Finding Missed Compiler Optimizations by Differential Testing (preprint)

Gergö Barany
Inria Paris
France
gergo.barany@inria.fr

Abstract

Randomized differential testing of compilers has had great success in finding compiler crashes and silent miscompilations. In this paper we investigate whether we can use similar techniques to improve not the correctness but the quality of the generated code: Can we compare the output of different compilers to find optimizations performed by one but missed by another?

We have developed a set of tools for running such tests. We compile C code generated by standard random program generators and use a custom binary analysis tool to compare the output programs. Depending on the optimization of interest, the tool can be configured to compare features such as the number of total instructions, multiply or divide instructions, function calls, stack accesses, and more. A standard test case reduction tool produces minimal examples once an interesting difference has been found.

We have used our tools to compare the code generated by GCC, Clang, and CompCert. We have found previously unreported missing arithmetic optimizations in all three compilers, as well as individual cases of unnecessary register spilling, missed opportunities for register coalescing, dead stores, redundant computations, and missing instruction selection patterns.

Keywords optimization, differential testing, randomized testing

1 Motivation

Over the last few years, randomized differential testing has become a standard method for testing programming tools. Hundreds of compiler bugs, both compiler crashes and silent generation of incorrect code, have been found by random testing in many mature compilers for various programming languages [Eide and Regehr 2008; Lidbury et al. 2015; Midtgaard et al. 2017; Yang et al. 2011]. The method can be similarly effective for finding crashes and correctness bugs in static analyzers [Cuoq et al. 2012].

In this approach, randomly generated programs (satisfying certain correctness properties) are compiled with different compilers, executed, and their results compared. Different results for well-defined programs imply a bug in at least one of the compilers. Such interesting input programs can be reduced by a test case minimizer to often just a few lines of code that provoke the bug.

Unpublished preprint, 2017.

```
int fn1(int p, int q) {
  return q + (p % 6) / 9;
}
```

Figure 1. Missed optimization example: `(p % 6) / 9` is always zero. Clang used to be unable to eliminate this computation.

Given the success of this method in finding bugs, we wanted to test whether it can also be used to find missed optimizations: Is it possible to build a system that finds minimal examples of missed optimizations by compiling random C programs, comparing the generated code, and minimizing the input program?

The answer is yes. Figure 1 shows a small example of an arithmetic optimization missed by Clang found by our tool (with small manual modifications). The value of the expression `(p % 6)` is always in the interval $[-5, 5]$. Dividing any number in this interval by 9, with truncation toward 0 as prescribed by the C standard, gives a value of 0 for any value of `p`. Therefore GCC generates code to simply copy `q` to the return register and return. Until a recent fix motivated by our findings, Clang did not perform this arithmetic simplification and generated code to evaluate the modulo and divide operations.

In this paper we describe the following contributions:

- a differential testing methodology for finding missed optimizations in C compilers;
- a set of configurable tools for automating the method; and
- the experimental evaluation of the method and tools, with examples of previously unreported missed optimizations of various kinds in all three C compilers we tested.

In addition to the examples throughout this paper, an evolving list of previously unreported missed optimizations we found is available online at https://github.com/gergo-/missed-optimizations. We have filed reports or patches against all compilers tested, and some issues have been fixed.

2 Random differential compiler testing

When searching for miscompilations, randomized differential compiler testing proceeds by generating random programs, compiling them with different compilers, executing them, and comparing the results. In order to make these results comparable, the random programs must fulfill certain criteria: They must be complete programs that avoid invoking any undefined behaviors in the target language, as well as (for C and similar languages) implementation-defined features such as the order of evaluation of expressions.

In order to make results comparable, Csmith [Yang et al. 2011] generates programs that compute a checksum over the final values of all global variables and prints it to the standard output before exiting; Orange3 [Nagai et al. 2014] precomputes expected results during generation and adds corresponding conditional print statements to the program. In either case, the textual output of the test program compiled with different compilers can easily be compared by a driver script. Any difference indicates a miscompilation bug in at least one of the compilers under test. Other random generators such as CCG [Balestrat 2016] or ldrgen [Barany 2017] do not generate complete programs that are meant to be executed; they can still be useful for finding compiler crashes.

The generators mentioned above all target the C programming language, but similar tools exist for other languages, such as CLsmith for OpenCL [Lidbury et al. 2015], jsfunfuzz for JavaScript [Ruderman 2015], or efftester for OCaml [Midtgaard et al. 2017].

Having found the symptoms of a compiler bug, an important final phase is the reduction of the input program to a minimal example showing the bug. Automatically generated programs typically contain hundreds or thousands of lines of code. Finding the cause of a bug in such a large test case is tedious and often unrealistic. Most bugs also don't need large triggers: In a large-scale study of reported compiler bugs, Sun et al. [2016] report that test cases for compiler bugs 'are typically small, with 80 % having fewer than 45 lines of code'.

To help identify the actual bug, test case reducers simplify the input program by removing parts or by other transformations such as replacing variables by constants. C-Reduce [Regehr et al. 2012] is a generic reducer for C family programs that is run with a user-provided 'interestingness test'. This test is a program that returns 0 if the current version of the program is still 'interesting' (i. e., shows the buggy behavior one is investigating) and some other value otherwise. For finding miscompilations, the test will typically compile the program with the compilers under test and see if the resulting binaries still show different behavior; for finding compiler crashes, the test will check if the compiler still crashes. Starting from an initial program, C-Reduce explores a search space of incrementally reduced programs that are interesting according to the test provided. It stops when it has reached a minimal program that it cannot reduce further.

