Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages

VSEVOLOD LIVINSKII, University of Utah, USA
DMITRY BABOKIN, Intel Corporation, USA
JOHN REGEHR, University of Utah, USA

Compilers are part of the foundation upon which software systems are built; they need to be as correct as possible. This paper is about stress-testing loop optimizers; it presents a major reimplementation of Yet Another Random Program Generator (YARPGen), an open-source generative compiler fuzzer. This new version has found 122 bugs, both in compilers for data-parallel languages, such as the Intel® Implicit SPMD Program Compiler and the Intel® oneAPI DPC++ compiler, and in C++ compilers such as GCC and Clang/LLVM. The first main contribution of our work is a novel method for statically avoiding undefined behavior when generating loops; the resulting programs conform to the relevant language standard, enabling automated testing. The second main contribution is a collection of mechanisms for increasing the diversity of generated loop code; in our evaluation, we demonstrate that these make it possible to trigger loop optimizations significantly more often, providing opportunities to discover bugs in the optimizers.

CCS Concepts: • **Software and its engineering → Empirical software validation**; **Source code generation**.

Additional Key Words and Phrases: compiler testing, compiler defect, automated testing, random testing, random program generation, YARPGen

ACM Reference Format:
Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proc. ACM Program. Lang.* 7, PLDI, Article 181 (June 2023), 22 pages. https://doi.org/10.1145/3591295

1 INTRODUCTION

Machine learning, big data, and other recent trends have caused loop-specific compiler optimizations to increase in importance and in sophistication. For example, both GCC and LLVM can use the polyhedral model [Feautrier 1992a,b] to analyze and transform loop nests. LLVM's polyhedral optimizer—its "polly" subproject[1]—contains 24,700 lines of C++, and relies on another 175,000 lines of external support code. This much code, performing tricky symbolic reasoning, written in an unsafe language, and engineered to run quickly, might be expected to contain bugs and, in fact, a recent study of defects in GCC and LLVM [Zhou et al. 2021] found that "loop optimizations in both GCC and LLVM are more bug-prone than other optimizations." When a bug causes the compiler to crash, it is annoying; when it causes the compiler to emit incorrect object code, it is potentially dangerous. For example, Listing 1 shows a loop optimization bug in GCC that we discovered and reported.

[1] https://polly.llvm.org

Authors' addresses: Vsevolod Livinskii, vlivinsk@cs.utah.edu, University of Utah, USA; Dmitry Babokin, dmitry.y.babokin@intel.com, Intel Corporation, USA; John Regehr, regehr@cs.utah.edu, University of Utah, USA.

2475-1421/2023/6-ART181
https://doi.org/10.1145/3591295

```
void test(unsigned short a, unsigned short b, long long c) {
  for (char i = 0; i < (char)c; i += 5) {
    if (!b)
      var_120 = a;
    else
      var_123 = a;
  }
}
```

Listing 1. This function triggers a loop-related miscompilation bug in GCC (#102920); it was automatically reduced from a test case that we generated

```
void stencil(int* restrict in, int* restrict out, int n) {
    for (int i = 0; i < n; ++i)
        out[i] = (in[i - 1] + in[i] + in[i + 1]) / 3;
}
```

Listing 2. Even a very simple stencil loop like this triggers interesting optimizations that we want to test https://gcc.godbolt.org/z/aEs1daYWq

The research problem addressed by this paper is how to expose bugs in the implementations of sophisticated loop optimizers. The main hypothesis that we evaluate is "loop optimization bugs in compilers for C-like data parallel languages can be effectively discovered using randomly generated programs that are free of undefined behavior (UB), and that also contain idioms recognized by loop optimizers." Randomized testing is desirable because the space of inputs to a compiler is very large and—empirically—compiler developers are not capable of writing test cases by hand that reveal all of the defects in a production-grade compiler. We require generated programs to be UB-free because it is not possible to reach reliable conclusions from the observable behaviors of a program that executes UB. Randomized testing methods have been applied to compilers for at least 60 years [Sauder 1962] and they can be effective.

Although random generation of UB-free code is a problem that has been addressed by a number of previous papers, we are unaware of prior work specifically focusing on generating the kinds of expressive loop idioms that appear to be required to stress-test loop optimizers. Listing 2 shows an example of the kind of code that we want to generate; it is a stencil: a loop where each element of an output array is a function of a collection of nearby elements of an input array or arrays. In this case, each output element is the average of three neighboring input elements. To optimize this code, a compiler can notice that `in[i + 1]` in one loop iteration is the same value as `in[i]` in the next iteration, and also `in[i - 1]` in the iteration after that. Thus, while a naive translation of this loop would load all three values from RAM each time the loop body executes, an optimized version only needs to load one value from RAM, with the other two values being forwarded from previous iterations using registers.

Modern compilers for C and C++ use sophisticated loop transformations and auto-vectorization to achieve high performance, while data-parallel languages such as ISPC [Pharr and Mark 2012] and SYCL [Khronos® SYCL™ Working Group 2020] require less aggressive analysis and optimization since the languages directly expose fine-grained parallelism. However, compilers for all of these languages perform non-trivial code transformations, which are error-prone, especially when targeting modern CPU and GPU architectures. Our work targets all of these C-family languages by generating random programs in a high-level intermediate representation that supports loop

idioms and also static analysis to ensure UB-freedom; it can be lowered relatively straightforwardly to the four different concrete syntaxes. As a starting point, we used our previous version of YARPGen [Livinskii et al. 2020], which was designed to target scalar optimizations in C and C++ compilers, but we almost entirely re-implemented it to support loops and to output multiple languages. In this paper, we will refer to the old implementation as YARPGen v.1, and to the new one as YARPGen or YARPGen v.2 when it is necessary. The version that we started with contained 8,619 lines of C++; to patch that program (using Git's patch facility) to the current YARPGen v.2 requires removing 6,295 lines of C++ while adding 10,099.

YARPGen v.2 was able to detect 66 previously-unknown bugs in GCC, 28 in LLVM, 16 in the Intel® oneAPI DPC++ compiler, and 12 in Intel® ISPC. Furthermore, although these targets were not a primary focus for us, we found two bugs in the Intel® Software Development Emulator[2] and two in the Alive2 translation validation tool [Lopes et al. 2021]. We reported all of these bugs and most of them have been fixed, showing that compiler developers consider the kind of bugs that we find to be worthwhile. The research contributions of this paper include a static UB avoidance mechanism for loops, and our loop generation policy mechanism for creating programs that help us find interesting and difficult-to-trigger compiler bugs.

2 BACKGROUND: UNDERSPECIFIED ASPECTS OF C-FAMILY PROGRAMMING LANGUAGES

To support generating high-quality object code across a wide variety of target platforms, the C language and its descendants are somewhat underspecified: they give implementations substantial freedom to make convenient and efficient choices. These choices come in three flavors.

First, *implementation-defined behaviors* are those where the compiler must make a consistent choice and also document it. For example, the size of the `int` type, and the range of values that it supports, are implementation-defined. These behaviors do not concern us in this work. In principle, they limit the scope of differential testing, which can only be done across compilers that agree on a collection of important implementation-defined behaviors. However, in practice, most modern compilers of interest agree on these.

Second, *unspecified behaviors* are those where there compiler must choose from a collection of alternatives, with no requirement for consistency. For example, the order in which arguments to a function are evaluated is unspecified. In practice, these behaviors are few enough and benign enough that they can be avoided fairly easily—for example, by ensuring that function arguments do not have side effects.

Finally, *undefined behaviors* in C-family languages are untrapped error conditions: the standard imposes no requirement on the behavior of a program that, for example, accesses out-of-bounds memory. There are hundreds of undefined behaviors (UBs) and it is, in general, difficult to statically guarantee their absence. Avoiding UB while generating expressive loops is a primary contribution of our work.

