Check for updates

PDF Download
3729305.pdf
19 February 2026
Total Citations: 1
Total Downloads: 537

Latest updates: https://dl.acm.org/doi/10.1145/3729305

Published: 10 June 2025
Accepted: 06 March 2025
Received: 15 November 2024

Citation in BibTeX format

RESEARCH-ARTICLE

Divergence-Aware Testing of Graphics Shader Compiler Back-Ends

DONGWEI XIAO, Hong Kong University of Science and Technology, Hong Kong, Hong Kong

SHUAI WANG, Hong Kong University of Science and Technology, Hong Kong, Hong Kong

ZHIBO LIU, Hong Kong University of Science and Technology, Hong Kong, Hong Kong

YITENG PENG, Hong Kong University of Science and Technology, Hong Kong, Hong Kong

DAOYUAN WU, Hong Kong University of Science and Technology, Hong Kong, Hong Kong

ZHENDONG SU, Swiss Federal Institute of Technology, Zurich, Zurich, ZH, Switzerland

Open Access Support provided by:

Hong Kong University of Science and Technology

Swiss Federal Institute of Technology, Zurich

Divergence-Aware Testing of Graphics Shader Compiler Back-Ends

DONGWEI XIAO, Hong Kong University of Science and Technology, China
SHUAI WANG*, Hong Kong University of Science and Technology, China
ZHIBO LIU, Hong Kong University of Science and Technology, China
YITENG PENG, Hong Kong University of Science and Technology, China
DAOYUAN WU, Hong Kong University of Science and Technology, China
ZHENDONG SU*, ETH Zürich, Switzerland

Graphics shaders are the core of modern 3D visual effects, enabling developers to create realistic, real-time rendering of 3D scenes. Shaders are specialized programs written in high-level shading languages like GLSL, and graphics shader compilers translate these high-level shader programs into low-level binaries that run on GPUs. These shader compilers are complex programs with multiple layers: front-end, middle-end, and back-end. Despite significant development efforts from industrial GPU vendors such as NVIDIA and AMD, graphics shader compilers still contain bugs that can impact downstream applications and lead to negative consequences from poor user experience in entertainment to accidents in driving assistance systems. Because they are complex and deep in the compilation pipeline, the back-ends of shader compilers are particularly challenging to test. Our empirical exploration shows that state-of-the-art testing tools for shader compilers do not specifically target the back-ends and are thus ineffective in uncovering back-end bugs.

This work fills this gap and introduces ShaDiv, an automated testing tool specifically designed to uncover bugs in the back-ends of graphics shader compilers. To this end, ShaDiv generates test inputs with two novel, carefully designed strategies to support the unique computational models of the back-ends, namely control and data flow divergence among GPU threads. ShaDiv deliberately perturbs divergence patterns in both the control and data flow of shader programs to effectively trigger back-end optimizations. Our evaluation of ShaDiv on graphics shader compilers from four mainstream GPU vendors uncovered 12 back-end bugs. Further comparison with existing shader compiler testing tools shows that ShaDiv achieves a 25% coverage increase in the back-end components and finds four times as many back-end bugs.

CCS Concepts: • **Software and its engineering → Compilers**; **Software testing and debugging**; • **Computer systems organization → Single instruction, multiple data**.

Additional Key Words and Phrases: Shader Compiler, Testing, Divergence, GPU, SIMT, Back-End

ACM Reference Format:
Dongwei Xiao, Shuai Wang, Zhibo Liu, Yiteng Peng, Daoyuan Wu, and Zhendong Su. 2025. Divergence-Aware Testing of Graphics Shader Compiler Back-Ends. *Proc. ACM Program. Lang.* 9, PLDI, Article 202 (June 2025), 25 pages. https://doi.org/10.1145/3729305

*Corresponding authors.

Authors' Contact Information: Dongwei Xiao, Hong Kong University of Science and Technology, Hong Kong, China, dxiaoad@cse.ust.hk; Shuai Wang, shuaiw@cse.ust.hk, Hong Kong University of Science and Technology, Hong Kong, China; Zhibo Liu, Hong Kong University of Science and Technology, Hong Kong, China, zliudc@cse.ust.hk; Yiteng Peng, Hong Kong University of Science and Technology, Hong Kong, China, ypengbp@cse.ust.hk; Daoyuan Wu, Hong Kong University of Science and Technology, Hong Kong, China, daoyuan@cse.ust.hk; Zhendong Su, ETH Zürich, Zürich, Switzerland, zhendong.su@inf.ethz.ch.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/6-ART202
https://doi.org/10.1145/3729305

1 Introduction

Graphics rendering is the process of generating images from 2D or 3D models. It enables a wide range of applications, from virtual reality and gaming [CNN 2024] to computer-aided manufacturing [Inc. 2024] and robotics [CornilleMarch 2024]. Graphics rendering has been gaining increased attention in recent years due to the growing demand for realistic and immersive visual effects. The market for graphics rendering has reached 4.4 billion USD in 2023 and is expected to grow at a compound annual growth rate of 25% [Wadhwani 2024].

To produce high-quality graphics while also achieving high performance, developers express desired visual effects, such as lighting and texture mapping, using *shader programs*. A shader program is typically written in a high-level language and describes the expected visual effects. The shader program is often executed on a graphics processing unit (GPU). To run on hardware platforms, the shader program must be translated and optimized into low-level machine code specific to the target hardware, such as NVIDIA or ARM GPUs.

The translation and optimizations of high-level shader programs into GPU executables are performed by *graphics shader compilers*. Graphics shader compilers are developed by leading GPU vendors like NVIDIA over the course of decades. Despite their high engineering quality, shader compilers can still contain bugs that lead to incorrect rendering results, which can in turn cause confusing visual effects for users.

Graphics shader compilers use a three-phase pipeline: front-end, middle-end, and back-end. The front-end and middle-end handle hardware-agnostic optimizations like dead code elimination and constant folding. The back-end, on the other hand, is specific to the GPU. Unlike C/C++ programs, which are executed on CPUs, shader execution on GPUs features the Single Instruction, Multiple Threads (SIMT) model [Novak 2014; Sampaio et al. 2014], which is designed to exploit the massive parallelism of GPUs. The SIMT model exposes more opportunities and challenges for compiler back-ends to optimize. Different from CPU-based compilers like GCC and LLVM, shader compilers have the most complex optimizations in the back-end, with three times more code than the front-end and middle-end combined [3D 2024], and involve GPU-specific optimizations like execution mask insertion and divergence-aware register allocation to ensure high performance.

Bugs in the shader compiler back-ends are hard to detect. The back-end is the last stage in the compilation pipeline, meaning a testing input program must carefully pass the front-end and middle-end optimizations before triggering interesting back-end optimizations, making it challenging to explore the optimization space of the back-end. Despite significant advances in compiler testing, generating test cases that can stress deep optimizations is generally considered difficult [Even-Mendoza et al. 2023; Livinskii et al. 2023]. We empirically find that existing tools for testing shader compilers [Donaldson et al. 2017; Xiao et al. 2023] can hardly test the back-end optimizations (see further details in Section 6.4). As such, there is a pressing need for a dedicated testing approach to uncover bugs in the back-ends of shader compilers.

We propose ShaDiv, a systematic and automated testing framework to uncover bugs in graphics shader compiler back-ends. ShaDiv generates test programs using carefully designed strategies inspired by the unique SIMT execution model of shader programs. We observe unique challenges in compiling for the SIMT model, namely, control flow divergence and data flow divergence. Accordingly, we design two generation strategies to stress-test the related back-end optimizations, thereby increasing the chance of finding bugs in compiler back-ends.

Experiments on four shader compilers from leading GPU vendors uncover a total of 14 bugs, with 12 being in the compiler back-end; 11 of these bugs are confirmed and 6 are fixed. Compared to existing works [Donaldson et al. 2017; Xiao et al. 2023], ShaDiv achieves a 25% relative increase in line coverage of the back-end code and finds 4 times as many back-end bugs. We release ShaDiv at [ShadeDiv-Artifact 2024] to boost future research. Our contributions are as follows:

- Conceptually, this research addresses a critical yet underexplored area in compiler testing: graphics shader compiler back-ends. These back-ends, featuring the SIMT execution model, present unique challenges distinct from those encountered in traditional CPU-based compiler testing, and require dedicated testing strategies.
- Technically, our approach leverages divergence and liveness analysis, tailored to the SIMT execution model, to guide the generation of effective test programs. We design a program generator that carefully incorporates the analysis results, thereby generating programs with diverse divergence and liveness characteristics that can thoroughly stress the back-end optimizations.
- Empirically, we advanced the state-of-the-art in shader compiler testing by finding four times as many back-end bugs and improving the line coverage by 25% compared with existing works. To date, ShadeDiv has identified 14 bugs, with 11 confirmed and 6 already fixed, demonstrating its practical value for shader compiler developers.

2 Background

2.1 Graphics Rendering and Shader Programs

Graphics Shader Programs. In real-time rendering, developers often use *graphics shader programs* to define visual effects by manipulating lighting, textures, and geometry. These programs are typically written in high-level C-like Domain-Specific Languages (DSLs) like GLSL [Group 2024a].

```
 1 uniform vec3 o;
 2 varying vec3 coord;
 3 void main() {
 4   float r = 1.0;
 5   vec3 camera = vec3(0, 0, -1);
 6   float d = length(camera - o);
 7   if (d < r) {  // uniform
 8     color = BLACK;
 9     return;
10   }
11   vec3 ray = normalize(coord - camera);
12   float hit = ray_dist(camera, ray, o, r);
13   if (hit > 0.0) { // divergent
14     color = get_color(hit, d, ray);
15   } else {
16     color = WHITE;
17   }
18 }
```

Fig. 1. A graphics shader program drawing a ball.

Example. Fig. 1 shows a simple graphics shader program that draws a ball. A GPU thread executes this program by taking a pixel's coordinates `coord` as input and outputs the pixel's color `color`. Since the pixel coordinates vary across pixels, `coord` is qualified with the `varying` keyword. The shader also takes as input the ball center's coordinates `o`, which is of the built-in type `vec3` (defined as a 3D vector with components $[x, y, z]$). The variable `o` is qualified as `uniform` since the value of `o` is the same for all pixels at any given time. In lines 7–9, if the distance `d` between the camera and the ball center is less than the ball radius `r`, i.e., the camera is inside the ball, the color is set to black. Otherwise, the camera shoots a ray in the direction of the pixel (line 11). In line 12, the `ray_dist` function calculates the distance a ray travels to hit the ball or returns `-1` if it misses. The pixel color is determined by ray-ball intersection (lines 13–16).

