Metamorphic Testing for (Graphics) Compilers

[Short Paper]

Alastair F. Donaldson
Imperial College London
alastair.donaldson@imperial.ac.uk

Andrei Lascu
Imperial College London
andrei.lascu10@imperial.ac.uk

ABSTRACT

We present strategies for metamorphic testing of compilers using opaque value injection, and experiences using the method to test compilers for the OpenGL shading language.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

Compiler testing; metamorphic testing; OpenGL; graphics

1. INTRODUCTION

Practically all software deployed today has been compiled or is interpreted at runtime, and methods for testing compilers and interpreters have thus received a lot of research attention (see e.g. [4, 5, 6, 7, 8]). A notable method for testing compilers is *random differential testing*, popularised by the Csmith tool [8], whereby a program is randomly generated (a process known as *fuzzing*) and then compiled by different compilers at multiple optimisation levels. Mismatches in the behaviour of the resulting binaries indicate compiler bugs.

A recent alternative strategy is *equivalence modulo inputs* (EMI) testing [4]. Given a well-defined, deterministic program P, EMI testing involves first performing code coverage analysis of P with respect to an input I to identify *I-dead* statements: statements not covered by I. From P, a series of program variants, $P_1, P_2, \ldots, P_n$ can be created, with each P_i obtained by mutating or deleting a subset of the I-dead statements of P. Each P_i should behave identically to P *when executed on input I*; deviations in behaviour are indicative of compiler bugs. Thus, EMI testing is an example of *metamorphic testing* [1]—the programs are in a *metamorphic relationship* with one another and with P, with respect to I. A metamorphic approach has also been used to test compilers by generating input programs that are equivalent by construction [7].

To appear in Proceedings of the 1st International Workshop on Metamorphic Testing, MET'16, May 16, 2016, Austin, TX, USA

In recent work on testing compilers for the OpenCL many-core programming language, we experimented with a variation of EMI testing where instead of identifying existing I-dead code, we injected I-dead code into programs [5]. This works by introducing a new program input variable, v say, and injecting conditional code of the form:

```
if(φ(v)) { /* injected statements */ }
```

where ϕ is a side effect-free predicate over v. By setting v to a runtime value that causes $\phi(v)$ to evaluate to *false*, we make the injected statements "dead-by-construction". As a result, the program should behave identically with or without this injection, so long as the injected statements are syntactically correct and well-typed. Because the runtime value of v is *opaque* to the compiler, the compiler must compile and optimise the program to behave correctly and efficiently for *any* value of v that does not invoke undefined behaviour. The code injection will influence the manner in which the compiler processes the program (at a minimum affecting the way the program is parsed), and this may identify compilation bugs if the injection exposes behavioural differences.

In Section 2 we argue that our approach to metamorphic compiler testing via *opaque value injection* can be applied more broadly, and hypothesise that the technique has the potential to uncover "bugs that matter"—high-priority compiler bugs that a compiler developer should urgently fix. We also argue that our metamorphic testing method is suited to finding bugs in compilers for languages whose semantics are either unclear or offer an envelope of possible behaviours, in which case pure fuzzing may be ineffective.

To support our claims, in Section 3 we present preliminary experience using metamorphic testing to find bugs in compilers for GLSL, the OpenGL shading language [3], evaluated on GPUs and drivers from Intel and NVIDIA. Our method is able to identify minimal changes to open source graphics shaders that should not lead to perceptible changes in the rendered image, yet lead to drastically different results (see Figure 1). We find that a simple image comparison metric suffices to ignore cases where metamorphic injections lead to slight variations in the rendered image (permissible due to the loose specification of floating-point semantics in GLSL), while flagging up cases where the image is likely to be deemed incorrect by a human observer.

2. METAMORPHIC COMPILER TESTING VIA OPAQUE VALUE INJECTION

We discuss strategies for injecting opaque values into input programs, outline some metamorphic transformations

that opaque values enable, and explain that it is straightforward to compute, from a bug-inducing program, a minimal set of transformations that expose the bug. We argue why this approach may be effective in finding high-priority bugs, as well as coping with imprecise or under-specified language semantics. Though the approach we propose is general, we illustrate our ideas by referring to our testing of GLSL compilers, detailed further in Section 3.

