Instruction Visibility in SPEC CPU2017

Andrei Rimsa Álvares

CEFET-MG, Brazil

José Nelson Amaral

University of Alberta, Canada

Fernando Magno Quintão Pereira

UFMG, Brazil

Abstract

When analyzing the impact of compiler optimizations upon the execution of benchmarks, it is important to know which portion of the executed code can be influenced by the compiler, and which portion cannot. From the compiler perspective, the program text is *visible*. Pre-compiled library code is *invisible*, because the compiler can only improve the program text. Thus, an important question, from the point of view of compiler developers, is what proportion of the instructions executed by these benchmarks are visible when they are executed with reference inputs. This paper answers this question for the 43 programs of SPEC CPU2017 benchmark suite, when compiled with GCC and executed on an x86 processor. This information helps both computer architects and compiler designers to assess the potential gain of application-level code transformations. On average, 92.2% of the instructions executed by SPEC CPU2017 are visible. In contrast, 10.8% of the instructions executed in a typical run of GNU CoreUtils are visible. Different optimization levels tend to preserve these ratios. This stark contrast indicates that developers should exercise caution when projecting performance results from compiler-based code transfor-

Email addresses: andrei@cefetmg.br (Andrei Rimsa Álvares), jamaral@ualberta.ca (José Nelson Amaral), fernando@dcc.ufmg.br (Fernando Magno Quintão Pereira)

mations measured in SPEC CPU2017 onto other applications. Amdahl's law is a reminder that the performance improvement to a system is limited by the portion of execution time that cannot be affected by the change.

---

1. Introduction

The Standard Performance Evaluation Corporation (SPEC) was founded in 1988 to create standard benchmark suites to evaluate the performance of computer systems [1]. Since then, SPEC has evolved to provide benchmarks for Cloud Computing, Graphics, Java, Storage, Virtualization, Web services and Power Consumption, including SERT, a suite widely used to evaluate computer server efficiency. For the compiler research community, the SPEC CPU benchmark suite has been the dominant source of programs used to predict the effects of innovation on the performance of computing systems [2]. This suite focuses on compute-intensive applications that stress the CPU and the higher levels of the memory hierarchy. The first collection of SPEC CPU benchmarks in this series was announced in October 1989 [3]. The most recent release is the SPEC CPU2017 benchmark suite. SPEC CPU2017 is divided into four suites: `intspeed`, `fpspeed`, `intrate` and `fprate`. Programs in these collections are implemented in either C, C++ or Fortran.

SPEC CPU has been fundamental to the standardization of performance measurement in the computing industry. This impact has, unsurprisingly, been of great consequence to the development of compilation technologies. As testimony to this statement, the many editions of Patterson and Hennessy's classic textbook, including its latest version [4], rely on different releases of SPEC CPU to justify hardware design decisions. While compiler developers use the entire benchmarks, many efforts to subset SPEC benchmark suites led to the capture of the behavior of the programs with shorter simulations for computer architecture studies [5, 6, 7]. Therefore, compiler writers and computer architects tend to employ SPEC CPU as a beacon: good code optimizations should perform well in this benchmark collection. Such importance, however, elicits one

question from a compiler's perspective:

> [**Q**] How much of the gains obtained in SPEC CPU can be extrapolated to benefits into actual applications?

To answer the above question, this paper analyzes the provenance of machine instructions processed during the execution of SPEC CPU2017 programs. This study uses InstrGrind, a dynamic analyzer, based on valgrind [8], that counts executed instructions and classifies them as *visible* and *invisible*. The former comes from the source code of the benchmark; the latter, from libraries invoked during its execution. Visible instructions determine the proportion of the program that the compiler can modify.

When we asked a group of ten experienced compiler developers, who have knowledge and experience with the SPEC CPU benchmark suites, which proportion of the instructions executed by these benchmarks are visible to the compiler, their average answer was remarkably off the mark: 57.5% instead of the 92% that we measured. This paper sheds light on the representativeness of the performance results obtained by testing with SPEC CPU benchmarks. To assess the representativeness of SPEC CPU2017, this paper uses InstrGrind to measure instruction provenance in the GNU CoreUtils library. GNU CoreUtils consists of 106 utility programs used in UNIX-like operating systems. Examples include `cat`, `mv` and `ls`. The GNU CoreUtils programs are the epitome of system applications: they interact heavily with the operating system, are input bound, and are massively used in production environments. The comparison of instruction provenance in SPEC CPU2017 and GNU CoreUtils leads to the following results:

- The proportion of visible instructions found in SPEC CPU2017 when compiled using gcc with -O0 is, on average, 0.922.
- The same ratio in GNU CoreUtils, when compiled with gcc at -O0 is, on average, 0.108; hence, substantially smaller than SPEC CPU2017's.

Thus, whereas most of SPEC CPU2017's executed instructions are visible, most of GNU CoreUtils' are invisible.

- The disparity between these ratios remains for code generated with higher levels of optimization. For instance, the ratio observed in SPEC CPU2017 compiled using gcc with -O2 is 0.701. Similar behavior holds for GNU CoreUtils: the proportion of visible instructions is still low, at 0.073.

Summary of Methodology. Following SPEC's recommended methodology [1], all the averages of ratios are geometric means. Measurements in SPEC CPU2017 use the reference inputs of each benchmark. Modified versions of the scripts distributed with GNU CoreUtils are used to execute programs in this collection. This study reports results obtained in a 16-core Intel(R) Xeon(TM) E5-2630 running at 2.40GHz, with 32GB of RAM on Linux CentOS 7.5. Benchmarks were compiled with gcc version 7.3.1. Results reported in this paper indicate that there is greater potential for compiler-based code transformations to impact the performance of SPEC CPU2017 than the performance of applications from other domains. This observation is confirmed by the total execution time on different levels of compiler optimization. The execution time of SPEC CPU2017 at level 2 (gcc with -O2) is 45% lower than at level 0 (gcc with -O0). For GNU CoreUtils this reduction is 1%. The rest of this paper provides further details about our empirical analysis. Given that the provenance information depends only on the portion of the code that comes from source and from libraries, the conclusions from this study should remain true for different runtime settings, although this study does not provide data to corroborate this observation.

This paper in perspective. A new dynamic tool was crafted specifically to aid in our investigations: InstrGrind (Section 4). This tool was built with the expertise acquired when building CFGgrind, another tool that was the subject of previous publications of ours [9, 10]. CFGgrind and InstrGrind are dynamic analysis tools implemented on top of valgrind. The former is much more precise: it produces a high-level description of the parts of a program that

have been exercised by a given input, i.e., its control flow graph. However, such precision makes it unnecessarily slow for the goal pursued in the present work: to separate visible from invisible instructions in SPEC CPU2017. Therefore, InstrGrind emerged as a natural derivation of our former work to fit these new objective. It is faster, yet less general, as described in Section 2.

