Program Reconditioning: Avoiding Undefined Behaviour When Finding and Reducing Compiler Bugs

BASTIEN LECOEUR, Imperial College London, UK
HASAN MOHSIN, Imperial College London, UK
ALASTAIR F. DONALDSON, Imperial College London, UK

We introduce *program reconditioning*, a method for allowing program generation and differential testing to be used to find miscompilation bugs, and test-case reduction to be used to simplify bug-triggering programs, even when (a) the programming language of interest features undefined behaviour (UB) and (b) no tools exist to detect and avoid this UB. We present two program generation tools based on our reconditioning idea: GLSLsmith for the OpenGL Shading Language (GLSL), a widely-used language for graphics programming, and WGSLsmith for the WebGPU Shading Language (WGSL), a new language for web-based graphics rendering. GLSL features many UBs, but unlike for languages such as C and C++ no tools exist to detect them automatically. While the WGSL language *specification* features very limited UB, early WGSL implementations *do* exhibit UB, for reasons of initial implementation simplicity, making it challenging to test them to quickly detect and eliminate unrelated miscompilation bugs. Thanks to reconditioning, we show that GLSLsmith and WGSLsmith allow differential testing and test-case reduction to be applied to compilers for GLSL and WGSL for the first time, despite the unavailability of UB detection techniques for these languages. Through a large testing campaign, we have found 24 and 33 bugs in GLSL and WGSL compilers, respectively. We present experiments showing that when reconditioning is *disabled*, compiler testing leads to a high rate of test programs that *appear* to trigger miscompilation bugs, but actually just feature UB. We also present a novel approach to managing floating-point roundoff error using reconditioning, implemented for both GLSL and WGSL.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Software testing and debugging**.

Additional Key Words and Phrases: Randomised testing, test-case reduction, compiler testing, undefined behaviour, OpenGL, WebGPU

ACM Reference Format:
Bastien Lecoeur, Hasan Mohsin, and Alastair F. Donaldson. 2023. Program Reconditioning: Avoiding Undefined Behaviour When Finding and Reducing Compiler Bugs. *Proc. ACM Program. Lang.* 7, PLDI, Article 180 (June 2023), 25 pages. https://doi.org/10.1145/3591294

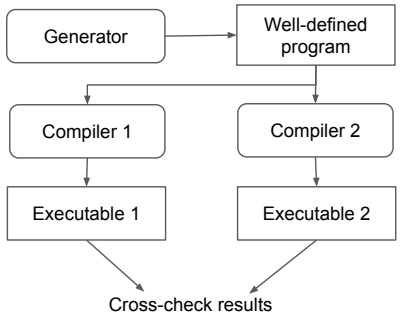
1 INTRODUCTION

Random differential testing [McKeeman 1998] has been shown to be very effective at finding *miscompilation* bugs, where a compiler generates incorrect code for a given source program. For example, the Csmith [Yang et al. 2011] and YARPGen [Livinskii et al. 2020] tools have used differential testing to find hundreds of miscompilations in GCC and LLVM.

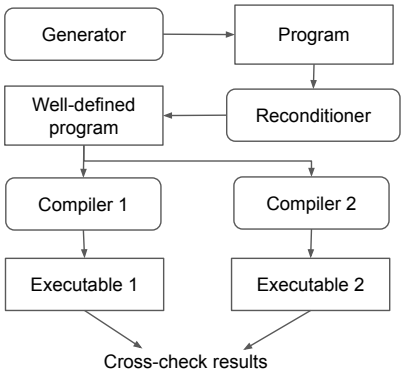
A miscompilation is identified when a generated program yields different execution results after it is compiled by different compilers (or by one compiler at multiple optimisation levels): a mismatch indicates that at least one of the compilers has miscompiled the program. However, in a language

Authors' addresses: Bastien Lecoeur, Department of Computing, Imperial College London, UK, bastien.lecoeur20@imperial.ac.uk; Hasan Mohsin, Department of Computing, Imperial College London, UK, hasan.mohsin18@imperial.ac.uk; Alastair F. Donaldson, Department of Computing, Imperial College London, UK, alastair.donaldson@imperial.ac.uk.

2475-1421/2023/6-ART180
https://doi.org/10.1145/3591294

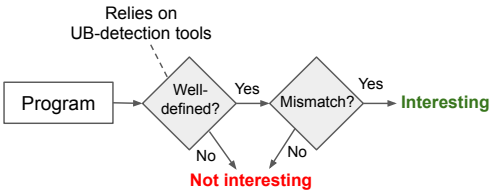


(a) Standard approach, where logic for ensuring UB-freedom is baked into the generator

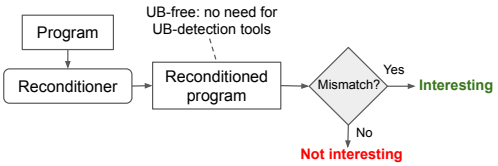


(b) Extracting logic for ensuring UB-freedom into a separate *reconditioning* step

Fig. 1. Differential compiler testing, without and with reconditioning



(a) Standard approach, which relies on external UB-detection tools



(b) Reconditioning the program avoids the need for UB-detection tools

Fig. 2. Interestingness test for test-case reduction, without and with reconditioning

that features *undefined behaviour* (UB), this is only true if the generated program is *well-defined*, i.e. guaranteed (according to the language semantics) to compute a deterministic result. Program generators such as Csmith and YARPGen incorporate careful mechanisms to ensure that generated programs are well-defined by construction. This differential testing setup is illustrated in Figure 1a: notice that the *generator* component produces a well-defined program.

As randomly-generated programs tend to be too large and complex for human developers to understand, an *automated test-case reduction* tool is an essential component of a randomised compiler testing setup. Reducing a program that triggers a miscompilation involves systematically attempting to simplify the program subject to an *interestingness test* that must check two conditions:

(1) the program remains well-defined;
(2) the program continues to trigger a mismatch with respect to the compilers under test.

An interestingness test based on these conditions is illustrated in Figure 2a. Condition 2 is clearly essential: a mismatch is what indicates that a miscompilation has occurred. But Condition 1 is *also* essential: without it, test-case reduction may yield a small program that leads to a result mismatch between compilers because it triggers UB, and *not* due to a genuine miscompilation.

A standard approach for ensuring Condition 1 is to invoke one or more external UB-detection tools. This reliance on UB-detection tools is acceptable for mainstream languages such as C and

C++, for which mature tools (e.g. AddressSanitizer [Serebryany et al. 2012]) are available. However, such tools do not exist for more niche or emerging languages, and might take many person years or decades to build. One such language is the OpenGL Shading Language (GLSL), which features a range of UBs but no UB-detection tools. Another is the WebGPU shading language (WGSL), part of the new WebGPU standard for in-browser graphics rendering [W3C 2023b]. According to the WGSL language specification [W3C 2023a], the language has relatively few UBs. However—for reasons we elaborate on in §2.2—current implementations of WGSL generate code in downstream target languages in a naive fashion that may trigger UBs in those target languages. This means that WGSL *as currently implemented* features a number of UBs. It would clearly be a waste to build tools for detecting these *temporary* UBs, as they will disappear when WGSL compilers mature. Still, it is desirable to deploy randomised compiler testing early, to quickly find miscompilation bugs in WGSL compilers, and the presence of temporary UBs makes this difficult.

The problem. The problem we address is: How can we apply differential testing and test-case reduction to languages that feature UB (or whose current implementations feature UB) in the *absence* of UB-detection tools, to yield minimised UB-free programs that trigger miscompilations?

Our solution: reconditioning. Our idea for working around the absence of UB-detection tools is very simple. Usually, the program generator component of a differential compiler testing setup contains logic to ensure that a program is well-defined by construction (as shown in Figure 1a). We propose extracting this logic into its own tool: a source-to-source translator that takes in an arbitrary program that *might* exhibit UB, and emits a program that *definitely does not* exhibit UB. We call this tool a *reconditioner*, and it leads to the revised differential testing workflow of Figure 1b.

This change has little observable impact on program generation: well-defined programs are used for differential testing either way. (In §3 we explain why we nevertheless apply reconditioning during differential testing, rather than having the generator produce well-defined programs directly.) However, the extraction of UB-avoidance logic into a reconditioner allows a different take on test-case reduction. Instead of relying on external tools to detect UB (Figure 2a), the interestingness test can invoke the reconditioner to *eliminate* any UB in the candidate reduced program before it is used to check for a mismatch between compilers (Figure 2b). The reconditioned program is UB-free by construction, so there is no need for external UB-detection tools. This makes the approach ideally suited for languages (like GLSL and WGSL) where such tools do not exist.

The test-case reducer yields a small program that, upon reconditioning, triggers a miscompilation. The reconditioned program can be provided as a reproducing test case in a bug report.

At first sight it may appear that building a reconditioner is as hard as building a UB-detection tool. However, a reconditioner merely needs to rewrite its input program so that *potential* UBs are eliminated. This is a much easier task than detecting the presence of actual UBs that will be triggered when the program runs. As an example, consider integer division by zero—a common UB. A reconditioner can reliably and trivially *avoid* this UB by replacing every expression of the form e_1/e_2 with a conditional expression of the form $(e_2 == 0\ ?\ e_1 : e_1/e_2)$. In contrast, dynamically detecting actual divisions-by-zero would require more sophisticated runtime instrumentation, which would be particularly hard to put into practice for languages that target unconventional processors such as GPUs.

Contributions. We make the following contributions. **(1)** We introduce the simple idea of reconditioning as a way of avoiding the need for UB-detection tools during compiler testing. **(2)** We show that reconditioning is *practical* by building reconditioning-based compiler testing tools, GLSLsmith and WGSLsmith, for two real-world programming languages, GLSL and WGSL, bringing differential testing to these languages for the first time. The engineering effort required to build the reconditioning steps for these tools was not high, and we argue that it is far lower than the effort that

would have been required to build full-blown UB-detection tools. **(3)** We present a novel method for reconditioning programs that use floating-point arithmetic to avoid roundoff error. **(4)** We present a large testing campaign and experimental evaluation showing that GLSLsmith and WGSLsmith are effective at finding bugs in WGSL an GLSL compilers (leading to the detection of 24 GLSL and 33 WGSL compiler bugs, 18 and 13 of which have been fixed already), and that reconditioning is an essential part of the success of these tools: when reconditioning is disabled, either during program generation or during test-case reduction, differential compiler testing frequently leads to false alarms due to programs that trigger UB.