Injecting opaque values. The crux of our method is to augment a program with one or more *opaque values*: fresh variables that will take fixed values at runtime, but whose values are unknown to the compiler. The manner for achieving this varies between languages, but a straightforward method is usually apparent. For C programs, opaque values can be injected by passing additional command-line arguments to the program, by declaring additional program variables and populating their values by reading from a specific file, or by marking such variables as `volatile` and initialising them with desired concrete values (the compiler should assume nothing about the values of `volatile`-qualified variables, thus should not perform constant propagation based on their initial values).

The transformations described below require using opaque values to construct expressions that will evaluate to *true*, *false*, 1 and 0 at runtime. We denote these *opaque expressions* by $\mathbb{T}$, $\mathbb{F}$, $\mathbb{1}$, and $\mathbb{0}$, respectively. We refer to $\mathbb{T}$ as "an opaque *true* expression", and similarly for $\mathbb{F}$, $\mathbb{0}$ and $\mathbb{1}$.

To test GLSL compilers, we add a new `uniform` declaration [3, p. 46] to a shader—a two-element floating-point vector, `INJ`—which we populate with the value $(0.0, 1.0)$ at runtime. We can then define e.g. $\mathbb{T}$ = `INJ.x < INJ.y` and $\mathbb{0}$ = `INJ.x`. We could also use more elaborate expressions, e.g. defining $\mathbb{0}$ = `(INJ.x * `d`)`, where d is an expression produced by a fuzzer.

Example program transformations. So far we have experimented with metamorphic transformations based on *dead code injection* and *identity functions*. We describe these, and offer suggestions for additional transformations that may be useful for testing compilers.

Dead code injection. We discussed the use of an opaque *false* expression to inject "dead-by-construction" code in Section 1, by introducing conditional statements of the form `if(`$\mathbb{F}$`) { ... }`. The use of $\mathbb{F}$, instead of `false`, means that the compiler cannot automatically optimize away the injected code. Because it will *not* be executed, arbitrary syntactically valid code can be injected, including code that would invoke undefined behaviour. In [5] we experimented with injecting code generated by a fuzzer. In our GLSL testing framework we instead consider transplanting code fragments from one real-world shader program into another (see Section 3). Further research is needed to understand the relative effectiveness of these sources of code fragments.

Identity functions. Replacing an integer-valued expression e with `(`e` + `$\mathbb{0}$`)`, `(`e` - `$\mathbb{0}$`)`, `(`e` * `$\mathbb{1}$`)`, `(`e` / `$\mathbb{1}$`)` or `(`$\mathbb{T}$` ? `e` : `d`)` (for an arbitrary expression d, e.g. produced by a fuzzer, and where $\mathbb{0}$ and $\mathbb{1}$ are cast to integer type if necessary) should clearly have no effect on program behaviour. Similarly, a Boolean-valued expression e can be replaced with `(`e` && `$\mathbb{T}$`)` or `(`$\mathbb{F}$` || `e`)`, and one can imagine many further semantics-preserving *identity functions* over program expressions. The use of opaque expressions prevents the compiler from optimising away identity function applications, though our results in Section 3 show that non-opaque expressions can still be effective in triggering compiler bugs. Identity functions can be recursively applied, so that an expression e can be replaced with e.g. `(`$id_1(e)$` + `$id_2(\mathbb{0})$`)`, where $id_1(e)$ and $id_2(\mathbb{0})$ denote applications of further identity functions to e and $\mathbb{0}$.

Our idea is that by transforming program expressions into syntactically richer but semantically equivalent forms, identity functions will help to exercise under-tested compiler optimisations. Errors in the implementation of such optimisations may then be identified through behavioural differences.

Other metamorphic transformations. There is broad scope for investigating further metamorphic program transformations hinging on opaque values. For example, when testing a compiler for a language that supports pointers, we could manufacture complex potential aliasing scenarios by injecting dead-by-construction code that manipulates the fields of linked structures in interesting ways. As another example, we could obfuscate the control flow graph (CFG) of a program by injecting dead-by-construction `break`, `continue`, `return` and `goto` statements. In each example, the aliasing conditions and control paths that are possible at runtime are not affected, but the injections may cause the compiler to compute different static over-approximations to this information. As compiler optimisations are known to be sensitive to aliasing information and CFG structure, these transformations may be effective at exposing bugs. Metamorphic program transformations proposed in the context of program generation [7] could also be applied in our setting.