Shader Execution Pattern: SIMT Model. Shader execution on GPUs features unique characteristics compared to traditional CPU execution. GPU shader execution is highly parallel, where a large number of GPU *threads* are executed simultaneously. The threads are grouped into *warps*. Each warp contains multiple threads (e.g., 32 threads). GPU threads in the same warp execute the same program instruction at the same time but with different data. The unique execution pattern of GPU threads and warps is typically referred to as the *SIMT* (Single Instruction, Multiple Threads) [Garland and Kirk 2010; Nickolls and Dally 2010; Nickolls and Kirk 2009] execution model.

Importantly, since different threads take different input data, it is common that threads in the same warp will take different control flow paths, which is referred to as *divergent control flow*. For instance, some threads may take the true branch of an `if` statement, while others take the false branch. Because threads in a warp execute the same instruction simultaneously, managing divergent control flow is a key challenge. GPUs handle this by executing both branches, using *execution masks*, a special set of registers, to activate/inactivate threads for each branch. Idle threads still execute instructions but discard the results, thus ensuring program correctness.

Example. Fig. 2 illustrates the execution of three threads, T1, T2, and T3, inside a GPU warp that executes the program in Fig. 1. Assume that the camera is outside of the ball, and the rays in T1 and T2 hit the ball, while the ray in T3 misses. The control flow and basic blocks (labeled from ① to ⑤) of the shader program are shown in Fig. 3. The basic blocks ① (lines 4 to 7) and ③ (lines 11 to 13) are always executed by all threads, thus all threads are active when executing the instructions in those blocks. GPU execution distinguishes two types of branching instructions: uniform branching and divergent branching. Uniform branches are branches where all threads in the GPU kernel uniformly take either the true or false branch at any given time, while divergent branches are those where threads in the same warp can potentially take different branches due to their input data, e.g., the pixel coordinates in this example. The determination of uniform/divergent branches is decided statically by the compiler based on divergence analysis, which we will introduce soon afterward. The branch condition in line 7 is uniform. Since all threads take the same branch, the branch can be simply implemented with a conditional `jump` instruction. Since the camera center `o` is outside of the ball, the basic block ② (lines 8–9) is skipped. The branch condition in line 13 is divergent: T1 and T2 execute the branch where the ray hits the ball, while T3 executes the other branch. The GPU will execute both branches for all threads but will use execution masks to force T3 to be idle when executing block ④, and T1 and T2 to be idle in block ⑤. When a thread is inactive, its execution results are discarded, i.e., the value of `color` will not be tampered with.

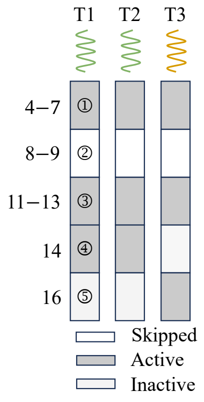

Fig. 2. SIMT model.

2.2 Graphics Shader Compilers

Graphics shader compilers are responsible for compiling and optimizing high-level shader programs into GPU executables in order to achieve high rendering performance. We will show the unique challenges in the back-end of shader compilers in this section.[1]

① **Control/Data Flow Divergence Analysis** in graphics shader compilers is a process that identifies and manages differences in execution paths and data usage among shader threads. Divergent control flow occurs when threads within a wrap take different execution paths, while divergent data flow arises when threads hold different values for the same variable at the same program point. Unmitigated divergence leads to significant performance bottlenecks, including thread idling and increased resource consumption. Divergence analysis often influences other compilation processes and optimizations, such as instruction scheduling, register allocation, and parallel execution strategies, ultimately ensuring efficient and correct shader execution on the GPU.

[1]We will use the terms "shader compilers" and "graphics shader compilers" interchangeably throughout the paper.

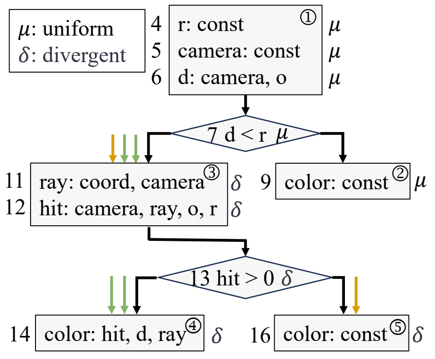

Fig. 3. Control/data flow of the shader program.

Example. Fig. 3 shows the control and data flows of the shader program in Fig. 1. Each line in the gray box represents the data dependence relationship, in which the left-hand side of the colon is the variable being assigned, and the right-hand side is the variables (or constants) that the assignment depends on. We use the symbol μ to denote that a variable or control flow is uniform, and δ to denote that a variable or branching is divergent. The divergence labels are shown on the right of each statement or branch condition. In lines 4 and 5, the variable `r` and `camera` only depend on constants, and thus are labeled as uniform. The variable `d` in line 6 depends on the constant `camera` and a uniform input `o` and is also uniform. The branch condition in line 7 depends on uniform variables `d` and `r`. Hence, the control flow analysis labels this branch statement to be uniform. In line 11, the pixel coordinates `coord` vary across threads, thus tainting the variable `ray` as divergent. In turn, the `hit` variable is tainted as divergent since it depends on `ray`. The `if` statement in line 13 is divergent, as its condition involves `hit`, meaning its truth value can differ among threads. Upon encountering divergent branches, the shader compiler must emit instructions for setting the execution masks to control the activation status of threads. If a branch condition is divergent, all statements in both the true and false branches are labeled as divergent regardless of data dependence. Thus, `color` in lines 14 and 16 are tagged as divergent.

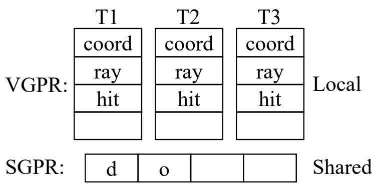

Fig. 4. Register allocation.

② **Divergence-Aware Register Allocation.** Register allocation handles the mapping of variables in the shader program to the physical registers in the GPU. GPUs have two sets of register files: Vector General Purpose Registers (VGPRs) and Scalar General Purpose Registers (SGPRs). VGPRs are local to each thread, while SGPRs are shared among threads in the same warp. To ensure correctness, the variables whose value can be different among threads must be stored in VGPRs, while uniform variables can be mapped to both VGPRs and SGPRs. However, GPUs have a limit on the number of VGPRs and SGPRs. If the register pressure (the number of allocated registers) exceeds the limit, register spilling occurs, which temporarily stores the spilled registers in the slower memory and incurs significant performance degradation. As such, compilers must carefully allocate registers to minimize the register pressure while avoiding violations of program correctness. Register allocation also has to account for the live ranges of variables, which are the intervals between the first and last use of a variable. Register allocation is an NP-complete problem in traditional compiler design [Pereira and Palsberg 2005], and the divergence of control/data flow on GPUs further complicates this problem.

Example. Fig. 4 shows register allocation for the program in Fig. 1. Suppose the register limit for VGPRs is 12, and the limit for SGPRs is 4. Since `coord`, `ray`, and `hit` are divergent (see Fig. 3), they must be stored in VGPRs, which are local to each thread. The variables `d` and `o` are uniform, holding consistent values across threads. While `d` and `o` could be stored in VGPRs, this would require $2 \times 3 = 6$ registers. Given that 9 VGPRs are already allocated for divergent variables, allocating `d` and `o` to VGPRs would exceed the limit, leading to register spilling. Since SGPRs are shared among threads, only two registers are needed to represent `d` and `o` for all threads, saving 7 registers. Hence, the compiler chooses to allocate `d` and `o` to SGPRs.

```
1  uniform int uni_input;
2  varying vec2 coord;
3  void main() {
4    int t = 0;
5    if (uni_input != 2)  // uniform
6      for (int i = 0; i < 2; i++) {
7        if (coord.x >= 56) // divergent
8          continue;        // divergent
9        t = 1;             // divergent
10     }
11   else {...} // uniform
12 }
```

(a) Uniform-divergent interleavings causes the AMD compiler to hang.

```
1  void main() {
2    for (int i = 0; i < 2; i++) {
3      if (coord.x > 56) {
4        c = int(g); // Define c
5        break;
6      }
7      d = 2 | d;
8      g = 0;
9    }
10   d = coord.x ^ c;// Extend use of c
11   ...  // Assign color based on c, d
12 }
```

(b) Extended uses of the variable c leads to incorrect rendered images in the AMD compiler.

Fig. 5. Error-triggering shader programs found by uniform-divergent interleavings and extended liveness.

3 Motivation

ShaDiv generates shader programs with diverse divergence and liveness characteristics to thoroughly test shader compiler backends. Our focus on these two aspects stems from their inherent complexity and their fundamental role in backend optimization. Examination of an open-source shader compiler [3D 2024] shows that divergence information is embedded within the intermediate representation (IR) and used extensively throughout the backend for optimization. Liveness analysis, similarly, enables key optimizations like instruction scheduling and post-register allocation. Furthermore, thread divergence causes long-standing problems for shader compiler designers [Jerez 2018], with some advocating for the development of specialized IRs to better manage divergence [Ekstrand 2022; Glob 2022]. As we will demonstrate in Section 6.3, ShaDiv's generation schemes prove highly effective in comprehensively stress-testing a diverse set of backend optimizations. Below, we provide two examples to illustrate the importance of divergence and liveness in testing shader compiler back-ends.

A Bug Discovered by Uniform-Divergent Interleavings. Graphics shader compilers need to generate code that conforms to the SIMT execution model of GPUs, which requires handling both uniform and divergent control flows. In Fig. 5a, the shader program interleaves uniform and divergent control flows. The `if` statement in line 5 is uniform, as it is controlled by the uniform input variable `uni_input`. The `for` loop contains a divergent branching (line 7), which causes the control flow to diverge in the following statements in the loop body (lines 8 and 9). The compiler needs to perform execution mask insertion to ensure program semantics for the divergent statements. When compiling the *divergent* statements in lines 7 and 8, the AMD compiler creates a temporary execution mask to ensure their semantics. During inserting execution masks for another *divergent* statement in line 9, the compiler, however, mistakenly deallocates the temporary mask, leading to a violation of semantics and a hanging during runtime execution. The developers noted this bug as a "good catch" [Anonymous 2024]. The discovery of such a bug requires the synergistic effects of uniform and divergent control flows in lines 5 to 11. Without carefully designed generation schemes, it can be hard to generate the divergence interleavings that can trigger this bug.