The Orange3 random generator includes its own specialized reduction mode. Both of these reducers work well in practice. They often produce very small examples, at times with further opportunities for manual cleanup.

```
char fn2(float p) {
  if (isfinite(p) &&
      CHAR_MIN <= p && p <= CHAR_MAX) {
    return (char) p;
  }
  return 0;
}
```

(a) Example input for useless spilling by GCC. Guards added manually for well-defined behavior on all inputs.

```
vcvt.u32.f32 s15, s0   ; float -> unsigned int
sub sp, sp, #8         ; allocate stack frame
vstr.32 s15, [sp, #4]  ; spill float reg
ldrb r0, [sp, #4]      ; reload to int reg
add sp, sp, #8         ; free stack frame
```

(b) Annotated ARM code generated by GCC for the `float` to `char` conversion. Spilling is useless, the last four instructions could be replaced by a direct copy (`vmov r0, s15`).

Figure 2. Example of unnecessary spill code generation.

3 Searching for missed optimizations

Our goal is to exploit this existing differential testing infrastructure in novel ways to find missed optimizations.

Consider the case of inefficient register allocation, resulting in more spill code than is strictly needed for a given function. The amount of spill code can be computed for different compiled programs and can thus form the basis of an interestingness test. If we can start from a program on which some compiler generates useless spills, a reducer can then produce a corresponding program that is minimal but still shows a difference in spilling between compilers.

Figure 2 shows an example program found by our tool, then manually extended with guards against undefined behavior on overflow. The figure also shows part of the ARM assembly code[1] generated for this program by GCC. The conversion between floating-point and integer types is performed by the `vcvt` instruction, which puts its result into a floating-point register. This value must be copied to the integer return register `r0`. It can then be zero-extended from 8 bits to implement the truncation

[1] In all our ARM examples, we generate code for the ARMv7-A instruction set with VFPv3-D16 hardware floating-point extensions.

to `char`, which is unsigned on the target machine. GCC models the copy and zero extension by spilling an integer word to the stack and reloading a byte, whereas Clang and CompCert simply generate a register-to-register copy instruction. (CompCert then zero-extends. Clang does not, as the guards ensure that the value is already in the correct range.) We have reported this issue; a GCC developer found that the spill is chosen because it is not marked as having higher cost than a zero-extension pattern with a direct copy.

Our research hypothesis was that this process can (a) be seeded efficiently with randomly generated programs, (b) be generalized to many kinds of missed optimizations, and (c) reduce programs to reasonably small examples that show clearly useless code that should be optimized away. In other differential testing approaches, compilers are used 'correctness oracles' for each other. Our goal was to use different compilers' outputs as 'optimization oracles'.

Our method thus works as follows:

1. generate a random C program
2. compile the program with different compilers to obtain different binaries
3. if binaries show an 'interesting' difference:
   - reduce to minimal program showing the same 'interesting' difference

We use randomly generated programs instead of open source software because random generators provide an unlimited supply of programs with characteristics of our choice: With or without floating-point operations, loops, branches, etc., and with tunable parameters such as the number of variables or maximum size of basic blocks.

For generating programs, we have experimented with the Csmith and ldrgen tools. By default, Csmith generates a complete application with a `main` function and typically several other functions. We use command line flags to request generation of a single function and suppress generation of `main`. We also disable generation of global variables because we have found that different compilers generate different code for loading the addresses of globals; these inessential differences introduced unnecessary complications when comparing binaries.

As Csmith marks all functions variables `static`, we post-process its output to remove this keyword, since otherwise the compiler would be free to generate an empty object file. In contrast, ldrgen always generates a single non-`static` function and at most a single global variable that has not caused us problems.

Both generators were useful for finding missed optimizations. Csmith covers a larger subset of the C language than ldrgen, allowing us to find some issues that could not be found using ldrgen. On the other hand, Csmith often produces large amounts of dead code, leading to trivial binary code generated even for complex-looking input source code. In contrast, ldrgen tries to avoid generating dead code, and binaries compiled from ldrgen-generated code are typically larger and more complex than those compiled from Csmith-generated code of similar size.

```
int fn3(double c, int *p, int *q) {
  int i = (int) c;
  *p = i;
  *q = i;
  return i;
}
```

(a) Program causing Clang to generate redundant code.

```
vcvt.s32.f64    s2, d0  ; convert double->int
vstr    s2, [r0]        ; store to *p
vcvt.s32.f64    s2, d0  ; convert again
vcvt.s32.f64    s0, d0  ; convert yet again
vmov    r0, s0          ; copy to return reg
vstr    s2, [r1]        ; store to *q
```

(b) Annotated ARM code generated by Clang. The type conversion is performed three times; one time would suffice.

Figure 3. Redundant repeated type conversions introduced by Clang for a single conversion in the source.