Automatic test case reduction. Each of the metamorphic transformations described above can easily be *reversed*. It is thus straightforward to repeatedly reverse transformations to home in on a minimal subset of transformations that induces a behavioural difference.

Let P be a program and let P' denote the program obtained by applying transformations $t_1, \ldots, t_n$ to P. Suppose P and P' behave differently when compiled and executed, indicating a possible compiler bug. Test case reduction proceeds by repeatedly reversing a randomly chosen transformation t_i, re-applying t_i if its removal causes P and P' to behave identically, until no remaining transformation can be reversed while preserving the behavioural difference. The process can be accelerated by attempting to reverse multiple transformations in a single reduction step.

It may also be possible to simplify a transformation whose reversal removes the behavioural difference. For instance, if an identity function has transformed expression e to `(`$\mathbb{T}$` ? `e` : `d`)`, where d is complex, d could be replaced by a simpler expression. Similarly, a dead code injection could be simplified by removing a subset of the injected statements.

Test case reduction for randomly generated programs requires careful use of analysis tools to avoid introducing undefined behaviour during the reduction process [6], otherwise a reduction attempt for a large bug-inducing test case tends to lead to a small, useless program that invokes an undefined behaviour. In contrast, reversal of metamorphic transformations cannot introduce undefined behaviour.

Finding bugs that matter. It is well known that compiler bug reports are treated by compiler developers with varying priorities. A compiler bug-finding tool should ideally find "bugs that matter": bugs that compiler developers will regard as having a high priority to fix.

Our experience using fuzzing to test OpenCL compilers [5] is that although we could find small test cases that indisputably exposed bugs, many of the bugs appeared unlikely

to affect practical OpenCL kernels. For example, OpenCL programmers make infrequent use of structs and unions, and use pointers in a limited fashion, yet many of the bugs we found involved nested structures with pointer fields.

A benefit of our proposed metamorphic approach, also associated with the EMI testing method that inspired our work [4] (but not with an approach that generates equivalent random programs [7]), is that it starts with an *existing* program and produces a minimal change to the program that exposes an erroneous behavioural difference with respect to a given compiler. If the original program carries high value, e.g. if it is an important test case for core functionality, and if the bug-inducing difference is small, it seems reasonable that the compiler bug could affect real-world programs, thus it is plausible that there would be some urgency associated with fixing the bug. Our results in Section 3 show that in some cases very small changes to open source shader programs can lead to drastic differences in the rendered image.

Coping with imprecise semantics. Many programming languages allow an envelope of acceptable behaviours for floating-point operations, and "fast math" compiler optimisations that change floating-point semantics are desirable in domains where a degree of variation in results is acceptable. Testing compilers is challenging in this setting: two different compilers applied to a single program may legitimately produce binaries that give different results when executed, and a metamorphic transformation that would be semantics-preserving over the real numbers (e.g. $e \rightarrow$ `(`e `+ 0)`) may lead to a behavioural difference by influencing compiler optimisation (e.g. by inhibiting constant folding).

This issue affects random differential testing, EMI testing and our metamorphic approach. We hypothesise that our metamorphic approach (and, for the same reason, EMI testing) is better-suited to coping with floating-point behavioural differences compared with random differential testing. Without special measures during generation, a randomly generated program over floating-point data may be subject to more severe accumulation of rounding errors than would typically occur in a real-world program. This may lead to dramatically different outputs when the program is processed by compilers that apply different optimisations and/or executed on architectures for which corner-case aspects of floating-point arithmetic (e.g. whether denormals are flushed to zero) are implementation-defined. In contrast, metamorphic testing compares program variants using one compiler and architecture, so consistent hardware rounding modes can be expected, and differences in compiler optimisations arise only due to metamorphic transformations. Our early results for testing GLSL compilers (Section 3) show that metamorphic transformations do lead to small differences in rendered images, but that these small differences are easy to distinguish from the dramatic changes in image content associated with compiler bugs.