A Bug Discovered by Extended Liveness. The GPU architecture features a unique memory structure, with VGPRs storing divergent variables and SGPRs storing uniform or divergent variables. The memory-related optimizations, such as register allocation and register spilling, must not only reduce

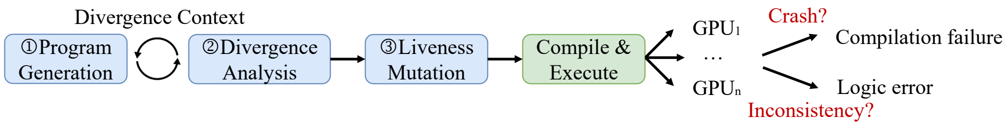

Fig. 6. Testing pipeline of ShaDiv.

register pressure but also conform to the divergence type constraints of VGPR/SGPR. Triggering such optimizations requires certain def-use chains in the shader program. Fig. 5b shows an example bug uncovered by extending the uses of the variable c. After introducing an additional read to c (highlighted in red) in line 10 by the XOR operation, the program becomes ill-compiled by the AMD compiler and produces incorrect rendering images. The extended usage of c introduces an additional def-use relation to the definition site of c in line 4 (also highlighted). The assignment to c in line 4 is divergent since it is guarded by an if-condition that depends on the divergent variable coord.x. The read of c in line 10 is also divergent. The extended def-use relation of c couples the data flow with the divergence of the control flow, which significantly complicates the memory-related optimizations in the compiler back-end, leading to the manifestation of the bug. The discovery of such a bug requires generating test programs with extended liveness of variables that can expose the complex data flow divergence patterns. It thus motivates us to design the liveness extension mutation in ShaDiv.

4 Design of ShaDiv

Study Scope. ShaDiv focuses on testing the correctness of the shader compiler back-ends. In general, ShaDiv is able to capture two types of bugs:

- *Compilation failures.* This type of bug causes obvious misbehaviors, such as crashes or memory access violations. These bugs can happen during the compilation process (compilation-time) or when running the compiled binary (runtime). We do not distinguish between the compilation-time or runtime failures, as we find existing shader compilers, which are typically shipped in proprietary GPU drivers, do not offer clear stack traces or error messages to differentiate them.
- *Logic errors.* This type of bug does not directly cause obvious misbehaviors, but causes the compiled binary to render incorrect images during the execution. These bugs are more subtle and stealthy, and are potentially harder to identify by normal users than compilation failures.

Fig. 6 shows the testing pipeline of ShaDiv. The pipeline consists of three major components:
① G_{rand}: *Random Program Generation.* This phase generates random shader programs as testing inputs. To ensure the validity of the generated programs and lower the implementation burdens, G_{rand} first generates a customized IR and then translates the IR to the concrete shader languages. The use of IR allows us to abstract the syntax and semantics of shader languages, and to simplify the generation and mutation of shader programs.
② G_{cdiv}: *Divergence Analysis.* This phase augments the random program generator by generating shader programs with complex control flow divergence patterns. To that end, G_{cdiv} maintains a divergence context which is updated whenever a new statement is generated from G_{rand}; the program generator then samples from the divergence context to generate interleavings of uniform and divergent statements. The cooperation between G_{rand} and G_{cdiv} works iteratively on generating each statement until the program is fully generated. We will detail G_{cdiv} in Section 4.2.
③ M_{liv}: *Liveness Mutation.* This stage complicates the data flow patterns of shader programs by mutating the liveness range of variables in the generated program from G_{rand}. The mutation first performs liveness analysis towards the shader program to identify the liveness information of each variable. Then, M_{liv} extends the liveness range of some variables by introducing additional

Assignment	*assign*	::=	$v := e$
Variable	v	::=	*valid variable name*
UnaryOp	$\diamond$	::=	`not` | `negate`
BinaryOp	$\otimes$	::=	$+ \mid - \mid \times \mid \ldots$
Constant	c	::=	*valid values of type* φ
BasicType	φ	::=	`int` | `bool` | `float` | `vec`$\{2, 3, 4\}$
DivergenceType	ω	::=	$\delta \mid \mu$
Expression	e	::=	$c \mid v \mid e_1 \otimes e_2 \mid \diamond e$
Statement	s	::=	*assign* | if e then s else s | for v in $e..e$ $\{s\}$

Fig. 7. Selected syntax of the Λ_{ShaDiv}.

DivergenceEnv	Σ	::=	$v \longrightarrow \omega$
BranchDivStack	Φ	::=	$List[\omega]$

Fig. 8. Auxiliary data structures in Λ_{ShaDiv}.

define-use relationships. The mutated program stresses the data flow divergence of the shader program. The details of M_{liv} will be discussed in Section 4.3.

The generated program after the M_{liv} phase is then translated from the IR to the specific shader languages. The translated program is then compiled and executed on a set of GPU devices under test. If any of the GPU crashes during compilation or execution, we report it as a compilation failure. If the execution succeeds, we collect the rendered images from the different GPUs and compare them to identify inconsistencies (detailed in Section 6.2), which are reported as logic errors.

4.1 Random Program Generation

To facilitate the generation and mutation of shader programs, we design an IR, namely Λ_{ShaDiv}, that abstracts the syntax and semantics of shader languages. Λ_{ShaDiv} is designed to be simple and easy to map to concrete shader language syntax. The IR does not intend to capture all the features of shader languages: per our observation, complex high-level operations, e.g., structs and built-in math functions, are unlikely to affect the back-end behaviors (see discussions in Section 7). We show the selected syntax of Λ_{ShaDiv} in Fig. 7. The `vec` type in Λ_{ShaDiv} represents a 2D, 3D, or 4D vector in the coordinate systems. We model the divergence type ω in Λ_{ShaDiv}, which can be either divergent (δ) or uniform (μ). A variable is divergent if its value can be different across threads in a warp at a given program instruction, and uniform otherwise.

Statement Generation. Algorithm 1 illustrates the statement generation algorithm, RandGenStmts, which randomly creates a list of statements to compose the program. It takes a set of variables (*var_set*) and a maximum statement count (*MAX_N*) as input. The algorithm randomly determines the number of statements to generate, up to *MAX_N*. For each statement, it randomly chooses a statement type (assignment, if, or for). Assignment statements are created by randomly selecting a variable from *var_set* and generating a type-compatible expression using GenExpr (introduced soon). If-statements involve generating a boolean condition and recursively generating the then- and else-branches, decrementing *MAX_N* to prevent infinite recursion. For-loops are generated by randomly determining a start value and a short iteration range (1–3 iterations), and then recursively generating the loop body, again decrementing *MAX_N*. The generated statement is then added to the list, and the function returns the complete list of statements.

Algorithm 1 Statement generation.

```
1: function RandGenStmts($var\_set$: Input variable set, $MAX\_N$: Maximum number of statements)
2:     $n \leftarrow$ Sample$(0..\max(0, MAX\_N))$
3:     $stmts \leftarrow []$
4:     for _ $\in 1..n$ do
5:         $stmt\_type \leftarrow$ Sample({assign, if, ...})
6:         if $stmt\_type$ = assign then
7:             $v \leftarrow$ Sample$(var\_set)$
8:             $\varphi \leftarrow$ BasicTypeOf$(v)$
9:             $expr \leftarrow$ GenExpr$(\varphi, var\_set, 0)$
10:            Append$(stmts, v := expr)$
11:        else if $stmt\_type$ = if then
12:            $cond \leftarrow$ GenExpr(bool, $var\_set$, 0)
13:            $then\_stmt \leftarrow$ RandGenStmts$(var\_set, MAX\_N - 1)$
14:            $else\_stmt \leftarrow$ RandGenStmts$(var\_set, MAX\_N - 1)$
15:            Append($stmts$, if $cond$ then $then\_stmt$ else $else\_stmt$)
16:        else if $stmt\_type$ = for then
17:            $start \leftarrow$ Sample$(0..INT\_MAX - 3)$
18:            $end \leftarrow$ Sample$(1..3) + start$
19:            $body \leftarrow$ RandGenStmts$(var\_set, MAX\_N - 1)$
20:            Append($stmts$, for $i$ in $start..end$ {$body$})
21:    return $stmts$
```

Algorithm 2 Expression generation.

```
1: function GenExpr($\varphi$: Basic type, $var\_set$: Input variable set, $depth$: Expr depth)
2:     if $depth \geq MAX\_EXPR$ then          ▷ $MAX\_EXPR$ is the maximum depth of the expression tree
3:         $expr\_type \leftarrow$ Sample({const, var})
4:     else
5:         $expr\_type \leftarrow$ Sample({const, var, unop, binop})
6:     if $expr\_type$ = const then
7:         $c \leftarrow$ Sample(valid constants of type $\varphi$)
8:         return $c$
9:     else if $expr\_type$ = var then
10:        $var\_set \leftarrow$ Filter$(var\_set, \varphi)$          ▷ Filter variables with the specified basic type
11:        $v \leftarrow$ Sample$(var\_set)$
12:        return $v$
13:    else if $expr\_type$ = unop then
14:        $\diamond \leftarrow$ Sample({not, negate})
15:        $e \leftarrow$ GenExpr$(\varphi, var\_set, depth + 1)$
16:        return $\diamond\, e$
17:    else if $expr\_type$ = binop then
18:        $\otimes \leftarrow$ Sample$(\{+, -, \times, \ldots\})$
19:        $e_1 \leftarrow$ GenExpr$(\varphi, var\_set, depth + 1)$
20:        $e_2 \leftarrow$ GenExpr$(\varphi, var\_set, depth + 1)$
21:        return $e_1 \otimes e_2$
```

Generation of Expressions. Algorithm 2 shows the generation of expressions in G_{rand}. The function GenExpr takes as input the basic type of the expression to generate, the set of variables *var_set* that are available to use, and the depth of the expression to generate in the expression tree. If the current depth exceeds a globally configured maximum depth *MAX_EXPR* of the expression tree, the function will only generate leaf nodes, i.e., constants and variables. Otherwise, the function is also allowed to generate binary and unary expressions. The function generates valid constants of the specified basic type if the intended expression is a constant (line 7). If the expected expression is a variable, the function filters the variables in *var_set* to only include variables with the specified basic type, and then randomly selects a variable from the filtered set (line 11). If the expression type is unary or binary expressions, the function recursively generates the subexpressions while increasing the expression depth by 1 (lines 15, 19, and 20).