3 GENERATING UB-FREE PROGRAMS WITH EXPRESSIVE LOOPS

Our code generator's output should be expressive: it should be syntactically and semantically interesting in the sense that it triggers as many code paths in the compiler (and, in particular, the loop optimizer) as possible. On the other hand, its output must not execute undefined behaviors, and ideally it should avoid UB without using the kind of pervasive dynamic checking that, for example, Csmith [Yang et al. 2011] used. These goals are in tension; this section describes how we achieve them both. Figure 1 provides an overview. Generation proceeds in three main steps:

[2] https://www.intel.com/content/www/us/en/developer/articles/tool/software-development-emulator.html

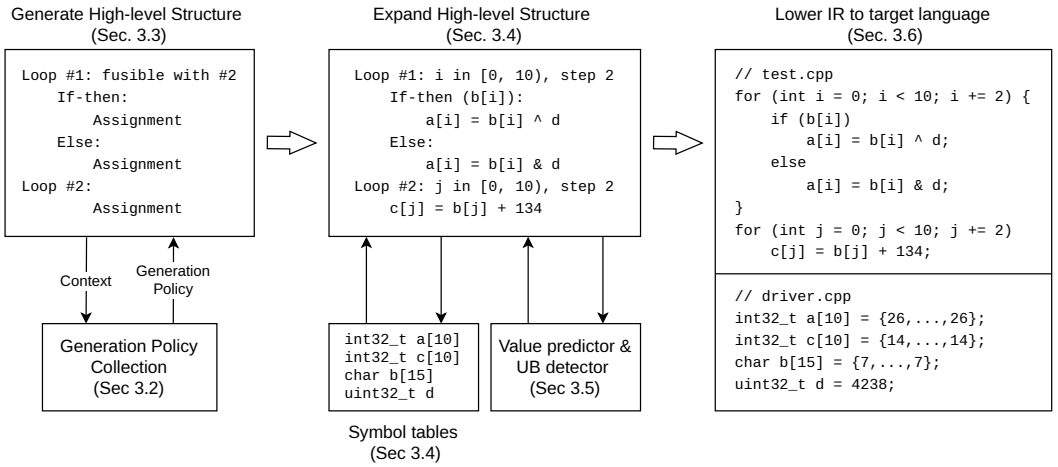

Fig. 1. Overview of how YARPGen v.2 generates code

creating a high-level program skeleton, fleshing out the skeleton with details such as arrays and operations on them, and then lowering our intermediate representation (IR) to a concrete syntax.

3.1 Test Oracles

Software testing requires an *oracle* to determine if some execution of the system under test was correct or incorrect. In addition to the trivial oracle that looks for abnormal termination of the compiler process, YARPGen v.2 supports:

- *Ground truth:* As a side-effect of undefined behavior avoidance, YARPGen precomputes the effect of executing the randomly generated code. Thus, a miscompilation can be signaled if the result of running the compiled code differs from this prediction.
- *Differential:* We compute a checksum of the effect of executing the randomly generated code; this is used for differential testing where the result is not known in advance, but a bug is signaled when two different compilers (or two different modes of the same compiler, such as `gcc -O0` and `gcc -O3`) disagree with each other.

Both oracles are necessary in practice. The first one is useful when differential testing is impossible, for example because there is only one compiler for a given language. This is the case for Intel® ISPC, where we have found bugs that affect all modes of the compiler, such as issue #1768; differential testing is incapable of finding this bug. However, pre-computing the result of a program interferes with automated test-case reduction tools (e.g., C-Vise [Liška 2022], C-Reduce [Regehr et al. 2012], and Perses [Sun et al. 2018]) because they make non-semantics-preserving changes during reduction. Thus, it is crucial that we do not rely on the code producing a specific answer—this is where differential testing becomes most useful. (This sort of test-case reduction relies on external tools to reject reduction steps that trigger undefined behaviors.)

3.2 Loop Generation Policies

Generation policies is an idea introduced in our previous paper about YARPGen v.1 [Livinskii et al. 2020]. We used this as something of a catch-all term for mechanisms that were used to increase the probability of generating test cases that were believed to be desirable, and that (without generation policies) were being generated only rarely, if at all. In this paper, we have adapted this idea, broadly

construed, to generating loops. As a simple example, consider that we would like to test *loop fusion*, an optimization that can merge adjacent loops when they have identical iteration spaces and also lack dependencies that block fusion. The odds of a pair of fusible loops being organically generated is quite low, but it is easy to generate such a pair simply by making the decision to do so.

The basic principle behind generation policies is that we cannot find bugs in optimizations that we cannot trigger. A rule of thumb that we have used is that if we run across an interesting loop transformation that is supported by more than one compiler, then we should at least consider creating a generation policy that targets it. We used existing bugs reports, unit tests, test suites, and general knowledge of common compiler optimizations as sources of inspiration. We analyzed these to identify interesting patterns that were missing from YARPGen v.2 at the time, and tried to fit them into the existing infrastructure or extend it to support them. A difficult issue is helping YARPGen trigger specific transformations without constraining expressiveness to such an extent that fuzzing effectiveness is compromised. What we attempt to do is abstract away the essence of the pattern that triggers the transformation, while leaving to random chance as many of the details of instantiating the pattern as possible. The resulting fuzzing technique ends up being something like a gray-box fuzzer, but with a human—a YARPGen developer—in the loop, adding generation policies as required in response to gaps in code coverage. We have added loop generation policies to YARPGen v.2 piecemeal over a period of several years as we learned how to break loop optimizers; the rest of this section describes them. They are not mutually exclusive; they compose to increase the diversity of the generated code even further. For example, YARPGen v.2 can generate a combination of reduction and stencil (`a += (b[i - 1] + b[i] + b[i - 1]) / 3`) that will compute a reduction of three elements of the array.

Loop sequences and loop nests. These are represented as a first-class elements in the fuzzer's IR; they are the main factor that determines the high-level shape of the generated code. They are essential for making coordinated decisions involving multiple loops. For example, to trigger a loop interchange optimization, we have to generate a loop nest that contains array accesses with a column-major order. To trigger loop fusion, adjacent loops have to have the same iteration spaces. To trigger other loop optimizations, "perfectly nested loops" where all assignment operators are in the innermost loop must be generated.

Array access patterns. Applications often access arrays in idiomatic patterns [McCool et al. 2012], and optimizing compilers have adapted to provide specific support for some of these. These patterns are determined by the interaction between arrays and loops' iteration spaces. First, we can arbitrarily decide the relation between array dimensionality and the loop nest depths (fewer, same, or more). Second, we can pick the order of the induction variables used in the array subscripts—in order of the loop nest or not, and whether to use the same induction variable to access multiple array dimensions.

This approach allows us to achieve good expressiveness and to mimic various access patterns that are found in real applications. For example, if we decide that the array dimension matches the loop depth and use the same induction variables for all dimensions (`a[i][i]`), we will get a diagonal traversal of a matrix. Another option is to use in-order traversal of an array with some constant indexes to get a slice; this generation policy causes all of these special cases to happen regularly. We consider array access patterns to be a set of rules that govern generation of individual array subscripts—subsequent policies such as stencils make more involved decisions that are synchronized across multiple array subscripts.

Stencils. Stencil codes (Listing 2) are ubiquitous in image processing and scientific applications, such as finite difference methods. Given the huge number of degrees of freedom available to a naive

```
extern int a[], b[];
void foo() {
  for (int i = 0; i < 20; ++i)
    a[i] = b[i];
}
```

Listing 3. Code snippet that triggers LLVM's `memcpy` idiom recognizer

random program generator, it is not likely that stencils will naturally be generated. Therefore, we implemented a stencil pattern as a first-class IR element, that can be used in arithmetic expressions inside loops. Our stencil generation policy results in loops that use multiple constant offsets from a single induction variable into an array or arrays. This gives us direct, fine-grained control over things like stencil size, stride, number of array dimensions used, and number of arrays used, that would be difficult to get control over if we only used unstructured random generation.

Vectorizable loops. Automated vectorization is a sophisticated program transformation performed by most of the compilers we are targeting. It tends to be somewhat fragile, and can be defeated by a number of program properties such as:

- data dependencies across loop iterations
- library functions outside of a limited set
- unpredictable iteration spaces
- unpredictable control flow, particularly related to exiting the loop

Furthermore, it is often the case that only the innermost loop of a loop nest gets vectorized. If we want to heavily stress autovectorizers, we need to ensure that all prerequisites are met sufficiently often. In our initial experiments, this did not happen naturally, so we added a "vectorizable" loop property that ensures that they will be fulfilled, satisfying our goal of generating many vectorizable loops, but not compromising on our ability to express more general loops.

Reductions. These computations—common in real applications—reduce the dimensionality of data, for example by summing the elements of a loop, computing the smallest element, etc. YARPGen's reduction generation policy provides a generalized version of this kind of computation where a randomly generated function is applied element-wise to an input array, resulting in an output array of reduced dimensionality.