Paper structure. After providing necessary background (§2) we give a more detailed overview of our reconditioning idea (§3). We then describe how we have put reconditioning into practice for GLSL and WGSL, including our novel method for handling floating-point operations (§4), and present our reconditioning-based compiler testing tools, GLSLsmith and WGSLsmith (§5). We present results from an uncontrolled bug-finding campaign using these tools, and controlled experiments assessing the extent to which reconditioning avoids UB (§6). After a discussion of related work (§7) we summarise our contribution and discuss ideas for future work (§8).

2 BACKGROUND

For ease of presentation, we regard the input on which a program will be executed as being part of the program. We also restrict attention throughout to single-threaded programs. Even though GPUs exhibit massive parallelism, correct compilation of the individual threads that execute a GPU program is vital for overall correctness.

We use *undefined behaviour* (UB) to refer to any behaviour that prevents the program from producing a deterministic, implementation-independent result. This includes "catch fire" UB in C-family languages (e.g. division by zero), unspecified behaviour (e.g. evaluation order for function arguments in many languages), and implementation-defined behaviour (e.g. floating-point roundoff in graphics shader programs). We say that a program is *well-defined* if it does not trigger UB, according to this broad definition, when executed.

2.1 GLSL and Its Undefined Behaviours

The OpenGL Shading Language (GLSL) [Kessenich 2019; Simpson and Kessenich 2019] is a language for expressing graphics and compute workloads to run on GPU devices, and is associated with the OpenGL [Segal and Akeley 2019] and OpenGL ES [Leech 2019] APIs. GLSL programs, called *shaders*, can be broadly divided into *graphics* shaders, used for rendering images, and *compute* shaders, which perform GPU computations on data buffers in a manner similar to kernels in CUDA [Nvidia 2021] and OpenCL [Khronos Group 2023b].

Undefined behaviour in GLSL. Through a careful study of the GLSL language specification, we have categorised all the types of UB that the language features. We summarise these (excluding concurrency-related UBs), noting key differences compared with UB in C.

Arithmetic: GLSL features several arithmetic UBs with counterparts in C, e.g. integer division by zero, division of `INT_MIN` by -1, modulo operations where either argument is negative, and over-shifting. Unlike in C, signed integer overflow is *well*-defined (having 2s-complement semantics), and the `abs` function is well-defined for all values (`abs(INT_MIN)` is defined as `INT_MIN`).

Built-ins: Many GLSL-specific built-in operations have associated UB, e.g. the `clamp`(a, b, c) function, which clamps `a` into the range $[b, c]$, yields an undefined result if $b > c$.

Floating-point: Floating-point arithmetic is based on the IEEE-754 standard [IEEE 1985], but unlike in C, GLSL implementations have a lot of freedom to deviate from this standard. For example: GLSL does not allow the floating-point rounding mode to be controlled nor queried; whether denormalised

values are represented or flushed to zero is implementation-defined; and the language specification states that "implementations are permitted to perform such optimisations that effectively modify the order or number of operations used to evaluate an expression, even if those optimisations may produce slightly different results relative to unoptimized code" [Kessenich 2019, pages 97 and 103]. This means that the results of inexact floating-point operations, for which the real-valued result of an operation is not an exact floating-point number, may vary across platforms.

Uninitialised and out-of-bounds accesses: Accessing uninitialised variables leads to UB (unlike in C, global variables are not default-initialised). Undefined behaviour is also triggered when arrays, vectors and matrices are accessed out-of-bounds.

Order of evaluation: Though evaluation order for function input parameters is defined in GLSL (left-to-right), GLSL features output function parameters. The order in which these are written, and the evaluation order of most operators and of array and structure initialisation expressions, is undefined. Care must be taken to avoid UB if such constructs feature side-effecting expressions.

Infinite loops: GPU programs are expected to terminate, to avoid the displays of devices from freezing up. Thus infinite or very long-running loops constitute a form of UB in GLSL: the OS watchdog detects when a GPU workload is hogging resources and kills the GPU process, leading to unpredictable results [Sorensen and Donaldson 2016].

Lack of UB-detection tools for GLSL. The glslang reference front-end [Khronos Group 2023a] checks whether a given GLSL program is syntactically correct and well-typed, but does not give insight into whether dynamic UB might be triggered. To our knowledge, no UB-detection tools exist for GLSL, making it a prime target language for reconditioning, which avoids the need for UB-detection tools during test-case reduction.

UB-detection tools for C do not help. Although GLSL has C-like syntax, it is neither a superset nor a subset of C: it is a distinct language with many syntactic and semantic differences. This means that the rich set of UB-detection tools for C cannot be used to detect UBs in GLSL.

2.2 WGSL and Its Undefined Behaviours

The WebGPU Shading Language (WGSL) [W3C 2023a] is a recent graphics shading language designed to work with WebGPU [W3C 2023b], a new API for web-based graphics rendering. WGSL is similar in essence to various other graphics shading languages, including GLSL (see §2.1) and Microsoft's High Level Shading Language [Microsoft 2019], featuring both graphics and compute shaders. However, it has an entirely new syntax and its own semantic peculiarities. As with GLSL, WGSL does not include dynamic memory allocation, and array sizes are always known.

WebGPU is a JavaScript API that exposes various graphics- and compute-related functionality. Instead of requiring GPU makers to support WebGPU natively in their device drivers, the intention is that browser developers will implement WebGPU on top of lower-level native graphics APIs, such as OpenGL ES [Leech 2019] or Vulkan [Group 2019] on Android, Vulkan on Linux, Direct3D [Microsoft 2021] on Windows, and Metal [Apple 2023] on iOS and OSX. Consequently, a WebGPU implementation must feature a compiler that ingests WGSL and emits code in the relevant shading language for the native graphics API: GLSL for OpenGL, SPIR-V [Kessenich et al. 2022] for Vulkan, HLSL for Direct3D and the Metal Shading Language [Apple 2022] for Metal. The generated code can then be compiled by the GPU driver of the client platform.

Two main WGSL compilers are being developed at present: tint from Google, used in their Dawn implementation of WebGPU for Chromium [Google 2023b], and naga from Mozilla, used in their wgpu implementation of WebGPU for Firefox [Rust Graphics Mages 2023].

Fundamental undefined behaviour in WGSL. The WGSL specification takes care to give fully-defined semantics to most operations in the language. However, there are three behaviours

that are not well-defined: out-of-bounds array accesses, infinite loops and floating-point roundoff errors; these lead to UB for similar reasons as in GLSL (see §2.1).

Unlike in GLSL, uninitialised variables have a well-defined default value in WGSL (e.g., 0 and false for integers and booleans, respectively).

Temporary undefined behaviour in WGSL. The WGSL specification provides clear semantics for various arithmetic operators that have UB edge cases in many other languages. For example, it mandates 2s complement semantics for signed arithmetic, and specifies that division-by-zero yields the value of the numerator. However, at present neither of the two WGSL compilers, tint and naga, generate code that respects these semantics. Instead, they naively compile WGSL arithmetic operations into corresponding operations in the downstream shading languages that they target. Because these target languages *do* feature arithmetic UBs (e.g., division-by-zero is undefined in all downstream languages), this means that WGSL *as currently implemented by tint and naga* does feature various arithmetic UBs. We refer to these kinds of UBs as *temporary* UBs. Our experimental evaluation (see §6) is performed with respect to versions of tint and naga that suffer from temporary UB. Rectifying this situation is on the roadmap for both tint and naga, but in the meantime it is desirable to be able to apply compiler testing techniques to these tools, to allow early detection of unrelated bugs. Thus it is important to have a means of working around this temporary UB.

Lack of UB-detection tools for WGSL. The tint and naga compiler front-ends can be used to check whether a WGSL program is syntactically valid and well typed, but no UB-detection tools for WGSL exist. In terms of a long-term investment, it would clearly be a waste of time to build UB-detection tools for the *temporary* UBs discussed above. As we explain in §3 and §4, reconditioning provides a solution that avoids the need to build such tools.

2.3 Test-case Reduction and UB-detection Tools

Test-case reducers are usually variations on the widely-used delta debugging algorithm [Zeller and Hildebrandt 2002], and are parameterised by an *interestingness test* [C-Reduce Project 2023], provided by the user of the test-case reduction tool, which determines whether a test case triggers the bug of interest. The reducer repeatedly tries smaller or simpler test cases, using the interestingness test to determine whether they trigger the bug. When a simpler test case is deemed interesting it is recorded as the best test case seen so far, and the reducer proceeds to attempt to simplify it further.

The burden of UB-detection tools. When reducing a C/C++ miscompilation using the C-Reduce [Regehr et al. 2012] tool, the recommendation is to write an interestingness test that uses several program analysis tools to check for potential UB [C-Reduce Project 2023]. This leads to an interestingness test of the form illustrated in Figure 2a. The analyses that are typically used when deploying C-Reduce in practice include: compiling the program with pedantic warnings enabled, rejecting the program if specific warnings (e.g. related to uninitialised memory) are issued; interpreting the program using Frama-C [Cuoq et al. 2012] with various safety checks enabled; and running the program after adding compile-time instrumentation using AddressSanitizer [Serebryany et al. 2012] to detect buffer overflows. We estimate that this suite of external UB-detection tools has taken several person decades to construct, and this time has only been invested because C and C++ are among the most widely-used programming languages in the world.

Such analysis tools are unlikely to be available for less widely-used (yet still important) languages. As discussed in §2, UB-detection tools are *not* available for the GLSL and WGSL language that we study in this paper. Furthermore, there are good reasons for wishing to apply compiler testing to WGSL implementations as soon as possible, but the *temporary* UB exhibited by current WGSL compilers—for which building UB-detection tools would be a poor long-term investment—make this challenging without an alternative solution.

3 RECONDITIONING: THE IDEA

We now elaborate on the overview of our reconditioning approach presented in §1.

A program generator such as Csmith [Yang et al. 2011] is designed with a built-in "box of tricks" for avoiding UB. This leads to generated programs that are UB-free by construction, as shown in in Figure 1a. When the time comes to *reduce* a miscompilation bug, the test-case reducer starts with a well-defined program, provided by the generator thanks to its box of tricks. Thereafter the reducer must take care to ensure that the simpler programs it considers *remain* well-defined (or, rather, they should be discarded as uninteresting if they trigger UB). This has traditionally required the use of external UB-detection tools, as illustrated by the interestingness test illustrated in Figure 2a. As explained in §2.3, this is a barrier to reliable automated reduction of miscompilation bugs in languages where UB-detection tools are not available.