4.2 Divergence Analysis

Design Goal. The G_{cdiv} component is designed to generate shader programs with complex divergence patterns. Each statement is associated with a divergence type, which can be either divergent (δ) or uniform (μ). A statement s as divergent if any of the expressions composing the statement is divergent, and uniform otherwise. When generating a program with statements $s_1, s_2, \ldots, s_n$, G_{cdiv} aims to ensure that their associated divergence types $\omega_1, \omega_2, \ldots, \omega_n$ include a mix of uniform and divergent types, contributing to the overall complexity of the divergence pattern. To that end, G_{cdiv} independently samples the divergence type $\omega_i, i \in [1, n]$ for each statement s_i according to a Bernoulli distribution $\mathcal{D}$, which specifies the probability of each divergence type:

$$\omega_i \simeq \mathcal{D} = \begin{cases} \delta & \text{with probability } p_\delta \\ \mu & \text{with probability } 1 - p_\delta \end{cases} \tag{1}$$

The value of p_δ controls the proportion of divergent statements in the generated program, thus creating varied divergence patterns. We set $p_\delta = 0.2$ in our experiments to generate programs with a moderate amount of divergence.[2]

Technical Challenge. Our preliminary program generation efforts found that blindly generating programs can easily lead to all statements being either uniform or divergent. Single divergence patterns such as all statements being uniform/divergent can be easily compiled and optimized by the compiler back-ends, failing to adequately stress the compiler's optimization capabilities. As such, it is imperative to generate programs with a mix of uniform and divergent statements to provide more challenging test cases for the compiler.

To that end, G_{cdiv} explicitly models the divergence of each statement when generating the program, and generates programs that contain a mix of uniform and divergent statements. G_{cdiv} augments the random program generator G_{rand} by introducing a divergence analysis routine that tracks the divergence information and assists the generation process. Each time a new statement is to be generated, G_{cdiv} samples the divergence type of the statement according to the divergence distribution $\mathcal{D}$. It consults the results from divergence analysis during the generation process to ensure that the generated statement has the specified divergence type. The divergence analysis then tracks the divergence information from the newly generated statement. The generation process and divergence analysis work iteratively until the program is fully generated.

Divergence Environment. In graphics shader languages, programmers cannot directly specify the divergence type of a variable. Instead, it is determined automatically through program analysis. To manage this, G_{cdiv} uses a divergence environment, denoted as Σ. This environment keeps track of the divergence type (ω) for each variable. Additionally, a branch divergence stack, Φ, is used to monitor the divergence types of conditions within nested if-statements. This stack stores the divergence type of each conditional branch that the current code location is nested inside, reflecting the program's control flow divergence. These auxiliary data structures are illustrated in Fig. 8.

Divergence Analysis. The divergence environment Σ and the branch divergence stack Φ jointly determine the divergence type of an expression and are updated during the program generation process. We will first show the inference rules on the divergence type of an expression, and then introduce the rules for updating the divergence information for different kinds of statements.

[2]We will discuss the influence of hyperparameters, including p_δ, in Section 5.

Algorithm 3 Divergence type inference.

```
1: function DivInfer(e: Expression, Σ: Divergence environment, Φ: Branch divergence stack)
2:     for each variable $v \in \text{Vars}(e)$ do
3:         if $\Sigma(v) = \delta$ then
4:             return $\delta$
5:     for each divergence type $\omega \in \Phi$ do
6:         if $\omega = \delta$ then
7:             return $\delta$
8:     return $\mu$
```

Divergence Type Inference. Algorithm 3 shows the algorithm to infer the divergence type of an expression. The function DivInfer iterates all the variables referred to in the expression (denoted as $\text{Vars}(e)$). If any of them is divergent, the expression is inferred as divergent (lines 2–4). Moreover, if any of the branch conditions in the branch divergence stack is divergent, meaning that the analyzed expression is nested within a divergent branch, the expression is also inferred as divergent (lines 5–7). If neither of the conditions holds, the expression is inferred as uniform (line 8).

Algorithm 4 Divergence update rules for assignment statements.

```
1: function AssignUpdate(s: Assignment statement, Σ: Divergence environment, Φ: Branch divergence stack)
2:     $v \leftarrow \text{LHS}(s)$
3:     $e \leftarrow \text{RHS}(s)$
4:     $\omega \leftarrow \text{DivInfer}(e, \Sigma, \Phi)$
5:     $\Sigma(v) \leftarrow \omega$
```

Divergence Update Rules for Assignment Statements. Algorithm 4 illustrates the divergence update rules for assignment statements. The function AssignUpdate takes as input an assignment statement s, the divergence environment Σ, and the branch divergence stack Φ. It extracts the variable v being assigned from the left-hand side of the statement ($\text{LHS}(s)$) and the expression e assigned to v from the right-hand side ($\text{RHS}(s)$). The divergence type of the assignment expression e is inferred using the DivInfer function (line 4). The divergence type of variable v is then updated to the divergence type of e.

Algorithm 5 Divergence update before analyzing the branches of an if-statement.

```
1: function PreBranchUpdate(e: Branch condition, Σ: Divergence environment, Φ: Branch divergence stack)
2:     $\omega \leftarrow \text{DivInfer}(e, \Sigma, \Phi)$
3:     push$(\Phi, \omega)$
```

Divergence Update Rules for If-Statements. The environment update rules for if-statements require two separate updates: one before and one after the divergence analysis for the branches. Algorithm 5 shows the function PreBranchUpdate, which describes the update rules for the branch conditions and is called before entering the if-branches. It first infers the divergence type of the condition expression e using the DivInfer function (line 2) and then pushes the inferred divergence type onto the branch divergence stack (line 3) to record the control flow divergence.

Once the code generation for both the then- and else-branch is complete, the PostBranchUpdate function in Algorithm 6 is called. This signals the end of the nested if-statement, and the branch divergence stack is popped (line 2). During the generation of these branches, separate divergence environments, denoted as Σ_{then} and Σ_{else}, are used to independently track the divergence within the then- and else-branch (more details on these environments later). After the branches are generated,

Algorithm 6 Divergence update after analyzing the branches of an if-statement.

```
1: function PostBranchUpdate($\Sigma_{then}$: Divergence environment for then-branch, $\Sigma_{else}$: Divergence environment for else-branch, $\Phi$: Branch divergence stack)
2:     pop($\Phi$)
3:     $\Sigma_{merged} \leftarrow \{v : \Sigma_{then}(v) \mid v \in dom(\Sigma_{then}) \wedge v \notin dom(\Sigma_{else})\} \cup \{v : \Sigma_{else}(v) \mid v \in dom(\Sigma_{else}) \wedge v \notin dom(\Sigma_{then})\}$
4:     for each variable $v \in dom(\Sigma_{then}) \cap dom(\Sigma_{else})$ do
5:         if $\Sigma_{then}(v) = \delta \vee \Sigma_{else}(v) = \delta$ then
6:             $\Sigma_{merged}(v) \leftarrow \delta$
7:         else
8:             $\Sigma_{merged}(v) \leftarrow \mu$
9:     return $\Sigma_{merged}$
```

these two divergence environments are merged (lines 3–8). In line 3, the merged environment, Σ_{merged}, is initialized with divergence type mappings that exist in only one of the two branches. The function *dom* returns the set of variables (keys) present in a divergence environment. For a variable present in both Σ_{then} and Σ_{else}, its divergence type in Σ_{merged} is determined as follows: if the variable is divergent in either branch, it is marked as divergent; otherwise, it is considered uniform (lines 4–8). Finally, the merged divergence environment is returned.

Divergence Update Rules for For-Statements. In our program generator, we only generate for-loops with bounded iteration ranges. As such, the analysis of for-loops is straightforward — the divergence of the loop variable is uniform, and then the loop body is recursively analyzed. For brevity, we omit the details of the update rules for for-statements.

Divergence-Aware Program Generation. Algorithm 7 extends the pure random program generator of Algorithm 1 by incorporating the divergence analysis results. The differences are highlighted in commented lines.

Divergence Type Sampling. Each time before a statement is generated, the divergence type of the statement is sampled according to the divergence distribution $\mathcal{D}$ (line 6). Variables visible in the current scope are further filtered to obtain *div_var_set*, denoting the set of variables with consistent divergence types with the statement (line 7).

Assignment Statement Generation by G_{cdiv}. When generating an assignment statement, the generation of the right-hand side expression is allowed to use variables in *div_var_set* (line 11) to ensure that the composed expression has consistent divergence types with the specified divergence type. In line 12, the divergence environment is updated according to the AssignUpdate rule to track the new divergence type of the assigned variable.

If-Statement Generation by G_{cdiv}. First, a condition expression is created using variables from the *div_var_set*. This expression is designed to align with the specified divergence type (line 15). The PreBranchUpdate function is called to track the updated divergence information. The divergence environment is then forked into two copies, each tracking the divergence of the then-branch and else-branch, respectively (lines 16–17). The two branches are recursively generated with their respective divergence environments (lines 18–19). Finally, the PostBranchUpdate function is called to reflect the divergence information update after the exit of the if-statement (line 20).

For-Statement Generation by G_{cdiv}. The generation of the for-statement does not require extra divergence analysis, as we only generate loops that terminate in a fixed number of iterations. During the generation process, we recursively generate the body of the loop with the current divergence environment and branch divergence stack (line 25).