Loops over bytes. Loops that iterate over a vector of bytes are common, and optimizing compilers like to turn them into more efficient computations when possible. For example, open-coded loops over bytes can sometimes be turned into faster implementations such as the system-provided `memset` or `memcpy` routine (Listing 3). These idiom recognizers tend to be fragile, and they are highly desirable targets for stress testing. To achieve this, YARPGen v.2 employs a byte-loop pattern by creating a loop header that iterates through bytes, and populates the loop body with randomly generated expressions.

Compiler-specific loop attributes. It can be useful to override a compiler's built-in cost functions using pragmas such as LLVM's `#clang vectorize`, `#clang interleave`, and `#clang unroll`. We optionally add them to loops when generate tests for C and C++. It is safe to do so, because when a compiler encounters unknown pragmas, it simply emits warning messages.

3.3 Generating the High-Level Program Structure

Because our focus is on intra-procedural loop optimizations, YARPGen v.2 generates a single function when it is run. Its first step, structure generation, performs top-down construction of skeleton code that it will later flesh out with details. The high-level code includes elements found in C-family languages such as conditionals and assignments, but it also contains larger structural elements such as loop sequences and loop nests. During this step, generation policies are invoked to determine properties of various program elements. For example, generation policies supply the maximum depth of a loop nest and the maximum length of a loop sequence. Other generation-policy-based decisions are also made during this phase, such as a loop sequence being marked as containing loops that have a common iteration space, or an individual loop being marked as auto-vectorizable. These attributes will be used later to guide how YARPGen fills in the detailed code. The reason that we first create the high-level skeleton is that this led to a pleasing separation of concerns in the random generator, making it more modular and debuggable than it otherwise would have been. For example, as a debugging aid YARPGen can print a representation of the high-level structure, which can be inspected readily because it lacks complicating details.

3.4 Expanding the Skeleton

The first step in fleshing out the skeleton program is to generate global data items that the random code will access. Data is split into a table of inputs, whose values are known and will not be changed by the randomly generated code, and outputs, whose values will be inspected in order to detect miscompilation bugs. Since YARPGen does not attempt to look for bugs in floating point optimizations, these globals are integer-typed scalars and arrays. On the input side, only scalar variables are created in this step, with arrays being created later, on the fly, in conjunction with loop generation, to ensure that array sizes match up with loop iteration spaces. Output variables and arrays are created during the expansion of assignments later. If any of the global variables end up being unused by the randomly-generated code, they are removed before IR is lowered to a concrete syntax. The values stored in these global variables are known to YARPGen—which uses them for its undefined behavior analysis—but are opaque to the compiler that is being tested. We currently rely on separate compilation to provide opacity; if we ever test a compiler that is sufficiently aggressive with cross-file optimization, we will have to be heavier handed. For example, the initialized globals could be placed in a dynamically loaded library.

After global input variables have been created, YARPGen proceeds to fill in the IR for the generated code. Expression nodes in the IR are expanded top-down; the targets of assignments are allocated in the output symbol table, exposing the resulting values to external compiler-correctness-checking. Each scope gets its own symbol table; these local tables contain iterators in addition to scalars and arrays. Iterators are treated separately by the undefined behavior analysis: they are used as induction variables in loops, but they are not involved in arbitrary randomly generated computations. Local symbol tables store the type, dimension, scope, initial value, and current value of each data item.

To expand a loop header, YARPGen first has to take into account constraints imposed by high-level loop properties that come from generation policies. For example, a loop that is part of a fusible sequence will have the same iteration space as its neighboring loops. A loop performing a stencil computation will likely have an iterator that does not (quite) run over the entire array, because the stencil will access array elements offset from the iterator's position. Initially, each loop is characterized by start, end, and step values that are constants, but YARPGen randomly replaces some of these constants with expressions that are partially opaque to the compiler (by depending on external variables), but end up evaluating to the same values that the constants would have held.

```
varying int max(varying int a, varying int b) {
    return a > b ? a : b;
}
```

Listing 4. ISPC code for a parallel max operation; varying indicates a vector data type. This function is compiled into a single masked vector instruction: `vpmaxsd zmm0, zmm0, zmm1` https://ispc.godbolt.org/z/MraWhd5Tb

This gives us some interesting diversity in how loop iteration spaces look to the compiler without making UB avoidance more complicated than it already is.

3.5 Static Undefined Behavior Avoidance for Loops

Loops potentially lead to two kinds of UB: out-of-bounds array accesses and arithmetic UB inside the loop body. Whereas Csmith [Yang et al. 2011] relied on dynamic checks to avoid both kinds of UB, our hypothesis is that pervasive conditional control flow in loop bodies would hamper some of the optimizations that we wish to test. Existing research [Even-Mendoza et al. 2020] appears to corroborate this suspicion.

YARPGen's undefined behavior avoidance is entirely static, and uses concrete value tracking, which was pioneered by the Orange family of random program generators [Nagai et al. 2012, 2013, 2014] and extended in our previous paper about YARPGen v.1 [Livinskii et al. 2020]. For example, when generating a shift operator in a C-family language, the shift exponent must be non-negative and also smaller than the bitwidth of the value being shifted. Thus, if YARPGen v.1 wanted to generate `x << y` in a situation where `y == -1000`, it would instead generate something like `x << (y + 1005)`. As long as the code being generated is loop-free, this strategy maintains the invariant that the already-generated program fragment is UB-free. Thus, when the tool terminates, the entire program is UB-free. For scalar code this approach worked well and YARPGen v.2 also uses it. This UB avoidance strategy is not altogether straightforward to extend to loops, though both Orange and YARPGen v.1 had some limited solutions for this. Their approaches are summarized in Section 6, so here we will give a brief overview. Orange3 simulated every loop iteration, avoiding UB any time that it would have happened by adding values from an array specifically initialized with values that avoid UB. This approach is computationally expensive, and can produce UB avoidance artifacts in the generated code, similar to Csmith's [Yang et al. 2011] wrapper functions. Orange4 [Nakamura and Ishiura 2016] simply ensured that each generated loop would execute at most once. This approach is not able to test loop optimizations that rely on run-time properties of the code, such as loop trip count for some case of unrolling or vectorization. YARPGen v.1 had experimental support for loops that, like Orange4, ensured that the first iteration would not trigger UB; but then it also ensured that subsequent iterations of the loop would see the same values.

YARPGen v.2 uses this same approach—ensuring that all loop iterations see the same values—as one of its two approaches to generate UB-free loop code. This is possible because we keep track of every iteration space that we generate, as well as dimensions of the arrays, so that these can be matched up. It also helps that we maintain a clean separation between variables that are only used as inputs, and those that are only used as outputs, making value tracking pretty straightforward even inside loop bodies. We rely on separate compilation (Section 3.4) to ensure that the compiler cannot notice that we have used these strategies, forcing it to analyze the general case. Separate compilation cannot, however, stop the compiler from observing runtime characteristics of our code. For example, at runtime, ISPC maintains an execution mask to track active program instances[3]

[3]An ISPC program instance is similar to a CUDA thread or an OpenCL work-item https://ispc.github.io/ispc.html#basic-concepts-program-instances-and-gangs-of-program-instances

```
for (int i = 0; i < N; ++i) {
    a[i] = (i % 2 == 0)    ? (b[i] + c[i]) : (b[i] - c[i]);
    d[i] = (i % 2 == zero) ? (e[i] * f[i]) : (e[i] / f[i]);
}
```

Listing 5. Addition of two arrays, where UB is avoided by conditional access, based on the iterator. In the second case the condition is obfuscated with a variable whose value is opaque to the compiler.

in order to use masked vector instructions to describe control flow, rather than using explicit branching. For example, the `max` function (Listing 4) is compiled into a single instruction. On hardware targets that do not support masking directly, the cost of applying a mask is high, and it becomes profitable to check for the "all-on" state of the mask at runtime, and execute a separate, simpler code path in that case. Our first method produces loops that operate on the same values in each iteration. Therefore, the mask will always contain the same value, and we will only test the "all-on" code path and will never trigger the general path. We would instead like the execution mask to contain diverse values so all code paths based on it are tested.

To ensure that we can test this sort of optimization, we developed a novel generation mechanism that allows arrays to use multiple values (but without attempting to analyze all loop iterations, which is computationally infeasible). We do this by logically separating the iteration space of the loop into subsets. At present, we support partitioning into two subsets—even and odd elements—but we plan to expand and generalize this support in the future. In this scheme, the iterator walks through the array in an alternating odd-even pattern, so the loop body performs a different computation for the even and odd values. For example, if we start at element one and use three as a step, we will get the required pattern. Optionally, YARPGen can hide the splitting criterion from the compiler, as shown in Listing 5, using an opaque variable. A limitation of this method is that arrays can have different values only along one of the axes. For example, in a case of three-dimensional array, alternating values can be in rows, columns, or planes, but only in one of these for any given loop nest.