2. Valgrind as a Platform for Developing Dynamic Analysis

InstrGrind, the tool developed for this paper, performs a type of *dynamic analysis*. Dynamic analyses are code-understanding techniques that extract information from the execution of programs while *static analyses* extract information from the text of programs without running them. Both can be used to analyze the behavior of programs to extract properties of interest. When applied to binary programs, the information obtained from static analyses can be used to modify such programs by adding, changing, or removing instructions. Dynamic analyses are not usually used to change the program's binary, rather they monitor the execution of the program to either report statistics about the execution or to recover control-flow or data-flow information. When combined with binary optimization, dynamic analysis may change the program during execution. There exist several frameworks onto which dynamic analyses can be implemented, such as `gprof` [11], `Pin` [12], `QEMU` [13], `DynamoRio` [14] and valgrind [8]. These frameworks support building tools to perform dynamic analyses and code transformations. They have been used before to find parallelization opportunities [15, 16], memory errors [17, 18], and to protected against control-flow based attacks [19]. InstrGrind capitalizes on the infrastructure of valgrind to analyze the visibility of instructions (Section 4).

valgrind is a virtual machine that intercepts instructions of a host physical machine, originally created by Julian Seward, in the early 2000's [17]. It supports different architectures, such as x86, ARM, PowerPC and MIPS, among others. Interception happens by compiling instructions from the host machine, in a just-in-time fashion, to an intermediate representation called VEX IR. Tools plugged into valgrind can process this representation, either running

dynamic analyses on them or transforming them. Code in this representation is then translated back into the host architecture's instruction set. The host instructions run directly on the target device. There are several tools built onto VALGRIND. The most well known among them is probably `memcheck` [8], a tool to detect memory-related bugs, such as memory leaks and write-after-free bugs, for instance. Other examples of VALGRIND tools include `cachegrind`, a cache profiler that performs detailed simulations of the L1, D1 and L2 caches, and `callgrind`, an extension to `cachegrind` that produces extra information about call graphs.

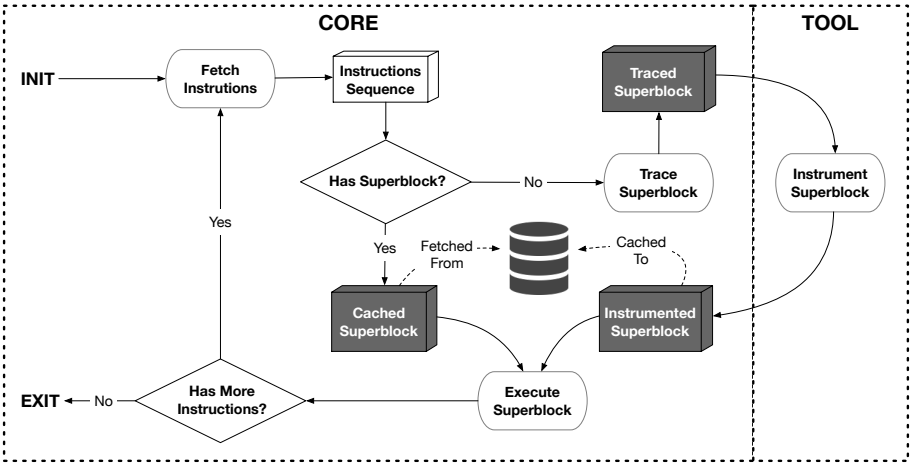

Figure 1: VALGRIND's execution workflow.

VALGRIND's architecture is divided into two main parts: the core that provides low-level support, such as program loading, just-in-time translation to and from VEX IR, and other useful services; plus the tool that actually performs the instrumentation, like `memcheck` and others. Figure 1 shows a diagram of VALGRIND's execution workflow. At the core, VALGRIND first identifies a sequence of instructions that can be executed as a unit. If this sequence is viewed for the first time, then these instructions are translated into VEX IR to form a superblock. This superblock is thus passed to the tool to perform the instrumentation. Afterwards, the superblock is returned to VALGRIND's core and cached for fast retrieval. Finally, this superblock is translated back to the

host's assembly instruction set using a just-in-time compiler for execution. In case of a previously seen sequence of instructions, VALGRIND recovers its cached instrumented superblock and proceed with its execution.

A tool described in previous work, CFGGRIND [9, 10], recovers control-flow graphs dynamically. In principle, it could also be used to check the visibility of instructions. However, although it yields information that is invaluable to the analyses of executable programs, the goal of this paper can be achieved with a simpler dynamic analysis. This observation motivates the design of INSTRGRIND. It shares some infrastructure with CFGGRIND, but without the unnecessary fine-grained information required by the construction of control-flow graphs. The next sections shall provide details on how INSTRGRIND works, while introducing its *raison d'être*: the separation between the visible and the invisible instructions executed by a program.

3. Visible and Invisible Instructions

In a computer application, only part of the instructions executed come from the application's source code. The rest of the instructions comes from dynamically linked libraries and routines added by the compiler, such as initialization (pre-main code) and finalization (post-main code). This distinction is formalized by Definition 1.

Definition 1 (Visibility). *Given a compiled program P with source code S, and a compiler C, the* visible *instructions of P are the instructions that were produced by C's code generator for statements that appear in S. Every other instruction required for the execution of P is an* invisible *instruction.*

Example 1 (Visibility). *Figure 2 shows* `countdown.c`*, a C program whose function* `main` *invokes three functions that belong to the* `libc` *library:* `strtol`*,* `printf` *and* `sleep`*. This program is dynamically linked to the* `libc` *library file in Figure 2. The executable file contains the translated x86-64 assembly instructions of the* `main` *function, while the library file contains many functions, including the three functions invoked by this program. According to Definition 1,*

the instructions in the executable file that originated from the **`main`** *function are* visible*, every other instruction, including the ones implemented in library functions such as* **`strtol`**, **`printf`** *and* **`sleep`**, *is* invisible*. Instructions originated from dynamically linked libraries are not necessarily considered invisible. As long as the instructions are derived from the program's source code, as stated by Definition 1, it does not matter if they are dynamically linked or not. These instructions will be considered visible.*

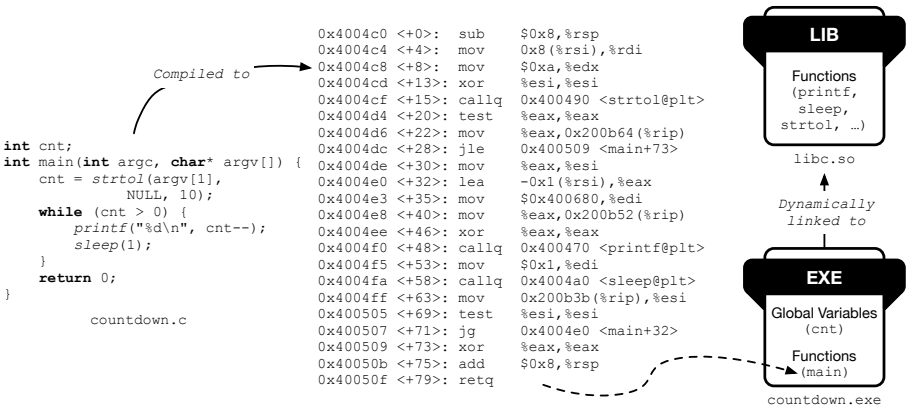

Figure 2: Example code in C (left) compiled to x86-64 assembly instructions (middle) produced the executable file that is dynamically linked to the `libc` library (right).