Algorithm 7 Divergence-aware statement generation.

```
1: function DivGenStmts($var\_set$: Variable set, $MAX\_N$: Max #stmts, $\Sigma$: Divergence environment, $\Phi$: Branch divergence stack)
2:     $n \leftarrow$ Sample$(0..\max(0, MAX\_N))$
3:     $stmts \leftarrow []$
4:     for _ $\in 1..n$ do
5:         $stmt\_type \leftarrow$ Sample({assign, if, ...})
6:         $\omega \leftarrow$ Sample$(\mathcal{D})$                      ▷ Sample the divergence type according to the distribution
7:         $div\_var\_set \leftarrow$ Filter$(\Sigma, var\_set, \omega)$    ▷ Filter variables with the specified divergence type
8:         if $stmt\_type$ = assign then
9:             $v \leftarrow$ Sample$(var\_set)$
10:            $\varphi \leftarrow$ BasicTypeOf$(v)$
11:            $expr \leftarrow$ GenExpr$(\varphi, div\_var\_set, 0)$     ▷ Generate an expression with the filtered variables
12:            AssignUpdate$(v := expr, \Sigma, \Phi)$                  ▷ Update the divergence environment
13:            Append$(stmts, v := expr)$
14:        else if $stmt\_type$ = if then
15:            $cond \leftarrow$ GenExpr(bool, $div\_var\_set$, 0)   ▷ Generate a condition expression with the filtered variables
16:            PreBranchUpdate$(cond, \Sigma, \Phi)$                   ▷ Update the branch divergence stack
17:            $\Sigma', \Sigma'' \leftarrow$ Copy$(\Sigma)$                              ▷ Fork the divergence environment
18:            $then\_stmt \leftarrow$ DivGenStmts$(var\_set, MAX\_N - 1, \Sigma', \Phi)$   ▷ Recursively generate the branches
19:            $else\_stmt \leftarrow$ DivGenStmts$(var\_set, MAX\_N - 1, \Sigma'', \Phi)$
20:            $\Sigma \leftarrow$ PostBranchUpdate$(\Sigma', \Sigma'', \Phi)$          ▷ Merge the divergence environments
21:            Append($stmts$, if $cond$ {$then\_stmt$} else {$else\_stmt$})
22:        else if $stmt\_type$ = for then
23:            $start \leftarrow$ Sample$(0..INT\_MAX - 3)$
24:            $end \leftarrow$ Sample$(1..3) + start$
25:            $body \leftarrow$ DivGenStmts$(var\_set, MAX\_N - 1, \Sigma, \Phi)$
26:            Append($stmts$, for $i$ in $start..end$ {$body$})
27:    return $stmts$
```

4.3 Liveness Extension

Design Goal. M_{liv} mutates the generated shader program from G_{cdiv} to complicate the divergence patterns of the data flow in the shader program. It aims to extend the liveness ranges of variables in the program. Given a shader program P with statement sequence $s_1, s_2, \ldots, s_n$, suppose the variable v is defined[3] at statement $s_i, i \in [1, n]$. M_{liv} introduces m new uses of the variable v at statements $s_{j_1}, s_{j_2}, \ldots, s_{j_m}, \forall k \in [1, m].s_{j_k} \in (i, n]$. By such generation scheme, the liveness of the variable v, $\textsc{Liv}(v)$, is extended from its original range OriLiv(v) to the newly added use sites in the program:

$$\textsc{Liv}(v) := \left[s_i, s_{j_1}\right] \cup \left[s_i, s_{j_2}\right] \cup \cdots \cup \left[s_i, s_{j_m}\right] \cup \textsc{OriLiv}(v) \tag{2}$$

Technical Challenge. Extending the liveness of variables requires introducing additional def-use relations. A naïve approach to implementing the liveness extension is to add uses at random locations for each definition site in the program. However, randomly adding uses can violate the divergence type maintained by G_{cdiv}, or even worse, introduce reading from uninitialized memory, which can lead to undefined behaviors during execution. To address this issue, we design a def-use aware liveness extension algorithm that crafts additional variable uses to the program.

Reachable Site Construction. For each definition site s that defines a variable v in the program, we add additional uses for v by mutation (introduced soon). We need to determine candidate sites for creating the additional uses. The candidate sites must satisfy the following property: for the given statement s' in the candidate sites, there is no other definition site of v between s and s'. In other words, by creating a variable read to v in s', we can establish a def-use relationship between

[3]We use the term "define" to refer to assigning a value to a variable.

s and s'. We denote the candidate sites as the definition reachable sites $\textsc{DefReach}(s, v)$, which are the set of statements that can be reached from s without encountering another definition site of v. $\textsc{DefReach}(s, v)$ can be computed by a fixed-point iteration algorithm similar to liveness analysis [Allen and Cocke 1976] that traverses the control flow graph of the program.

Introducing Additional Uses. After obtaining the definition reachable sites, which form the candidates for extending the liveness of variables, we introduce additional uses for each of the definition statements. The liveness extension algorithm is shown in Algorithm 8. The function *LivExt* takes as input the definition statement s that assigns a value to a set of variables, the divergence environment Σ, and the branch divergence stack Φ. For each variable v that is defined in the statement, the function first retrieves the definition reachable sites as potential candidates (*mut_set*) for mutation. It then infers the divergence type of the statement through its right-hand side expression (the definition statement is an assignment statement) using the DivInfer function (line 4). The algorithm then filters out the candidates with inconsistent divergence types (lines 5–7). A statement s' is randomly sampled from the candidates (line 8), and its assignment expression e is extracted from the statement (line 9). We then compose a binary expression with e that issues a variable read to v (lines 10 and 11). The newly constructed expression e' is inserted back into the statement s' (line 12). As such, we introduce extra uses for the variable v, and the liveness range of the variable includes an additional range from s to s'. By repeating the process for all statements in the program, we can effectively complicate the data flow in the program and stress-test the register/memory-related optimizations in the compiler back-ends.

Algorithm 8 Liveness extension.

```
1:  function $\textit{LivExt}$($s$: Definition statement, $\Sigma$: Divergence environment, $\Phi$: Branch divergence stack)
2:      for $v \in \textsc{Kill}(s)$ do
3:          $mut\_set \leftarrow \textsc{DefReach}(s, v)$
4:          $\omega \leftarrow \textsc{DivInfer}(\text{RHS}(s), \Sigma, \Phi)$      ▷ Infer the divergence type of the definition (assignment) statement
5:          for $mut\_s \in mut\_set$ do
6:              if $\textsc{DivInfer}(\text{RHS}(mut\_s), \Sigma, \Phi) \neq \omega$ then
7:                  $mut\_set \leftarrow mut\_set \setminus \{mut\_s\}$      ▷ Filter out statements with inconsistent divergence type
8:          $s' \leftarrow \textsc{Sample}(mut\_set)$
9:          $e \leftarrow \text{RHS}(s')$
10:         $\otimes \leftarrow \textsc{Sample}(\text{valid binary operators})$      ▷ Randomly select a binary operator
11:         $e' \leftarrow e \otimes v$
12:         $s' \leftarrow \textsc{Subs}(s', e, e')$      ▷ Substitute the expression with the new expression
```

5 Implementation

We implemented ShaDiv in over 7K lines of Python code. We first generate an IR for the shader program, and then translate it to the GLES 3.0 shader language [Group 2024c] through a simple rule-based translator. In line with research on testing traditional C/C++ compilers [Livinskii et al. 2020], we have limited support for floating-point arithmetic to avoid generating numerically unstable programs. We limit the floating-point values to the range $[-4.0, 4.0]$ to avoid large numerical errors. We also use the program reconditioning technique [Lecoeur et al. 2023], which wraps operations with safe arithmetic functions, to avoid undefined behaviors in the generated shader programs.

Hyperparameter Settings. As with other generation-based testing tools such as CSmith [Yang et al. 2011], we also model the probability of generating a specific program feature, such as if-statements and assign-statements, using a uniform probability distribution. We also maintain hyperparameters like the maximum number of nested if-statements and the maximum number of statements in a basic block, to constrain the size of the test cases. To ensure the reproducibility of our experiments, we release all the hyperparameters and the probability distribution that we used in our experiments in [ShadeDiv-Artifact 2024]. We list several key hyperparameters in Table 1. To evaluate the sensitivity of our findings to these hyperparameters, we ran five fuzzing campaigns with randomly mutated settings for 24 hours. We found that these campaigns differ by at most one bug. The consistent bug detection rates indicate that ShaDiv's performance is not significantly affected by specific hyperparameter values.

Table 1. Representative hyperparameters in the experiments.

Hyperparameter	Value
Probability of generating divergent statements	0.2
Probability of generating an if-statement	0.2
Probability of generating an assign-statement	0.5
Maximum #nested if-statements	5
Maximum #statements in a block	12

6 Experiment Results

6.1 Experiment Setup

Evaluation Targets. We evaluate ShaDiv on the graphics shader compilers from four leading GPU vendors: Intel, NVIDIA, AMD, and ARM. We show the GPU hardware information in Table 2. Three of the evaluated GPUs are for desktops, and one is for mobile devices. Since graphics shader compilers are shipped with the GPU drivers, we also show the driver versions in Table 2. We use the latest version of the drivers (by the time of writing) to ensure that we are using the most recent bug fixes, which is a practice recommended by the GPU vendors. All of our tested shader compilers are commercial products with high quality and were extensively fuzzed by a fuzzing company acquired by Google [Donaldson et al. 2020; Ltd. 2018]. Our experiments are conducted on the Vulkan API [vul 2024], which is a widely used graphics API to interact with the GPU hardware. We generate 40K testing input and run them on the four GPUs under test.

Table 2. Hardware and driver versions used for the experiments.

Vendor	Hardware	Driver Version
Intel	Intel HD Graphics 730	32.0.101.5972
NVIDIA	GeForce GTX 3060	560.94
AMD	Radeon RX 6400	32.0.11037.4004
ARM	Mali-G715	v1.r38p1

6.2 Overall Effectiveness and Efficiency

Testing Time. We record the time of generating the testing inputs and the time of compiling and executing the generated programs. The program generation incurs a low overhead, with an average of 45 milliseconds to generate a program. The compilation and execution time, however, is significantly higher, with an average of 7.2 seconds per program. It takes on average 80 hours to compile and execute 40K testing inputs on a single GPU under test.

The time statistics are consistent with the complexity of the generated programs. The complex divergent patterns of both the data and control flow pose a unique challenge to the back-end compilers, which can lead to a long compilation and execution time. Despite the long execution time, the complexities of the generated programs are essential to stress the back-end compilers and uncover potential bugs. We will show that the long execution time is worthwhile, as ShaDiv is able to uncover a large number of bugs in the back-end compilers.

Identifying Errors. As mentioned in Section 4, we capture both crashes and deviant outputs as errors. For crashes, we rely on the return code of the execution process to determine whether a crash occurs, as most of the tested shader compilers are shipped with graphics drivers, which are closed-source and do not provide crash traces. It is thus impossible for us to distinguish whether the crash happens during compilation or runtime execution. For deviant outputs, we compare the outputs with a CPU-based shader compiler, the SwiftShader [Google 2024], which is an industry-strength shader compiler maintained by Google. Since CPU-based shader compilers do not have the challenges of GPU-based compilers and are considered simpler, we use SwiftShader as the reference to determine the correctness of the outputs. We compare the output images from the tested GPUs with the SwiftShader. If the average pixel-wise difference between the compared images is larger than a threshold, we deem the output as incorrect. Our threshold selection follows the same practice as GLFuzz and FShader, i.e., selecting thresholds above which human eyes can identify the image difference. We set the threshold to a safe value of 20 so that the false positives are minimized.