3. TESTING GLSL COMPILERS

We report on preliminary experience applying our metamorphic approach to test compilers for GLSL, the OpenGL shading language [3]. Reliable GLSL compilers are required for portable rendering across GPUs from multiple vendors, and compiler reliability is particularly relevant in the context of safety-critical graphics processing [2].

Tooling framework. Our *injector* tool takes two GLSL shaders, a *recipient* and a *donor*. The `INJ` opaque value is added to the recipient. Each point in the recipient has a percentage chance (controlled by the user) of being selected for injection. At each selected point, a block of code, randomly selected from the donor, is added using *dead code injection*. Free variables in the donated code are either substituted with appropriately-typed variables available in the recipient, or declared at the injection point (the choice is made randomly). After donation, a random percentage of expressions in the enlarged recipient are selected, to which *identity functions* are applied. We have implemented the identity functions described in Section 2 (including expression fuzzing) and several variations thereof, and apply identity functions to expressions with floating-point and signed/unsigned scalar and vector types.

Vendor	Intel	NVIDIA
GPU	Iris Graphics 6100	GeForce GTX 980M
Driver	20.19.15.4352	352.63
OS	Windows 10.0.10240	Ubuntu 15.10
Host CPU	Intel Core i3-5157U	Intel Core i7-4720HQ

Table 1: The platforms used for our experiments

Our *launcher* tool uses a given vertex and fragment shader to render an image that is then saved to disk. We currently use a trivial vertex shader, applying metamorphic transformations to fragment shaders only.

To search for compiler bugs, we use a script that takes a recipient shader and a directory of donor shaders. The script generates the *reference image* associated with the unmodified recipient, then repeatedly invokes the injector and launcher tools, choosing a random donor for each injection into the recipient. We validate each injected shader using the OpenGL reference compiler (`https://www.khronos.org/opengles/sdk/tools/Reference-Compiler/`), discarding invalid shaders (our prototype injector tool sometimes yields invalid programs). Valid injected shaders that produce an image different from the reference image are flagged.

On finding a difference, our *reducer* tool uses the iterative reduction strategy outlined in Section 2 to find a minimal set of simplified injections that trigger the difference. In some cases we are able to manually simplify the shader further.

Impact of floating-point semantics. The GLSL specification allows for some flexibility in the way floating-point operations are implemented on different GPUs. For example [3, p. 83]: *"Any denormalized value ... can be flushed to 0. The rounding mode cannot be set and is undefined. NaNs are not required to be generated."* A degree of flexibility in the optimisations that a compiler may perform is also provided: [3, p. 88] *"Without any [precision] qualifiers, implementations are permitted to perform such optimizations that effectively modify the order or number of operations used to evaluate an expression, even if those optimizations may produce slightly different results relative to unoptimized code."* As such, we hypothesised that in the absence of a compiler bug, metamorphic transformations might still trigger small differences in rendered images.

To account for this, we use the OpenCV library (`http://opencv.org/`) to compare images based on the *chi-square* distance between their associated colour histograms, regarding images as distinguishable if and only if this distance is larger than a given threshold value. Our hypothesis was

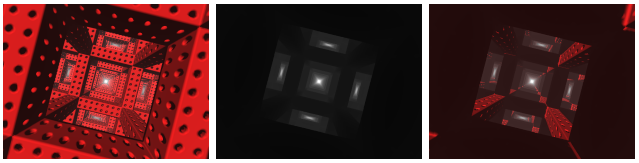

Figure 1: Reference image (left) and images due to *Intel* (middle) and *NVIDIA* (right) compiler issues

that we would be able to find a reasonable threshold to differentiate between image differences due to compiler bugs (exceeding the threshold), vs. small differences arising due to floating-point issues (lying below the threshold).

Experimental results. We have experimented in an exploratory fashion with 16 fragment shaders from `http://glslsandbox.com`, used both as recipients and donors. We searched for bugs in GLSL compilers from Intel and NVIDIA using the platforms detailed in Table 1, which we refer to as *Intel* and *NVIDIA*.

We illustrate our findings with an example recipient shader which, unmodified, produces the left-hand image of Figure 1.[1] The shader is 74 lines of GLSL after preprocessing, which is fairly large compared with typical fragment shaders.