The concept of visibility leads to the notion of *Visibility Ratio* of a program execution. Visibility is a static property of an instruction, established by its provenance. However, the visibility ratio of a program execution is determined dynamically, because it depends on the number of instructions from each category (visible or invisible) that is processed.

Definition 2 (Visibility Ratio). *Let V be the number of* visible *and I be the number of* invisible *instructions executed during a run of a program P with workload W. The visibility ratio of (P, W) is $\frac{V}{V+I}$.*

4. Analyzing Instruction Provenance

The measurement of the visibility ratio of a program execution requires instrumentation that counts how many times each instruction is executed. This in-

strumentation must be dynamic — it cannot be statically inserted into the code because a compiler only has access to visible instructions. Thus, InstrGrind, a tool that we built on top of valgrind [8], is used to perform the instrumentation. The choice for valgrind is due to our familiarity with it. Any other dynamic binary instrumentation framework could have been used instead, such as Pin [12] or Qemu [13].

4.1. Instrumenting Groups of Instructions

Although every instruction in an execution must be accounted for, there is no need to instrument all of them. Instrumenting all the instructions would lead to an impractical runtime overhead. Instead, InstrGrind instruments groups of instructions. Each group contains a set of consecutive instructions that are always executed as a unit. The head of the group is the first instruction executed by the program, or an instruction immediately executed after an operation that diverges the execution flow, such as jumps, branches, calls, returns, etc. The tail of the group is the last instruction executed by the program, or an instruction that causes flow divergence. Except for the last instruction, all other instructions of the group cannot divert the flow of control.

A group of instructions is not the same as a *basic block*. Instead, groups can be viewed as the dynamic manifestation of basic blocks. A basic block is computed during compilation and thus the targets of all static branches are known when the first and last instructions of a basic block are determined. A group of instructions may contain multiple targets of control flow that have not yet been discovered. Thus, instructions may belong to multiple groups as the control flow is discovered during the execution of the program. The groups found in programs instrumented with InstrGrind for this study contain, typically, six to seven instructions. Thus, grouping reduces the instrumentation overhead up to sevenfold.

Example 2. *Figure 3 shows how the instructions observed during the execution of function* `main`*, seen in Example 1, are divided into groups. This division happens while instructions are executed, not statically. When a group is discovered,*

Group Data Structure	Group Instrumentation

G1 {
  instrs = [0x4004c0..0x4004cf]
  execs_count
}
	0x4004c0: **G1.execs_count++**
	  sub $0x8,%rsp
	0x4004c4: mov 0x8(%rsi),%rdi
	0x4004c8: mov $0xa,%edx
	0x4004cd: xor %esi,%esi
	0x4004cf: callq 0x400490 <strtol>

G2 {
  instrs = [0x4004d4..0x4004dc]
  execs_count
}
	0x4004d4: **G2.execs_count++**
	  test %eax,%eax
	0x4004d6: mov %eax,0x200b64(%rip)
	0x4004dc: jle 0x400509

G3 {
  instrs = [0x4004de..0x4004f0]
  execs_count
}
	0x4004de: **G3.execs_count++**
	  mov %eax,%esi
	0x4004e0: lea -0x1(%rsi),%eax
	0x4004e3: mov $0x400680,%edi
	0x4004e8: mov %eax,0x200b52(%rip)
	0x4004ee: xor %eax,%eax
	0x4004f0: callq 0x400470 <printf>

G4 {
  instrs = [0x4004f5..0x4004fa]
  execs_count
}
	0x4004f5: **G4.execs_count++**
	  mov $0x1,%edi
	0x4004fa: callq 0x4004a0 <sleep>

G5 {
  instrs = [0x4004ff..0x400507]
  execs_count
}
	0x4004ff: **G5.execs_count++**
	  mov 0x200b3b(%rip),%esi
	0x400505: test %esi,%esi
	0x400507: jg 0x4004e0

G6 {
  instrs = [0x4004e0..0x4004f0]
  execs_count
}
	0x4004e0: **G6.execs_count++**
	  lea -0x1(%rsi),%eax
	0x4004e3: mov $0x400680,%edi
	0x4004e8: mov %eax,0x200b52(%rip)
	0x4004ee: xor %eax,%eax
	0x4004f0: callq 0x400470 <printf>

G7 {
  instrs = [0x400509..0x40050f]
  execs_count
}
	0x400509: **G7.execs_count++**
	  xor %eax,%eax
	0x40050b: add $0x8,%rsp
	0x40050f: retq

Figure 3: (Left) data-structures used to represent groups of instructions: a list of instruction addresses, plus a counter. (Right) dynamic instrumentation, in boldface, that precedes the execution of a group of instructions.

an auxiliary structure is created for it, as seen in the left column of Figure 3. This structure holds a counter, incremented via binary instrumentation, as seen in the right column of Figure 3 in boldface. Thus, whenever a group of instructions is visited by the program flow, its execution counter is incremented by one. An instruction may belong to more than one group. For instance, the operation at address `@0x4004e0` *belongs to groups G3 and G6.*

4.2. Classifying Instructions

Instructions are classified into visible or invisible using as reference their location in memory. In order to classify instructions, InstrGrind uses a mapping that associates object files with the range of addresses that they cover, once loaded into memory. A range of addresses is defined by a base address and a size. As noted in Definition 1, instructions located in ranges belonging to source files are classified as visible and all other instructions are classified as invisible. Example 3 shows how this classification works in practice.

```
1 /usr/lib64/libc-2.17.so:0x4c459a0:1376079
2 /usr/lib/valgrind/vgpreload_core-amd64-linux.so:0x4a24580:568
3 /usr/lib/valgrind/instrgrind-amd64-linux:0x58000150:1764106
4 /usr/lib64/ld-2.17.so:0x4000ad0:111936
5 /home/user/countdown:0x4004c0:402
```

Figure 4: Files mapping onto memory with range addresses specified as base address and size.

Example 3. *Figure 4 shows the mapping that* InstrGrind *creates to analyze the execution of the* `countdown` *program seen in Example 1. Visible instructions start at the address* `0x4004c0`*, and cover the next 402 bytes (Line 5). Instructions that exist in* valgrind*'s address space (Lines 2-3 of Fig. 4) are not counted because they are not part of the normal execution of the program. Instructions from* `libc` *(Line 1) and from* `ld` *(Line 4) are classified as invisible.*

In some large computer applications part of the application code, which must be classified as visible, is offloaded to shared libraries. In such cases, an analyst must identify which address ranges in memory corresponds to visible or invisible instructions. For all the benchmarks analyzed in this study, the determination of visible and invisible address ranges is automatic because the files that compose the compiled program originate the instructions marked as visible.