Differentiating Back-End Errors. Our experiments are performed on Vulkan APIs. When compiling a Vulkan shader program, the front- and middle-end shader optimizations are performed by the SPIR-V Tools [Group 2024b], and the intermediate representation from the middle-end, the SPIR-V [Group 2024d], is then passed to the back-end compiler for the back-end optimizations and code generation. If an error occurs in the front-end or middle-end, then all the back-end compilations will be affected; if an error occurs in the back-end compiler of a certain GPU, then it is likely that not all the GPUs will have the same error. As such, we deem an error as a back-end error if it is only triggered on a specific GPU, and a front-end or middle-end error if it is triggered on all the GPUs. We may have false negatives in this classification, as some back-end errors may be triggered on all GPUs due to the similarity of the back-end compilers. However, we do not observe such cases in our experiments.

Uncovered Error-Triggering Inputs. We show the number of error-triggering inputs in Table 3. In total, with 40K generated shader programs, we uncovered over 2K error-triggering inputs that induce either crashes or deviant outputs. We also show the number of errors that happen in the compiler back-end. Among over 2.7K error-inducing inputs in total, there are 2.5K inputs that trigger errors in the compiler back-ends. Such a high proportion indicates our effectiveness in uncovering bugs in the back-end compilers, which can be attributed to the carefully designed back-end-oriented generation schemes in ShaDiv.

Table 3. #Error-triggering inputs discovered by 40K testing inputs.

Vendor	#Crashes		#Deviant Outputs		Total Errors	
	Total	Back-end	Total	Back-end	Total	Back-end
AMD	1,788	1,760	60	22	1,848	1,782
Intel	449	429	21	6	470	435
NVIDIA	185	180	82	45	267	225
ARM	0	0	137	125	137	125
Total	2,422	2,369	300	198	2,722	2,567

Error Deduplication and Bug Analysis. To decide the unique root causes, a.k.a. bugs, that trigger the errors, we first reduce the error-triggering inputs with Perses [Sun et al. 2018] and C-Reduce [Regehr et al. 2012]. Two authors of this paper, who are also experts in graphics shaders, individually check the minimized programs to verify that the programs are not false positives

Table 4. Uncovered bugs in the tested graphics shader compilers.

Vendor	#Crash Bugs	#Deviant Output Bugs	Total	Confirmed	Fixed
Intel	1	2	3	2	0
AMD	5	1	6	6	4
NVIDIA	1	1	2	0	0
ARM	0	1	1	1	0
Non-Back-End	1	1	2	2	2
Total	8	6	14	11	6

(discussed soon) and then cross-check with each other to reach a consensus. This step is costly and takes a month to complete. We then group the error-triggering inputs with similar erroneous symptoms and program features (e.g., language constructs, control/data flow patterns). In total, we find 14 unique groups of error-triggering inputs. To avoid creating a large number of bug reports, we report one input in each group to the developers for confirmation. 11 are promptly confirmed by the developers. Since the remaining three have distinct erroneous symptoms and program features, we believe that they are genuine and unique bugs. We list the uncovered bugs in Table 4.[4] Six bugs are fixed by the time of writing, and it takes time for the developers to fix the remaining bugs given the complexity of the back-end compilers. Due to the proprietary and closed-source nature of the back-end compilers, vendors did not disclose the root causes of identified bugs. Understanding the root causes through the compiler binary code is also challenging since the binary is highly complex and optimized, making it difficult to recover the high-level logic.

False Positive (FP) Analysis. We consider three cases as FPs: ① the input shader program contains UBs; ② the erroneous behaviors are not reproducible; ③ the deviant outputs are due to the numerical instability, or the deviant images are visually indistinguishable from the reference images. We do not find any type ① FPs, which is expected since our generated programs are designed to be UB-free (see Section 5). We find 28 cases of type ② FPs and 19 cases of type ③ FPs. In total, we have less than 2% of FPs in our experiments, meaning that our uncovered error-triggering inputs are mostly due to true compiler bugs.

Bug Examples. To understand the characteristics of our uncovered bugs, we showcase several representative error-triggering shader programs in Fig. 9 and Fig. 10. Each bug is triggered by a unique divergent interleaving or extended liveness pattern, which is carefully designed by ShaDiv to stress the back-end compilers.

Bug_1: Divergent Loop Breaks Cause the AMD Compiler to Hang. In Fig. 9a, we show a bug uncovered by the divergence interleavings from G_{cdiv}. The variable `t` receives the value from the x-value of the pixel coordinates `coord`, and is thus a divergent variable; the variable `w` receives a constant value and is uniform. Inside the nested for-loop at lines 2 and 3, the three if-conditions have distinct divergence patterns: the conditions in lines 4 and 9 are divergent, while the condition in line 7 is uniform. Although the nested loop should exit in 4 iterations at most (the loop guardians `uni1` and `uni2` are uniform input variables set to 2 and 0, respectively), the AMD compiler hangs.

Bug_2: Divergent Returns Induces Deviant Outputs. We show a program inducing deviant outputs in the AMD compiler in Fig. 9b. The output pixel color `color` is uniformly assigned with values at the beginning and the end of the program. Inside the for-loop, there are three return statements guarded by different divergence conditions. The return statements in lines 5 and 9 are divergent,

[4]The distribution of confirmed and fixed bugs varies among GPU vendors, influenced by their manpower and our communication speed. Some vendors use non-technical customer support for bug reports, which can raise hurdles in a "down to the earth" discussion with developers.

```
 1 int w = 0, t = int(coord.x);
 2 for (int i = 0; i < uni1; i++) {
 3   for (int j = uni2; j < 2; j++) {
 4     if (t < uni1) // divergent
 5       break;      // divergent
 6   }
 7   if (w > 0)      // uniform
 8     w = t;        // divergent
 9   if (!w)         // divergent
10     break;
11 }
```

(a) Example bug that triggers hangings in the AMD shader compiler.

```
 1 color = ...              // uniform
 2 for (int i = 0; i < 1; i ++) {
 3   if (coord.x < 56.0)    // divergent
 4     if (coord.x < 56.0) // divergent
 5       return;
 6   if (i > 2)             // uniform
 7     return;
 8   if (coord.x < 56.0)    // divergent
 9     return;
10 }
11 color = ...              // uniform
```

(b) Example bug that triggers deviant outputs in the AMD shader compiler.

Fig. 9. Example error-triggering shader programs found by divergent interleavings.

```
1 int f = uni1, g = 1, t = -40000;
2 for (int i = t; i > t - 4; i -= 2)
3   if (!g)
4     g = f + 1;
5   else
6     g = f / g; // Extended use of g
```

(a) A bug causing segmentation faults in the Intel shader compiler.

```
1 int m = 0, n = 0, o = 0;
2 for (int i = 1; i < 3; i++) {
3   o = int(coord) ^ m; // Extend m
4   m = int(coord) == o;
5   n = vec4(m);
6 }
```

(b) A bug causing segmentation faults in the AMD shader compiler.

Fig. 10. Example error-triggering shader programs found by extended liveness.

while the one in line 7 is uniform. Such divergence interleavings confuse the AMD compiler, which leads to incorrect assignment to the output color.

Bug$_3$*: Extended liveness Causes Segfaults in the Intel Compiler.* Fig. 10a shows a program that triggers segmentation faults in the Intel compiler. The liveness extension from M_{liv} adds a read to the variable g in the division operation in line 6. Without the extension, g is considered dead after line 3 since it will be overwritten. The added read to g, however, effectively extends its liveness to line 6, and the Intel compiler fails to handle such cases correctly and crashes.

Bug$_4$*: Introduced Variable Read Leads to Crashes in the AMD Compiler.* Fig. 10b shows a case where the extended liveness causes the AMD compiler to throw a segmentation fault. The variable `m` is initialized with 0 at the beginning, yet is overwritten in line 4 without being read. By introducing a read to `m` through the XOR operation in line 3, the liveness of `m` in line 1 is extended to line 4, which then causes the segmentation fault in the AMD compiler.

6.3 Effectiveness of Individual Components

To understand the effectiveness of individual components in ShaDiv, we enable each component separately and generate 10K testing inputs for each component. Since G_{cdiv} and M_{liv} depend on the generated program from G_{rand}, we always enable G_{rand} when evaluating the individual contributions of G_{cdiv} and M_{liv}. For comparison purposes, we also set up the full ShaDiv under the same settings. We run the generated testing inputs on all four shader compilers, then collect error-triggering inputs and analyze the underlying bugs.

Table 5. Effectiveness of individual components by generating 10K testing inputs each.

Component	#Crashes	#Deviant Outputs	Total #Errors	#Bugs
G_{rand}	25	1	26	2
G_{cdiv}	135	7	142	6
M_{liv}	491	27	518	8
ShaDiv	654	97	751	11

We show the results in Table 5. G_{rand} uncovers the least number of errors, with 26 in total. This is expected, since without the divergence interleavings and extended liveness, the generated programs are less likely to trigger interesting optimizations in the back-end compilers and are thus less likely to uncover bugs. G_{cdiv} finds 142 error-triggering inputs, a significantly higher number than G_{rand}. The divergence interleavings generated by G_{cdiv} are effective in stressing the control flow optimizations in the back-end compilers, which leads to a higher number of errors. M_{liv} also improves the effectiveness, with 518 error-triggering inputs in total. M_{liv} generates programs with extended liveness ranges and significantly complicates the data flow-related optimizations and code generation in the compilers. None of the three components, however, is as effective as the full ShaDiv, which uncovers the most number of errors and bugs. By combining the strengths of G_{rand}, G_{cdiv}, and M_{liv}, ShaDiv is able to generate complex shader programs that exhibit both divergent control and data flow patterns, which are essential for bug detection.