Experimenting with 100 metamorphic variants based on this shader, we found that the resulting images always differed from the reference image, on both *Intel* and *NVIDIA*. The differences were typically very small, and visually imperceptible to us. For example, replacing an expression `normalize(vec3(0.1, 0.4, 0.0))` with `normalize(vec3(0.1 * INJ.y, 0.4, 0.0))` (recall that `INJ.y` is set to 1.0) led to a difference in one out of 640×480 pixels on *Intel*, with the R component of an RGB pixel being decreased by 1 (with colour values lying in the integer range $[0, 255]$). We speculate that multiplication by `INJ.y`, which is not a compile-time constant, may affect the floating-point optimisations performed by the compiler. This is legitimate according to the GLSL specification, as discussed above.

We also found small pixel differences between the reference images computed on *Intel* and *NVIDIA*, and between the images produced on these platforms for any single metamorphic variant.

The remaining images in Figure 1 illustrate that in some cases we observed radical differences due to metamorphic transformations. The middle image was rendered on *Intel* and arises from replacing expression `diffuse` with `(false ? mix(targetDepth, time, false) : diffuse)`, where `diffuse`, `targetDepth` and `time` are `float` variables and `mix` is a GLSL built-in function [3]. (Issue reported to Intel, awaiting confirmation.) The right-hand image was rendered on *NVIDIA*. Provoking this bug required several identity functions to be applied simultaneously, with the most complex example replacing the expression `p + vec3(-EPS, 0.0, 0.0)` with:

```
vec3(((p + vec3(-EPS, 0.0, 0.0))[0]), ((p + vec3(-EPS,
  0.0, 0.0))[1]), (false ?  -EPS : p[2])) + vec3(0.0),
```

where `p` is a 3D floating-point vector. These expressions are indeed equivalent, modulo floating-point effects. (Issue reported to NVIDIA, who have confirmed reproduction but not yet whether the issue is indicative of a bug). We also found and reported two *Intel* front-end bugs, where the compiler rejects valid expressions produced by our expression fuzzer. (Both issues confirmed and fixed by Intel.)

These metamorphic transformations do not actually make use of the opaque values provided by `INJ`. Given that the injections are based on compile-time constants, we are surprised that the compiler does not optimise them away. We have also found bug-triggering variants that depend on dead code injection based on opaque values.

We have found that images indicative of compiler bugs are usually radically different form the reference, as Figure 1 shows, and in many cases a black image is produced. In our experiments so far, a chi-squared threshold of 5 has sufficed to identify all bug images, but lets through a small number of images that exhibit significant pixel-level variation, yet to us look visually identical to the reference image.

[1]See `http://glslsandbox.com/e#29059` for an animation based on the shader.

4. CONCLUSIONS AND FUTURE WORK

We have argued that metamorphic testing using opaque value injection can be employed generally to test programming language implementations, and demonstrated that this method is effective at exposing bugs in compilers for GLSL, an interesting domain because of its imprecise rules on floating-point semantics. The next steps for our GLSL project include extending our tool chain to implement a wider range of metamorphic transformations, testing GLSL compilers from a wider range of vendors, and eliciting feedback from vendors on the bugs our technique finds. It would also be interesting to investigate applying this metamorphic testing approach to implementations of other programming languages.

5. ACKNOWLEDGEMENTS

Our thanks to Ajit Dingankar (Intel) and Vinod Grover (NVIDIA) for liaising regarding compiler issues, to Paul Thomson and Thomas Wahl for feedback, and to Thibaud Lutellier for suggesting using the chi-square histogram distance. This work is supported in part by the EPSRC-funded HiPEDS CDT and a gift from Intel Corporation.

6. REFERENCES

[1] T. Chen, S. Cheung, and S. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Hong Kong University of Science and Technology.

[2] Khronos Group. Khronos invites industry participation to create safety critical graphics and compute standards, `https://www.khronos.org/news/press/`, August 2015. Retrieved 26 January 2016.

[3] Khronos Group. The OpenGL shading language, languge version 4.50, revision 5, January 2015.

[4] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *PLDI*, 2014.

[5] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson. Many-core compiler fuzzing. In *PLDI*, 2015.

[6] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *PLDI*, 2012.

[7] Q. Tao, W. Wu, C. Zhao, and W. Shen. An automatic testing approach for compiler based on metamorphic testing technique. In *ASPEC*, 2010.

[8] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.