5. Measuring Visibility Ratio

This section answers three research questions:

- **RQ1**: What is the visibility ratio of SPEC CPU2017's execution with reference inputs?
- **RQ2**: What is the visibility ratio in GNU CoreUtils' execution with its standard test inputs?
- **RQ3**: How code optimizations impact these ratios?
- **RQ4**: Is there a positive correlation between visibility ratio and the speedup achieved by compiler optimizations?

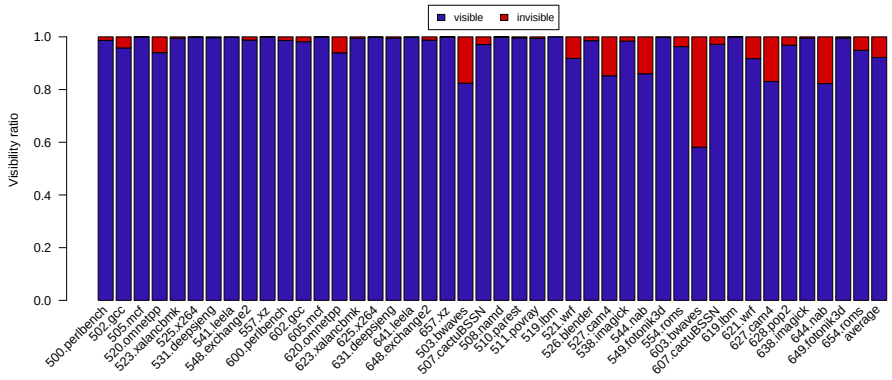

Figure 5: Visibility ratio of SPEC CPU2017 programs compiled using GCC with -O0, running with reference inputs.

All the experiments were executed on a 16-core Intel(R) Xeon(R) E5-2630 at 2.40GHz with 16GB of RAM running Linux CentOS 7.5. Programs are compiled with GCC v7.3.1.

5.1. RQ1: Visibility Ratio in SPEC CPU2017

Figure 5 shows the visibility ratio of instructions executed for the SPEC CPU2017 benchmark suite with reference inputs. Each benchmark was compiled with -O0 and executed once using InstrGrind with its reference input. This experiment took approximately 9 days to complete for the entire SPEC CPU2017 suite.

Most of the instructions observed by InstrGrind are visible. For the 43 programs available in SPEC CPU2017, 37 of them had visibility ratio higher

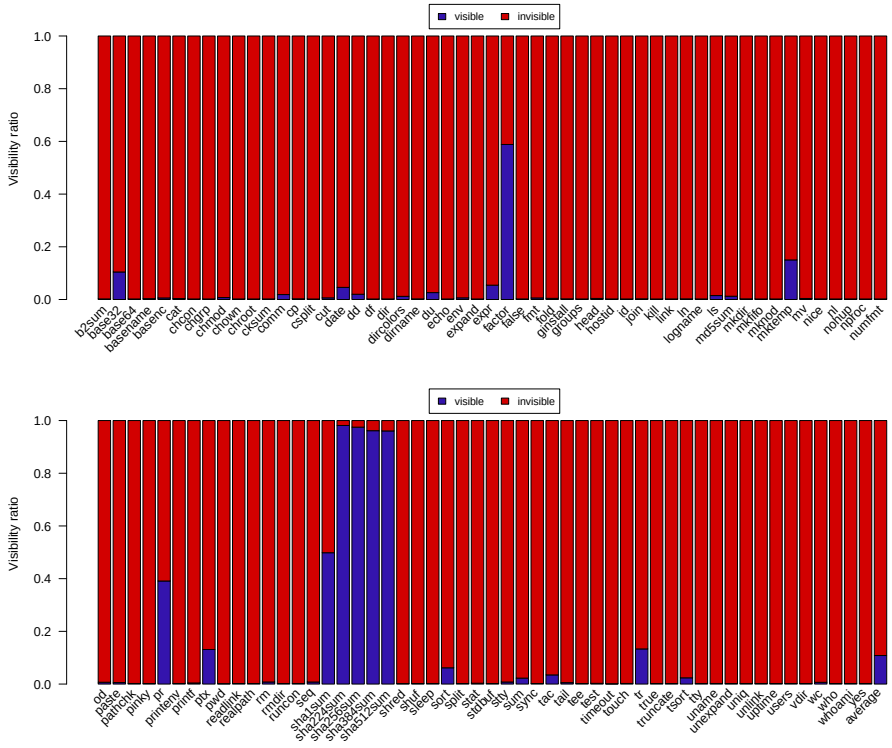

Figure 6: Visibility ratio of GNU CoreUtils programs compiled using gcc with -O0.

than 0.9; 5 more of them higher than 0.8. The only divergence was `603.bwaves`, with a visibility ratio of about 0.6. The average visibility ratio reported in the charts is computed by summing the number of visible instructions over all the benchmark executions and dividing by the total number of instructions executed by all the benchmarks in the suite. For SPEC CPU2017 this average is 0.922.

Most of the instructions observed in a typical run of SPEC CPU2017 are *visible*. This result confirms our expectations given that SPEC CPU2017 focuses on CPU performance and the curation of the benchmark suite must ensure that the entire code can be distributed under a SPEC license agreement. Thus, benchmark authors tend to avoid extensive library usage for a number of reasons, two of which standout: predictability and licensing. Predictability tends to hinder library usage because libraries may change without the benchmark developer having any control over their evolution. Licensing issues, in turn, might

prevent the integration of a library into a benchmark that is not open-source.

5.2. RQ2: Visibility Ratio in GNU CoreUtils

Figure 6 shows the visibility ratio observed during the execution of GNU CoreUtils with the standard inputs available in its test suite. All programs were compiled with -O0. This test suite was modified to invoke each program, with its reference input, using InstrGrind. The entire suite was executed once. The numbers that follow represent the results for 105 of the 106 programs available in GNU CoreUtils. The exception is the *[* (open bracket) program that was not exercised by the GNU CoreUtils' test suite. The average visibility ratio is 0.108. A few calculation-oriented programs, such as `factor` and the cryptographic routines, break this tendency, as Figure 6 shows. However, these programs are exceptions within GNU CoreUtils. The conclusion of this experiment is that instructions processed in a typical run of GNU CoreUtils' programs are, in general, invisible.

5.3. RQ3: The Impact of Compiler Optimizations

When producing code for a program P, a compiler has an effect only on the source code of P; that is, on its visible instructions (Def. 1). Hence, it is natural to assume that the larger the visibility ratio of P, averaged across different executions, the larger the impact of compiler optimizations on P. Figures 7 and 8 confirm this intuition.