Figure 3 shows one example of a missed optimization found using a Csmith-generated seed program. This function receives pointers as arguments and contains assignments to their target. Clang used to duplicate the type conversion for each use of the variable `i`. We have reported this issue, and a partial fix eliminating the conversions but leaving some unnecessary copies has been added to Clang. This missed optimization could not have been found using only ldrgen because its current version never generates assignments through pointers.

Our system for interestingness tests on binaries is described in the following section. In the spilling example of Figure 2, the interestingness test searched for different numbers of loads through the stack pointer. Other tests may consider features such as function calls or certain classes of arithmetic instructions.

For reducing to minimal programs showing missed optimizations, we use C-Reduce. C-Reduce has many different passes that implement different minimizing transformations. We use the default pass set, except for disabling a pass that transforms local variables into global ones. The reason is the same as above: Accesses to globals introduce uninteresting differences in binaries.

The entire process sketched above is tied together by a shell script that simply calls a generator, the compilers, and the reducer. All the power in the process comes from

these tools. Reduced programs are checked by a human to see if they are trivial, duplicates, or really interesting missed optimizations.

4 Finding optimization differences

As we use standard random program generators and a standard test case reduction tool, the only optimization-specific part of our toolchain is the interestingness test. This interestingness test compares the code generated by different compilers. We implemented it in a tool we tentatively call `optdiff`.

Our design goal was to build a tool that can be easily reconfigured for different kinds of missed optimization tests. We also wanted it to be able to model at least an interestingness test for register allocation as discussed above. That is, we wanted to be able to identify different amounts of spill code (in particular, reloads of spilled values) in the outputs of different compilers. This means being able to identify opcodes and their arguments in order to find all loads whose address uses the stack pointer register as the base. As a spill reload in a loop is generally much less efficient than a reload outside of a loop, we also wanted to be able to estimate an execution frequency for every instruction. For this we needed access to a loop nesting tree or a control flow graph (CFG) for the compiled code. Finally, we wanted our tool to be retargetable to at least the ARM and x86-64 architectures.

After considering various assembly code parsers and binary analysis libraries, we settled on the angr binary analysis framework[2] [Shoshitaishvili et al. 2016] as the basis for our tool. This framework collects various tools useful for cross-platform binary analysis, such as an executable loader, a library of architecture descriptions, and predefined program analyses. Its API is presented as a Python library. Importantly for us, angr can construct a CFG for the binary it is given. Further, it gives access to the actual assembly opcodes and operands in the binary; while searching for an appropriate library, it surprised us that many binary analysis frameworks only provide generic platform-independent representations of the executable, but not assembly instructions.

Our `optdiff` tool is a Python program that uses angr's binary loader to load code from the binaries to be compared. For simplicity, we only look at the physically first nontrivial function in each binary. (Some platforms such as ARM use support functions from the compiler's runtime library to implement operations such as integer divisions. When calling `optdiff`, we replace these functions by 'trivial' stubs that simply return 0.) As our C code generators are configured to generate exactly one function, and as C-Reduce never introduces new functions, this suffices for our present purposes.

[2] http://angr.io/

```
@checker
def instructions(arch, instr):
    """Number of instructions."""
    return 1
```

(a) The trivial checker for counting instructions.

```
@checker
def loads(arch, instr):
    """Number of memory loads."""
    op = instr.insn.mnemonic
    if is_arm(arch):
        if op == 'ldrd':
            return 2
        elif re.match('ldm.*', op):
            return len(instr.insn.operands)-1
        return bool(re.match('v?ldr.*', op)
    ... # other architectures
```

(b) The checker for counting memory accesses (excerpt).

Figure 4. Two example checkers from `optdiff`.

We then use the algorithm of Wei et al. [2007] to construct a loop nesting tree for the input program and use that as a basis for basic block frequency estimation. Frequency estimation proceeds by assigning the function's entry block the frequency 1, then propagating frequencies in the CFG such that (a) every block's frequency is distributed equally to its successors at the same loop nesting level, and (b) loop entry blocks get assigned a frequency of 8 times their predecessors outside the loop (i. e., we assume that every loop iterates 8 times).

`optdiff` then uses functions we call *checkers* to compute a *score* for each binary. The score is a number expressing the prevalence of the feature of interest (e. g., the amount of spill code) in the input program. Typically, a lower score indicates a program considered more efficient according to some abstract criterion such as the number of instructions, the number of spill reloads, or the number of divide instructions.

A checker is a function that computes a local score for each instruction in the input program. The total score for the program is the sum of the individual instruction scores weighted by the estimated block frequencies computed before. Thus for a checker c and an input function f consisting of blocks b with frequencies (weights) w_b containing instructions i, the total score s is given by:

$$s = \sum_{b \in f} w_b \cdot \sum_{i \in b} c(i)$$

Checkers are small Python functions annotated with the `@checker` 'decorator', a Python metaprogramming device for registering these functions in `optdiff`'s table

of checkers. Checkers return a number or a boolean with `True` treated as 1, `False` treated as 0. Figure 4 shows the source code of two example checkers predefined in `optdiff`: the trivial checker for counting instructions, and the ARM-specific part of the checker for the number of memory loads. As ARM has load instructions for doublewords and a flexible load-multiple instruction, this checker counts the number of registers written by the instruction. Opcodes are strings, and we recognize related groups of instructions using regular expressions. This opens up room for mistakes if the regular expression is too specific or too general. However, we have found that in such cases, we can quickly identify errors from the minimal programs reduced according to a faulty criterion.