The key idea behind reconditioning is to *extract* the generator's box of tricks into a stand-alone *reconditioner* component that turns a program with potential UB into one that is definitely well defined. This reconditioner can then be used *during test-case reduction* to avoid UB *without the need for external UB-detection tools*, via the interestingness test illustrated in Figure 2b.

Reconditioning as a just-in-time transformation before cross-checking. The use of reconditioning changes the compiler testing and test-case reduction workflows slightly, so that (a) the generator yields a non-reconditioned program that *might* trigger UB, (b) the reducer attempts to simplify a non-reconditioned program that *might* trigger UB, but (c) reconditioning is *always* applied to a program before it is used to cross-check a pair of compilers, so that a compiler is *always* presented with a reconditioned program that does *not* trigger UB. Point (a) is illustrated by Figure 1b: the generated program is reconditioned before cross-checking occurs. Point (b) is illustrated by Figure 2b: the reducer provides a program that might exhibit UB to the interestingness test, but the interestingness test reconditions the program before cross-checking it against the compilers under test. If a bug has been found by a program $Q = \text{recondition}(P)$ (the reconditioned version of a program P), then reducing the *non*-reconditioned program P, but reconditioning just-in-time as part of the interestingness test, ensures that the shader initially compiled and executed during reduction is exactly the same program that found the bug. Applying test-case reduction to Q, and reconditioning during the interesting test, would lead to $\text{recondition}(Q) = \text{recondition}(\text{recondition}(P))$ being considered, which might not trigger the miscompilation of interest.

The reconditioned reduced test case should be used to file a bug report. With reconditioning, test-case reduction no longer yields a minimised program that triggers a miscompilation bug. Instead, it yields a minimised program that triggers a miscompilation bug *after it has been reconditioned*. The reconditioned version of the minimised program should thus be provided when the miscompilation bug is reported to developers, as the non-reconditioned version might trigger UB.

Writing a reconditioner is easier than writing a UB-detector. A reconditioner has the freedom to *change* the test program to work around *possible* UB. We give several examples of this in §4. For the GLSL and WGSL languages we have studied, the required changes are very straightforward. For example, uninitialised variable accesses can be avoided via a trivial pass that modifies every variable declaration that does not have an initialiser so that the variable is initialised to some default value. In contrast, writing a UB-detection tool to accurately identify uninitialised memory accesses—such as the MemorySanitizer tool for C/C++ [Stepanov and Serebryany 2015]—is a big undertaking.

We cannot accurately measure the difference in effort required to build a reconditioner for GLSL or WGSL compared with the effort required to build detection tools for all of the UBs we recondition against; this would require going and building those tools, which would be self-defeating. We argue throughout §4 why the effort required to implement reconditioning for each category of UB is substantially lower than the effort that would be required to build an associated detection tool.

Suitability of languages as targets for reconditioning. Reconditioning is intended to help with compiler testing for languages that feature UB, but for which mature UB-detection tools do not exist, rather than for languages such as Java (which avoids UB by design), or C/C++ (which is well served by UB-detection tools).

In the remainder of the paper we show that reconditioning works well for GLSL and WGSL, which are both complex languages with many fundamental UBs, in the case of GLSL (§2.1), or temporary UBs with respect to current implementations, in the case of WGSL (§2.2). GLSL and WGSL share a lot in common with other graphics programming languages such as SPIR-V [Kessenich et al. 2022] (used by the Vulkan and OpenCL programming models) and HLSL [Microsoft 2019] (the Direct3D shading language). We expect reconditioning to also work well for these important languages.

Graphics languages typically feature limited use of pointers and dynamic allocation. Leaving aside the fact that C/C++ are well-served by numerous UB-detection tools, reconditioning full-blown C/C++ programs might require very intrusive changes (e.g. adding machinery to drag around information about pointer targets), or sophisticated analyses (defeating the aim of reconditioning). Reconditioning could certainly be applied to sizeable *fragments* of these languages, for which necessary pointer analysis is easy. All compiler testing tools we are aware of are restricted to language fragments to some degree.

Furthermore, reconditioning and the use of UB-detection tools do not have to be mutually-exclusive. For some languages it may be the case that detection tools are available for various UBs, and reconditioning can be put in place to handle others. For example, we are not aware of scalable off-the-shelf tools for detecting when floating-point roundoff error occurs in C/C++ programs. The reconditioning approach for floating-point that we present in §4.3, which modifies a program to eliminate roundoff error for particular floating-point operators, could be leveraged to aid in testing aggressive floating-point optimisations in C/C++ compilers, with external UB detection tools used to handle UBs unrelated to floating-point.

Applications of reconditioning beyond UB. Recall from §2.2 that we use reconditioning to work around the fact that current WGSL implementations feature *temporary* UB. A related potential application of reconditioning is as a means of working around known compiler bugs. If a known bug is triggered by a particular source code idiom, a reconditioning step could be temporarily introduced to rewrite this idiom away, making it more likely that randomised testing will reveal other bugs, and preventing test-case reduction from "slipping" [Chen et al. 2013] to the known bug.

4 RECONDITIONING FOR GLSL AND WGSL

We now provide details of reconditioning steps for avoiding all the kinds of UB exhibited by the GLSL and WGSL languages, as summarised in §2.1 and §2.2, including the temporary UBs exhibited by current implementations of WGSL. We group these reconditioning steps into three categories: (1) reconditioning of arithmetic and built-in operators, which are very similar to the measures the Csmith program generator [Yang et al. 2011] takes to avoid UB when generating programs (§4.1); (2) reconditioning steps we have specifically designed for handling GLSL and WGSL, but that are conceptually very straightforward (§4.2); (3) a more sophisticated reconditioning step that avoids roundoff error for a number of floating-point arithmetic operators (§4.3). The fact that these reconditioning steps are very simple is key to our proposal: it means the engineering overhead associated with building a reconditioner is low; we estimate much lower than the overhead that would be required to build associated UB detectors.

In what follows, we provide illustrative examples for GLSL; the details for WGSL are similar.

```
int SAFE_DIV(int A, int B) {
  return B == 0 || A == INT_MIN && B == -1 ? A / 2 : A / B;
}

ivec3 SAFE_DIV(ivec3 A, ivec3 B) {
  return any(equal(B, ivec3(0))) || any(equal(A, ivec3(INT_MIN))) && any(equal(B, ivec3(-1)))
      ? A / ivec3(2) : A / B;
}
```

Fig. 3. Safe wrappers for scalar and vector operations

4.1 Reconditioning for Arithmetic and Built-in Operators (Inspired by Csmith)

Like Csmith, our reconditioners for GLSL and WGSL avoid arithmetic UB by rewriting potentially-dangerous arithmetic operations and built-in function calls with calls to special *wrapper* functions. In the case of WGSL, this suffices to take care of the *temporary* UBs discussed in §2.2, and the use of wrapper functions can be gradually reduced as compilers for WGSL mature.

As an example, a GLSL scalar or vector integer division expression e_1/e_2 is rewritten as a call to SAFE_DIV(e_1, e_2), implementations of which are shown in Figure 3 for integer and 3D integer vector types. A wrapper must be designed to generate *some* well-defined result in the case where the desired operation is unsuitable, but it does not matter which well-defined result.

We have written wrapper functions for all the GLSL operators and built-in functions that have associated UB, and for relevant WGSL operators and built-in functions that exhibit temporary UB. The reconditioner adds the definitions of all required wrapper functions at the start of the program.

The difference between the way Csmith and our tools use arithmetic wrappers is that Csmith emits wrappers when a program is generated, while the reconditioners of GLSLsmith and WGSLsmith add wrappers as a program transformation step. This allows arithmetic wrappers to eliminate UBs in programs that arise both from program generation and during test-case reduction.

Dynamically detecting arithmetic UBs would involve building tooling similar to the UndefinedBehaviorSanitizer tool for C/C++ [LLVM 2023], which uses runtime instrumentation to dynamically check for occurrences of dangerous operations such as division by zero. It should be clear that building a tool such as UndefinedBehaviorSanitizer from scratch would require a much larger engineering effort than defining suitable wrapper functions and building a transformation pass that replaces operators with calls to wrapper functions.

A minor drawback of inserting wrapper functions is that they form part of the final, reconditioned test case that is used to report a miscompilation bug. However, because our wrapper functions are simple, this has not impeded the effective reporting of bugs in practice (see §6.1). We study the size overhead of reconditioning in general—much of which comes from wrapper functions—in §6.2.

4.2 Other Straightforward Reconditioning Steps

We devised the following reconditioning steps in response to various GLSL and WGSL UBs. It should be easy to adapt these steps to work for languages with similar UBs.

Index bounding. To ensure that all array, vector and matrix indexing expressions are in bounds, each indexing expression e is replaced with an expression that constrains e to ensure that it falls within bounds. In the case of GLSL this involves changing e to SAFE_ABS(e) % N, where N is the known length of the array or vector. The SAFE_ABS wrapper is necessary because the % operator is only well-defined when both its arguments are positive, and because GLSL's regular abs built-in returns a negative value when its argument is INT_MIN, which SAFE_ABS avoids. Indexing expressions are reconditioned in WGSL in a similar manner.

```
int f (out int p0, out int p1, int p2);

void main() {
  int x = 0, y = 0, z = 0;
  // Write-write conflict on x (first two
  // parameters)
  f(x, x, x);
  // Write-write conflict on x, read-write
  // conflict on y
  int[4] A = int[4](x += 1, x *= y,
                    y += 2, z += 1);
  // Various conflicts on x
  x++ + ++x;
}
```

(a) Before reconditioning

```
int f (out int p0, out int p1, int p2);

void main() {
  int x = 0, y = 0, z = 0;

  int t1 = x;
  f(x, t1, x);

  int t2 = (x *= y);
  int[4] A = int[4](x += 1, t2,
                    y += 2, z += 1);
  int t3 = (++x);
  x++ + t3;
}
```

(b) After reconditioning

Fig. 4. Reconditioning to avoid undefined behaviour related to evaluation order

Without reconditioning, dynamically detecting out-of-bounds accesses could be achieved via a compile-time bounds-checking instrumentation using shadow memory, such as that implemented by AddressSanitizer for C/C++ [Serebryany et al. 2012]. While the more restricted use of pointers in GLSL and WGSL might make it easier to build such a tool for these languages compared with for C/C++, it would clearly be a much larger undertaking than writing a simple pass that constrains index expressions in the manner proposed above.