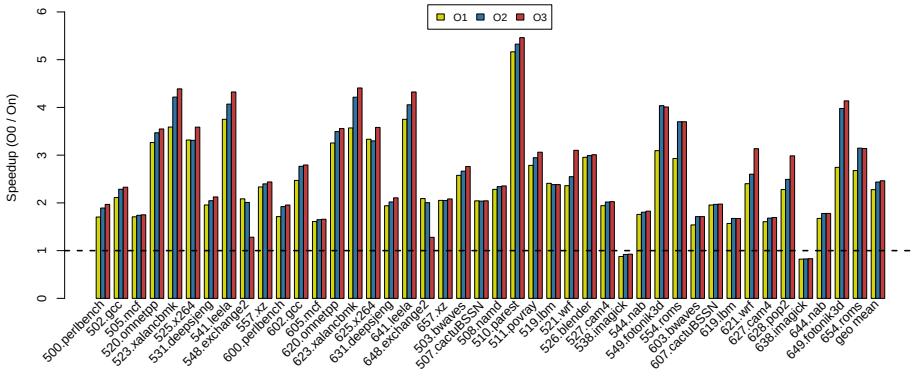

Figure 7: Speedup produced by different optimization levels of GCC, when applied onto the SPEC CPU2017 programs.

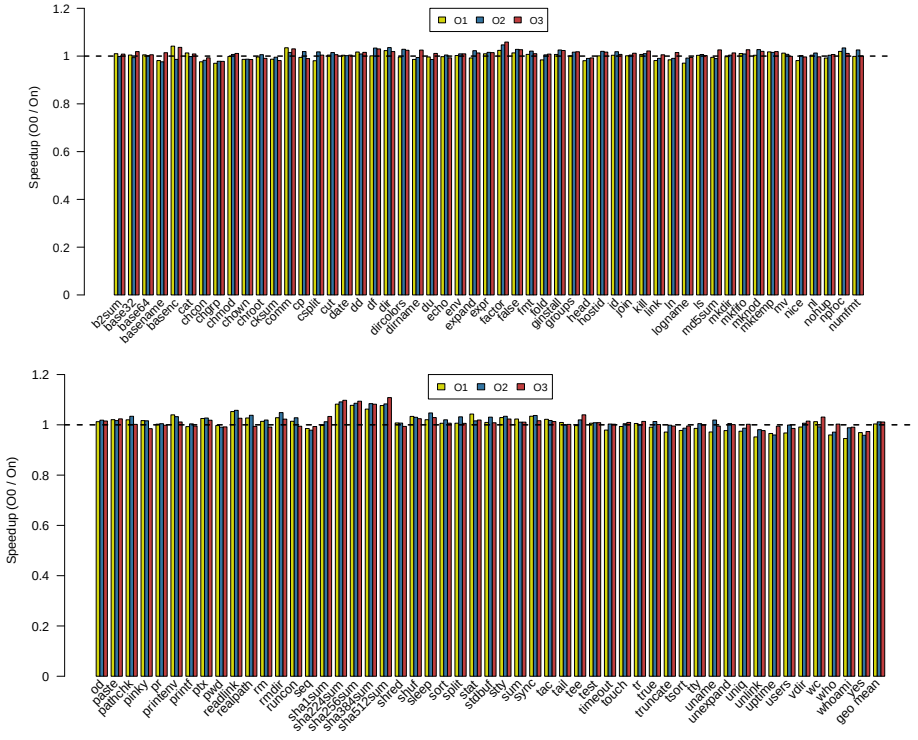

Figure 8: Speedup produced by different optimization levels of GCC, when applied onto the GNU CoreUtils programs.

Figure 7 shows the speedups obtained by the different optimization levels of GCC when applied onto SPEC CPU2017. Results are the average of three executions. Deviations are insignificant; thus, not reported. In total, one run of all unoptimized programs for SPEC CPU2017 takes approximately 45 hours. The following average speedups (geometric mean) are observed for SPEC CPU2017: $-O1 = 2.277$x, $-O2 = 2.437$x and $-O3 = 2.463$x.

Figure 8 shows similar data, this time considering the programs from GNU CoreUtils. The test suite was invoked 25 times, each invocation taking approximately 4 minutes. Deviations are, again, negligible; hence, omitted. Speedups are too low to be considered statistically significant[1]. On average

[1]Because the runtime differences produced by the three optimization levels were small, this experiment averages 25 samples, in contrast to the one used to produce the data in Figure 7, which averages 3.

(geomean), GCC -O3 delivers a speedup of 1.011 across the GNU CoreUtils' programs. This last experiment indicates that the visibility ratio strongly determines the effect of compiler optimizations onto programs.

Visibility ratios vary in face of different optimization levels; however, variations tend to be small. Figures 10 and 11 show the visibility ratio for optimized versions of the SPEC CPU2017's and GNU CoreUtils' programs, respectively. The optimized version (compiled with -O2) has less visible instructions than the non-optimized (compiled with -O0): a visibility ratio of 0.701 against 0.108. However, this difference is small to be of consequence. Optimizations bear a more noticeable effect onto SPEC CPU2017. Programs optimized with -O2 show a visibility ratio of 0.701, against 0.922 in the unoptimized programs. This difference is natural: SPEC CPU2017 has a larger code base, and a much larger visibility ratio than GNU CoreUtils; therefore, the compiler has more opportunities to optimize code.

5.4. RQ4: Correlation Between Visibility Ratio and Optimization Speedup

Intuitively, the more visible instructions a program contains, the more opportunities a compiler has to optimize its code. However, it is well-known that the same sequence of compiler optimizations produce very distinct effects on different programs. As an example, Silva *et al.* have recently shown that the benefits brought by the optimization levels of LLVM vary widely across programs [20]. Therefore, there is no expectation that higher visibility ratio causes higher optimization benefits. However, is there a correlation between visibility ratio and optimization speedup for the benchmarks and compilers in this study?

Figure 9 provides a visual comparison between the two measurements of interest. The vertical axis shows the visibility ratio (Definition 2) of the 148 benchmarks studied in this paper. The horizontal axis shows the speedup observed for the same set of benchmarks, when compiled using GCC with -O2. Speedup is measured as the running time of the unoptimized program divided by the running time of its optimized version.[2] Thus, the greater the benefit

[2]This section analyzes the effects of GCC with -O2; however, the same conclusions hold

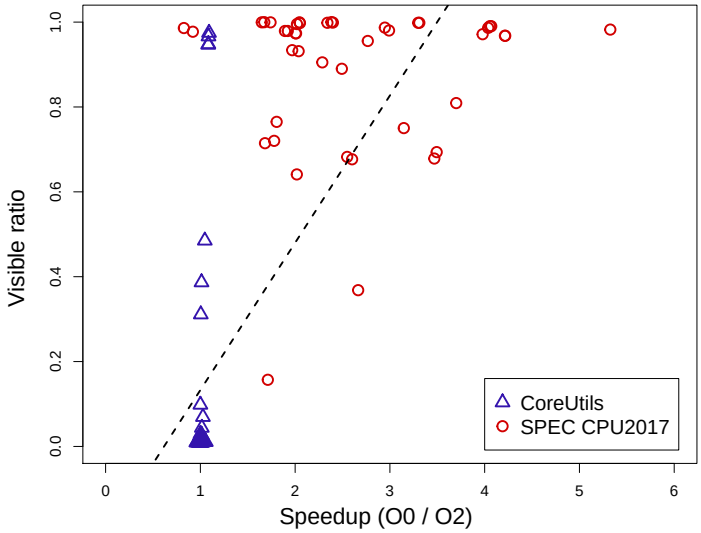

Figure 9: Scatter plot that relates the visibility ratio and the speedup produced by gcc with -O2 for the benchmarks considered in this paper.

of using GCC with -O2 over a program, the higher the value of the speedup. Figure 9 also shows a dashed regression line that relates visibility ratios and speedups.