Checkers are normal Python functions that can call each other; the spill reload checker (not shown) builds on the `loads` checker from Figure 4 to identify memory loads whose address operand uses the stack pointer register as the base.

Note that checkers are purely local: they look at each instruction in isolation, without information about the surrounding code, or any way of propagating information to other instructions. This is for simplicity, although it makes some interesting things impossible. In particular, this framework cannot be used for cycle-accurate performance estimation, which would necessitate propagation of information about the states of pipelines and caches between instructions.

The `optdiff` tool provides command line flags for customizing its operation. It exits with a status indicating 'interesting' if the scores for the input binaries differ. It also provides a flag for a minimum absolute difference to be considered interesting. In other cases, we are not just interested in whether the scores s_1 and s_2 for the two binaries differ, but want only to consider cases where a certain compiler is 'better' than another one. For this, `optdiff` provides flags specifying that only the case $s_1 < s_2$, or only the case $s_1 > s_2$, should be considered interesting.

At the time of writing, `optdiff` comprises 573 physical lines of Python code, of which 246 lines compute the loop nesting tree and estimate basic block frequencies, 171 lines are checker definitions, and the remaining 156 lines parse command line arguments, load binaries, call the other components to compute scores, and exit with an appropriate result. There are currently 14 checkers, some of which are architecture-specific or not yet implemented for all architectures. These include checkers for the total number of instructions; memory loads and stores from the stack or from any address; register copies; additive or multiplicative integer arithmetic operations; additive, multiplicative, or arbitrary floating-point arithmetic operations; x86-64 packed (SIMD) instructions; and function calls.

```
int N;
double fn5(double *p1) {
  int i = 0;
  double v = 0;
  while (i < N) {
    v = p1[i];
    i++;
  }
  return v;
}
```

Figure 5. Example loop extensively optimized by GCC on x86-64. The loop could instead be replaced by a simple branch as done by GCC on ARM and by Clang on both platforms.

The x86-64 packed instructions are an interesting case of a checker where a lower score does not necessarily indicate a 'better' program. Indeed, a higher number of such instructions may mean that one compiler succeeded in vectorizing a loop but another didn't. We have not found such cases so far, but the search did result in an interesting issue shown in Figure 5. This function always returns either 0 if `N` ≤ 0, or `p1[N-1]` otherwise. Thus the loop can be replaced by a simple branch, as only the last iteration matters. Clang is able to perform this optimization on all platforms. GCC performs the optimization on ARM, but on x86-64, it generates a complex unrolled loop instead.

5 Dealing with undefined behavior

One particular feature of our system is the absence of checks for undefined behavior in the programs we generate or reduce. Such checks are standard in other randomized differential approaches that look for miscompilations [McKeeman 1998; Nagai et al. 2014; Yang et al. 2011] since undefined programs may be compiled differently by different compilers, without this being an indication of a miscompilation. For this reason, random generators like Csmith or Orange3 exclude certain undefined behaviors at generation time (for example, Csmith uses flow-sensitive pointer analysis) or generate code to guard against undefined behavior at runtime by using a library of safe wrapper functions for arithmetic operations.

For our search for missed optimizations, hiding most arithmetic operations behind a function call would be inconvenient: It would prevent almost all arithmetic simplifications that we may want to test for.

As another difference to our approach, differential bug-finding tools detect differences in compilation by executing the generated programs. For this, they include concrete inputs for their code, typically as initialized global variables.

In contrast, we never execute our programs, only compare the generated binary code. We therefore do not need input data. Inputs are function parameters with unknown values provided by a hypothetical caller. Some values for these parameters may lead to undefined behavior. For example, in Figure 5, the pointer `p1` may be `NULL` or otherwise invalid; in Figure 3, certain values of the `double` parameter may overflow the `int` it is converted to.

This is completely normal for C programs; compilers expect programmers to call such functions with valid arguments and typically do not give guarantees if such implicit preconditions are violated. On the other hand, they do not go out of their way to pessimize the cases where there *are* such violations; they just generate code as if such cases never occurred. Therefore, pragmatically, we expected no problems in practice with comparing such programs in which certain input values may cause undefined behavior.

All in all, our experiments confirmed that this was usually the case for undefined behavior during expression evaluation that was dependent on input values. On the other hand, we did find cases where the undefined behavior was unconditional and independent of input data. As a representative example, we did sometimes observe reduced functions such as the following:

```
int fn(int a) {
  int x = 0;
  int b = a / x;
  return b;
}
```

Neither GCC nor Clang warn about the unconditional division by zero in this case, although Clang recognizes it and 'optimizes' it to a function that returns immediately. GCC does not do this, thus the generated binaries differ, and this program is considered 'interesting' by some of our checkers. However, as the function does not admit any well-defined execution, we do not consider this an interesting optimization.

As a slightly more advanced example, other cases reduce the similar functions with divide expressions along the lines of `a / !!a`. This is well-defined and equal to `a` if `a` is nonzero, undefined otherwise. Such expressions do admit well-defined executions, but nevertheless the different code generated by different compilers is not what we would consider an interesting missed optimization.

