruction set that makes up a program. Soot [17] could be used to obtain the same effect in Java. That is, in fact, a direction that we hope to further explore in the future.

6 FUTURE WORK

While in this work we presented the way we define and measure invisible instructions, we plan on applying these concepts on a few practical scenarios. One such scenario is in detailed studies of the impact of compiler optimizations. Take for instance a Dead Store Elimination optimization, whose goal is to remove redundant store operations. If one applies this optimization and observes a reduction of 2% in the number of stores in a given program, it might seem like a modest result. However, if this program's ratio of invisible stores is 90%, it means the optimization reduced the number of stores *it could analyze* by 20%, which is a much more expressive result.

We also believe invisible instructions can be a useful metric when measuring software characteristics. When comparing programs in different programming languages, for example, the ratio of invisible instructions could be used as a way to quantify abstraction. That is, how much of the implementation of a program is left to the compiler in a given language? Similarly, it could also be used as a way to compare the quality of the code generated by different compilers, or for different architectures. All of these are research directions we plan on pursuing in future work.

7 CONCLUSION

This paper investigated the problem of distinguishing which parts of a binary come from the source code and which have been inserted by the compiler. To this end, we have introduced the notion of invisible instructions. To measure the prevalence of such instructions, we implemented an analysis on top of the LLVM Compiler Infrastructure and Pin, a dynamic binary instrumentation tool from Intel. We have executed our analysis on 98 benchmarks taken mostly from the LLVM test-suite. Our experimental results lead us to believe that a considerable fraction of a program is invisible to the eyes of a compiler optimization pass.

ACKNOWLEDGMENT

We thank the SBLP referees for their many insightful comments and criticism. Guilherme Leobas has received the support of a scholarship from the Brazilian Ministry of Education (through CAPES). Fernando Pereira has received the support of the Brazilian Research Council (CNPq).

REFERENCES

[1] Gene M. Amdahl. 1967. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference (AFIPS '67 (Spring))*. ACM, New York, NY, USA, 483–485. DOI: http://dx.doi.org/10.1145/1465482.1465560

[2] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490. DOI: http://dx.doi.org/10.1145/115372.115320

[3] Arnaldo Carvalho De Melo. 2010. The new linuxfiperffitools. In *Slides from Linux Kongress*, Vol. 18.

[4] Gregory J. Duck and Roland H. C. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *CC*. ACM, New York, NY, USA, 132–142.

[5] Sumit Gulwani. 2010. Dimensions in Program Synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP '10)*. ACM, New York, NY, USA, 13–24. DOI: http://dx.doi.org/10.1145/1836089.1836091

[6] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 317–330. DOI: http://dx.doi.org/10.1145/1926385.1926423

[7] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-free Programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 62–73. DOI: http://dx.doi.org/10.1145/1993498.1993506

[8] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 3–14.

[9] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[10] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[11] Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*. IEEE, 330–335.

[12] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200. DOI: http://dx.doi.org/10.1145/1065010.1065034

[13] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[14] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *PLDI*. ACM, New York, NY, USA, 89–100.

[15] David A Patterson and John L Hennessy. 2013. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Newnes.

[16] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX*. USENIX Association, Berkeley, CA, USA, 28–28.

[17] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *CASCON*. IBM Press, 13–.

[18] Reinhold P Weicker. 1984. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM* 27, 10 (1984), 1013–1030.

[19] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2013. Formal Verification of SSA-based Optimizations for LLVM. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 175–186. DOI: http://dx.doi.org/10.1145/2491956.2462164