Table 1 reports three different types of correlation coefficients produced from the data in Figure 9. Pearson correlation coefficient measures how linearly related are the visibility ratios and speedups: the closer to 1.0 the stronger is the linear relation between the two measurements. We report p-values in Table 1: the closer to zero, the more unlikely that the correlation results have been observed by chance. However, this statistical test, at least when used with Pearson's coefficient, assumes that the data is bivariate normal [21]. The speedups that we observe after optimizing the programs failed the Shapiro-Wilk [22] Normality Test; hence, they cannot be considered to come from a normal distribution. Nevertheless, Pearson's R^2 remains a consistent estimator of correlation between populations. The two populations that we compare still

with the two other levels of optimizations tuned for speed, GCC -O1 and GCC -O3. According to Figures 7 and 8, speedups produced by the different levels present similar geometric means, with -O2's gains being higher than -O1's, and slightly lower than -O3's.

present finite variances and finite covariance [23]. Therefore, we report Pearson's coefficient, plus two other measures of correlation.

The two other coefficients in Table 1, Spearman's and Kendall's, measure *rank correlation*, that is, the tendency that if one of the variables grows, the other will grow too. Spearman's rank gauges how well the two measurements can be related by a monotonic function. Kendall's rank measures the similarity of the orderings of the sequences formed by the two populations. In the case of the rank coefficients, we would observe 1.0 if for any two benchmarks, b_i and b_j, we have that $\mathit{VisibilityRatio}(b_i) > \mathit{VisibilityRatio}(b_j) \Rightarrow \mathit{Speedup}(b_i) > \mathit{Speedup}(b_j)$.

Suite	#Progs	Pearson	Spearman	pv	Kendall	pv	Shapiro-Wilk
All	148	0.74	0.83	≈ 0	0.73	≈ 0	≈ 0
CUtils	105	0.63	0.77	≈ 0	0.74	≈ 0	0.00013
Spec	43	0.10	-0.05	0.73	-0.03	0.79	0.02874

Table 1: Correlation coefficients between visibility ratio and optimization speedup observed within different benchmark suites. The p-value of the correlation test, pv, checks the alternative hypothesis that the coefficient of determination is zero. The lower is this value, the less likely is the possibility that the observed correlation results from chance. Shapiro-Wilk is a normality test performed on the population of speedups. Results below 0.1 are not considered to come from a normal distribution in most statistical studies [24]. Normality is an assumption in Pearson's test of significance [25]; hence, we do not report its p-value.

Table 1 shows that for the set formed by the programs in SPEC CPU2017 and in GNU CoreUtils, there is a statistically significant correlation between visibility ratio and the speedups achieved after optimization. This observation also holds for the programs in GNU CoreUtils alone. Regardless of the coefficient considered, the p-value returned by any of the correlation tests is close to zero for the ensemble of all the benchmarks and for GNU CoreUtils alone. However, for the programs in SPEC CPU2017, when considered alone, there is no statically significant correlation between visibility ratio and speedups. There are two reasons for this effect. First, the number of benchmarks considered in this case is smaller: correlation is measured in 43 data points. However, the most important reason to explain the absence of strong correlation is the fact that compiler optimizations act upon programs in heterogeneous ways. This observation has already been made in previous work, and is the core motivation behind the search for customizable optimization sequences [20, 26, 27, 28].

Example 4 illustrates this issue.

Example 4. *In* SPEC CPU2017, *`603.bwaves` has the lowest visibility ratio but does not result in the worst speedups when optimized. Similarly, high visibility ratios (e.g.,* `508.namd` *and* `510.parest`*) do not result in the highest speedups. A benchmark might have 80% of the code visible; however, its visible part might not be suitable for compiler improvements. In another benchmark, in turn, the 40% visible part may contain code that is suitable for performance-relevant code transformations. Therefore, although the compiler can only improve visible code, the nature of said improvement still depends on how suitable this code is for the transformations available in the compiler.*

Therefore, the visibility ratio of a program running with a certain workload does not bring enough information — if considered alone — to predict accurately the effect that compiler optimizations have on the program. Nevertheless, the visibility ratio is one of the factors that determines the impact of these optimizations, as the high degrees of correlations reported in Table 1 indicate. Following Amdahl's Law, the visibility ratio can be effectively used to estimate the limits of potential benefits for future compiler optimizations.

5.5. Reflection on the Results

Are these visibility ratios an experimental confirmation of the expectation of compiler developers or are they new information for some of them? An informal consultation from July of 2018 indicated that there was no consensus about visibility. At that time we asked ten researchers what proportion of a program in SPEC CPU2006 is visible[3]. These engineers and academics have experience with compilers: four of them have been part of CGO's program committee. The other six were professionals actively working with the following compilers, within different companies: LLVM (Apple, Facebook and ARM), Visual Studio (Microsoft), JavaScript Core (STMicroelectronics), and V8 (Google). We quote

[3]We chose SPEC CPU2006 because this suite was more well-known than SPEC CPU2017 at the time when the questions were sent.

below the e-mail sent to the different researchers and professionals, asking them about their feeling on the proportion of invisible instructions:

> We are working on a project to count the number of "visible" and "invisible instructions" in a program. A visible instruction is, for instance, the instruction that we can see in the LLVM IR. An invisible instruction is an operation that we cannot see, for instance, instructions inserted to implement the ABI, or that are part of some external library. We have already counted this ratio in several benchmarks. I've decided to write to the compiler experts that I know, asking them what is the proportion between:
>
> - **VS:** Stores in the source code of a C program, and
> - **TS:** all the stores produced during the execution of the program.

Answers came out all over the spectrum. The average was $\mathbf{VS}/\mathbf{TS} = 0.575$. This value is substantially lower than the average (arithmetic mean) observed for SPEC CPU2017: 0.92. Variance in the answers was high: the standard deviation was 0.31. Only two researchers gave us answers greater than 0.9. Two answers were less than 0.3. The highest ratio someone guessed was 0.95; the lowest was 0.2.