6.4 Comparison with Existing Tools

We compare the effectiveness and efficiency of ShaDiv with other existing tools that also aim to find bugs in graphics shader compilers. To the best of our knowledge, GLFuzz [Donaldson et al. 2017] and FShader [Xiao et al. 2023] are the most relevant tools to compare with ShaDiv. We set up GLFuzz, FShader, and ShaDiv to generate 10K testing inputs according to their default settings, respectively. Since both GLFuzz and FShader are mutation-based approaches and require seed shader programs as a basis for mutation, we use their own seed programs ported from their original repositories. We run all the tools on the four shader compilers mentioned in Section 6.1. Each tool is allowed to run for 24 hours. To mitigate the impact of randomness, we repeat each experiment four times and report the average statistics.

Table 6. Comparison with existing tools under the budget of 10K testing inputs and 24-hour execution time.

Tool	#Crashes		#Deviant Outputs		Total #Errors		#Bugs	
	Total	Back-end	Total	Back-end	Total	Back-end	Total	Back-end
ShaDiv	107	99	5	3	112	102	5	4
GLFuzz	58	11	42	0	100	11	5	1
FShader	8	0	77	0	85	0	2	0

Overall Comparison Results. We report the results of the comparison study in Table 6. The number of deviant errors found by GLFuzz and FShader is significantly higher than by ShaDiv, which is expected since both GLFuzz and FShader are based on metamorphic testing and are tailored towards finding deviant outputs. In spite of that, ShaDiv excels in uncovering back-end bugs and finds 4 times as many back-end bugs as GLFuzz and FShader combined. Although both GLFuzz and ShaDiv find 5 bugs in total, ShaDiv finds 4 back-end bugs, while GLFuzz only finds 1. Without the carefully designed divergence interleavings and extended liveness patterns, GLFuzz and FShader can hardly detect the back-end bugs that ShaDiv uncovers, which requires specific

Table 7. Coverage improvement on individual components of the open-source compiler back-end.

Register spilling	Execution mask insertion	Instruction scheduling	Register allocation
72%	27%	17%	14%
Assembler	**Instruction selection**	**Lowering**	**Post-register allocation**
25%	49%	17%	17%

control and data flow patterns to trigger. We have shown some examples of such bugs in Fig. 9 and Fig. 10.

Coverage Comparison. To understand the difference in the effectiveness of testing shader compiler back-ends, we run ShaDiv, FShader, and GLFuzz for 24 hours and measure the line coverage on the compiler back-end. However, nearly all the tested compilers are closed-source due to the proprietary nature of the GPU drivers, which makes it impossible to measure the code coverage directly. We measure the coverage on an open-source AMD compiler back-end [3D 2024], which is, to the best of our knowledge, the only production-ready compiler back-end with publicly available source code. We show the line coverage achieved over 24 hours in Fig. 11. GLFuzz and FShader achieve nearly identical performance, with 41.5% and 41.9% of line coverage, respectively. ShaDiv achieves 52.1% of line coverage and improves by 25% over GLFuzz and FShader.

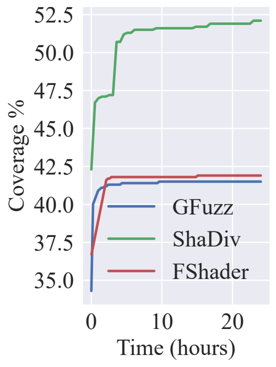

Fig. 11. Line coverage on the AMD compiler back-end over 24 hours.

Analysis on Individual Components. We further inspect the coverage improvement by ShaDiv for individual components in the compiler back-end. The coverage improvement for each component is listed in Table 7. Although our method is designed to generate programs with divergence and liveness features, it does not only stress-test these two components but also the vast components in the compiler backends. Components like register spilling and instruction selection have also seen significant coverage improvement due to the influence of divergence and liveness patterns. Such observation indicates that ShaDiv is able to thoroughly stress-test compiler back-ends. ShaDiv also finds three bugs in different components of the open-source compiler back-end, including instruction optimizer, execution mask insertion, and instruction lowering, demonstrating the variety of back-end bugs that ShaDiv is able to uncover.

7 Discussion

Testing vs. Verification. ShaDiv adopts a testing approach to validate the correctness of the shader compiler back-ends. In line with other testing works [Le et al. 2014, 2015; Yang et al. 2011], ShaDiv cannot guarantee the absence of bugs in the back-ends of shader compilers. Verification, on the other hand, is a heavyweight approach that can formally prove the correctness of the compilers. However, shader compiler back-ends are highly complex and composed of intricate optimization passes. It is thus challenging to formally verify the correctness of the back-ends. Our testing-based approach offers less than 2% of false positives, and the error-triggering inputs can be used as a basis for developers to debug and patch the bugs.

Accommodating Variances of Different GPUs. ShaDiv generates test cases based on the special features in the SIMT model of shader executions, which must handle the divergence issue in control/data flow. The SIMT model is a fundamental execution model in modern GPUs. As such, we deem our approach to be *general* and *applicable* to a wide range of shader compiler back-ends. We have also demonstrated in Section 6.4 that ShaDiv is able to achieve promising code coverage in the back-end. In spite of that, we are also aware that different GPUs may have nuanced differences in their hardware architectures. For instance, the register file size and the instruction set architecture may vary across different GPUs. Such differences can lead to different heuristics in the back-end optimizations and may require specialized test case generation schemes for individual GPUs. Nonetheless, since we have open-sourced ShaDiv, the community can easily extend ShaDiv to launch specialized testing towards a particular GPU by modifying test case generation strategies.

White-Box Testing vs. Black-Box Testing. ShaDiv is a black-box testing tool that generates test cases without relying on internal implementations of compilers. We believe that black-box testing is more practical and general than white-box testing, as it does not require the source code of the compiler back-ends. To the best of our knowledge, the source code of mainstream shader compiler back-ends is proprietary and not publicly available. As such, it is challenging to perform white-box testing. In contrast, black-box testing does not require the compiler source code and can be easily applied to test proprietary shader compilers. In addition, black-box testing is more general as it does not focus on a particular implementation of the compiler, and can be applied to various shader compilers with minimal effort.

Challenges to Coverage-Guided Fuzzing. Coverage-guided fuzzing, while effective in many contexts, faces significant obstacles when applied to graphics shader compilers. Instrumenting these compilers to collect coverage data is a major challenge. The closed-source and proprietary nature of many compilers precludes source-level instrumentation. Moreover, we have not identified readily available tools suitable for binary instrumentation in this domain. Commercial shader compilers are integrated into graphics drivers, necessitating substantial reverse-engineering efforts for effective instrumentation. Existing tools like NVBit [Villa et al. 2019] are limited to intercepting API calls between the application and the driver, and cannot analyze the internal state of the graphics drivers. Furthermore, the computational overhead of binary instrumentation is substantial. Given that our uninstrumented fuzzing campaigns already require 80 hours, the added cost would be prohibitive.

Alternatives to Our Testing Approach. We have considered several alternative approaches to testing shader compilers, yet found them ineffective in uncovering back-end bugs. We have preliminarily explored generating programs towards components that are not specific to the SIMT execution model, like value numbering and instruction lowering, yet we only discovered trivial bugs. We have also tried to generate programs that manipulate the shared memory. The access to the shared memory in shaders is typically performed by sampling the textures, which can induce floating-point errors during the sampling process and severe false positive issues (over 60% per our observation). In contrast, our method only incurs less than 2% of false positives and is more suitable for identifying compiler bugs.

Extension to Other GPU Compilers. ShaDiv is evaluated on fragment shader compilers. Other graphics shaders, such as vertex shaders, share the same compiler as fragment shaders. Hence, bugs found by ShaDiv on fragment shader compilers may also affect other types of graphics shader compilations. CUDA programs and compute shaders also feature the SIMT execution model; we believe that ShaDiv can also benefit the testing of those compilers, provided with extra engineering efforts to support their specific language features. Extension to SIMD vectorization compilers, however, requires different testing strategies due to differences in the execution model.

Limitations of Language Feature Support. ShaDiv currently supports a subset of the GLSL language features. We do not generate test cases that involve structures, arrays, or a variety of built-in GLSL functions. We hypothesize that supporting more language features may not be helpful in testing the back-ends of shader compilers. Structures and arrays will be resolved into scalar variables in the scalarization pass of the compiler middle-end, and built-in functions will be inlined into basic arithmetic operations. Hence, the high-level language features in the source program will be invisible to the back-end. Although there can be bugs that can only be triggered by our unmodelled language features, we hypothesize that they are less likely to be back-end bugs.

8 Related Work

Property-Based Testing. Property-based testing (PBT) [Claessen and Hughes 2000; Xiong et al. 2024] relies on defining high-level properties that the system must satisfy and uses randomized input generation to explore a vast input space. Pałka et al. [Pałka et al. 2011] generate typed functions on lists to test Haskell compilers. Midtgaard et al. [Midtgaard et al. 2017] test OCaml compilers by generating programs according to a specified type and effect system. Frank et al. [Frank et al. 2024] generate well-typed functions while avoiding unused arguments. Our random program generator shares a similar goal as these works by generating well-typed programs in a top-down manner. Since the typing systems of graphics shader languages are relatively simple, e.g., no higher-order functions, the type constraints are straightforward to satisfy. Effective testing of shader compiler back-ends, however, requires testing inputs with tailored properties besides passing type checking. Our approach crafts shader programs with unique characteristics specific to the SIMT model, an area that existing PBT works haven't explored.

Compiler Testing. One of the prominent methods for testing compilers is metamorphic testing (MT) [Chen et al. 2020], which asserts metamorphic relations (MRs) for the tested compiler. MT has been applied to C compilers [Le et al. 2014, 2015; Li et al. 2024a; Sun et al. 2016], just-in-time compilers [Li et al. 2023], zero-knowledge compilers [Xiao et al. 2025], multi-party computation compilers [Li et al. 2024b], and deep learning compilers [Ma et al. 2023; Xiao et al. 2022]. Differential testing (DT) [McKeeman 1998] checks the correctness of compiled executables by comparing the outputs of different compilers or compiler versions. DT has been applied to test C compilers [Even-Mendoza et al. 2022; Livinskii et al. 2020; Theodoridis et al. 2022a,b; Yang et al. 2011; Zhang et al. 2017], decompilers [Liu and Wang 2020; Lu et al. 2024], and webassembly compilers [Liu et al. 2023]. Our work employs DT to check the correctness of the graphics shader compiler back-ends. There are also works that explore testing of a specific compiler component, e.g., the typing system [Chaliasos et al. 2022; Sotiropoulos et al. 2024] and loop optimizations [Livinskii et al. 2023]. Our work shares a similar study scope with these works by focusing on the back-end of graphics shader compilers.