We have experimented with including a static analyzer in the interestingness test to exclude the cases where every execution path leads to undefined behavior. We ran the Eva abstract interpreter [Blazy et al. 2017], part of the Frama-C program analysis platform [Kirchner et al. 2015]. This analyzer emits warnings when it cannot prove that certain operations are safe, and a more severe error when it finds that there are no valid execution paths in the program. Adding a check for such errors and exiting the interestingness test with a 'not interesting' status avoids reducing input programs to the division by zero case above. However, we found that this only pushes the reduction process in the direction of the latter, 'maybe divide by zero' case, which does not help us discover interesting missed optimizations.

```
unsigned fn6(unsigned a, unsigned p2) {
  int b;
  b = 3;
  if (p2 / a)
    b = 7;
  return b;
}
```

Figure 6. In unsigned arithmetic, the condition is true iff `p2` $\geq$ `a`. Clang compares directly, GCC divides.

Avoiding such uninteresting, partially undefined cases in practice is thus an open problem. For now, we use the following workarounds:

- These cases appear relatively often when simply counting instructions, but we found them to be sufficiently rare when using more specialized checkers. With these checkers, we simply accept that they will occur from time to time, and restart the search for a missed optimization. This is just a special case of the need to manually inspect and classify the reduced programs generated by our tool.
- There are many undefined cases of arithmetic or bitwise operations on signed integers, but in practice we only had such problems with division, modulo, and pointer dereference/indexing. We can configure the ldrgen program generator not to generate such operations, which allows us to focus on more interesting things.

As Figure 1 shows, we are still able to find interesting missed optimizations involving divisions. This is also the case if the divisor is a variable, as in the example of Figure 6. The division in the `if` condition can be replaced by a much cheaper direct comparison, something GCC fails to do on ARM.

Loops are a more problematic case. With our instruction count checker, C-Reduce likes to reduce functions containing loops to nonsensical code like the following:

```
void fn(char p3, int p5) {
  double v;
```

```
    while (p3)
      v = !p5;
    (int)v & 1;
}
```

Entering an infinite loop without externally visible side effects (such as I/O) is undefined in C, and different compilers 'optimize' this program differently. This, too, sends our reduction process down uninteresting paths.

The Eva static analyzer reports errors on functions that it can prove are definitely non-terminating, but it does not complain about conditional non-termination as in the case above. We have briefly experimented with the CPAchecker platform [Beyer and Keremoglu 2011] in the hope that its termination analysis would catch such cases of functions that do not terminate for all inputs. Unfortunately, we found that it returned 'unknown' in too many cases to be practical for our purposes.

We therefore leave the case of loops as an open problem for future work. Despite the interesting find of the example in Figure 5, the false positive rate due to non-terminating functions was too high for meaningful use. In most of our experiments we have therefore disabled the generation of loops. We intend to revisit this in the future with other termination checking tools.

6 Experimental evaluation

We have evaluated our method of finding missed optimizations in a series of unstructured experiments performed occasionally over the course of about five months, incrementally testing and evolving our `optdiff` tool and our methods in response to our observations. We used up-to-date development versions of all compilers at optimization level `-O3` and with `-fomit-frame-pointer` (where supported) to make spill code easier to identify.

6.1 Development of the project

It became clear early on that Csmith tends to generate large amounts of dead code, which often leads to very small and uninteresting binaries even for large C functions. After searching for other random generators, we eventually adopted ldrgen and have used it almost exclusively for most of our experiments. The two generators are complementary: Csmith's larger supported fragment of the C language allowed us to find the issues in Figures 3 and 7, while arithmetic optimizations are more quickly found using ldrgen. The library of `optdiff` checkers began with a simple checker for counting instructions and a very simple load checker that gradually evolved into the more sophisticated form shown in Figure 4. Other checkers were added as we observed patterns in generated programs and wanted to focus on more specific features.

```
struct S0 {
  int f0;
  int f1;
  int f2;
  int f3;
};

int f7(struct S0 p) {
  return p.f0;
}
```

Figure 7. The function returns the structure's first field, which on ARM is passed in register `r0`. This is also the return register; the function could return immediately. GCC first generates useless code to spill the structure to the stack, then reload the first field.

As a side-effect of various issues encountered over the course of the project, we have submitted patches or bug reports to the Csmith, angr, and ldrgen developers.

The missed optimizations we collected were thus found with a mix of various versions of different tools, reflecting the exploratory nature of this work. We varied the command line flags of generators and checker definitions to search for issues of various kinds.

For most of the project, we focused on comparing GCC and Clang, mostly generating code for ARM due to familiarity with its assembly language. While we are particularly interested in register allocation and spilling, it turned out that searching for differences in spill reloads often finds issues that are not directly related to register allocation. In particular, we found several arithmetic simplification issues while trying to find differences in register allocation. A common pattern was a function with more arguments than fit in argument registers, having to receive some arguments on the stack. If a computation on one of these stack arguments is redundant and is optimized away by one compiler but not by the other, this will show up in the compiled code as a difference in the number of reads from the stack. As this is a roundabout way of searching for arithmetic optimization opportunities, we implemented more direct checkers. In contrast, when searching for actual differences in register allocation, we typically tell the random code generator to limit the number of arguments according to the target platform's argument registers. Even so, we keep finding more spills due to differences in arithmetic optimizations (as less optimized code uses more registers) than due to real differences between register allocators.