The main advantage of loop partitioning is that it permits YARPGen v.2 to reuse its static UB-avoidance mechanism in loop bodies with minimal overhead in terms of time taken to generate random code: the UB avoidance mechanism runs twice per loop instead of once.

This method has several advantages over the previous approaches. It allows us to test divergent values in the loop body without any wrapper functions, it does not create UB avoidance artifacts, and it only requires us to analyze a small, fixed number of loop iterations, minimizing overhead. This approach is one of the two major contributions of this paper.

3.6 Lowering YARPGen IR to Multiple Languages

Because the four C-family languages supported by YARPGen v.2 are fairly similar, and because its IR has been designed taking all of them into account, lowering our intermediate representation to any of them is reasonably straightforward. One useful thing that we can do during lowering, when the target language supports several similar constructs, is to choose from them randomly. For example, when targeting ISPC and SYCL, we randomly choose between emitting regular and data-parallel loops, driving the compiler down different code paths.

The two most similar of the supported languages are C and C++. From the YARPGen point of view, they are almost identical. The difference between them includes different UB rules for the left-shift operator, supported types, and standard library functions. The rest of the machinery is shared.

Support for ISPC, on the other hand, is more involved. The main complexity comes from its explicit vector types, which means that YARPGen has to support an additional type parameter to

capture this in its IR. This type property is orthogonal to C++ type properties, which add another layer of type casting rules. They have to be supported in the form of explicit casts, as well as implicit casting rules that are similar to integral promotion and arithmetic conversions in C++. We also extend support for standard library calls to reductions and vector-wise operations.

YARPGen's SYCL support is limited, and did not require any augmentation of the fuzzer IR. We were able to satisfy the limitations of parallel SYCL loops by changing generation parameters, such as maximum depth of data-parallel loops or set of allowed standard library functions. The main modifications were implemented in the lowering component of the fuzzer, where we had to support data transfer between the test driver and test function.

3.7 High-level Intermediate Representation

The high-level IR in YARPGen v.2 is designed to support expressive, UB-free loops, and also to lower to multiple target languages. This IR is roughly analogous to a compiler IR, in the sense that our IR nodes represent program objects such as types, values, statements, and expressions. It is, overall, simpler than a compiler's IR because YARPGen does not need to support the wide variety of analyses and transformations that an optimizing compiler supports. Rather than discovering properties of something like the iteration space of a loop, YARPGen tends to make a decision about the iteration space ahead of time, and then subsequently generates code having the desired properties.

Our IR supports generation of expressive, UB-free loops in two ways. First, we carefully chose what elements have a first-class representation in the IR; these include loop sequences, loop nests, and stencils. Compiler IRs, in contrast, would usually represent these elements implicitly as collections of more primitive nodes. By directly representing higher-level abstractions, we can more easily enforce high-level properties such as creating perfect loop nests or synchronizing the iteration spaces of a sequence of loops. The second way that our IR supports expressiveness and UB-freedom is by supporting a variety of auxiliary elements that track the program environment and the current state of the generation process. For example, generating a loop that can be reliably vectorized requires restricting its behavior in several ways (Section 3.2); the loop context object helps track these.

Supporting lowering to multiple programming languages is mainly a matter of avoiding commitment to particular representations too early. For example, matrix multiplication in ISPC uses data-parallel loops with carefully constructed operations to avoid data-dependency conflicts, whereas C++ uses simple loop nests. These code fragments look quite different at the syntax level, but since they perform the same operation, we represent them using the same IR construct. Similarly, vector types in ISPC serve some of the same functions as arrays in other languages; we have IR elements that abstract over this representation issue.

3.8 Generating a Test Harness

Besides a file containing the randomly generated function, YARPGen v.2 also emits a header file containing declarations for global variables and a driver file that contains a main function and also definitions for all global variables. The randomly generated code typically receives some of its inputs via parameters and others via global variables. The main function initializes data items, calls the randomly generated function, and then looks at its return value and also the values of global variables; to support the differential testing oracle it prints a checksum of these outputs, and to support the ground truth oracle it checks for mismatches with its value predictions.

3.9 Limitations

Features not yet supported by YARPGen v.2 include:

- Floating point math
- Dynamic memory allocation
- Support for multiple random functions—generated code includes function calls, but only to standard library code
- First-class pointers and pointer arithmetic—YARPGen v.2 currently only supports the limited kinds of pointers that occur when an array value decays into a pointer type
- Non-standard vector extensions, such as intrinsic functions that give C or C++ code access to specific vector instructions

Some of these limitations are inherent to our design. For example, we did not set out to find bugs in vector intrinsics, and in fact optimizing compilers more or less leave these intrinsics alone during compilation, treating them as opaque. Fixing other limitations is clearly desirable, and is a matter of putting more engineering resources into YARPGen. For example, supporting multiple functions that call each other should be fairly straightforward given the infrastructure that we have already created. Finally, some limitations seem difficult to lift. The most prominent example is absence of floating-point (FP) support. This is a serious limitation, but it is on par with the state of the art in C++ compiler fuzzing. To the best of our knowledge, there exist only two compiler fuzzers (Orange [Nagai et al. 2014] and YARPGen v.1 [Livinskii et al. 2020]) that have addressed the issue of correct compilation of IEEE FP, and both of them have serious limitations and did not appear to generate significant results. We discuss this issue and possible solutions further in Section 5.

4 EVALUATION

This section describes the bugs that have been found using YARPGen v.2, and evaluates its ability to trigger various loop optimizations. These results—including the number of reported bugs—are completely separate from our previous work on YARPGen v.1 [Livinskii et al. 2020].

4.1 Summary of a Testing Campaign

Over the last three years, we used YARPGen v.2 to test then-current versions of GCC and LLVM, as well as occasionally testing Intel® ISPC, the Intel® oneAPI DPC++ compiler, and Alive2. An up-to-date list of bugs found by YARPGen, that we have reported, is available online.[4] In all cases, compilers targeted various flavors of x86-64. We checked that YARPGen v.2 can be used to test ARM programs (using an Apple M1 chip), but we did not perform a thorough testing campaign for that target. Our testing focused on commonly used optimization levels: `-O0` and `-O3`. Jiang et al. [2022] have shown that esoteric compiler options can increase the number of detected bugs. However, we avoided these since, in our experience, compiler developers are often not motivated to fix issues that are encountered far away from the default optimization pipeline.

The top-level results of our testing campaign are:

- 66 bugs in GCC. All of these were either fixed or assigned to compiler developers, showing that bugs discovered by YARPGen are valued by the GCC developers. 32 of these were miscompilation bugs, 31 were compiler crashes, and three were cases where the compiler failed to terminate. 56 of the bugs were in the middle-end optimizations, seven in the x86-64 backend, two were in an inter-procedural optimization, and one was in a front-end. 22 of the bugs we reported affected more than one released version of the compiler. 13 of the bugs that we reported were independently rediscovered by users. Table 1 shows the 50 most recently reported bugs.
- 28 bugs in LLVM. 18 of these were fixed, one was confirmed, one was resolved, and eight remain unacknowledged. 23 of these bugs were compiler crashes and five were miscompilation

[4] https://github.com/intel/yarpgen/blob/main/bugs.rst

bugs. 15 were in a middle-end optimization, 11 in the x86-64 backend, and two were not classified. A full list is in Table 2.
- 12 bugs in Intel® ISPC. All of these have been fixed. Seven of them were compiler crashes and five were miscompilation bugs. Seven were in a middle-end optimization and five in the x86-64 backend. A full list is in Table 3.
- 16 bugs in the Intel® oneAPI DPC++ compiler. Nine were miscompilation bugs and seven were compiler crashes. Nine were in a middle-end optimization, three in the x86-64 backend, and four remain unclassified. Intel® oneAPI DPC++ bugs were reported to a non-public bug tracker.
- Two bugs in Intel® SDE and two bugs in Alive2.

Overview of reported bugs. We looked for patterns in the bugs that we found. Please note that since our testing campaign ran over several years, during which we continued to develop YARPGen, features that we added earlier were used in more test cases—so there are likely to be some biases in these informal results.

In terms of the overall program structure, we determined that perfect loop nests of depth two with unknown trip count were the most common bug trigger. We have reported bugs with bigger loop depths (up to five), but their combined presence was roughly the same as that of depth two. As for array access patterns, we found that slicing along one of the array dimensions was the one that most commonly triggered bugs. We also found that the most buggy loop optimization component of GCC was related to vectorization, whereas in LLVM instruction selection was the buggiest component.