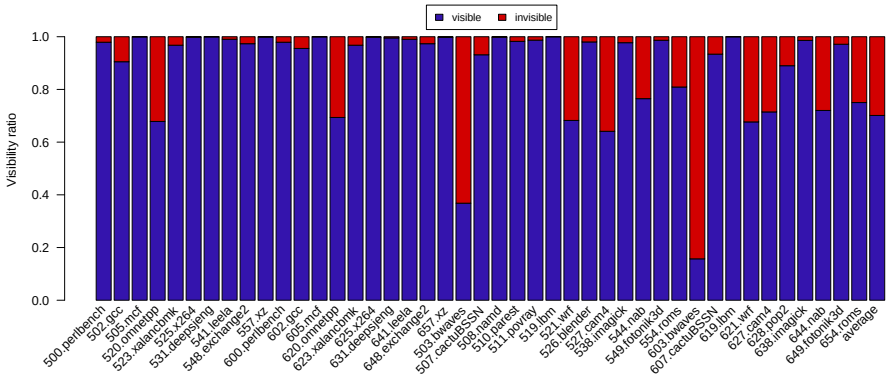

Figure 10: Visibility ratio of SPEC CPU2017 programs compiled using GCC with -O2, running with reference inputs.

6. Related Work

The categorization of dynamic instances of program instructions is not a novel endeavor. In their classic textbook, Patterson and Hennessy's discuss the dynamic count of opcodes executed for the twelve integer benchmarks in

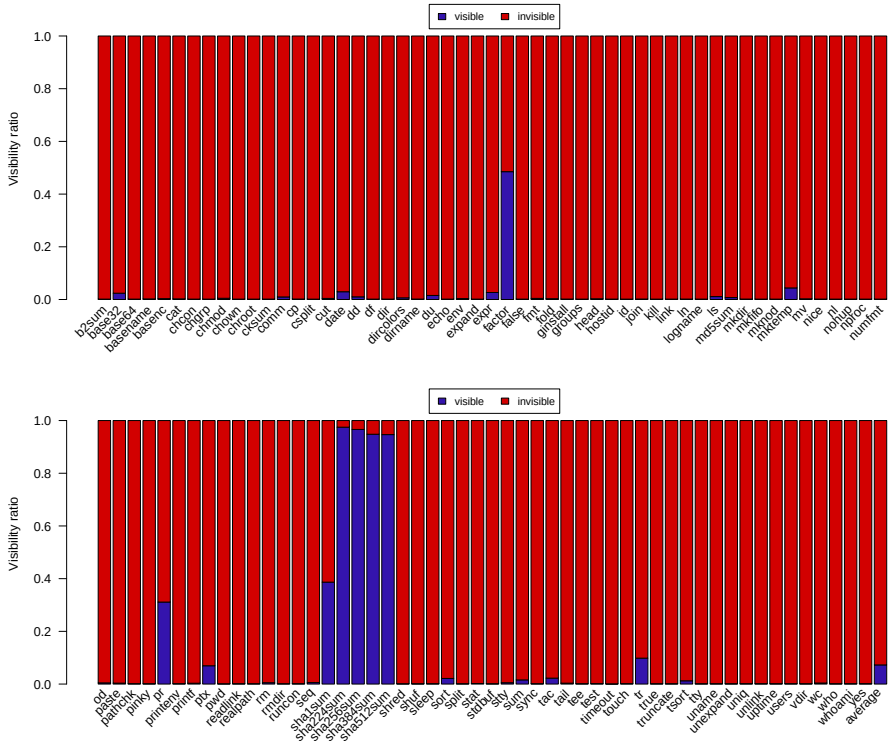

Figure 11: Visibility ratio of GNU CoreUtils programs compiled using gcc with -O2.

the SPEC CPU2006 benchmark suite [4]. Similarly, Guthaus et al. present a comprehensive analysis of the distribution of instructions across MiBench and in the SPEC CPU2000 benchmark suite [29]. In this analysis, instructions are split into four major types: control, memory, integer arithmetics and floating point arithmetics. However, previous work has not distinguished visible from invisible instructions.

Historically, SPEC CPU has been the subject of several statistical studies [6, 5, 2]. Since the release of SPEC CPU2017, numerous research groups have already performed analyses on it [30, 31, 32, 33, 34]. These studies focus on some particular aspect of that benchmark suite, such as potential redundancies [32], code coverage [31, 34] or memory characterization [33]. None of these papers analyze the visibility ratio in SPEC CPU2017, nor the impact of compiler optimizations on these benchmarks. Amaral et al. curated the

Alberta Workloads for the SPEC CPU2017 Benchmark Suite [35]. Based on their analysis of how program behavior varies with workloads, the expectation is that the visibility ratios will not change significantly when the benchmarks are run with different inputs.

Leobas *et al.* [36] classify an instruction as visible if it corresponds directly to operations in the LLVM representation of the program. Instructions inserted by the compiler, such as loads and stores used to spill variables, are classified as invisible. Their goal is to estimate the overhead caused by the code that the compiler creates outside of a programmer's control. Their definition is in contrast to Definition 1, in which the category of an instruction depends on its location in memory. Differently than the work of Leobas *et al.*, this paper supports the assessment of the impact that compiler transformations might have on programs.

7. Conclusion

This paper measured the visibility ratio of instructions executed by benchmarks in SPEC CPU2017, when run with reference inputs on a Linux-based processor with the x86 architecture. This study reveals that most of the executed instructions are visible — they originate from code available to the compiler. In contrast, most of the instructions executed in a typical run of programs in GNU CoreUtils are invisible. The origin of invisible instructions are libraries that these applications invoke during their execution. These findings confirm the intuition of some developers but will be surprising to others. Visibility measurements are important because architectures and compilers are often evaluated with SPEC CPU2017, which shows high visibility ratio; however, they are applied onto applications like GNU CoreUtils, which have low visibility ratio.

The ability to estimate the visible part of a program is important, because visibility enables a variation of Amdahl's Law for compilers. Although Amdahl's Law was formulated in the context of parallel processing, the same idea applies to an optimizing compiler: optimizations can only enhance code that

they can change. In other words, visibility limits the amount of improvement that compiler-based code transformations can have on programs. Therefore, by knowing what is the visible portion of the executed code, developers and compiler engineers can better estimate the benefits of compiler optimizations on that code.

Software:. InstrGrind is available at `https://github.com/rimsa/instrgrind` under the GNU GPL 2.0 license. Readers are encouraged to use it to measure visibility in other architectures, benchmarks, compilers or input sets.

Acknowledgement

Fernando Pereira has been supported by CNPq (Grant 406377/2018-9) and FAPEMIG (Grant PPM-00333-18). While visiting UFMG, Nelson received a scholarship from CAPES (Edital CAPES PrInt).

References

[1] K. M. Dixit, Overview of the SPEC benchmarks, in: J. Gray (Ed.), The Benchmark Handbook for Database and Transaction Systems (2nd Edition), Morgan Kaufmann, 1993, pp. 489–521.

[2] A. Phansalkar, A. Joshi, L. K. John, Analysis of redundancy and application balance in the spec cpu2006 benchmark suite, in: ISCA, ACM, New York, NY, USA, 2007, p. 412–423. doi:10.1145/1250662.1250713.

[3] K. M. Dixit, The SPEC benchmarks, Parallel Computing 17 (10-11) (1991) 1195–1209. doi:10.1016/S0167-8191(05)80033-X.

[4] D. A. Patterson, J. L. Hennessy, Computer Organization and Design MIPS Edition: The Hardware/Software Interface, Newnes, 2013.