6.2 Results

In the early stages of the project, we reported some issues we found interesting to the compilers' bug trackers. As

these are not correctness bugs, they are treated with low priority and mostly remained unfixed. We therefore stopped submitting them and decided to aggregate all in a public list. After we published this preliminary list online, there was some renewed interest by LLVM developers, and one previously reported issue was fixed.

It is difficult to count missed optimizations as we cannot be sure which seemingly related issues are really caused by the same piece of code. However, to the best of our knowledge, we have identified the following numbers of previously unreported, distinct missed optimization issues that we would consider worth changing and assume to be reasonably simple.

GCC In GCC, we have identified twelve issues, for which we have filed five reports so far, including the examples in Figures 2, 6, 7, and 8. One of our reports contained two issues that we now suspect to have separate underlying causes and that we now count separately. All of our reports were confirmed by developers, and one issue has been fixed in GCC's development version.

Clang For Clang/LLVM we reported one issue (Figure 3), which has been fixed in the development version. Another report was opened by somebody based on, and giving credit to, a collection of our examples of missed arithmetic optimizations of division and modulo operations (including the one in Figure 1); this, too, was fixed. For a third issue we did not file a report but directly submitted a patch, which is awaiting code review.

CompCert We have not tested CompCert extensively but have identified eight separate missed optimizations and have submitted patches for two of them.

We caution against interpreting these numbers as an indication of the quality of the various compilers' code generators. None of the issues we have found so far are likely to result in significant performance differences in actual applications. Our experiments were purely exploratory and not quantitative. We have made no effort to be 'fair' in the sense of investing equal amounts of work in finding missed optimizations in each compiler.

6.3 Classification of the issues found

We have tried to identify the underlying reasons for the missed optimizations we found. Our analysis is based on the patches we developed ourselves, the patches developed by compiler maintainers, code inspection, and comments by maintainers on the issues we reported. We have not been able to classify all issues.

6.3.1 Forgotten/faulty rules

Pattern matching rules for instruction selection, arithmetic simplifications, and in various program analyses are important parts of every optimizing compiler. With large numbers of rules and possible interactions between them, forgetting rules or getting the associated costs and priorities wrong is an unsurprising source of mistakes.

Of the examples in this paper, the ones in Figures 1, 2, and 3 fall into this category. The first one is part of a group that are considered missing rules in LLVM's instruction simplifier by LLVM developers. The second is confirmed by a GCC developer to be due to a missing cost annotation in a match rule. The third was due to redundant instruction selection patterns that, in the words of an LLVM developer, could 'confuse the cost logic' in the selector, leading to code duplication. Another LLVM example we have found and submitted a patch for is a missing rule for selecting the ARM `vnmla` 'floating-point multiply accumulate with negation' instruction for expressions of the form `-(a * b) - c`; the selector only has a pattern for the symmetric case of `-c - (a * b)`.

Four of the issues we identified in CompCert also fall into this category: a missing constant-folding rule for the modulo operator; a missing instruction selection rule for ARM's `movw` move-immediate instruction; missing constant folding rules for the ARM `mla` integer multiply-add instruction; and missing reassociation rules for multiplications by constants of the form `c1 * x * c2` to `x * (c1 * c2)`. We have submitted patches for the `movw` and `mla` issues.

```
int fn8_1(int p1) {
  int a = 6;
  int b = (p1 / 12 == a);
  return b;
}
```

(a) The division and comparison could be optimized to a subtraction and a comparison, but GCC failed to do this.

```
int fn8_2(int p1) {
  int b = (p1 / 12 == 6);
  return b;
}
```

(b) GCC was able to optimize this equivalent variant.

Figure 8. Example of a confirmed phase-ordering problem in GCC. The rule for simplifying the division and comparison used to be run before constant propagation.

6.3.2 Phase ordering

Phase ordering issues are a well-known and much researched problem in compiler construction. Some of the problems we found are due to such ordering problems. One example is shown in Figure 8. The branch condition in either case is equivalent to $72 \leq \texttt{p1} < 84$ and can be optimized to a subtraction and two signed comparisons

(or a single unsigned comparison) instead of the expensive division. GCC managed to do this in the second case, but not the first one. We reported this issue as a baffling case of seemingly failed constant propagation. It turned out that the matching rule for this optimization was only run before constant propagation, but not after. In response to our report, GCC developers have moved this rule from the early matching phase to a later one.

A trivial phase ordering case we found in CompCert concerns the conditions of useless branching statements like `if (c) {} else {}`. The corresponding useless jump instructions are cleaned up by one of the backend passes, but this pass only runs after dead code elimination. The code evaluating the condition `c` is therefore left in the program.

6.3.3 Unimplemented optimizations

Some optimizations may be missing from compilers simply because developers have not thought of implementing them, or have lacked the resources to do so. We collect some such cases because we find them interesting as opportunities to learn from other compilers' behavior. Clearly, these should not be considered mistakes of the same kind as forgotten cases or faultily implemented optimizations such as those discussed above.