21% of the bugs we discovered were in a compiler backend, despite the fact that our target is high-level loop optimizations. We believe that there are several factors involved here; for example, in some cases middle-end optimizations open up additional possibilities for backend optimizations. Also, loop optimizations such as vectorization are tied closely to the compiler's backend, with details such as the presence of masking being important. A dedicated testing campaign for compiler backends [Boushehri et al. 2022; Rong et al. 2022] would almost certainly be more effective at finding backend bugs than our approach is. However, that kind of approach risks finding less relevant bugs due to generating non-canonical IR, whereas our approach always presents backends with canonical IR that is emitted by a compiler middle end.

Duplicate GCC bug reports. In several cases, we found a recently-introduced GCC bug around the same time that others reported it. For example, bugs #103119, #105621, and #106605 were reported by compiler users. Additionally, another research group was running a similar compiler testing campaign concurrently with ours, but using mutation-based test-case generation. They found GCC bugs #103399, #106417, #107228, #108668 before we did. In all seven cases, we reported the bug within two days of others having reported it. Also, one bug (#96693) was erroneously marked as a duplicate despite the fact that we had reported it about a week earlier.

Impact of open-sourcing YARPGen. A minor complicating factor in our testing campaign is that during it, we released YARPGen v.2 as open source software. We did this because we knew of several individuals who were specifically interested in fuzzing loop optimizations, and we judged that helping them (and others like them) out was more important than being the only users of our tool. For example, Martin Liška, a GCC developer, reported eight bugs that he found using our tool, including #101256. We did not know that they were using our tool, but we discovered this after the fact and included their bugs in our list of GCC bugs. However, there may have been others using YARPGen, who we did not know about.

Table 1. YARPGen v.2 found 66 bugs in GCC, this table shows the latest 50 of them. "ICE" stands for "Internal Compiler Error." "ASGD" stands for "Assigned." We shortened names of the components and bug descriptions.

#	ID	Status	Type	Component	Description
1	95649	fixed	ICE	tree-opt	ICE during GIMPLE pass: cunroll
2	95717	fixed	ICE	tree-opt	ICE during GIMPLE pass: vect: verify_ssa failed
3	95916	fixed	ICE	tree-opt	ICE during GIMPLE pass: slp : verify_ssa failed
4	96022	fixed	ICE	tree-opt	ICE during GIMPLE pass: slp in operator[], at vec.h
5	96755	fixed	ICE	target	ICE in final_scan_insn_1, at final.c with -O3 for skx
6	98048	fixed	ICE	tree-opt	ICE in build_vector_from_val, at tree.c
7	98064	fixed	ICE	tree-opt	ICE in check_loop_closed_ssa_def, at tree-ssa-loop-manip.c
8	98069	ASGD	wrong code	tree-opt	Miscompilation with -O3
9	98211	fixed	wrong code	tree-opt	Wrong code at -O3
10	98213	fixed	timeout	tree-opt	Never ending compilation at -O3
11	98302	fixed	wrong code	target	Wrong code on aarch64
12	98308	fixed	ICE	tree-opt	ICE in vect_slp_analyze_node_ops, at tree-vect-slp.c
13	98381	fixed	wrong code	tree-opt	Wrong code with -O3 -march=skx
14	98513	fixed	wrong code	tree-opt	Wrong code with -O3
15	98640	fixed	wrong code	tree-opt	GCC produces incorrect code with -O1 and higher
16	98694	fixed	wrong code	target	Wrong code for loops with -O3 for skx and icx
17	99777	fixed	ICE	tree-opt	ICE in build2, at tree.c with -O3
18	99927	fixed	wrong code	rtl-opt	Wrong code
19	100081	fixed	timeout	tree-opt	Compile time hog in irange
20	101014	fixed	timeout	tree-opt	Big compile time hog with -O3
21	101256	fixed	wrong code	tree-opt	Wrong code with -O3
22	102511	fixed	wrong code	tree-opt	Wrong code for -O3: first element of the array is skipped
23	102572	fixed	ICE	tree-opt	ICE for skx in vect_build_gather_load_calls, at tree-vect-stmts.c
24	102622	fixed	wrong code	tree-opt	Wrong code with -O1
25	102696	fixed	ICE	tree-opt	ICE in vect_build_slp_tree, at tree-vect-slp.c for skx and icx
26	102788	fixed	wrong code	tree-opt	Wrong code with -O3
27	102920	fixed	wrong code	tree-opt	Wrong code with -O3
28	103037	ASGD	wrong code	tree-opt	Wrong code with -O2
29	103073	fixed	ICE	ipa	ICE in insert_access, at ipa-modref-tree.h
30	103122	fixed	ICE	tree-opt	ICE in fill_block_cache, at gimple-range-cache.cc
31	103361	fixed	ICE	tree-opt	ICE in adjust_unroll_factor, at gimple-loop-jam.c
32	103489	fixed	ICE	tree-opt	ICE with -O3 in operator[], at vec.h
33	103517	fixed	ICE	tree-opt	ICE in as_a, at is-a.h with -O2 -march=skx
34	103800	fixed	ICE	tree-opt	ICE in vectorizable_phi, at tree-vect-loop.c with -O3
35	104551	fixed	wrong code	tree-opt	Wrong code with -O3 for skx, icx, and spr
36	105132	fixed	ICE	tree-opt	ICE in in operator[], at vec.h with -O3 for skx
37	105139	fixed	wrong code	target	vmovw instruction with an incorrect argument for -O3 for spr
38	105142	fixed	wrong code	tree-opt	Wrong code with -O2
39	105189	fixed	wrong code	tree-opt	Wrong code with -O1
40	105587	fixed	ICE	target	ICE in extract_insn, at recog.cc (error: unrecognizable insn)
41	106070	fixed	wrong code	tree-opt	Wrong code with -O1
42	106292	fixed	wrong code	tree-opt	Wrong code with -O3
43	106630	fixed	ICE	tree-opt	ICE: Segfault signal terminated program cc1plus with -O2
44	106687	fixed	wrong code	tree-opt	Wrong code with -O2
45	107404	fixed	wrong code	target	Wrong code with -O3
46	108166	fixed	wrong code	tree-opt	Wrong code with -O2
47	108365	ASGD	wrong code	c++	Wrong code with -O0
48	108647	fixed	ICE	tree-opt	ICE in upper_bound, at value-range.h with -O3
49	109341	ASGD	ICE	ipa	ICE in merge, at ipa-modref-tree.cc
50	109342	fixed	wrong code	tree-opt	Wrong code with -O2

Table 2. YARPGen v.2 detected 28 bugs in LLVM. “ICE” stands for “Internal Compiler Error.” “RES” stands for “Resolved.” “CONFR” stands for “Confirmed.” We shortened names of the components and bug descriptions.

#	ID	Status	Type	Component	Description
1	42819	fixed	ICE	Backend: X86	ICE: “Cannot select: X86ISD::SUBV_BROADCAST”
2	42833	fixed	wrong-code	Backend: X86	Incorrect result with -O3 -march=skx
3	46178	fixed	ICE	Backend: X86	Assertion ‘idx <size()’ in combineX86ShufflesRecursively
4	46471	new	ICE	new-bugs	Assertion “Uses remain when a value is destroyed!”
5	46525	fixed	ICE	new-bugs	Assertion ‘!verifyFunction(*L->getHeader()->getParent())’
6	46561	fixed	wrong-code	new-bugs	Wrong code with -O1
7	46586	fixed	wrong-code	Backend: X86	Wrong code with -O2
8	46661	fixed	ICE	new-bugs	InstCombine stuck in an infinite loop after 100 iterations
9	46680	fixed	ICE	ScalarOpt	InstCombine stuck in an infinite loop after 100 iterations
10	46950	fixed	ICE	new-bugs	UNREACHABLE at Transforms/Vectorize/LoopVectorize.cpp
11	47098	fixed	ICE	Opt	Polly during “Polly - Forward operand tree” on skx
12	47292	RES	ICE	Opt	ICE in polly with -O3
13	48326	fixed	ICE	Backend: X86	Assertion “Invalid child # of SDNode!”
14	48422	fixed	ICE	Opt	Assertion “Unknown counts for blocks that dominate latch!”
15	48445	fixed	ICE	Opt	Assertion “Partial READ accesses not supported”
16	48554	fixed	ICE	isl	ICE: polly/lib/External/isl/isl_ast_build_expr.c
17	50109	fixed	ICE	Opt	UNREACHABLE at Transform/ManualOptimizer.cpp
18	51797	new	ICE	new-bugs	InstCombine stuck in an infinite loop after 100 iterations
19	51798	fixed	ICE	LoopOpt	Assertion ‘hasVectorValue(Def, Instance.Part)’
20	51906	new	wrong-code	LoopOpt	LICM introduces load in writeonly function (UB)
21	51923	new	ICE	LoopOpt	Clang segfaults with loop unroll(enable)
22	52002	new	ICE	new-bugs	Assertion “Uses remain when a value is destroyed!”
23	52273	new	ICE	LoopOpt	Assertion ‘!verifyFunction(*L->getHeader()->getParent())’
24	52335	new	wrong-code	Backend: X86	Incorrect result with -O1 -march=skx
25	52504	fixed	ICE	Backend: X86	Assertion “Cannot use this version of ReplaceAllUsesWith!”
26	52560	fixed	ICE	Backend: X86	Cannot select: t60: v8i16 = X86ISD::VZEXT_MOVL t55
27	52561	CONFR	ICE	Backend: X86	Assertion “Can’t BITCAST between types of different sizes!”
28	58616	new	ICE	new-bugs	Assertion “Expected vector-like insts only.”