Initialisation enforcement. Accessing uninitialised data leads to UB in GLSL (§2.1), but not in WGSL (§2.2). Reconditioning to enforce initialisation is trivial: all data for which initialisers are lacking are set to 1 (integer), 1.0 (floating-point) or true (boolean). It might appear that the choice of initialiser could lead to knock-on UBs, e.g. if a variable `x` is initialised to 1, and then the expression `x - 1` is used as the second argument to the `/` operator, but this cannot occur since all relevant arithmetic operators are reconditioned via wrapper functions (see §4.1).

Without reconditioning, detecting accesses to uninitialised memory would require sophisticated dynamic analysis, such as that provided by the MemorySanitizer tool [Stepanov and Serebryany 2015]; prior work adapting this kind of analysis to support test-case reduction for OpenCL kernels proved to be a major undertaking [Pflanzer et al. 2016]. Clearly implementing a pass that adds a default initialiser expression to every uninitialised variable is a much smaller undertaking than building a dynamic analysis for uninitialised memory from scratch.

Loop limiters. Recall from §2.1 and §2.2 that extremely long-running or infinite loops lead to UB in GLSL and WGSL. The loops that GLSLsmith and WGSLsmith generate have a high chance of being non-terminating, since their exit conditions are arbitrary generated expressions. Even if these generation tools took steps to ensure termination of generated loops, it is very likely that this guarantee would be spoiled during test-case reduction due to the reducer removing parts of the bodies of loops that are needed to ensure termination.

Our GLSL and WGSL reconditioners rein in loop execution using per-loop *limiters*. A loop's limiter is a zero-initialised counter variable that is incremented at the start of each loop iteration and immediately tested against a constant upper bound; the loop breaks if this upper bound is reached. Loop limiter logic is inserted at the start of each loop body to ensure that `continue` statements in the loop cannot bypass the limiter.

A possible issue with the use of loop limiters is that they may suppress optimisations such as loop unrolling, by making it hard to statically determine the number of times that a loop will execute.

Controlling evaluation order. Evaluation order is well-defined in WGSL, but can lead to UB in GLSL (see §2.1). The reconditioner must guard against side-effecting operations occurring in undefined orders leading to UB, and output (`out` and `inout`) function parameters committing their results to call-site l-values in undefined orders. It would be easy but overly restrictive to completely eliminate the use of side-effecting expressions in contexts where evaluation order is undefined. Instead, the reconditioner exploits the fact that GLSL does not feature pointers and aliasing to perform a precise conflict analysis. The analysis works by examining the arguments of an operator left-to-right and considers three scenarios for a variable v: **write after write**: v is written more than once, all but the first modifying expression are extracted into temporaries; **write after read**: v is written to after being read, all modifying expressions are extracted into temporaries; **read after write**: v is read after being written, all reading expressions are extracted into temporaries. This analysis operates on a per call site basis; inter-procedural analysis is not required.

Figure 4 shows a `main` function before and after this reconditioning, where `main` calls another function `f` whose first two arguments are `out` parameters. Notice that the reconditioned `main` still features a number of side-effecting arguments to expressions; i.e. reconditioning is not too restrictive. As discussed in §6.1, we have found bugs in GLSL compilers that depend on side-effecting expressions, so the effort to avoid being too restrictive has been worthwhile.

We are not aware of sanitiser tools for other languages that directly detect evaluation ordering issues, though the CompCert interpreter has a mode in which evaluation order is randomised, which can be useful for identifying when a C program is sensitive to evaluation order [Leroy 2023]. While requiring a more intricate rewriting than the other reconditioning steps we have presented, our method for controlling evaluation order is still fairly straightforward, and probably easier to implement compared with adapting some existing GLSL compiler to use a random evaluation order.

4.3 Reconditioning to Avoid Floating-point Roundoff Error

We now explain how GLSL and WGSL shaders that feature floating-point operations are reconditioned to avoid roundoff error. We use "floating-point" to refer to 32-bit floating-point numbers, but the approach we present can be applied to other floating-point types.

Overview. We give the intuition for our approach by explaining how a program that only uses floating-point addition (and no other floating-point operators) could be reconditioned to avoid roundoff error. Let $+_F$ and $+_R$ denote floating-point and real addition, respectively. There are many pairs of floating-point numbers (x, y) for which $x +_R y$ is also a floating-point number. In such cases, the IEEE-784 standard [IEEE 1985] enforces that $x +_F y = x +_R y$—i.e., floating-point addition admits no possibility of roundoff error.

Suppose we can find a non-empty set of floating-point numbers S such that for any $x, y \in S$ it is possible to determine reliably, using only floating-point arithmetic, whether $x +_F y \in S$. Then we can recondition a program as follows:

- Change every floating-point program input value $I \notin S$ to some value $I' \in S$, and every floating-point literal $L \notin S$ appearing in the program to some literal $L' \in S$;
- Replace every floating-point addition expression $e_1 + e_2$ with $\texttt{MAKE\_IN\_RANGE}(e_1 + e_2)$, where $\texttt{MAKE\_IN\_RANGE}(x)$ returns x if $x \in S$, and some *default* element of S otherwise.

Implementation as a reconditioning step. Our reconditioners for GLSL and WGSL implement the above idea for the `+`, `-` and `*` floating-point operations, taking the set S to be a set of contiguous *integer* values that can be represented precisely in floating-point from, excluding the value 0:

$$S = \{x \in \mathbb{Z} \mid -2^{24} + 1 < x < 2^{24} - 1\} \setminus \{0\}.$$

```
float MAKE_IN_RANGE(float x) {
  return (abs(x) < 0.1 || abs(x) >= 16777216.0)
    ? 10.0 /* A default in-range value */ : x;
}
```

Fig. 5. Wrapper function to bring a floating-point value back into the well-defined range

The fact that every number in this integer subset S can be represented in floating-point follows from basic properties of the representation. This is *almost* the largest contiguous range of integer values that can be precisely represented in floating-point: the values -2^{24} and 2^{24} also have a precise floating-point representation; we discuss below why these values and 0 are deliberately omitted.

For any integers $x, y \in S$, and any operator $\circ \in \{+, -, *\}$, it is easy to show that whenever $z = x \circ y \notin S$ then either: $z = 0$, $z \leq -2^{24}$, or $z \geq 2^{24}$. That is, computation can only leave S by hitting zero or an integer whose magnitude is 2^{24} or greater. Since the integers -2^{24} and 2^{24} *also* have a precise floating-point representation, floating-point comparisons can precisely determine whether $z \in S$ despite the fact that the integer z might not be representable in floating-point. It is for this reason that -2^{24} and 2^{24} are excluded from S: their absence from S, but the fact that they have a precise floating-point representation, makes them suitable for detecting departure from S.

The `MAKE_IN_RANGE(`x`)` wrapper is shown in Figure 5. The check `abs(x) < 0.1` is used to exclude 0 from the set S (there is nothing special about the use of 0.1 here; any positive floating point number less than 1.0 would work). We exclude 0 because it has two floating-point representations: positive and negative. Legitimate implementation-defined behaviour related to the sign of 0 can yield different results across platforms if a shader makes a control flow decision that compares a floating-point expression with zero. As an experiment we tried including 0 in the set S, and found that we indeed got false alarm miscompilation bug reports when fuzzing using GLSLsmith.

Instead of defining S in terms of multiples of 1, we could have used multiples of 2^i for any $-126 \leq i \leq 103$ (the justification for this follows from details of the floating-point representation). We chose to use multiples of 1 because then every member of S can be precisely represented as an integer, allowing us to support casting operations between integer and floating point types, as well as the floating-point increment and decrement operators.

Our approach extends to floating-point vectors, and to a variety of built-in functions that are based on addition, subtraction and multiplication (such as the fused multiply-add builtin, `fma`); the reconditioners of §5 support such operations.

Limitations. Our floating-point reconditioning method supports a limited set of operators, and does not cater for e.g. floating-point division or numerous GLSL and WGSL floating-point built-ins. By restricting to integers it avoids certain interesting parts of the floating-point range, e.g. values very close to zero. These limitations might limit bug-finding ability, but the implementation-defined floating-point semantics of GPU programming languages (see §2.1) make generating well-defined programs that use such features—whether via reconditioning or otherwise—fundamentally difficult.

5 THE GLSLSMITH AND WGSLSMITH COMPILER TESTING TOOLS

We now provide a brief overview of GLSLsmith (§5.1) and WGSLsmith (§5.2), explaining how we have integrated these tools with a number of off-the-shelf test-case reducers (§5.3).

5.1 Overview of GLSLsmith

GLSLsmith is a program generator for GLSL, written in Java. When invoked, GLSLsmith generates a *ShaderTrap* script. ShaderTrap [Google 2023c] is a language and tool from Google that supports running GLSL shaders in a self-contained manner. A ShaderTrap script comprises a series of input

and output buffer declarations, commands to fill buffers with specific input values, one or more shaders to be executed, and commands to assert properties about output buffers after shader execution. ShaderTrap takes care of issuing the OpenGL API calls required for buffer creation, marshalling of data between CPU and GPU, as well as shader compilation and execution. The ShaderTrap file that GLSLsmith generates contains a single, randomly-generated compute shader designed to be executed by one thread. The shader operates on a number of input and output buffers, the sizes and types of which are generated at random.

GLSLsmith generates a shader as an abstract syntax tree (AST), using a set of recursive functions that carry around a program context object. We have designed the generator to produce a syntactically- and type-correct shader in one pass, without the need for backtracking. The AST is then pretty-printed into the ShaderTrap script. GLSLsmith supports declarations and expressions over scalars and vectors of integer and boolean types in full, as well as all built-in functions over these types. The complete set of compound control-flow statements is supported: `if`/`else` and `switch` statements, `while`, `for` and `do-while` loops, and `break` statements inside loops and switch constructs, and `continue` statements in loops can be generated. Floating-point variables and operations are generated in a limited fashion as discussed in §4.3.

Effort required to implement reconditioning in GLSLsmith. The GLSLsmith reconditioner leverages an existing parser for GLSL from the GraphicsFuzz project [GraphicsFuzz project authors 2023], is in turn based on ANTLR [Parr 2023]. At time of writing, the GLSLsmith code base is around 10k lines of Java, of which around 1.8k are devoted to implementing reconditioning. We estimate that the engineering time devoted specifically to reconditioning issues was around 1 person-month.