The two such interesting examples we found both concern optimization of floating-point expressions in Clang but not in the other compilers we tested. The first of these is strength reduction of a statement of the form `i = i * 10.0` where `i` is a 32-bit `int` variable. According to the C standard, this multiplication is to be performed as `double` due to the type of the floating-point constant. However, Clang correctly recognizes that any possible result of this operation is an integer that can be represented exactly in a 64-bit IEEE `double`'s 53 bits of mantissa. If the result overflows the range of `int`, the operation is undefined, and any behavior is allowed; otherwise, the result is mathematically correct and identical to multiplying `i` by the integer 10. Clang therefore generates an integer multiply instead of converting `i` to `double`, multiplying as `double`, then converting back.

The second case is simplification of a floating-point expression of the form `x + 0`, where the floating-point variable `x` was initialized from an `int`. In general, a floating-point add of 0 may not be optimized away because `x` may be a negative zero. However, this case is impossible if it was initialized from `int`, so in this particular case Clang's transformation is sound.

6.3.4 Other/uncategorized issues

We have not been able to conclusively categorize some other issues, such as those of Figures 5, 6, and 7. We suspect that all of these cases fall into the category of faulty rules, either triggering unwanted code transformations (Figures 5 and 7) or failing to trigger a presumably existing beneficial transformation (Figure 6).

Another possible source of missed optimizations is simply bad luck due to the fact that many computationally hard problems, in particular related to register allocation, must be solved using heuristics. We do not have any examples that we can definitely blame on weak or unlucky heuristics.

6.4 Case study: finding regressions

The compilers compared by our method do not have to be completely different, they can also be different versions or different optimization levels of the same compiler. Such a comparison can be useful for finding regressions in optimizations [Iwatsuji et al. 2016].

To see whether our system was able to do this as well, we tested two development versions of GCC just before and after the fix for the issue in Figure 8. Searching for cases where the new version produced more instructions than the old, in 8 hours we generated 8097 programs, of which 16 matched the interestingness test and were reduced to minimal examples. (Generating and testing programs is fast; the bulk of the time is in reduction, which commonly takes on the order of 30 minutes to an hour.)

We found a case that could be considered a regression: In code like

```
a = 4;
if (x / a) ...
```

where the divisor is a power of 2, the division and comparison against 0 were previously implemented using a single shift that updated the condition code register. After the change, this is no longer treated specially but compiled to two instructions (an add and a compare).

7 Related work

To our knowledge, there has not been much work on direct, (semi-)automatic comparison of the code generated by different compilers.

7.1 Assembly-level missed optimization search

We are aware of a single tool that uses a similar approach to ours for finding missed optimizations, presented in a short paper by Iwatsuji et al. [2016]. This system generates random programs using the Orange3 generator, compiles them with different compilers, and counts instructions in the generated assembly code. If the generated codes are sufficiently different, Orange3's built-in test case reducer produces a minimal example. The authors compared GCC and Clang as well as two different versions of GCC against each other; they found differences between the compilers as well as regressions in the

newer version of GCC. Several issues were reported by the authors and fixed by the compilers' developers.

While this system is broadly similar to ours, its design choices are different in almost all respects: Generated programs are straight-line code without branches. Binaries are compared by pure (static) instruction count, not by more specific features such as spill reloads or certain classes of arithmetic as in our checkers. This limits the tool's usefulness for finding certain missed optimizations. The authors state that they only searched for arithmetic optimizations. The instruction counts n and m for different programs were not compared by absolute difference as in our system, but by whether the relative difference $|\frac{n}{m}|$ was below a threshold of 0.6 or 0.7. It is not clear to us why this design was chosen. However, we note that some of the authors' examples access inputs as global variables with hard-coded initializers. As we discussed in Section 3, we have found the use of global variables to introduce uninteresting variation in the generated code, and a threshold-based comparison of relative differences might be a useful guard against such cases.

Another difference to our approach is that the Orange3 system always produces well-defined, complete programs that can be run with inputs provided in global variables. In contrast, we focus on individual functions without predefined inputs. As we discussed in Section 5, undefined behavior is a double-edged sword, but admitting certain cases of potentially undefined functions may allow our tools to find more missed optimizations.

7.2 AST-based missed optimization search

A different system related to the one discussed above was presented by Hashimoto and Ishiura [2016]. It also uses the Orange3 program generation system and its reducer, but finds missed optimizations with an interesting twist: It generates random unoptimized C programs, then optimizes these programs itself on the abstract syntax tree (AST) level. The unoptimized and optimized ASTs are both unparsed to C source files, compiled, and the generated assembly codes compared by counting instructions. One interesting aspect of this work is that one does not need to compare two different C compilers; the optimization oracle is the program generation system itself. The interestingness test is a refinement of the model of Iwatsuji et al. [2016]. Although the AST-level optimization was restricted to constant propagation and folding, the authors found several missed optimizations in both GCC and Clang.

7.3 Superoptimization

Superoptimization is a technique originally introduced by Massalin [1987] for finding the smallest code sequence equivalent to a given piece of input code. In the original formulation, a superoptimizer enumerates all assembly code sequences up to a given length and checks them for equivalence with the target function. Exhaustive enumeration is primitive, while the equivalence check must be engineered to be as efficient as possible. Modern superoptimizers can be stochastic, symbolic (based on SAT or SMT solvers), or combine several of these techniques [Phothilimthana et al. 2016].