Table 3. YARPGen v.2 detected 12 bugs in Intel® ISPC. “ICE” stands for “Internal Compiler Error.” We shortened names of the components and bug descriptions.

#	ID	Status	Type	Component	Description
1	1719	fixed	ICE	middle-end	Division by zero leads to ICE
2	1729	fixed	ICE	middle-end	Assertion failed: “ci != NULL”
3	1762	fixed	ICE	middle-end	ICE: “scatterFunc != NULL”
4	1763	fixed	wrong-code	middle-end	Wrong code for avx2-i64x4
5	1767	fixed	ICE	backend	Assertion “Getting TableId on SDValue()”
6	1768	fixed	wrong-code	middle-end	Uniform and varying types have different rounding rules.
7	1771	fixed	wrong-code	backend	Wrong code for avx2-i64x4
8	1788	fixed	ICE	middle-end	InstCombine stuck in an infinite loop after 1000 iterations
9	1793	fixed	wrong-code	backend	Wrong code for avx2-i32x16
10	1806	fixed	wrong-code	middle-end	ISPC produces wrong code with bool type iterator
11	1844	fixed	ICE	backend	ICE: “Unexpected illegal type”
12	1851	fixed	ICE	backend	LLVM assertion ‘Def == PreviousDef’ failed

Speed of bug fixes in LLVM. Our testing campaign for LLVM did not go as well as the one for the GCC. Following standard practices for responsible external bug-finding, we avoided flooding their bug tracker with issues, and instead kept the number of outstanding bug reports under a small limit. Alas, the LLVM developers often did not fix the issues that we reported very quickly, causing us to report fewer such bugs than we otherwise could have.

Partial bug fixes. An interesting facet of running a fuzzing campaign is observing cases where a fuzzer-triggered bug is fixed only incompletely. In a typical situation, the reported test case is fixed but without fully addressing the root cause of the bug; in this case, YARPGen finds other ways to trigger the issue soon after the initial, incomplete fix lands. This happened six times for GCC bugs during our testing campaign.

4.2 Covering Optimizations

A compiler fuzzer cannot find bugs in optimizations that it cannot trigger. Our loop generation policy mechanism is explicitly designed to trigger more loop optimizations more often; in this section we evaluate its ability to do that. We use optimization counters[5] that the LLVM developers have provided in order to evaluate our ability to trigger optimizations. These counters are, in effect, a high-level, domain-specific code coverage metric.

Experimental setup. We threw out a number of counters that we judged to be irrelevant, including those for counting experimental optimizations, for bookkeeping unrelated to optimizations, and for counting optimizations that cannot be triggered from C or C++, such as those related to garbage collection. To do this we extracted all 1,360 counters from the LLVM source code and manually analyzed the resulting list in order to select those that we judged are related to loops, vectorization, or are enabled by other loop optimizations. Some of these counters were easy to identify (e.g., `loop-unroll.NumUnrolled` and `loop-vectorize.LoopsVectorized`); others required looking closely at the surrounding source code. In summary, we made a good-faith effort to pick a set of optimization counters that actually count loop-related optimizations. There are 238 of these.

We used LLVM 15.0.3[6] as the basis for these experiments. We compare YARPGen's ability to trigger optimizations with that of a version of YARPGen where we disabled loop optimization policies, and also with LLVM's performance test suite.[7] This suite can optionally include various versions of the SPEC® CPU benchmark; we included SPEC® CPU2017 and configured it as directed by the LLVM developers.[8] For the benchmark suite, we simply compiled it and tallied up the number of times each optimization counter was triggered. For YARPGen v.2 (both with and without generation policies) we compiled randomly-generated programs over a 24-hour period using all cores on a machine with two AMD™ EPYC 7502 32-core processors.

Results of comparing YARPGen with the benchmark suite. Out of our set of 238 optimization counters, YARPGen v.2 (with generation policies) and the LLVM test suite, together, are able to trigger 80 of them. Eight were exclusively triggered by the test suite. Thus, YARPGen v.2 is able to test 90% of the optimizations that are triggered by the applications in LLVM's test suite (including SPEC® CPU2017).

Results of comparing YARPGen with and without generation policies. Only one of the 72 loop-related optimization counters (`gvn.MaxBBSpeculationCutoffReachedTimes`) was exclusively triggered by YARPGen v.2 with generation policies, the rest were triggered by both versions. We used Welch's

[5]This is our term; LLVM simply calls them "statistics."
[6]https://github.com/llvm/llvm-project/releases/tag/llvmorg-15.0.3
[7]https://github.com/llvm/llvm-test-suite/releases/tag/llvmorg-15.0.3
[8]https://github.com/llvm/llvm-test-suite/tree/main/External/SPEC

Table 4. Out of the 72 LLVM optimization counters that YARPGen v.2 can trigger, this table lists the most loop-relevant ones, and reports the average ratio between how many times that counter is triggered with generation policies (GP), as opposed to without them. For each of these counters, the data support the claim that generation policies trigger that counter more often, at a 95% confidence level.

Opt. counter name	GP to no GP ratio
licm.NumHoisted	10.29
licm.NumMovedCalls	3.48
licm.NumMovedLoads	20.31
licm.NumPromoted	12.34
licm.NumSunk	2.36
loop-delete.NumBackedgesBroken	2.04
loop-delete.NumDeleted	11.33
loop-idiom.NumMemSet	6.17
loop-instsimplify.NumSimplified	5.61
loop-peel.NumPeeled	2.40
loop-rotate.NumInstrsDuplicated	6.11
loop-rotate.NumInstrsHoisted	2.24
loop-rotate.NumNotRotatedDueToHeaderSize	4.69
loop-rotate.NumRotated	6.02
loop-simplify.NumNested	5.22
loop-simplifycfg.NumLoopBlocksDeleted	4.31
loop-simplifycfg.NumLoopExitsDeleted	9.29
loop-simplifycfg.NumTerminatorsFolded	6.90
loop-unroll.NumCompletelyUnrolled	10.97
loop-unroll.NumRuntimeUnrolled	9.79
loop-unroll.NumUnrolled	10.67
loop-unroll.NumUnrolledNotLatch	5.99
loop-vectorize.LoopsAnalyzed	4.09
loop-vectorize.LoopsEpilogueVectorized	11.55
loop-vectorize.LoopsVectorized	32.42

t-test to put each triggered optimization counter into one of three categories. For the first one, we can say with 95% confidence that generation policies are better (YARPGen v.2 with them triggers the counter more times than YARPGen v.2 without them); for the second category, we can say with 95% confidence that generation policies are worse; for the last one, the null hypothesis that neither version of YARPGen v.2 is better at triggering this particular counter. By this test, generation policies are better for 71 counters; worse for none; and, for one counter (`indvars.NumElimRem`) no conclusion can be drawn from our data. Table 4 shows a subset of these counters, along with the ratio between how many times each is triggered with and without generation policies. The geometric mean of this ratio across all 72 counters is 9.14. Overall, generation policies appear to be an effective way to trigger loop optimizations more often.