5.2 Overview of WGSLsmith

WGSLsmith is a program generator for WGSL, written in Rust. WGSLsmith can be invoked to generate a WGSL shader program, along with a JSON file containing data used to initialise input buffers used by the shader. It also includes a *harness* that can be used to run arbitrary WGSL compute shaders. Given a WGSL shader, the harness will invoke the necessary WebGPU calls to create input and output buffers, set up an execution pipeline and run the shaders, finally reading back data from the output buffers. Generated shaders can contain input and output buffers of random sizes and layouts.

Shaders are executed using the two main WebGPU implementations: Dawn [Google 2023b] and wgpu [Rust Graphics Mages 2023]. It is possible to select the specific back-end and physical device to use when executing a shader, and a shader can be executed against multiple such configurations for differential testing, comparing the values of the output buffers.

WGSLsmith generates shaders in a similar manner to GLSLsmith, using a single pass approach that produces syntactically-correct and well-typed shaders. It supports a wide range of language features, including scalar and vector expressions, user-defined types (structs) and functions. Supported control-flow constructs include: `if`/`else` and `switch` statements, `loop`, `while` and `for` loops, `break` statements inside loops and switch constructs, `continue` statements in loops and `fallthrough` statements in `switch` cases. Limited support for float-point is implemented similarly to GLSLsmith, and WGSLsmith additionally has basic support for pointers.

Effort required to implement reconditioning in WGSLsmith. The WGSLsmith uses a Rust parsing library, pest [pest project authors 2023], to parse WGSL programs. At time of writing, the WGSLsmith code base is around 11.2k lines of Rust, of which around 1.8k are devoted to implementing reconditioning. As for GLSLsmith, we estimate that 1 person-month of engineering effort was devoted to implementing reconditioning.

5.3 Integration with Off-the-shelf Reducers

Our reconditioning approach has allowed us to integrate GLSLsmith and WGSLsmith with a number of off-the-shelf test-case reducers, without requiring these reducers to have any in-built knowledge about UB in the WGSL and GLSL languages.

In principle, our tools can be made to work with any test-case reducer that operates on text files and uses an *interestingness test* to decide whether an attempted reduction operator should be deemed successful. In practice, we have integrated our tools with: C-Reduce, a specialised reducer for C/C++ programs [Regehr et al. 2012], which has a "not C" mode that been shown to work remarkably well for programs written in other languages that feature C-like syntax, such as brace-delimited blocks [Regehr 2012]; Perses [Sun et al. 2018], a grammar-based reducer that accepts a description of the grammar of the input format of interest and performs hierarchical delta debugging [Misherghi and Su 2006] with respect to this grammar; and Picire [Picire Project 2012], a state-of-the-art implementation of line-based delta debugging.

Our experience is that all these reducers work well when reconditioning is used, yielding small, easy to understand test programs. We have found Perses performs best with respect to both time taken for reduction and size of fully-reduced test cases. We hence use Perses for test-case reduction in our experiments (§6).

6 EXPERIMENTAL EVALUATION

6.1 Uncontrolled Bug-finding

We now discuss our experience using GLSLsmith and WGSLsmith to find bugs in GLSL and WGSL compilers via differential testing, demonstrating that the tools are effective at bug-finding. Reconditioning has been integrated into these tools from the start, and test-case reduction with reconditioning has been used in the process of triaging all of the bugs that we have reported. This provides evidence that a reconditioning-based solution can be effective for finding and reducing bug-triggering test cases without running into problems of UB.

Implementations under test. Throughout the development of GLSLsmith and WGSLsmith we have used the tools to target the OpenGL and WebGPU implementations summarised in Table 1 (with associated compilers for GLSL and WGSL). In the case of WebGPU, the **Dawn-*** platforms all use the tint compiler to convert WGSL into a downstream shading languages, while the **wgpu-*** platforms all use naga for this purpose. In all cases, the versions of tint and naga that we test exhibit *temporary undefined behaviour* as discussed in §2.2.

In the case of GLSLsmith we target the Vulkan platforms using ANGLE [Google 2023a], which implements OpenGL on top of Vulkan. This range of implementations covers discrete (NVIDIA) and integrated (Intel) GPUs, software renderers (LLVMpipe and SwiftShader), and both open source (non-NVIDIA) and proprietary (NVIDIA) drivers. During testing, each program was cross-checked against every OpenGL implementation, as it was possible to run them all on a single machine.

For WGSLsmith we test the Dawn [Google 2023b] and wgpu [Rust Graphics Mages 2023] implementations of WebGPU from Google and Mozilla, respectively, mainly generating code to target their Direct3D and Vulkan back-ends. In this case, we cross-checked results for all four Windows-based configurations during testing: **Dawn-D3D12**, **Dawn-Vulkan**, **wgpu-D3D12** and **wgpu-Vulkan** in Table 1. We had limited access to an Apple machine, so performed a small amount of end-to-end testing of the Metal back-ends of Dawn and wgpu, cross-checking the **Dawn-Metal** and **wgpu-Metal** configurations of Table 1. We were also able to use the Metal shader validator (available on Windows) to more extensively check that the Metal shading language code emitted by Dawn and wgpu is valid (even though we could not actually run the generated code on Windows).

Table 1. The OpenGL and WebGPU implementations we have tested

OpenGL implementations tested using GLSLsmith (Vulkan is targeted via ANGLE)		
Short name	Version(s)	Details
OpenGL-NVIDIA	*v460.80–v470.57*	NVIDIA OpenGL driver for GeForce GTX 1050Ti GPU
OpenGL-LLVMpipe	*21.1.1–21.1.7*	Mesa3D OpenGL software renderer
Vulkan-Swiftshader	*git revision 3bd2273b*	Google's Vulkan software renderer
Vulkan-Intel MESA	*Mesa3d 21.1.1–21.1.7*	Mesa3D Intel Vulkan driver for i7-8750H integrated GPU
Vulkan-NVIDIA	*v460.80–v470.57*	NVIDIA Vulkan driver for GeForce GTX 1050Ti GPU
WebGPU implementations tested using WGSLsmith		
Short name	Version(s)	Details
Dawn-D3D12	*git revision 2b4df7889*	Dawn with Direct3D12 back-end on Windows
Dawn-Vulkan	*git revision 2b4df7889*	Dawn with Vulkan back-end on Windows
Dawn-Metal	*git revision 2b4df7889*	Dawn with Metal back-end on macOS
wgpu-D3D12	*git revision f4c01052*	wgpu with Direct3D12 back-end on Windows
wgpu-Vulkan	*git revision f4c01052*	wgpu with Vulkan back-end on Windows
wgpu-Metal	*git revision f4c01052*	wgpu with Metal back-end on macOS

Testing approach. Throughout their development, we applied GLSLsmith and WGSLsmith to the OpenGL and WebGPU implementations of Table 1. This uncontrolled bug-finding involved bursts of activity where we would allow differential testing to run overnight or for several days using up-to-date builds of the implementations under test, after which we would reduce a selection of programs that triggered mismatches or crashes, identify a set of reduced programs that we were confident triggered distinct bugs, and report these bugs. In the process we made numerous bug-fixes and enhancements to GLSLsmith and WGSLsmith. We conducted additional bug-finding in response to improvements in our tools, or to bugs in the implementations under test being fixed.

Summary of bugs found. Using GLSLsmith we have found 24 distinct bugs, of which we have reported 19 (we held off reporting all bugs at once to avoid overwhelming compiler developers), of which 18 have been fixed. Using WGSLsmith we have found 33 distinct bugs, of which we have reported 27, of which 13 have been fixed. Based on a combination of manually analysing reduced bug-triggering test cases and feedback from developers to whom we reported these bugs, we are confident that they are all distinct. Table 2 and Table 3 categorise the bugs that we have discovered by target platform according to whether they expose a compiler crash (which for WGSL includes cases where the compiler generates syntactically incorrect code) or miscompilation. Reconditioning has been implemented in GLSLsmith and WGSLsmith from the start, so undefined behaviour was avoided in all the miscompilation bugs that we reported, providing evidence that the approach works well in practice. We performed a small amount of further simplification on automatically-reduced programs before reporting them. This typically involved (a) removing arithmetic wrappers introduced by reconditioning that we could tell were not necessary; (b) performing simplifications that were beyond the scope of the Perses test-case reducer; and (c) renaming variables and functions to make reduced test cases more readable. These changes tended to lead to the test case attached to a bug report being around 5–10 lines shorter than the test case arising directly from automated reduction. As discussed further in §7, it might be possible to avoid the need for (a) by leveraging recent work on detecting and eliminating redundant arithmetic wrappers in compiler testing tools [Even-Mendoza et al. 2020, 2022].

Example bugs. As illustrative examples, we discuss two bugs found by GLSLsmith—a miscompilation bug, and a crash bug that depends on GLSLsmith's support for floating-point (§4.3)—and a miscompilation bug found by WGSLsmith.

Table 2. Summary of bugs found using GLSLsmith

Compiler	Crash	Miscompilation
OpenGL-NVIDIA	2	8
OpenGL-LLVMpipe	4	2
Vulkan-Swiftshader	1	2
Vulkan-Intel	0	2
Vulkan-NVIDIA	0	2

Table 3. Summary of bugs found using WGSLsmith

Compiler	Crash	Miscompilation
Dawn-D3D12	3	4
Dawn-Vulkan	0	1
Dawn-Metal	4	3
wgpu-D3D12	10	6
wgpu-Vulkan	0	2
wgpu-Metal	5	3

```
void main() { // Pre: t0 == 0, t1 == 5
  t0 += t1;
  switch(t1 ^= t0) {
    case 0u: t2 = 1u; break;
    case 5u: t2 = 2u; break;
  }
}
```

Fig. 6. LLVMpipe miscompiled this shader so that the 5u switch case was taken

```
float f(float A) {
 return A >= 16777216.0 ? 2.0f : 1.0f;
}

void main() {
  float t1 = dot(1.0, f(1.0 + 1.0));
}
```

Fig. 7. This shader performs a well-defined floating-point computation, but crashes the NVIDIA OpenGL driver

LLVMpipe miscompilation found using GLSLsmith (Figure 6) [Freedesktop.org 2021]: The shader's input/output buffer includes integer variables `t0`, `t1` and `t2`, with `t0` initialised to 0 and `t1` to 5. The side-effecting switch condition should evaluate to 0, and the 0u switch case should be executed. LLVMpipe instead executed the 5u case, due to erroneously performing the side-effect in the switch condition twice. A Mesa developer confirmed this bug, commenting: "This bug (double evaluation of switch expression) affects all Mesa drivers. It's amazing that it survived for so long (looks like it existed at least since 2011)". The bug was promptly fixed.