Superoptimizers are interesting in the context of this work because they reveal possibly useful previously missed optimizations in the output of compilers, although using a very different strategy from ours.

7.4 Dynamic missed optimization search

Dynamic analysis is another approach for finding missed optimizations. In the work of Chen and Regehr [2010], benchmark functions from open source applications were instrumented to log input values, then extracted from the application, compiled with different compilers, and ran in isolation on realistic inputs. These runs were carefully timed and used to compare optimizations across compilers cycle-accurately. The authors were able to identify very detailed architecture-specific differences in optimizers, such as Clang generating an instruction with a 16-bit immediate operand where apparently 8- and 32-bit immediates are to be preferred on x86-64. Several of the issues found in this work were quickly fixed in Clang and GCC.

8 Conclusions

We have presented a methodology and supporting tools for finding missed optimizations in C compilers. Using standard tools where possible, we generate random C functions, compile them using different compilers, compare the binary codes, and reduce any interesting cases to a minimal example showing the same interesting property.

The interestingness of binaries is determined using a new tool that compares binaries according to various criteria that may be relevant to performance, such as the number of instructions, the number of arithmetic instructions of certain types, types of memory accesses such as loads from the stack indicating register spilling, or function calls. Our binary checker framework is easily extensible to search for new criteria for missed optimizations.

We have found missed optimizations in all three C compilers we tested: GCC, Clang, and CompCert. Several of the issues we reported to bug trackers have been fixed by developers, and we have prototyped patches for several others. The missed optimizations we found

fall into different categories such as unnecessary register spilling, missed arithmetic optimizations, redundant computations, and missing instruction selection patterns.

In the future we plan to investigate further the treatment of possibly nonterminating programs, interprocedural optimizations, new optimization checkers, and combinations of checkers.

References

Antoine Balestrat. 2016. CCG: A random C code generator. Source code repository. (2016). https://github.com/Mrktn/ccg

Gergö Barany. 2017. Liveness-Driven Random Program Generation. In *Pre-proceedings of the 27th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2017)*. https://arxiv.org/abs/1709.04421

Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

Sandrine Blazy, David Bühler, and Boris Yakobowski. 2017. Structuring Abstract Interpreters Through State and Value Abstractions. In *Verification, Model Checking, and Abstract Interpretation: 18th International Conference, VMCAI 2017, Paris, France, January 15–17, 2017, Proceedings*, Ahmed Bouajjani and David Monniaux (Eds.). Springer International Publishing, Cham, 112–130. https://doi.org/10.1007/978-3-319-52234-0_7

Yang Chen and John Regehr. 2010. Comparing Compiler Optimizations. Blog post. (2010). https://blog.regehr.org/archives/320

Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing static analyzers with randomly generated programs. In *Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012)*. http://www.cs.utah.edu/~regehr/papers/nfm12.pdf

Eric Eide and John Regehr. 2008. Volatiles Are Miscompiled, and What to Do About It. In *Proceedings of the 8th ACM International Conference on Embedded Software (EMSOFT '08)*. ACM, New York, NY, USA, 255–264. https://doi.org/10.1145/1450058.1450093

Atsushi Hashimoto and Nagisa Ishiura. 2016. Detecting Arithmetic Optimization Opportunities for C Compilers by Randomly Generated Equivalent Programs. *IPSJ Transactions on System LSI Design Methodology* 9 (2016), 21–29. https://doi.org/10.2197/ipsjtsldm.9.21

Mitsuyoshi Iwatsuji, Atsushi Hashimoto, and Nagisa Ishiura. 2016. Detecting Missed Arithmetic Optimization in C Compilers by Differential Random Testing. In *The 20th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI 2016)*. http://ist.ksc.kwansei.ac.jp/~ishiura/publications/C2016-10a.pdf

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Aspects of Computing* 27, 3 (2015), 573–609. https://doi.org/10.1007/s00165-014-0326-7

Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core Compiler Fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 65–76. https://doi.org/10.1145/2737924.2737986

Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS II)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 122–126. https://doi.org/10.1145/36206.36194

William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. 2017. Effect-driven QuickChecking of Compilers. *Proc. ACM Program. Lang.* 1, ICFP, Article 15 (Aug. 2017), 23 pages. https://doi.org/10.1145/3110259

Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. 2014. Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions. *IPSJ Transactions on System LSI Design Methodology* 7 (2014), 91–100. https://doi.org/10.2197/ipsjtsldm.7.91

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling Up Superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 297–310. https://doi.org/10.1145/2872362.2872387

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

Jesse Ruderman. 2015. jsfunfuzz. Source code repository. (2015). https://github.com/MozillaSecurity/funfuzz/tree/master/js/jsfunfuzz

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward Understanding Compiler Bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 294–305. https://doi.org/10.1145/2931037.2931074

Tao Wei, Jian Mao, Wei Zou, and Yu Chen. 2007. A New Algorithm for Identifying Loops in Decompilation. In *Proceedings of the 14th International Conference on Static Analysis (SAS'07)*. Springer-Verlag, Berlin, Heidelberg, 170–183. http://dl.acm.org/citation.cfm?id=2391451.2391464

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532