4.3 Code Coverage

Absolute code coverage numbers, for a compiler like GCC or LLVM, are tricky to interpret because these compilers support multiple source languages, multiple backends, and many configuration options. On the other hand, in controlled circumstances, relative code coverage numbers might

Table 5. Coverage of GCC source code

	Functions	Lines	Branches
YARPGen v.2	37.28%	34.96%	23.79%
SPEC® CPU 2017	44.84%	40.96%	27.96%
unit tests	79.51%	77.09%	55.94%
unit tests + YARPGen v.2	79.86%	78.16%	57.40%
unit tests + SPEC®	79.63%	77.45%	56.36%
unit tests + SPEC® + YARPGen v.2	79.93%	78.34%	57.62%

Table 6. Coverage of LLVM source code

	Functions	Lines	Branches
YARPGen v.2	22.30%	12.61%	12.37%
test suite (includes SPEC® CPU 2017)	27.99%	16.31%	17.14%
unit tests	84.26%	89.93%	73.58%
unit tests + YARPGen v.2	84.27%	89.97%	73.76%
unit tests + test suite	84.27%	89.97%	73.77%
unit tests + test suite + YARPGen v.2	84.28%	89.99%	73.85%

provide useful information. We collected code coverage for the LLVM 15.0.3[9] and GCC 12.2.0[10] implementations for the following inputs:

(1) the unit test suite that is distributed with the compiler
(2) SPEC® CPU2017 v1.0.1 (for GCC only)
(3) the LLVM test suite, including SPEC® CPU2017 v1.0.1 (for LLVM only)
(4) 24 hours of random testing with YARPGen v.2 in its default configuration, with the -O3 optimization flag, on an AMD™ Ryzen 9 5950X 16-core processor

The results are presented in Tables 5 and 6. YARPGen v.2 does not improve the coverage by much, nor does it provide very good coverage by itself. However, these results are in line with previously reported code coverage due to generative random fuzzers (for example, Table 3 from the Csmith paper [Yang et al. 2011] and Tables 8 and 9 from the YARPGen v.1 paper [Livinskii et al. 2020]). Our view is that coverage of functions, lines, and branches are simply not very good metrics for evaluating compiler fuzzers—the internal behavior of compilers is highly path- and value-sensitive.

4.4 Performance of YARPGen

We measured the CPU time used by each step in the random testing pipeline, when testing LLVM 15.0.3 and GCC 12.2.0 at their `-O0` and `-O3` optimization levels. YARPGen v.2 was used in its default configuration. The CPU usage was measured from the scripting infrastructure that drives random testing, and it does not include the CPU time used by that infrastructure itself, but we do not believe it to be significant. We conducted this experiment on a machine with an AMD™ Ryzen 9 5950X 16-core processor, using all cores. The results are presented in Table 7. The majority of processor time in this experiment, 77.7%, was spent in the compilers, and only 0.78% was spent in YARPGen v.2. Thus, it is not a bottleneck during random testing.

[9] https://github.com/llvm/llvm-project/releases/tag/llvmorg-15.0.3
[10] https://gcc.gnu.org/gcc-12/

Table 7. How CPU time is spent during random testing

Tool	Step	% of total CPU time
YARPGen v.2	generation	0.78%
gcc -O0	compilation	12.38%
	execution	5.31%
gcc -O3	compilation	38.63%
	execution	5.27%
clang -O0	compilation	7.63%
	execution	5.47%
clang -O3	compilation	19.10%
	execution	5.43%

5 FUTURE WORK

Making fuzzer specialization easier. As discussed in Section 3.2, generation policies need to be implemented by hand. A better route to extending a generative fuzzer like YARPGen might be to provide a domain-specific language for expressing generation policies. We have not yet pursued this research agenda, but Xsmith [Hatch et al. 2023] is an example of work in that direction: it uses a language specification to generate a fuzzer that can incorporate sophisticated program-generation techniques. However, it has not yet been applied to any domains that are as complex as generating UB-free C++. Another interesting approach would be to use feedback, such as coverage of the compiler under test, perhaps in combination with machine learning techniques, to automatically infer generation policies.

Floating-point support. The oracle problem is the main obstacle to using fuzzers to look for defects in compilers for languages that support IEEE floating point operations. The most common and interesting use case for FP optimizations, the "`-ffast-math`" compiler flag, allows compiler optimizations that lead to different numerical results across different compilers, platforms, and optimization levels. This means that almost every test case will look buggy to a tool that strictly applies differential testing. We experimented with an approach based on limiting the number of FP operations in a dependency chain, and then also allowing the result to be within a range of the expected result, before differential testing signaled a bug. This turned out to work poorly: the chain-length restriction limited the fuzzer's expressiveness, and coming up with an appropriate range for the result was difficult. We have come to believe that a much finer-grained approach that uses a formal methods tool to look at individual transformations performed by a compiler is probably a better way to detect miscompilation of floating point computations.

6 RELATED WORK

Compiler fuzzing has a long history; summary papers about compiler testing and fuzzing in particular were published by Boujarwah and Saleh [1997] and by Chen et al. [2020]. In this section we focus on prior work that potentially finds the same kinds of errors as YARPGen v.2, or implements mechanisms similar to generation policies; we briefly survey other approaches.

Generative compiler fuzzing. Csmith, developed by Yang et al. [2011], used a combination of whole program analysis and dynamic checks to avoid undefined behavior in generated tests. In particular, dynamic checks were used to eliminate UB in arithmetic operations and array subscripts. Subsequent research by Even-Mendoza et al. [2020, 2022] showed that wrapper functions impose a noticeable penalty on the code coverage and bug-finding ability of the fuzzer; this was one of the main motivations for YARPGen's preference for static UB-avoidance.

CLsmith, developed by Lidbury et al. [2015], is a modified version of Csmith that was created to test OpenCL compilers. Its test cases include vector types, intra-group communication, and atomic operations. Additionally, the developers supported equivalence modulo inputs [Le et al. 2014]—also known as metamorphic testing—to further enhance the bug-finding ability of CLsmith.

The Orange family of fuzzers pioneered the static UB avoidance approach, developing this idea for generating branch-free scalar code using concrete value tracking, over a series of papers [Nagai et al. 2012, 2013, 2014]. They subsequently developed two ways to extend this mechanism to loops. The first one, created by Nakamura and Ishiura [2015], effectively unrolls the loop inside the generator, analyzing all of its iterations. Any subexpression that causes UB at some iteration is augmented with array addition to avoid UB. For example, if `a[i] + b[i]` causes signed overflow for some value of `i`, it is replaced with `a[i] + (b[i] + tmp[i])`, where `tmp[i]` is initialized with zeroes except when a different value is required to avoid UB. This approach scales poorly in the presence of nested loops. Orange's second loop extension, developed by Nakamura and Ishiura [2016], allowed each loop to execute at most one iteration. Code generated by this approach is logically loop-free, but this fact can be hidden from the compiler by making the loop induction variable sufficiently opaque. YARPGen's approach is similar to these, but improves upon them in several ways. YARPGen v.1 [Livinskii et al. 2020] also extended the approach pioneered by Orange, but most of the improvements were targeted at scalar code.

Mutation-based compiler fuzzing. Rather than generating test cases from scratch, the mutation approach takes a seed program and modifies it. This has advantages and disadvantages; in our view, neither of these kinds of fuzzing subsumes the other. A family of mutation-based compiler fuzzers developed by Zhendong Su's research group [Le et al. 2014, 2015; Sun et al. 2016] holds the record in terms of reporting the most bugs for GCC and LLVM.

Fuzzing compilers using machine learning. A recent trend in compiler fuzzing is incorporating machine learning into the generator. This technique is not purely generative, since it requires example programs for training purposes, but it does not fit cleanly into the mutation-based approach either. For example, DeepFuzz, developed by Liu et al. [2019], uses the GCC test suite as training data; 82% of the generated programs are syntactically correct and can be compiled. DeepSmith, created by Cummins et al. [2018], targets OpenCL compilers; it generates tests that contain undefined or non-deterministic behavior, and uses compiler warnings and third-party tools to filter them out. A summary paper about application of machine learning techniques in fuzzing was published by Wang et al. [2020]. This approach has a huge advantage: the structure of the generated programs is inferred, rather than being painstakingly constructed by hand. However, it tends to produce programs that are not compliant with the relevant standards; for example, UB-freedom is a global property that, so far, appears to be out of reach of learned generators. Of course, depending on the goals of a testing campaign, it might be desirable to present compilers with non-conforming inputs. So far, machine-learning-based test case generators have not reached parity with hand-written tools in terms of bug-finding power.