[5] A. Phansalkar, A. Joshi, L. K. John, Subsetting the SPEC CPU2006 benchmark suite, SIGARCH Comput. Archit. News 35 (1) (2007) 69–76. doi:10.1145/1241601.1241616.

[6] A. A. Nair, L. K. John, Simulation points for SPEC CPU 2006, in: ICCD, IEEE, 2008, pp. 397–403.

[7] A. J. KleinOsowski, D. J. Lilja, MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research, IEEE Computer Architecture Letters 1 (1) (2002) 7–7.

[8] N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, in: PLDI, ACM, 2007, pp. 89–100. doi:10.1145/1250734.1250746.

[9] A. Rimsa, J. N. Amaral, F. M. Quintão, Efficient and precise dynamic construction of control flow graphs, in: SBLP, Association for Computing Machinery, New York, NY, USA, 2019, p. 19–26. doi:10.1145/3355378.3355383.

[10] A. Rimsa, J. Nelson Amaral, F. M. Q. Pereira, Practical dynamic reconstruction of control flow graphs, Software: Practice and Experience 51 (2) (2021) 353–384. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2907, doi:https://doi.org/10.1002/spe.2907.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2907

[11] S. L. Graham, P. B. Kessler, M. K. Mckusick, Gprof: A call graph execution profiler, in: CC, Association for Computing Machinery, New York, NY, USA, 1982, p. 120–126. doi:10.1145/800230.806987.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, Pin: Building customized program analysis tools with dynamic instrumentation, in: PLDI, ACM, New York, NY, USA, 2005, p. 190–200.

[13] F. Bellard, QEMU, a fast and portable dynamic translator, in: ATEC, USENIX, USA, 2005, p. 41.

[14] D. Bruening, T. Garnett, S. Amarasinghe, An infrastructure for adaptive dynamic optimization, in: CGO, IEEE Computer Society, USA, 2003, p. 265–275.

[15] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C. Luk, G. Lyons, H. Patil, A. Tal, Analyzing parallel programs with pin, Computer 43 (3) (2010) 34–41.

[16] F. Gruber, M. Selva, D. Sampaio, C. Guillon, A. Moynault, L.-N. Pouchet, F. Rastello, Data-flow/dependence profiling for structured transformations, in: Principles and Practice of Parallel Programming (PPoPP, Washington, DC, 2019, pp. 173–185.

[17] J. Seward, N. Nethercote, Using valgrind to detect undefined value errors with bit-precision, in: Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05, USENIX Association, USA, 2005, p. 2.

[18] D. Bruening, Q. Zhao, Practical memory checking with dr. memory, in: Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '11, IEEE Computer Society, USA, 2011, p. 213–223.

[19] V. Kiriansky, D. Bruening, S. P. Amarasinghe, Secure execution via program shepherding, in: Proceedings of the 11th USENIX Security Symposium, USENIX Association, USA, 2002, p. 191–206.

[20] A. F. d. Silva, B. N. B. de Lima, F. M. Q. a. Pereira, Exploring the space of optimization sequences for code-size reduction: Insights and tools, in: International Conference on Compiler Construction, Association for Computing Machinery, New York, NY, USA, 2021, p. 47–58. doi:10.1145/3446804.3446849.

[21] R. Newson, Parameters behind "nonparametric" statistics: Kendall's tau, Somers' D and median differences, Stata Journal 2 (1) (2002) 45–64.

[22] S. S. Shapiro, M. Wilk, An analysis of variance test for normality (complete samples), Biometrika 52 (3–4) (1965) 591–611. doi:doi:10.1093/biomet/52.3-4.591.

[23] P. Schober, C. Boer, L. A. Schwarte, Correlation coefficients: Appropriate use and interpretation, Anesthesia and Analgesia 126 (5) (2018) 1763–1768.

[24] P. Royston, Algorithm AS 181: The (W) test for normality, Journal of Applied Statistics 31 (1) (1982) 176–180. doi:10.2307/2347986.

[25] R. A. Armstrong, Should pearson's correlation coefficient be avoided?, Ophthalmic and Physiological Optics (2019) 1–12doi:https://doi.org/10.1111/opo.12636.

[26] S. Jain, U. Bora, P. Kumar, V. B. Sinha, S. Purini, R. Upadrasta, An analysis of executable size reduction by llvm passes, CSIT 7 (1) (2019) 105–110. doi:https://doi.org/10.1007/s40012-019-00248-5.

[27] S. Purini, L. Jain, Finding good optimization sequences covering program space, ACM Trans. Archit. Code Optim. 9 (4). doi:10.1145/2400682.2400715.

[28] A. F. da Silva, B. C. Kind, J. W. de Souza Magalhães, J. N. Rocha, B. C. F. Guimarães, F. M. Q. Pereira, AnghaBench: A suite with one million compilable C benchmarks for code-size reduction, in: CGO, IEEE, New York, NY, USA, 2021, pp. 378–390. doi:10.1109/CGO51591.2021.9370322.

[29] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown, Mibench: A free, commercially representative embedded benchmark suite, in: WWC-4, IEEE, 2001, pp. 3–14.

[30] J. Bucek, K.-D. Lange, J. v. Kistowski, SPEC CPU2017: Next-generation compute benchmark, in: ICPE, ACM, New York, NY, USA, 2018, p. 41–42. doi:10.1145/3185768.3185771.

[31] A. Limaye, T. Adegbija, A workload characterization of the SPEC CPU2017 benchmark suite, in: ISPASS, IEEE, New York, NY, USA, 2018, pp. 149–158. doi:10.1109/ISPASS.2018.00028.

[32] R. Panda, S. Song, J. Dean, L. K. John, Wait of a decade: Did SPEC CPU 2017 broaden the performance horizon?, in: HPCA, IEEE, New York, NY, USA, 2018, pp. 271–282.

[33] S. Singh, M. Awasthi, Memory centric characterization and analysis of spec cpu2017 suite, in: ICPE, ACM, New York, NY, USA, 2019, p. 285–292. doi:10.1145/3297663.3310311.

[34] Q. Wu, S. Flolid, S. Song, J. Deng, L. K. John, Hot regions in SPEC CPU2017, in: IISWC, IEEE, New York, NY, USA, 2018, pp. 71–77, invited Paper for the Hot Workloads Special Session. doi:10.1109/IISWC.2018.8573479.

[35] J. N. Amaral, E. Borin, D. Ashley, C. Benedicto, E. Colp, J. H. S. Hoffman, M. Karpoff, E. Ochoa, M. Redshaw, R. E. Rodrigues, The alberta workloads for the SPEC CPU 2017 benchmark suite, in: ISPASS, Belfast, Northern Ireland, 2018, pp. 159–168.

[36] G. V. Leobas, B. C. F. Guimarães, F. M. Q. Pereira, More than meets the eye: Invisible instructions, in: SBLP, Association for Computing Machinery, New York, NY, USA, 2018, p. 27–34. doi:10.1145/3264637.3264641.