NVIDIA floating-point crash found by GLSLsmith (Figure 7), reported privately to NVIDIA: The shader was reduced from a larger example featuring floating-point operations enabled by our support for floating-point (see §4.3). However, it causes the NVIDIA OpenGL driver to crash with the message "OpenGL does not allow swizzles on scalar expressions". NVIDIA have fixed this bug in response to our report.

Miscompilation in naga's SPIR-V back-end found by WGSLsmith [Rust Graphics Mages 2022]: Recall from §2.2 that all variables in WGSL are initialised to default values if no initialiser expression is provided. As a result, WGSLsmith is free to omit explicit initialiser expressions. This led to the discovery of a miscompilation bug in the SPIR-V back-end of naga, where a default-initialised global variable in WGSL was compiled into a global variable with *no* initialiser in SPIR-V. This is an UB in SPIR-V and led to the generated shader computing an unexpected result. The bug was fixed in response to our report.

6.2 Controlled Experiments

Our main claim is that reconditioning allows the combination of program generation, differential testing and test-case reduction to be used to find small UB-free test programs without the need for UB detection tools. To investigate whether reconditioning is really necessary in order to avoid UB, and to investigate the overhead of using reconditioning on reduced test cases, we present controlled experiments to answer the following research questions:

RQ1: To what extent is reconditioning essential for achieving UB-free reduced programs that trigger miscompilation bugs?

RQ2: What impact does reconditioning have on the size of reduced test cases that trigger miscompilation bugs, and on the performance of test-case reduction?

RQ3: What kinds of UB prove problematic when reconditioning is disabled, and which reconditioning steps are applied in fully-reduced test cases that trigger miscompilation bugs?

Experimental methodology. For each of GLSLsmith and WGSLsmith, we used the generation tool to generate 10,000 programs *without* reconditioning.[1] We call this set Unreconditioned. We then reconditioned each program in Unreconditioned to obtain a set of 10,000 reconditioned programs. We call this set Reconditioned. We cross-checked each program in Unreconditioned and each program in Reconditioned with respect to a representative pair of compilers, C_1 and C_2 (we discuss the specific compilers used in our experiments below), noting whether:

- **Crash:** at least one of C_1 or C_2 failed to compile the program;
- **Timeout:** the process of compiling and executing the program exceeded a given time limit for at least one of C_1 or C_2;
- **Mismatch:** both C_1 and C_2 successfully compiled the program, but the resulting binaries yielded different results—a sign of a possible miscompilation, or of UB being triggered;
- **Match:** both C_1 and C_2 successfully compiled the program, and the resulting binaries yielded identical execution results.

From the subset of programs in Unreconditioned flagged as "mismatch" we selected 50 programs at random (or all programs, if there were fewer than 50), and applied test-case reduction to these programs *without* reconditioning, reducing them with respect to the condition that they are still flagged as "mismatch". Call the resulting set of reduced test cases NoReconditioning.

For the subset of programs in Reconditioned flagged as "mismatch" we similarly selected 50 programs at random (or all programs, if there were fewer than 50). However, for each of these programs we conducted two separate test-case reduction runs: (1) a test-case reduction run on the (already) reconditioned program *without* applying reconditioning as part of the interestingness test; and (2) a test-case reduction run on the unreconditioned version of the program *with* reconditioning as part of the interestingness test, applying reconditioning to the resulting fully-reduced test case. Call the set of reduced test cases arising from reductions of the form (1) InitialReconditioning, and the set of reduced test cases arising from reductions of the form (2) FullReconditioning.

To recap, this led to three sets of reduced test cases:

- NoReconditioning: this represents the case where no attention is paid to undefined behaviour whatsoever, neither during program generation nor test-case reduction;
- InitialReconditioning: this represents the case where generated programs are well-defined, but where no attempt is made to prevent UB being introduced during test-case reduction;
- FullReconditioning: this is the approach we propose in this paper, where reconditioning is consistently used to guard against UB.

We then carefully manually examined each program in these three sets to determine which ones exhibited UB. Manual analysis was done by the authors of GLSLsmith and WGSLsmith, who have become experts in spotting UB in the process of building these tools, and involved working through the execution of each reduced test case, stopping as soon as a UB was observed. It is possible that mistakes were made during this manual analysis, but due to the small sizes of reduced shaders we are confident that such mistakes would be infrequent. Also, due to the lack of any automated tool support for UB detection in GLSL and WGSL, there was no alternative to a careful manual analysis.

[1] An exception to this is that, throughout this experiment, we always enabled reconditioning of loops using loop limiters (see §4.2). Without loop limiters, we found that GLSL shaders executing unbounded loop would lead to machine crashes and freezes, rendering our experiments infeasible, and that both naga and tint would reject shaders exhibiting loops that could be detected syntactically as being infinite if executed, leading to a low rate of usable shaders.

Finally, we noted size data (lines of code, and number of bytes) for the programs that we determined manually as being UB-free.

By construction, no programs in FullReconditioning should feature UB. To answer RQ1, we (a) check whether this is indeed the case, and (b) measure the extent to which programs in NoReconditioning and InitialReconditioning exhibit UB. This provides insight into the extent to which disabling reconditioning leads to useless test programs that trigger UB.

To answer RQ2, we compare the median size of UB-free programs (which should be all programs) in FullReconditioning with the median size of UB-free programs in NoReconditioning and InitialReconditioning. To assess performance, we compare the median wall clock time taken for these test-case reduction runs, and the median number of interestingness test invocations.

The manual analysis of UB in reduced test, described above, combined with an inspection of the reconditioning steps that are applied to test cases that trigger miscompilations, help to answer RQ3.

We performed this controlled experiment twice for each of GLSL and WGSL, using two pairs of GLSL compilers and two pairs of WGSL compilers, discussed below. We selected compilers that exhibited some known miscompilations identified via our uncontrolled bug-finding campaign (see §6.1), and selected two pairs per tool to ensure a degree of diversity, whilst being constrained by the machines that were available to us for experimentation. Each experimental run took around 2–4 days of dedicated machine time, depending on the compilers under test.

Experimental setup for GLSLsmith. As the first pair of GLSL compilers, we selected the two OpenGL drivers used during our bug-finding campaign (see OpenGL-NVIDIA and OpenGL-LLVMpipe in Table 1), using NVIDIA driver version 470.57 and LLVMpipe version 21.1.6. We refer to experiments using this configuration as **GLSL**$_1$. As the second pair of GLSL compilers, we used OpenGL-LLVMpipe version 21.1.7 (which is patched to avoid a critical miscompilation bug in version 21.1.6) and Vulkan-Swiftshader, git revision 3bd2273b. We refer to experiments using this configuration as **GLSL**$_2$. Experiments were run on a machine running Ubuntu 22.04 equipped with an Intel i7-8750H CPU and an NVIDIA 1050Ti GPU. GLSLsmith was configured with a timeout of 1 minute per shader execution. We used Perses for test-case reduction.

Experimental setup for WGSLsmith. As the first pair of WGSL compilers, we compared Dawn-Vulkan against wgpu-D3D12 (see Table 1), which we refer to as **WGSL**$_1$. As the second pair of WGSL compilers, we compared Dawn-D3D12 against wgpu-Vulkan, which we refer to as **WGSL**$_2$. In both cases we used Dawn git revision 2b4df7889 and wgpu git revision f4c01052. Experiments were run on a Windows 11 machine, with the WGSLsmith tools running in a Ubuntu 20.04 virtual machine and shaders executed natively on the host. The machine was equipped with an AMD Ryzen 3900X CPU and an NVIDIA 3070 GPU. WGSLsmith was configured with a timeout of 30s per execution. We used Perses for test-case reduction.

A note on timeouts. The timeouts used for experiments with GLSLsmith and WGSLsmith are different and were chosen based on practical experience using the tools "in the wild" (see §6.1). It is reasonable for them to be different because (a) the tools are used to test entirely different software stacks, and (b) we do not perform a comparison between GLSLsmith and WGSLsmith.

Results for RQ1. For each pair of GLSL and WGSL compilers (four pairs of compilers in total), Table 4 shows the number of crashes, timeouts, mismatches and matches obtained for the 10,000 test programs in Unreconditioned and Reconditioned. For GLSL$_1$ the mismatch rate is high even when reconditioning is enabled, and much higher still when reconditioning is disabled. Due to the size and complexity of the unreduced test cases in these sets it is not feasible to determine the root causes of all these mismatches without performing test-case reduction. Our investigation below into reduced shaders revealed that the mismatches exhibited when reconditioning was enabled are due to the LLVMpipe miscompilation bug discussed in §6.1 triggering very frequently, while the

Table 4. The rate of crashes, mismatches and matches for the test sets used in our controlled experiments

GLSLsmith	**$GLSL_1$**		**$GLSL_2$**	
Program set	Unreconditioned	Reconditioned	Unreconditioned	Reconditioned
# Crash	111	44	535	240
# Timeout	1	3	52	154
# Mismatch	2,349	909	3,022	56
# Match	7,539	9,044	6,391	9,550
WGSLsmith	**$WGSL_1$**		**$WGSL_2$**	
Program set	Unreconditioned	Reconditioned	Unreconditioned	Reconditioned
# Crash	9,321	7,692	511	765
# Timeout	127	268	220	299
# Mismatch	6	31	2,407	76
# Match	546	2,009	6,862	8,860

mismatches exhibited without reconditioning arise partly due to this bug and partly due to UB being triggered. For $GLSL_2$, which uses a version of LLVMpipe in which the miscompilation bug of §6.1 is fixed, the ratio of mismatches between Unreconditioned and Reconditioned is even higher, which we attribute in part due to this "low hanging" miscompilation no longer being present. For both configurations, we examined crashes that occur with vs. without reconditioning to see whether disabling reconditioning led to any new crashes. In the case of $GLSL_2$ we identified SwiftShader crash that occurs only when reconditioning is disabled, suggesting that the instrumentation introduced during reconditioning may sometimes mask bugs; this is in line with the findings of recent work on the effect of UB-avoidance mechanisms [Even-Mendoza et al. 2020, 2022].