Improving diversity in random test cases. An early approach to increasing the probability of triggering compiler optimizations is due to Burgess and Saidi [1996]. Their fuzzer, which produced FORTRAN tests, explicitly introduced common subexpressions, linear induction variables, and arithmetic expression patterns that the optimizer was known to be looking for.

Another approach to increasing diversity of generated tests is *swarm testing*, introduced by Groce et al. [2012]. The idea is to artificially change the probability of generating some program element at the beginning of generating a test case. So one generated test might be completely free of bitwise operators and another one might be dominated by them. YARPGen v.1 [Livinskii et al.

2020] adapted these ideas and applied them in a fine-grained way to its generation policies, which controlled contexts such as what kind of operators to generate, helping it find hard-to-trigger bugs. YARPGen v.2 builds directly on this idea, and expands it to loop optimizations and data-parallel languages. As far as we know, our work is the first to directly target these kinds of optimizations.

7 CONCLUSION

YARPGen v.2 is an open-source[11] generative fuzzer that can be used to find bugs in loop optimizations in compilers for C, C++, ISPC, and SYCL. Over a three-year period, it was able to detect 122 wrong code bugs and internal compiler errors; most of these have been fixed. Moreover, we reported 13 GCC bugs that were independently rediscovered by users, showing that at least 20% of the bugs we reported for GCC are of the kind that users actually encounter and then (typically) triage and reduce in a painfully manual fashion.

The first contribution of this paper is a novel static undefined behavior avoidance mechanism for loops that allows YARPGen v.2 to generate tests that are guaranteed to be compliant with language standards. Moreover, this method is applicable to multiple languages; combined with our syntax-independent IR, it allows us to target multiple languages with a single fuzzer. The second contribution is a collection of loop-specific generation policies that increase the number of applied loop and vector optimizations on average by 9.14 times for LLVM.

ACKNOWLEDGMENTS

This material is based on work supported by a grant from the Intel® Corporation. The authors would like to thank the GCC, LLVM, Intel® Implicit SPMD Program Compiler, and the Intel® oneAPI DPC++ compiler development teams for their help and cooperation. We would also like to especially thank Martin Liška, Richard Biener, and Jakub Jelinek for their overwhelming support and help with the GCC bugs that we reported. The authors would also like to thank the Flux research group[12] at the University of Utah for providing access to the Emulab cluster[13] that we used for our experiments. Finally, we thank the anonymous reviewers and our shepherd—Ayal Zaks—for their valuable feedback and suggestions.

REFERENCES

Abdulazeez S. Boujarwah and Kassem Saleh. 1997. Compiler Test Case Generation Methods: A Survey and Assessment. *Information and software technology* 39, 9 (1997), 617–625. https://doi.org/10.1016/S0950-5849(97)00017-7

Nader Boushehri, Ryan Berger, Stefan Mada, and John Regehr. 2022. Automated Translation Validation for an LLVM Backend. Presented at 2022 LLVM Developers' Meeting. https://llvm.org/devmtg/2022-11/slides/TechTalk18-AutomatedTranslationValidation.pdf [Online, accessed 03-16-2023].

C. J. Burgess and M. Saidi. 1996. The Automatic Generation of Test Cases for Optimizing Fortran Compilers. *Information and Software Technology* 38, 2 (Jan. 1996), 111–119. https://doi.org/10.1016/0950-5849(95)01055-6

Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36. https://doi.org/10.1145/3363562

Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler Fuzzing through Deep Learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 95–105. https://doi.org/10.1145/3213846.3213848

Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. 2020. Closer to the Edge: Testing Compilers More Thoroughly by Being Less Conservative about Undefined Behaviour. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1219–1223. https://doi.org/10.1145/3324884.3418933

[11] https://github.com/intel/yarpgen
[12] https://flux.utah.edu/
[13] https://www.emulab.net/

Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. 2022. CsmithEdge: More Effective Compiler Testing by Handling Undefined Behaviour Less Conservatively. *Empirical Software Engineering* 27, 6 (2022), 1–35. https://doi.org/10.1007/s10664-022-10146-1

Paul Feautrier. 1992a. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *International Journal of Parallel Programming* 21, 5 (1992), 313–347. https://doi.org/10.1007/BF01407835 Publisher: Springer.

Paul Feautrier. 1992b. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming* 21, 6 (1992), 389–420. https://doi.org/10.1007/BF01379404 Publisher: Springer.

Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm Testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 78–88. https://doi.org/10.1145/2338965.2336763

William Hatch, Pierce Darragh, and Eric Eide. 2023. Xsmith. https://gitlab.flux.utah.edu/xsmith/xsmith [Online, accessed 03-16-2023].

He Jiang, Zhide Zhou, Zhilei Ren, Jingxuan Zhang, and Xiaochen Li. 2022. CTOS: Compiler Testing for Optimization Sequences of LLVM. *IEEE Transactions on Software Engineering* 48, 7 (2022), 2339–2358. https://doi.org/10.1109/TSE.2021.3058671

Khronos® SYCL™ Working Group. 2020. SYCL™ Specification 1.2.1. https://registry.khronos.org/SYCL/specs/sycl-1.2.1.pdf [Online, accessed 12-08-2022].

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. *ACM SIGPLAN Notices* 49, 6 (2014), 216–226. https://doi.org/10.1145/2666356.2594334

Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. *ACM SIGPLAN Notices* 50, 10 (2015), 386–399. https://doi.org/10.1145/2858965.2814319

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)*. 65–76. https://doi.org/10.1145/2737924.2737986

Martin Liška. 2022. C-Vise. https://github.com/marxin/cvise [Online, accessed 11-08-2022].

Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 1044–1051. https://doi.org/10.1609/aaai.v33i01.33011044

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. In *Proceedings of the ACM on Programming Languages*, Vol. 4. 1–25. https://doi.org/10.1145/3485529

Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 65–79. https://doi.org/10.1145/3453483.3454030

Michael McCool, James Reinders, and Arch Robison. 2012. *Structured Parallel Programming: Patterns for Efficient Computation* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Eriko Nagai, Hironobu Awazu, Nagisa Ishiura, and Naoya Takeda. 2012. Random Testing of C Compilers Targeting Arithmetic Optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*. 48–53.

Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. 2013. Scaling up Size and Number of Expressions in Random Testing of Arithmetic Optimization of C Compilers. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2013)*. 88–93.

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

Kazuhiro Nakamura and Nagisa Ishiura. 2015. Introducing Loop Statements in Random Testing of C Compilers Based on Expected Value Calculation. In *Proc. of the Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2015)*. 226–227.

Kazuhiro Nakamura and Nagisa Ishiura. 2016. Random Testing of C Compilers Based on Test Program Generation by Equivalence Transformation. In *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. IEEE, 676–679. https://doi.org/10.1109/APCCAS.2016.7804063

Matt Pharr and William R. Mark. 2012. ISPC: A SPMD Compiler for High-Performance CPU Programming. In *2012 Innovative Parallel Computing (InPar)*. IEEE, 1–13. https://doi.org/10.1109/InPar.2012.6339601

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*. 335–346. https://doi.org/10.1145/2345156.2254104

Yuyang Rong, Stephen Neuendorffer, and Hao Chen. 2022. IRFuzzer: Improving IR Fuzzing with More Diversified Input. Presented at 2022 LLVM Developers' Meeting. https://llvm.org/devmtg/2022-11/slides/Lightning2-ImprovingIRFuzzingWithMoreDiversifiedInput.pdf [Online, accessed 03-16-2023].

Richard L. Sauder. 1962. A General Test Data Generator for COBOL. In *Proceedings of the Spring Joint Computer Conference* (May 1–3, 1962). 317–323. https://doi.org/10.1145/1460833.1460869

Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 849–863. https://doi.org/10.1145/2983990.2984038

Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-guided Program Reduction. In *Proceedings of the 40th International Conference on Software Engineering*. 361–371. https://doi.org/10.1145/3180155.3180236

Yan Wang, Peng Jia, Luping Liu, Cheng Huang, and Zhonglin Liu. 2020. A Systematic Review of Fuzzing Based on Machine Learning Techniques. *PlOS One* 15, 8 (2020), e0237749. https://doi.org/10.1371/journal.pone.0237749

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)*. Association for Computing Machinery, 283–294. https://doi.org/10.1145/1993498.1993532

Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. 2021. An Empirical Study of Optimization Bugs in GCC and LLVM. *Journal of Systems and Software* 174 (2021), 110884. https://doi.org/10.1016/j.jss.2020.110884

Received 2022-11-10; accepted 2023-03-31