Graphics Shader Compiler Testing. GLFuzz [Donaldson et al. 2017] and FShader [Xiao et al. 2023] are both based on MT to test graphics shader compilers. GLFuzz performs semantics-preserving mutations, while FShader fuses shaders by vision-inspired MRs and compares outputs with directly fused images. Spirv-fuzz [Donaldson et al. 2021] further extends GLFuzz with more test case generation schemes and an automated test case reduction algorithm to test SPIR-V compilers. The spirv-fuzz tool is now merged into GLFuzz. GLSLsmith [Lecoeur et al. 2023] fuzzes compute shaders by transforming randomly generated programs into UB-free programs. However, these tools are back-end agnostic and cannot effectively test back-end optimizations.

9 Conclusion

We propose ShaDiv, a generation-based testing tool designed specifically to uncover bugs in the graphics shader compiler back-ends. Coupled with two novel generation strategies, ShaDiv effectively uncovers 14 bugs in industrial shader compilers from four GPU vendors. We believe that ShaDiv can facilitate the development of more robust and reliable graphics shader compilers and can benefit users whose applications rely on the correctness of the shader compilers. We have released ShaDiv as an open-source tool at [ShadeDiv-Artifact 2024] and will maintain it constantly in the future. We welcome feedback and contributions from the community.

Acknowledgements

We thank the reviewers and shepherd for their valuable comments and suggestions. We also thank Prof. Wei Meng from the Chinese University of Hong Kong for his participation in the discussions of this work.

References

2024. Vulkan. https://www.vulkan.org/.

Mesa 3D. 2024. ACO: back end compiler for AMD GCN / RDNA GPUs. https://gitlab.freedesktop.org/mesa/mesa/-/blob/main/src/amd/compiler/README.md.

Frances E. Allen and John Cocke. 1976. A program data flow analysis procedure. *Commun. ACM* 19, 3 (1976), 137.

Anonymous. 2024. GPU hangs on AMD Radeon RX 6400 on a fragment shader. https://gitlab.freedesktop.org/mesa/mesa/-/merge_requests/297xx.

Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. 2022. Finding typing compiler bugs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* 183–198. https://doi.org/10.1145/3519939.3523427

Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 2020. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543* (2020).

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming.* 268–279. https://doi.org/10.1145/351240.351266

CNN. 2024. This VR modeling tool is revolutionizing 3D design | CNN Business. https://edition.cnn.com/2024/10/11/business/video/shapelab-3d-design-spc.

Tobias CornilleMarch. 2024. Software Tools For Robotics Landscape (2024) | Segments.ai. https://segments.ai/blog/software-tools-for-robotics-landscape/.

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

Alastair F Donaldson, Hugues Evrard, and Paul Thomson. 2020. Putting randomized compiler testing into production (experience report). In *34th European Conference on Object-Oriented Programming (ECOOP 2020).* Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 22–1. https://doi.org/10.4230/LIPIcs.ECOOP.2020.22

Alastair F Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. 2021. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* 1017–1032. https://doi.org/10.1145/3453483.3454092

Faith Ekstrand. 2022. In defense of NIR. https://www.gfxstrand.net/faith/blog/2022/01/in-defense-of-nir/.

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

Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023).* Association for Computing Machinery, New York, NY, USA, 1219–1231. https://doi.org/10.1145/3597926.3598130

Justin Frank, Benjamin Quiring, and Leonidas Lampropoulos. 2024. Generating Well-Typed Terms That Are Not "Useless". *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 2318–2339. https://doi.org/10.1145/3632919

Michael Garland and David B Kirk. 2010. Understanding throughput-oriented architectures. *Commun. ACM* 53, 11 (2010), 58–66. https://doi.org/10.1145/1839676.1839694

Glob. 2022. The trouble with SPIR-V, 2022 edition - Gob's blog. https://xol.io/blah/the-trouble-with-spirv/.

Google. 2024. GitHub - google/swiftshader: SwiftShader is a high-performance CPU-based implementation of the Vulkan graphics API. Its goal is to provide hardware independence for advanced 3D graphics. https://github.com/google/swiftshader.

Khronos Group. 2024a. Core Language (GLSL) - OpenGL Wiki. https://www.khronos.org/opengl/wiki/Core_Language_(GLSL).

Khronos Group. 2024b. GitHub - KhronosGroup/SPIRV-Tools. https://github.com/KhronosGroup/SPIRV-Tools.git.

Khronos Group. 2024c. OpenGL ES 3.0 Reference Pages. https://registry.khronos.org/OpenGL-Refpages/es3.0/.

Khronos Group. 2024d. SPIR Overview - The Khronos Group Inc. https://www.khronos.org/api/spir.

Autodesk Inc. 2024. Help | Maya Shaders | Autodesk. https://help.autodesk.com/view/ARNOL/ENU/?guid=arnold_for_maya_shaders_am_Maya_Shaders_html.

Francisco Jerez. 2018. Represent divergent control flow paths caused by non-uniform loop execution. https://gitlab.freedesktop.org/mesa/mesa/-/commit/4d1959e69328cf0d59f0ec7aeea5a2b704ef0c5f.

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226. https://doi.org/10.1145/2594291.2594334

Vu Le, Chengnian Sun, and Zhendong Su. 2015. Randomized stress-testing of link-time optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 327–337. https://doi.org/10.1145/2771783.2771785

Bastien Lecoeur, Hasan Mohsin, and Alastair F. Donaldson. 2023. Program Reconditioning: Avoiding Undefined Behaviour When Finding and Reducing Compiler Bugs. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 180:1801–180:1825. https://doi.org/10.1145/3591294

Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. 2023. Validating JIT compilers via compilation space exploration. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 66–79. https://doi.org/10.1145/3600006.3613140

Shaohua Li, Theodoros Theodoridis, and Zhendong Su. 2024a. Boosting compiler testing by injecting real-world code. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 223–245. https://doi.org/10.1145/3656386

Yichen Li, Dongwei Xiao, Zhibo Liu, Qi Pang, and Shuai Wang. 2024b. Metamorphic testing of secure multi-party computation (MPC) compilers. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1216–1237. https://doi.org/10.1145/3643781

Zhibo Liu and Shuai Wang. 2020. How far we have come: Testing decompilation correctness of C decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 475–487. https://doi.org/10.1145/3395363.3397370

Zhibo Liu, Dongwei Xiao, Zongjie Li, Shuai Wang, and Wei Meng. 2023. Exploring missed optimizations in webassembly optimizers. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 436–448. https://doi.org/10.1145/3597926.3598068

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–25. https://doi.org/10.1145/3428264

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 181:1826–181:1847. https://doi.org/10.1145/3591295

GraphicsFuzz Ltd. 2018. GraphicsFuzz is acquired by Google. https://www.graphicsfuzz.com/.

Yifei Lu, Weidong Hou, Minxue Pan, Xuandong Li, and Zhendong Su. 2024. Understanding and finding Java decompiler bugs. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 1380–1406. https://doi.org/10.1145/3649860

Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. 2023. Fuzzing deep learning compilers with hirgen. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 248–260. https://doi.org/10.1145/3597926.3598053

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

Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. 2017. Effect-driven QuickChecking of compilers. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–23. https://doi.org/10.1145/3110259

John Nickolls and William J Dally. 2010. The GPU computing era. *IEEE micro* 30, 2 (2010), 56–69. https://doi.org/10.1109/MM.2010.41

John Nickolls and David Kirk. 2009. Graphics and computing GPUs. *Computer Organization and Design: The Hardware/Software Interface, DA Patterson and JL Hennessy, 4th ed., Morgan Kaufmann* (2009), A2–A77.

Roman Novak. 2014. Loop optimization for divergence reduction on GPUs with SIMT architecture. *IEEE Transactions on Parallel and Distributed Systems* 26, 6 (2014), 1633–1642. https://doi.org/10.1109/TPDS.2014.2324587

Michał H Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*. 91–97. https://doi.org/10.1145/1982595.1982615

Fernando Magno Quintao Pereira and Jens Palsberg. 2005. Register allocation via coloring of chordal graphs. In *Asian Symposium on Programming Languages and Systems*. Springer, 315–329. https://doi.org/10.1007/11575467_21

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 335–346. https://doi.org/10.1145/2254064.2254104

Diogo Sampaio, Rafael Martins de Souza, Caroline Collange, and Fernando Magno Quintão Pereira. 2014. Divergence analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 35, 4 (2014), 1–36.

ShadeDiv-Artifact. 2024. Research artifacts. https://sites.google.com/view/ggenshader/.

Thodoris Sotiropoulos, Stefanos Chaliasos, and Zhendong Su. 2024. API-Driven Program Synthesis for Testing Static Typing Implementations. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 1850–1881. https://doi.org/10.1145/3632904

Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 849–863. https://doi.org/10.1145/2983990.2984038

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

Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. 2022a. Understanding and exploiting optimal function inlining. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 977–989. https://doi.org/10.1145/3503222.3507744

Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022b. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 697–709. https://doi.org/10.1145/3503222.3507764

Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 372–383. https://doi.org/10.1145/3352460.3358307

Preeti Wadhwani. 2024. 3D Rendering Market Size | Global Statistics Report, 2032. https://www.gminsights.com/industry-analysis/3d-rendering-market.

Dongwei Xiao, Zhibo Liu, Yiteng Peng, and Shuai Wang. 2025. Mtzk: Testing and exploring bugs in zero-knowledge (ZK) compilers. In *NDSS*. https://doi.org/10.14722/ndss.2025.230530

Dongwei Xiao, Zhibo Liu, and Shuai Wang. 2023. Metamorphic shader fusion for testing graphics shader compilers. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2400–2412. https://doi.org/10.1109/ICSE48619.2023.00201

Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Metamorphic Testing of Deep Learning Compilers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 1 (2022), 1–28. https://doi.org/10.1145/3489048.3522655

Yiheng Xiong, Ting Su, Jue Wang, Jingling Sun, Geguang Pu, and Zhendong Su. 2024. General and Practical Property-based Testing for Android Apps. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 53–64. https://doi.org/10.1145/3691620.3694986

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*. 283–294. https://doi.org/10.1145/3691620.3694986

Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*. 347–361. https://doi.org/10.1145/3062341.3062379

Received 2024-11-15; accepted 2025-03-06