Turning to the WGSL compilers, for $WGSL_1$ the majority of generated programs are rejected by the Direct3D FXC compiler after being compiled to HLSL by naga, leading to a very high rate of crashes. This is due both to bugs in Naga and numerous bugs and quirks in FXC, which is now largely unmaintained (yet cannot be bypassed). The crash rate is higher when reconditioning is disabled, but it leads to a low number of mismatches overall, and slightly more mismatches when reconditioning is enabled. For $WGSL_2$, the crash rate is much lower because tint triggers FXC issues less frequently than naga, and we see a massive difference in the mismatch rate between Unreconditioned and Reconditioned. Our investigation below into reduced test cases suggests that this difference is largely due to UB. Disabling reconditioning did not lead to the discovery of any additional crash bugs in the WGSL compilers under test.

We now consider the NoReconditioning, InitialReconditioning and FullReconditioning sets of reduced programs, discuss the extent to which they feature UB. The results are summarised in Table 5. For $WGSL_1$ the sizes of these sets is below 50, due to the limited number of result mismatches that were available (stemming from the very high rate of crashes). The results confirm that, as expected, none of the reduced programs in FullReconditioning feature UB. When reconditioning is completely disabled (NoReconditioning), the rate of UB is 74% or higher across all configurations. This shows that paying no attention to UB whatsoever leads to very unreliable differential testing results. When reconditioning is enabled during generation but disabled during reduction (InitialReconditioning), the results show that the rate of UB is less severe but still problematic: between 22% ($GLSL_1$) and 48% ($GLSL_2$). The main take-away from the experiment is clear: reconditioning *completely avoids* the problem of UB, allowing UB during program generation and/or reduction means that the results of miscompilation testing can no longer be trusted.

Results for RQ2. Table 6 shows data for median sizes (in bytes and lines of code) of reduced test cases that do not trigger UB, together with the median time (in seconds) and number of

Table 5. The extent to which reduced test cases exhibit UB when recondition is not applied at all, applied only during program generation, and applied consistently

GLSLsmith	**GLSL$_1$**			**GLSL$_2$**		
Program set	# programs	# with UB	UB rate	# programs	# with UB	UB rate
NoReconditioning	50	37	74%	50	43	86%
InitialReconditioning	50	11	22%	50	24	48%
FullReconditioning	50	0	0%	50	0	0%
WGSLsmith	**WGSL$_1$**			**WGSL$_2$**		
Program set	# programs	# with UB	UB rate	# programs	# with UB	UB rate
NoReconditioning	6	6	100%	50	43	86%
InitialReconditioning	24	7	29%	50	17	34%
FullReconditioning	17	0	0%	50	0	0%

Table 6. Median size and performance results across reductions with vs. without reconditioning

GLSLsmith	**GLSL$_1$**				**GLSL$_2$**			
	Size (b)	Size (LOC)	Time (s)	# i.t. calls	Size (b)	Size (LOC)	Time (s)	# i.t. calls
With recond.	314	19	230	133	632	29	225	221
No recond.	225	15	806	424	385	20	371	329
WGSLsmith	**WGSL$_1$**				**WGSL$_2$**			
	Size (b)	Size (LOC)	Time (s)	# i.t. calls	Size (b)	Size (LOC)	Time (s)	# i.t. calls
With recond.	974	46	228	886	1095	50	309	1006
No recond.	858	43	364	1146	876	44	455	1344

interestingness test (i.t.) calls associated with the reductions. Data are presented separately for reductions performed with reconditioning enabled (FullReconditioning), vs. reconditioning disabled (NoReconditioning and InitialReconditioning).

As expected, reconditioning leads to larger reduced test cases on average. However, the average number of lines remains in the tens, and the average size in bytes rarely exceeds 1Kb. Regarding time taken for reduction, the results of Table 6 show that, on average, reducing with reconditioning enabled is faster than reducing without reconditioning, involving a correspondingly smaller number of calls to the interestingness test. This makes sense because when reconditioning is used, reduction is applied to an *unreconditioned* test case, with reconditioning being applied as part of the interestingness test. In contrast, most of the reductions that did not use reconditioning are for test cases that come from the InitialReconditioning set. Recall that for these test cases, reconditioning was applied at program generation time, and the reconditioned test programs were reduced with reconditioning disabled. As a result, the reducer was typically faced with *larger* test cases, due to the size blow-up associated with reconditioning, leading to a longer reduction process.

Results for RQ3. Recall that manual analysis was used to determine which reduced shaders in InitialReconditioning and FullReconditioning featured UB. For each compiler configuration, Table 7 records the number of times each category of UB was identified as the source of UB in a reduced shader. For some shaders there may be multiple sources of UB; the numbers reflect the first UB that was spotted during manual analysis. The '-' entries are for UB categories that are not relevant to the shading language in question. Non-termination is not present as a UB category because, as discussed above, loop limiters were still used to avoid non-terminating loops even when all other reconditioning steps were disabled. The results show that while particular UBs dominate (uninitialised memory accesses for GLSL and floating-point roundoff errors for WGSL), all categories of UB show up in at least one reduced test case.

Table 7. Occurrences of each category of UB in reduced test cases

Category of UB	$GLSL_1$	$GLSL_2$	GLSL total	$WGSL_1$	$WGSL_2$	WGSL total
Out-of-bounds access	0	7	7	4	15	19
Builtin functions	10	3	13	2	0	2
Division	6	3	9	0	6	6
Modulo	3	14	17	1	10	11
Shift	1	15	16	1	2	3
Floating-point roundoff	7	0	7	3	21	24
Uninitialised memory	20	23	43	-	-	-
Operation order	1	2	3	-	-	-
Signed integer overflow	-	-	-	2	6	8

Turning to shaders in FullReconditioning, we investigated the reconditioning transformations that they feature, noting that many reduced shaders feature multiple such transformations. For the 100 GLSL shaders reduced across the $GLSL_1$ and $GLSL_2$ experiments, the remaining reconditioning transformations (with number of occurrences shown in parentheses) were: *arithmetic/built-in wrappers* (124), *index bounding* (85), *initialisation enforcement* (18), *controlling evaluation order* (1), and *loop limiters* (19). For the 67 WGSL shaders reduced across the $WGSL_1$ and $WGSL_2$ experiments, the remaining reconditioning transformations (with occurrences) were: *arithmetic/built-in wrappers* (42), *index bounding* (19), and *loop limiters* (4). This shows that reconditioning to avoid UB associated with arithmetic/built-in operators is critical, but that all reconditioning transformations of §4 appear in fully reduced bug-triggering shaders, except for transformations to avoid floating-point roundoff error. However, the large number of floating-point related UBs for WGSL in Table 7 show that this reconditioning step is essential to avoid spurious roundoff-related mismatches.

7 RELATED WORK

Compiler testing. A recent survey provides a comprehensive overview of the state-of-the-art in compiler testing [Chen et al. 2020]. The use of WGSLsmith as one of a variety of techniques used to harden implementations of WebGPU is discussed in an industry experience report [Donaldson et al. 2023]. Differential testing [McKeeman 1998] has been applied widely to compilers and compiler-like tools, via the influential Csmith program generator [Yang et al. 2011] and follow-on tools such as CLsmith [Lidbury et al. 2015], Verismith [Herklotz and Wickerson 2020], YARPGen [Livinskii et al. 2020]. Common to most such approaches is that they generate test programs that are free from UB by construction. GLSLsmith and WGSLsmith borrow much from the design of Csmith, but differ fundamentally because generated programs *may* exhibit UB before they are reconditioned; enforcement of UB-freedom is handled by a stand-alone reconditioning step that can be applied to both initially-generated programs and candidate programs considered during test-case reduction.

A line of work on extending the reach of compiler testing by relaxing methods for UB-avoidance involves using dynamic analysis to identify and remove safe arithmetic wrappers (such as those discussed in §4.1) that turn out not to be necessary in practice [Even-Mendoza et al. 2020, 2022]. It would be possible to integrate this with reconditioning by, in the reconditioning step (a) using standard reconditioning to add wrapper functions everywhere they might be needed, then (b) using dynamic analysis to identify and remove redundant wrappers. This could also be applied to identify and remove redundant uses of loop limiters (see §4.2).

A separate family of compiler testing techniques is based on metamorphic testing [Chen et al. 1998]. This includes the *equivalence modulo inputs* approach, which finds compiler bugs by identifying mismatches in programs that are equivalent by construction thanks to pruning of dead code [Le et al. 2014, 2015], or careful semantics-preserving mutation of live code [Sun et al. 2016].

Graphics shader compiler testing. The GLFuzz [Donaldson et al. 2017] and spirv-fuzz [Donaldson et al. 2021] tools have been used to find bugs in GLSL and SPIR-V compilers via the aforementioned metamorphic testing approaches. Both tools feature a different form of test-case reduction compared with differential testing approaches. Instead of reducing a single program that exhibits a difference between compilers, they start with two programs—an original program and a transformed program—and attempt to reduce the *transformations* that differentiate these programs. The end result is a pair of equivalent programs that differ minimally, rather than a single reduced program. The reduction capabilities of these methods are thus not directly comparable with GLSLsmith. A head-to-head bug-finding comparison between GLFuzz and GLSLsmith is also not possible because GLSLsmith works on compute shaders while GLFuzz works on graphics shaders.

Floating-point support in compiler testing. Our method for supporting floating-point operators is related to a method for testing compilers using arithmetic expressions [Nagai et al. 2014]. In this method, the idea is to generate arithmetic expressions with expected results that are known at generation-time. Like our approach, this uses the idea of restricting attention to subsets of floating-point numbers on which roundoff will not occur. The major difference is that our reconditioning step must support floating-point expressions in programs with arbitrary control flow, including loops. Thus our approach requires allowing computations to leave the special range and then be pulled back in by the `MAKE_IN_RANGE` function discussed in §4.3.

8 CONCLUSIONS

We have proposed *program reconditioning*, which avoids the need for using external program analysis tools to detect undefined behaviour when reducing miscompilation bugs, by *transforming* the program under test to get rid of undefined behaviour. Our GLSLsmith and WGSLsmith case studies show that reconditioning works well for two practical languages.

An avenue for future work is to investigate reconditioning as a way to handle undefined behaviour in languages that make use of challenging language features such as pointers. As discussed in §7, it would be interesting to combine reconditioning with recent methods for relaxing UB-avoidance mechanisms [Even-Mendoza et al. 2020, 2022], to counter the problem that these measures may suppress compiler optimisations. It would also be interesting to enhance GLSLsmith and WGSLsmith with recent approaches to accelerating and diversifying compiler testing [Chen et al. 2019, 2021].

Despite their major syntactic differences, the GLSL and WGSL languages have a lot in common, and as discussed in §4, many reconditioning steps required for these languages were similar. An interesting direction for future work could be to define an intermediate representation (IR) suitable for representing programs in a range of related languages, and performing recondition on the IR. This would avoid the need to build and maintain multiple reconditioners. As with any IR, there would likely be a trade-off, since a non-trivial amount of effort would be required to design an IR that could accommodate important differences between languages that it aims to generalise.

ACKNOWLEDGMENTS

We are grateful to Jack Clark, John Wickerson, Pingshi Yu and the anonymous reviewers of the PLDI 2022, FSE 2022 and PLDI 2023 conferences for their valuable feedback on earlier drafts of this work. This work was supported by the IRIS EPSRC Programme Grant (EP/R006865/1).

DATA AVAILABILITY

The GLSLsmith and WGSLsmith are both available as open source [Lecoeur 2023; Mohsin 2023]. We have made available an artifact capturing the versions of these tools at time of publication [Lecoeur et al. 2023]. The artifact includes a README file with instructions on how to use the tools, including to examine their reconditioning capabilities and reproduce selected bugs from §6.1, and also includes data sets related to the controlled experiments of §6.2.

REFERENCES

Apple. 2022. Metal Shading Language Specification. https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf, last accessed 2023-04-10.

Apple. 2023. Metal. https://developer.apple.com/metal/, last accessed 2023-04-10.

C-Reduce Project. 2023. Using C-Reduce. https://embed.cs.utah.edu/creduce/using/, last accessed 2023-04-10.

Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1 (2020), 4:1–4:36. https://doi.org/10.1145/3363562

Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. 2019. History-Guided Configuration Diversification for Compiler Test-Program Generation. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 305–316. https://doi.org/10.1109/ASE.2019.00037

Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2021. Coverage Prediction for Accelerating Compiler Testing. *IEEE Trans. Software Eng.* 47, 2 (2021), 261–278. https://doi.org/10.1109/TSE.2018.2889771

T.Y. Chen, S.C. Cheung, and S.M. Yiu. 1998. *Metamorphic Testing: a New Approach for Generating Next Test Cases.* Technical Report HKUST-CS98-01. Department of Computer Science, The Hong Kong University of Science and Technology.

Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Z. Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 197–208. https://doi.org/10.1145/2491956.2462173

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C - A Software Analysis Perspective. In *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7504)*, George Eleftherakis, Mike Hinchey, and Mike Holcombe (Eds.). Springer, 233–247. https://doi.org/10.1007/978-3-642-33826-7_16

Alastair F. Donaldson, Ben Clayton, Ryan Harrison, Hasan Mohsin, David Neto, Vasyl Teliman, and Hana Watson. 2023. Industrial Deployment of Compiler Fuzzing Techniques for Two GPU Shading Languages. In *Proceedings of the 16th IEEE International Conference on Software Testing, Verification and Validation, ICST 2023, Dublin, Ireland, June 16-20, 2023*. IEEE. To appear.

Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *PACMPL* 1, OOPSLA (2017), 93:1–93:29. https://doi.org/10.1145/3133917

Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpinski. 2021. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1017–1032. https://doi.org/10.1145/3453483.3454092

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 *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 1219–1223. https://doi.org/10.1145/3324884.3418933

Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. 2022. CsmithEdge: more effective compiler testing by handling undefined behaviour less conservatively. *Empir. Softw. Eng.* 27, 6 (2022), 129. https://doi.org/10.1007/s10664-022-10146-1

Freedesktop.org. 2021. Miscompilation of a switch case. https://gitlab.freedesktop.org/mesa/mesa/-/issues/5185, last accessed 2023-04-10.

Google. 2023a. ANGLE. https://opensource.google/projects/angle, last accessed 2023-04-10.

Google. 2023b. Dawn, a WebGPU implementation. https://dawn.googlesource.com/dawn/+/refs/heads/main/README.md, last accessed 2023-04-10.

Google. 2023c. ShaderTrap GitHub repository. https://github.com/google/shadertrap, last accessed 2023-04-10.

GraphicsFuzz project authors. 2023. GraphicsFuzz. https://github.com/google/graphicsfuzz, last accessed 2023-04-10.

The Khronos Vulkan Working Group. 2019. *Vulkan 1.1.141 - A Specification (with all registered Vulkan extensions).* Khronos Group. https://www.khronos.org/registry/vulkan/specs/1.1-extensions/pdf/vkspec.pdf, last accessed 2023-04-10.

Yann Herklotz and John Wickerson. 2020. Finding and Understanding Bugs in FPGA Synthesis Tools. In *FPGA '20: The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, February 23-25, 2020*, Stephen Neuendorffer and Lesley Shannon (Eds.). ACM, 277–287. https://doi.org/10.1145/3373087.3375310

IEEE. 1985. IEEE Standard for Binary Floating-Point Arithmetic.

John Kessenich (Ed.). 2019. *The OpenGL Shading Language Version 4.60.7.* Khronos Group. https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf, last accessed 2023-04-10.

John Kessenich, Boaz Ouriel, and Raun Krisch (Eds.). 2022. *SPIR-V Specification, Version 1.6, Revision 2, Unified.* Khronos Group. https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.pdf, last accessed 2023-04-10.

Khronos Group. 2023a. glslang GitHub repository. https://github.com/KhronosGroup/glslang, last accessed 2023-04-10.

Khronos Group. 2023b. The OpenCL Specification. https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf, last accessed 2023-04-10.

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 216–226. https://doi.org/10.1145/2594291.2594334

Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 386–399. https://doi.org/10.1145/2814270.2814319

Bastien Lecoeur. 2023. GLSLsmith GitHub repository. https://github.com/AaronGhost/glslsmith, last accessed 2023-04-10.

Bastien Lecoeur, Hasan Mohsin, and Alastair F. Donaldson. 2023. Artifact for "Program Reconditioning: Avoiding Undefined Behaviour When Finding and Reducing Compiler Bugs", PLDI 2023. https://doi.org/10.5281/zenodo.7819755

Jon Leech (Ed.). 2019. *OpenGL ES Version 3.2.* Khronos Group. https://www.khronos.org/registry/OpenGL/specs/es/3.2/es_spec_3.2.pdf, last accessed 2023-04-10.

Xavier Leroy. 2023. The CompCert C verified compiler: Documentation and user's manual. https://compcert.org/man/manual004.html, last accessed 2023-04-10.

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, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 65–76. https://doi.org/10.1145/2737924.2737986

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 196:1–196:25. https://doi.org/10.1145/3428264

LLVM. 2023. UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, last accessed 2023-04-10.

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

Microsoft. 2019. Reference for HLSL. https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-reference, last accessed 2023-04-10.

Microsoft. 2021. Direct3D 12 graphics. https://learn.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-graphics, last accessed 2023-04-10.

Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical Delta Debugging. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa (Eds.). ACM, 142–151. https://doi.org/10.1145/1134285.1134307

Hasan Mohsin. 2023. WGSLsmith GitHub repository. https://github.com/wgslsmith/wgslsmith, last accessed 2023-04-10.

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 Trans. Syst. LSI Des. Methodol.* 7 (2014), 91–100. https://doi.org/10.2197/ipsjtsldm.7.91

Nvidia. 2021. CUDA C++ Programming Guide, Version 11.2.1. https://docs.nvidia.com/cuda/archive/11.2.1/cuda-c-programming-guide/, last accessed 2023-04-10.

Terence Parr. 2023. ANTLR. https://www.antlr.org/, last accessed 2023-04-10.

pest project authors. 2023. pest. The Elegant Parser. https://github.com/pest-parser/pest, last accessed 2023-04-10.

Moritz Pflanzer, Alastair F. Donaldson, and Andrei Lascu. 2016. Automatic Test Case Reduction for OpenCL. In *Proceedings of the 4th International Workshop on OpenCL, IWOCL 2016, Vienna, Austria, April 19-21, 2016.* ACM, 1:1–1:12. https://doi.org/10.1145/2909437.2909439

Picire Project. 2012. Picire: Parallel Delta Debugging Framework. https://github.com/renatahodovan/picire, last accessed 2023-04-10.

John Regehr. 2012. Responsible and Effective Bugfinding. https://blog.regehr.org/archives/1679, last accessed 2023-04-10.

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 335–346. https://doi.org/10.1145/2254064.2254104

Rust Graphics Mages. 2022. Invalid result when using implicitly initialized array. https://github.com/gfx-rs/naga/issues/1748, last accessed 2023-04-10.

Rust Graphics Mages. 2023. The wgpu project. https://github.com/gfx-rs/wgpu, last accessed 2023-04-10.

Mark Segal and Kurt Akeley (Eds.). 2019. *The OpenGL Graphics System: A Specification Version 4.6 (Core Profile).* Khronos Group. https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf, last accessed 2023-04-10.

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

Robert J. Simpson and John Kessenich (Eds.). 2019. *The OpenGL ES Shading Language Version 3.20.6.* Khronos Group. https://www.khronos.org/registry/OpenGL/specs/es/3.2/GLSL_ES_Specification_3.20.pdf, last accessed 2023-04-10.

Tyler Sorensen and Alastair F. Donaldson. 2016. The Hitchhiker's Guide to Cross-Platform OpenCL Application Development. In *Proceedings of the 4th International Workshop on OpenCL, IWOCL 2016, Vienna, Austria, April 19-21, 2016.* ACM, 2:1–2:12. https://doi.org/10.1145/2909437.2909440

Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*, Kunle Olukotun, Aaron Smith, Robert Hundt, and Jason Mars (Eds.). IEEE Computer Society, 46–55. https://doi.org/10.1109/CGO.2015.7054186

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, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 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, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 361–371. https://doi.org/10.1145/3180155.3180236

W3C. 2023a. WebGPU Shading Language W3C Working Draft. https://www.w3.org/TR/WGSL/, last accessed 2023-04-10.

W3C. 2023b. WebGPU W3C Working Draft. https://www.w3.org/TR/webgpu/, last accessed 2023-04-10.

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 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. https://doi.org/10.1145/1993498.1993532

Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498

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